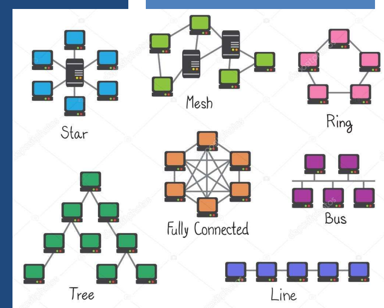
TCSS 558: APPLIED DISTRIBUTED COMPUTING



Chapter 6: Coordination

Consensus Algorithms

Chapter 7: Consistency and Replication

Wes J. Lloyd
School of Engineering and Technology
University of Washington - Tacoma

OBJECTIVES

- Homework 2
- Extra Credit – to be posted
- Active Reading Quiz
- Ch. 6 – Coordination
- 6.2 Logical clocks, Lamport clocks, Vector clocks
- 6.3 Distributed mutual exclusion
- 6.4 Election algorithms
- RAFT Consensus algorithm
- Chapter 7 Consistency and Replication

HOMEWORK 2 UPDATE

- Extension to Thursday 3/14 @ 11:59pm
- Please use extra time to ensure support for multithreading and concurrency
- More time to implement extra credit membership tracking methods
- 5 points extra credit for providing Maven build files (pom.xml)

SHORT-HAND-CODES FOR MEMBERSHIP TRACKING APPROACHES

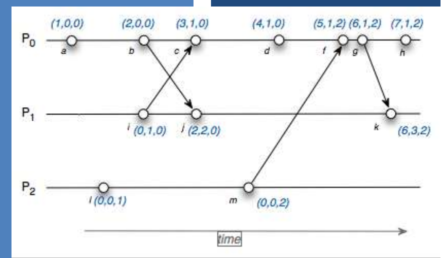
- Include readme.txt or doc file with instructions in submission
- Must document membership tracking method
- **S-1:** Static file membership tracking only = 0 pts
- **T-1:** TCP membership tracking only = +5 pts (*should be dynamic once servers point to membership server*)
- **U-1:** UDP membership tracking only = +10 pts (*automatically discovers nodes with no configuration*)
- **S+T-2:** Static file + TCP membership tracking = +15 pts (*Static file is not reread to refresh membership during operation*)
- **S+U-2:** Static file + UDP membership tracking = +15 pts (*Static file is not reread to refresh membership during operation*)
- **SD+U-2:** Static file + UDP membership tracking = +20 pts (*Static file is periodically reread to refresh membership during operation*)
- **T+U-2:** TCP + UDP membership tracking = 20 pts (*both dynamic*)

FEEDBACK – 3/6

- **For the static N-dimensional hypercube, broadcast requires N-1 messages**
- ***Does it mean that N-1 is the minimal message it can spread?***
- Message spreading algorithms are typically concerned with how to disseminate messages across unstructured adhoc topologies
  - As the size of the network is unknown, the goal is to experiment with different approaches to message spreading and spread termination
- N-1 is the minimum messages to fully disseminate a message starting at one node, to the entire hypercube

FEEDBACK - 2

- **What is the difference between a logical clock and a vector clock?**
- Covered today…
- Lamport clocks don't help to determine causal ordering of messages
- ***What is causality?***
- Vector clocks support the capture of causal histories and can be used as an alternative to Lamport clocks
  - Messages stamped with vector clocks
  - All processes tracks all others view of logical time



CH. 6.2: LOGICAL CLOCKS

L15.7

LOGICAL CLOCKS

- In distributed systems, synchronizing to actual time may not be required…
- It may be sufficient for every node to simply agree on a current time (e.g. logical)
- ***Logical clocks*** provide a mechanism for capturing chronological and ***causal*** relationships in a distributed system
- Think ***counters . . .***
- Leslie Lamport [1978] seminal paper showed that absolute clock synchronization often is not required
- Processes simply need to agree on the order in which events occur

LOGICAL CLOCKS - 2

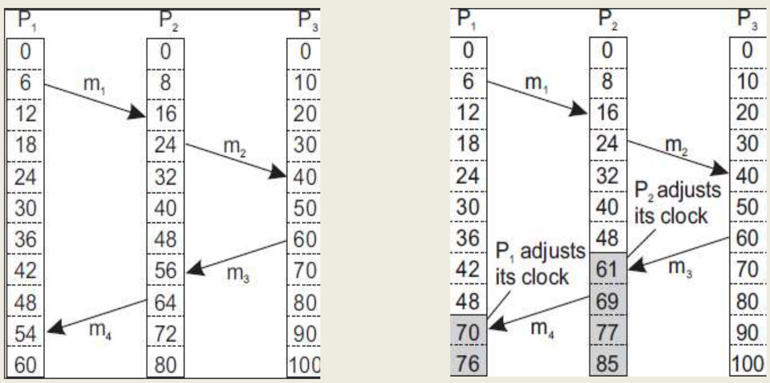
- **Happens-before relation**
- A→B: **Event A**, happens before **event B**…
- All processes must agree that **event A** occurs first
- Then afterward, **event B**
- Actual time not important. . .
- If **event A** is the event of proc P1 sending a msg to a proc P2, and **event B** is the event of proc P2 receiving the msg, then A→B is also true. . .
- The assumption here is that message delivery takes time
- Happens before is a transitive relation:
- A→B, B→C, therefore A→C

LOGICAL CLOCKS – 3

- If two events, say event X and event Y do not exchange messages, not even via third parties, then the sequence of X→Y vs. Y→X **can not be determined!!**
- Within the system, these events appear **concurrent**
- **Concurrent:** nothing can be said about when the events happened, or which event occurred first
- Clock time, C, must always go forward (increasing), never backward (decreasing)
- Corrections to time can be made by adding a positive value, but never by subtracting one

LOGICAL CLOCKS - 4

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



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

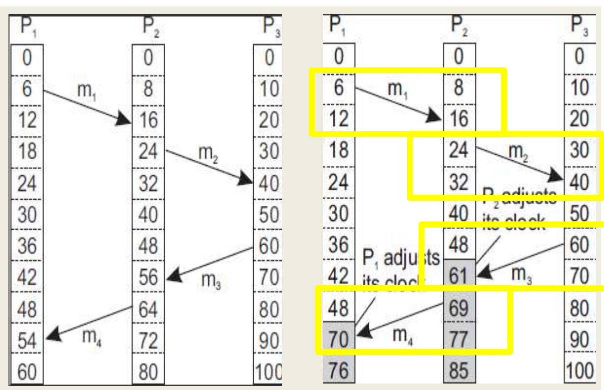
56: P2 receives m3

56: P2 clock reset=61

64: P2 sends m4 to P1

54: P1 receives m4

70: P1 clock reset=70



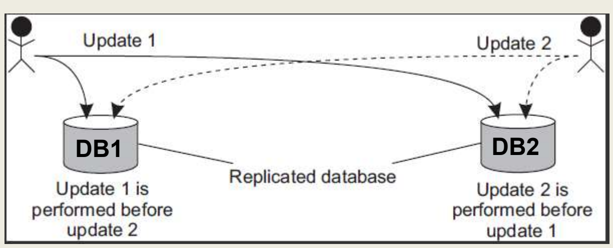
LAMPORT LOGICAL CLOCKS - IMPLEMENTATION

- Negative values not possible
- When a message is received, and the local clock is before the timestamp when then message was sent, the local clock is updated to message_sent_time + 1

1. Clock is incremented before an event: sending a message, receiving a message, some other internal event
   Pi increments Ci: Ci ← Ci + 1
2. When Pi send msg m to Pj, m's timestamp is set to Ci
3. When Pj receives msg m, Pj adjusts its local clock
   Cj ← max{Cj, ts(m)}
4. Ties broken by considering Proc ID: i<j; <40,i> < <40,j>

TOTAL-ORDERED MULTICASTING

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



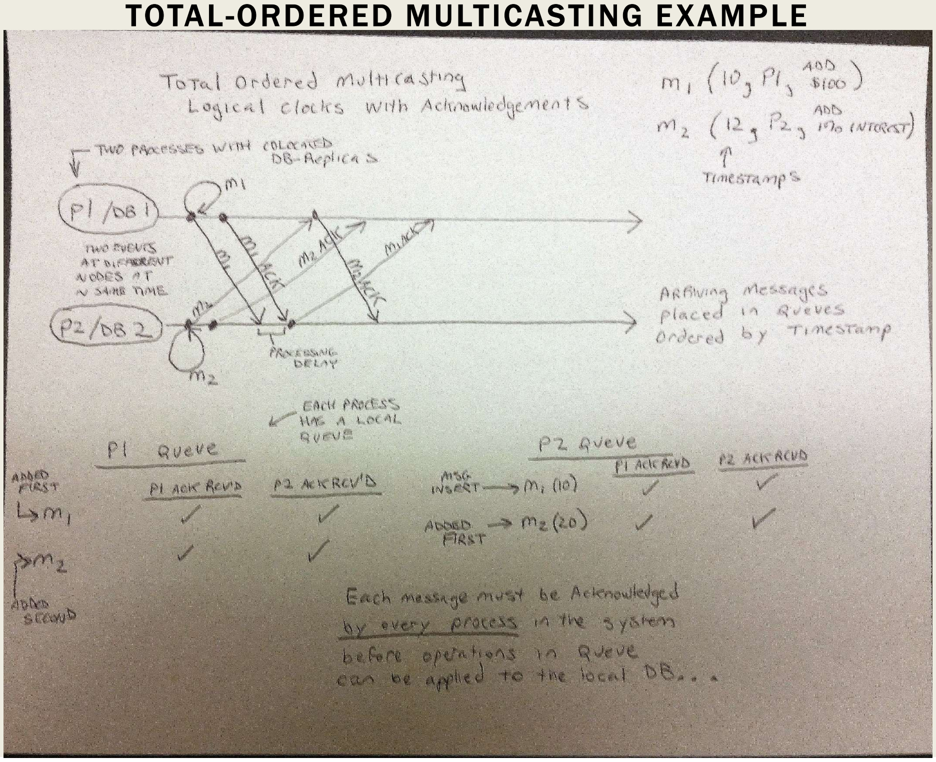
- **Initial Account balance**: $1,000
- **Update #1**: Deposit $100
- **Update #2**: Add 1% Interest
- Total Ordered Multicasting needed

TOTAL-ORDERED MULTICASTING EXAMPLE

- Two messages (m_1, m_2) must be distributed, to two processes (p_1, p_2)
- We assume messages have correct lamport clock timestamps
- m_1(10, p_1, add $100)
- m_2(12, p_2, add 1% interest)
- Each process maintains a queue of messages
- Arriving messages are placed into queues ordered by the lamport clock timestamp
- In each queue, each message must acknowledged by every process in the system before operations can be applied to the local database

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.15

TOTAL-ORDERED MULTICASTING EXAMPLE

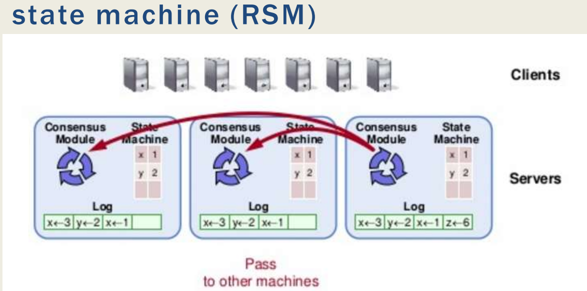


TOTAL-ORDERED MULTICASTING - 2

- Each message timestamped with local logical clock of sender
- Multicast message is conceptually *"sent"* to the sender
- Assumptions:
  - Messages from same sender received in order they were sent
  - No messages are lost
- When messages arrive they are placed in local queue ordered by timestamp
- Receiver multicasts acknowledgement of message receipt to other processes
  - Time stamp of message receipt is lower the acknowledgement
- This process ***replicates*** queues across sites
- Process delivers messages to application (database) only when message at the head of the queue have been acknowledged by ***every*** process in the system

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)

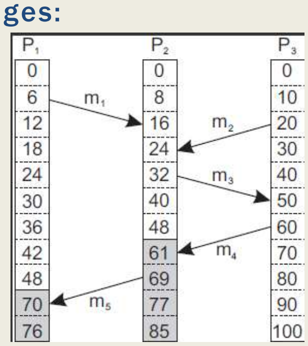


VECTOR CLOCKS

- 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

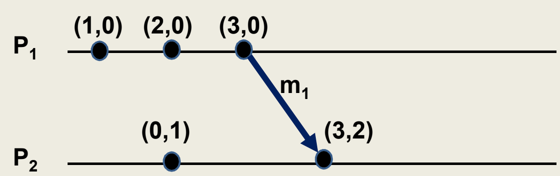
- Vector clocks keep track of <u>causal history</u>
- If two local events happened at process P, then the causal history H(p2) of event p2 is {p1,p2}

- 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

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.21

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)

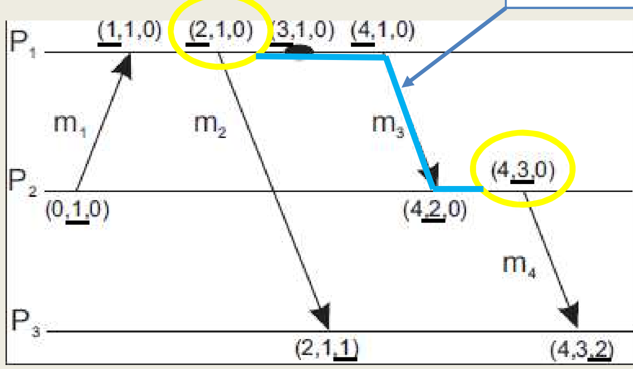
March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.22

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…

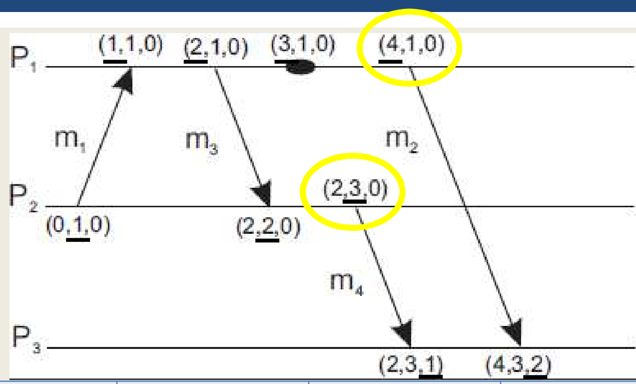
VECTOR CLOCKS EXAMPLE

- 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

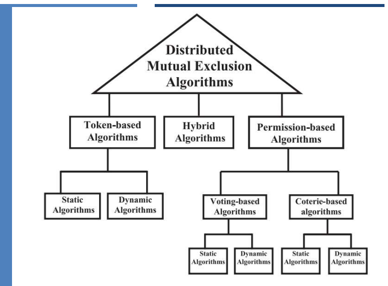


ts (m_2)	ts(m_4)	ts(m_2)<ts(m_4)	ts(m_2)>ts(m_4)	Conclusion
(4,1,0)	(2,3,0)	No	No	m2 and m4 may conflict

- 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

L15.27

DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

- Coordinating access among distributed processes to a shared resource requires **Distributed Mutual Exclusion**
- **Algorithms in 6.3**
- Token-ring algorithm
- Centralized algorithm
- Distributed algorithm (Ricart and Agrawala)
- Decentralized voting algorithm (Lin et al.)

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.28

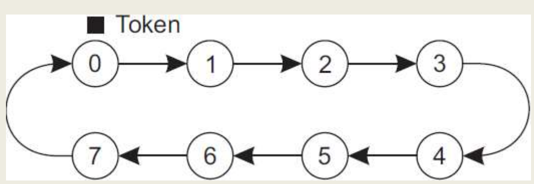
TOKEN-BASED ALGORITHMS

- 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

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.29

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

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.30

TOKEN-RING CHALLENGES

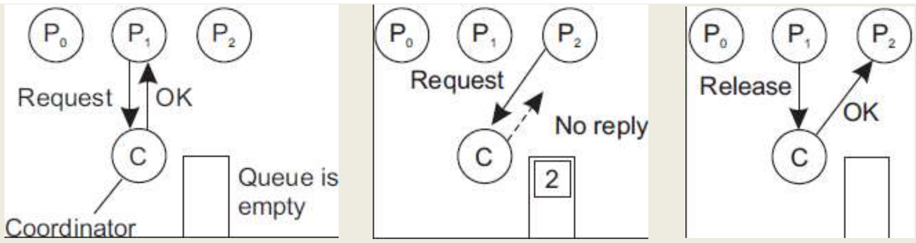
1. If token is lost, token must be regenerated
   - **Problem**: may accidentally circulate multiple tokens
2. Hard to determine if token is lost
   - What is the difference between token being lost and a node holding the token (***lock***) for a long time?
3. When node crashes, circular network route 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 ALGORITHMS - 3

- **Permission-based algorithms**
- Processes must require permission from other processes before first acquiring access to the resource
  - CONTRAST: Token-ring did not ask nodes for permission
- **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

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.35

DISTRIBUTED ALGORITHM - 2

- When each node receives a request message they will:

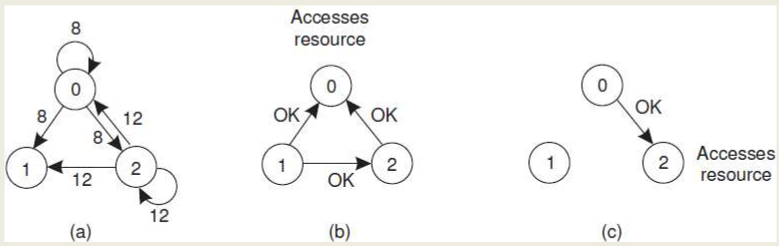
1. Say OK (*if the node doesn't need the resource*)
2. Make no reply, queue request (***node is using the resource)***
3. Perform a timestamp comparison (***if node is waiting to access the resource***), then:
   1. Send OK if requester has lower logical clock value
   2. Make no reply if requester has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

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.36

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
- Node 1 is not interested in the resource, it OKs both requests



- <u>In case of conflict, lowest timestamp wins!</u>
  - As seen in step (c)

CHALLENGES WITH DISTRIBUTED ALGORITHM

- <u>Problem:</u> Algorithm has N points of failure !
- Where N = Number of Nodes in the system

- <u>Problem:</u> When node is accessing the resource, it does not respond
  - Lack of response can be confused with <u>failure</u>
  - <u>*Possible Solution:*</u> 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

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.39

DECENTRALIZED ALGORITHM

- Lin et al. [2004], decentralized voting algorithm
- Resource is replicated N times
- Each replica has its own coordinator
- Accessing resource requires majority vote: Votes from m > N/2 coordinators
- **Assumption #1:** When coordinator does not give permission to access a resource (because it is busy) it will inform the requester

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.40

DECENTRALIZED ALGORITHM - 2

- **Assumption #2:** When a coordinator crashes, it recovers quickly, but will have forgotten votes before the crash.
- Approach assumes coordinators reset **arbitrarily** at any time
- **Risk**: on crash, coordinator forgets it previously granted permission to the shared resource, and on recovery it errantly grants permission again
- **The Hope**: if coordinator crashes, *upon recovery*, ***the node granted access to the resource has already finished before the restored coordinator grants access again . . .***

DECENTRALIZED ALGORITHM - 3

- Even with conservative probability values, the chance of violating correctness **is so low** it can be neglected in comparison to other types of failure
- Leverages fact that a new node must obtain a majority vote to access resource, ***which requires time***

N	m	p	Violation
8	5	3 sec/hour	$< 10^{-15}$
8	6	3 sec/hour	$< 10^{-18}$
16	9	3 sec/hour	$< 10^{-27}$
16	12	3 sec/hour	$< 10^{-36}$
32	17	3 sec/hour	$< 10^{-52}$
32	24	3 sec/hour	$< 10^{-73}$

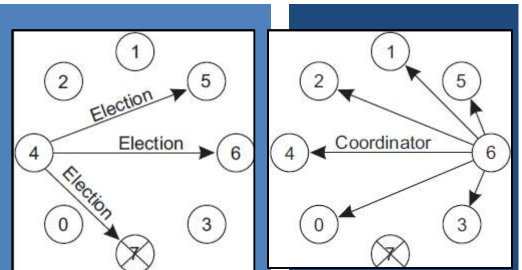
N	m	p	Violation
8	5	30 sec/hour	$< 10^{-10}$
8	6	30 sec/hour	$< 10^{-11}$
16	9	30 sec/hour	$< 10^{-18}$
16	12	30 sec/hour	$< 10^{-24}$
32	17	30 sec/hour	$< 10^{-35}$
32	24	30 sec/hour	$< 10^{-49}$

N = number of resource replicas, m = required “majority” vote

DECENTRALIZED ALGORITHM - 4

- **Back-off Polling Approach for *permission-denied*:**
- If permission to access a resource is denied via majority vote, process can poll to gain access again with a ***random*** delay (***known as back-off***)
- If too many nodes compete to gain access to a resource, majority vote can lead to low resource utilization
  - ***No one can achieve majority vote to obtain access to the shared resource***
- Problem Solution detailed in [Lin et al. 2014]

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.43



CH. 6.4: ELECTION ALGORITHMS

L15.44

ELECTION ALGORITHMS

- Many distributed systems require one process to act as a coordinator, initiator, or provide some special role
- Generally any node (or process) can take on the role
  - In some situations there are special requirements
  - Resource requirements: compute power, network capacity
  - Data: access to certain data/information
- Assumption:
  - Every node has access to a "node directory"
  - Process/node ID, IP address, port, etc.
  - Node directory may not know "current" node availability
- Goal of election: at conclusion all nodes agree on a coordinator

ELECTION ALGORITHMS

- Consider a distributed system with N processes (*or nodes*)
- Every process has an identifier id(P)
- Election algorithms attempt to locate the highest numbered process to designate as coordinator

- **Algorithms:**
- Bully algorithm
- Ring algorithm
- Elections in wireless environments
- Elections in large-scale systems

BULLY ALGORITHM

- When ***any*** process notices the coordinator is no longer responding to requests, it initiates an election
- Process P_k initiates an election as follows:
  1. P_k sends an ELECTION message to all processes with higher process IDs (P_{k+1}, P_{k+2}, … P_{N-1})
  2. If no one responds, P_k wins the election and becomes coordinator
  3. If one of the higher-ups answers, it takes over and runs the election.
- When the higher numbered process receives an ELECTION message from a lower-numbered colleague, it responds with “OK”, indicating it’s alive, and it takes over the election.

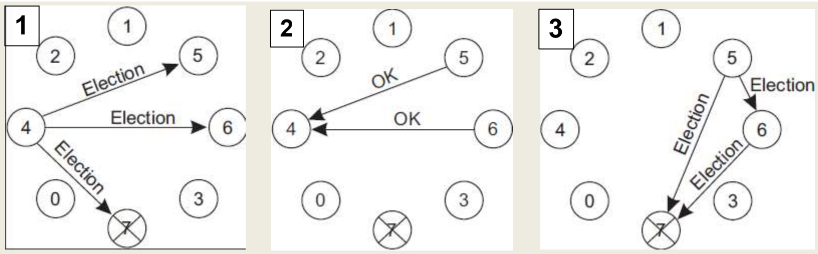
March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.47

BULLY ALGORITHM - 2

- The higher numbered process then holds an election with ***only*** higher numbered processes (nodes).
- Eventually ***all*** processes give up except one, and the remaining process becomes the new coordinator.
- The coordinator announces victory by sending all processes a message stating it is starting as the coordinator.
- If a higher numbered node that was previously down comes back up, it holds an election, and ultimately takes over the coordinator role.
- The process with the “*biggest*” ID in town always wins.
- Hence the name, **bully algorithm**

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.48

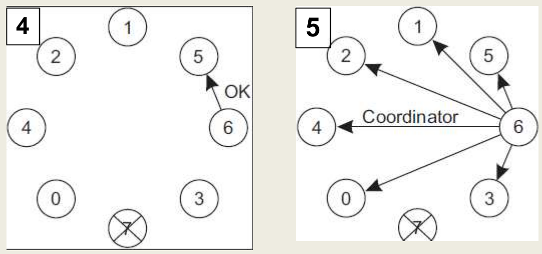
BULLY ALGORITHM - 3



[1] Process 4 holds an election

[2] Process 5 and 6 respond

[3] Process 5 and 6 each hold an election



[4] Process 6 tells Process 5 to stop

[5] Process 6 wins and tells everyone

BULLY SUMMARY

- Every node knows who is participating in the distributed system
  - Each node has a group membership directory
- First process to notice the leader is offline launches a new election
- GOAL: Find the highest number node that is running
  - Loop over the nodes until the highest numbered node is found
  - May require multiple election rounds
- Highest numbered node is always the ***"BULLY"***

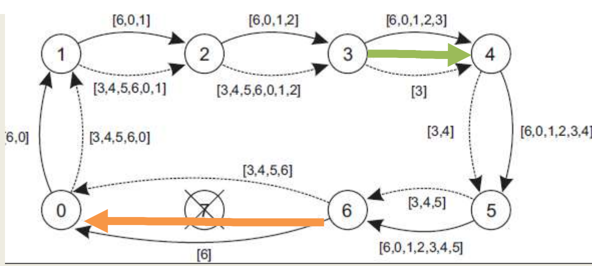
RING ALGORITHM

- Election algorithm based on network of nodes in a logical ring
- *Does not use a token*
- Any process (P_k) starts the election by noticing the coordinator is not functioning

1. P_k builds an **election message**, and sends to its successor
   - If successor is down, successor is skipped
   - Skips continue until a running process is found
2. When the **election message** is passed around, each node adds its ID to a *separate* **active node list**
3. When **election message** returns to P_k, P_k recognizes its own identifier in the **active node list**. Message is changed to COORDINATOR and "**elected(P_k)**" message is circulated.
   - Second message announces P_k is the NEW coordinator

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.51

RING: MULTIPLE ELECTION EXAMPLE



- **PROBLEM:** Two nodes start election at the same time: P_3 and P_6
- P_3 sends **ELECT(P_3)** message, P_6 sends **ELECT(P_6)** message
  - P_3 and P_6 both circulate ELECTION messages at the same time
- Also circulated with ELECT message is an **active node list**
- Each node adds itself to the **active node list**
- Each node votes for the highest numbered candidate
- P_6 wins the election because it's the candidate with the **highest ID**

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.52

ELECTIONS WITH WIRELESS NETWORKS

- Assumptions made by traditional election algorithms not realistic for wireless environments:
  - Message passing is reliable
  - Topology of the network does not change
- A few protocols have been developed for elections in ad hoc wireless networks
- Vasudevan et al. [2004] solution handles failing nodes and partitioning networks.
  - Best leader can be elected, rather than just a random one

VASUDEVAN ET AL. WIRELESS ELECTION

1. Any node (***source***) (P) starts the **election** by sending an ELECTION message to immediate neighbors (any nodes in range)
2. Receiving node (Q) designates sender (P) as parent
3. (Q) Spreads election message to neighbors, ***but not to parent***
4. Node (R), receives message, designates (Q) as parent, and spreads ELECTION message, ***but not to parent***
5. Neighbors that have already selected a parent immediately respond to R.
   - If ***all*** neighbors already have a parent, R is a leaf-node and will report back to Q quickly.
   - When reporting back to Q, R includes metadata regarding battery life and resource capacity
6. Q eventually acknowledges the ELECTION message sent by P, and also indicates the most eligible node (based on battery & resource capacity)

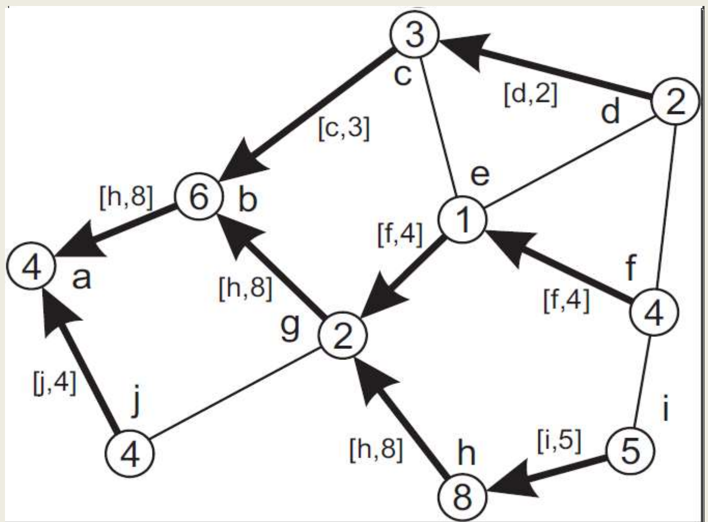
WIRELESS ELECTION - 2
SOURCE NODE: [A]

Node [A] initiates election

Election messages propagated to all nodes

Each node reports to its parent node with best capacity

Node A then facilitates Node H becoming leader



WIRELESS ELECTION - 3

- When multiple elections are initiated, nodes only join one
- Source node tags its ELECTION message with unique identifier, to uniquely identify the election.
- With minor adjustments protocol can operate when the network partitions, and when nodes join and leave

ELECTIONS FOR LARGE-SCALE SYSTEMS

- Large systems often require several nodes to serve as coordinators/leaders
- These nodes are considered ***"super peers"***
- ***Super peers*** must meet operational requirements:

1. Network latency from normal nodes to ***super peers*** must be low
2. ***Super peers*** should be evenly distributed across the overlay network (ensures proper load balancing, availability)
3. Must maintain set ratio of ***super peers*** to normal nodes
4. ***Super peers*** must not serve ***too many*** *normal nodes*

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.57

ELECTIONS FOR DHT BASED SYSTEMS

- DHT-based systems use a bit-string to identify nodes
- **Basic Idea**: Reserve fraction of ID space for super peers
- The first $\log_2(N)$ bits of the key identify super-peers
- m=number of bits of the identifier
- k=# of nodes each node is responsible for (Chord system)
- **Example:**
- For a system with m=8 bit identifier, and k=3 keys per node
- Required number of super peers is $2^{(k-m)} \cdot N$, where N is the number of nodes
  - In this case N=32
  - **Only 1 super peer is required for every 32 nodes**

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.58

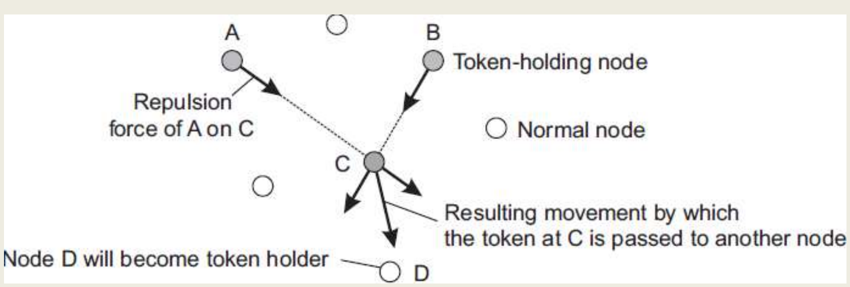
SUPER PEERS IN AN M-DIMENSIONAL SPACE

- Given an overlay network, the idea is to position superpeers throughout the network so they are evenly disbursed
- Use tokens:
- Give N tokens to N randomly chosen nodes
- No node can hold more than (1) token
- Tokens are "repelling force". Other tokens move away
- All tokens exert the same repelling force
- This automates token distribution across an overlay network

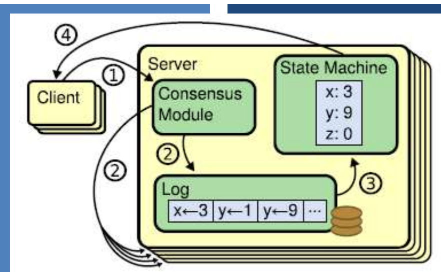
March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.59

OVERLAY TOKEN DISTRIBUTION

- Gossping protocol is used to disseminate token location and force information across the network
- If forces acting on a node with a token exceed a threshold, token is moved away
- Once nodes hold token for awhile they become superpeers



March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.60



RAFT CONSENSUS

L15.61

CONSENSUS IN DISTRIBUTED SYSTEMS

- Paxos Algorithm (originally published in 1989)
- Original algorithm by Leslie Lamport (logical clocks) for consensus
- Single decree Paxos: supports reaching agreement on a single decision
  - To agree on contents of a single log entry
- Multiple decree Paxos: use multiple instances of the protocol to facilitate series of decisions such as a log

- Ensures safety and liveness
- Changes in cluster membership
- Has been proven "correct" (e.g. via proofs)

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.62

PAXOS DRAWBACKS

- As reported by the inventors of RAFT . . .
  - *Diego Ongaro and John Ousterhout from Stanford University*
- Exceptionally difficult to understand
- Most descriptions focus on single-decree version
- Survey at the 2012 USENIX Symposium (UNIX Users Group, Advanced Computing Systems Association)
  - Few seasoned researchers comfortable with Paxos
  - Understanding typically requires reading multiple papers

PROBLEMS WITH PAXOS

- Problem 1: Single Decree Paxos
- Two stages
- Lacks simple intuitive explanation
- Hard to understand why the "single-decree" protocol works
- Used for agreement on just one log entry
- Problem 2: Lacks foundation for building practical implementation
- No widely agreed upon algorithm for multi-Paxos
  - Multi decree for agreement on an entire log file
- Lamport's multi-Paxos description has missing detail
  - Mostly focused on single decree

PROBLEMS WITH PAXOS - 2

- Other attempts to flesh out details are divergent from Lamport's own sketches
- Problem 3: Paxos architecture is poor for building practical systems
- Paxos' notion of consensus is for a single log entry
- Consensus approach can be designed around a sequential log
- Problem 4: Paxos approach uses a symmetric peer-to-peer approach vs. a leader-based approach
  - Works when just (1) decision
  - Having a leader simplifies making multiple decisions

RESULTING PROBLEMS

- Implementations of Paxos typically diverge as each develops a different architecture for solving the difficult problem(s) of implementing Paxos
- Paxos formulation is good for proving theorems about correctness, but challenging to use for implementing real systems
  - Though it has been used a fair bit
  - See paper: Consensus in the Cloud: Paxos Systems Demystified
- Observation: significant gaps between the description of the algorithm and the needs of a real-world system, result in final systems based on divergent, unproven protocols

DESIGN GOALS FOR RAFT

- Complete and practical foundation for building systems
  - Reduce design work for developers
- Safe under all conditions
- Efficient for common operations
- UNDERSTANDABLE
  - So Raft can be implemented and extended as needed in real world scenarios

DESIGN GOALS FOR RAFT - 2

- Raft decomposes consensus into sub-problems:
  - Leader election: leader election algorithms adjustable
  - Log replication: leader accepts log entries and coordinates replication across cluster enforcing log consensus
  - Safety: if any state machine applies a log entry, then no other server can apply a different log entry for the same log index
  - Membership changes: must migrate from old-configuration to new-configuration in a coordinated way

DESIGN GOALS FOR RAFT - 3

- Simplify the state space
- Reduce the number of states to consider
- Make system more coherent
- Eliminate non-determinisim
- LOGS not allowed to have holes
- Limit ways logs can be inconsistent

RAFT ALGORITHM BASICS

- Begins by electing a <u>leader</u>
- <u>Leader</u> manages log replication
- <u>LEADER ACTIVITIES</u>
  - Accepts log entries from other nodes
  - Replicates them on other servers
  - Tells nodes when safe to apply log entries to their state machines (KV store)
  - <u>Leader</u> can make decisions without consulting others
  - Data flows from <u>leader</u> → to nodes
  - When <u>leader</u> fails, a new <u>leader</u> is elected

RAFT BASICS - 2

- Server states: leader, (*)follower, candidate
  - (*) – initial state of every node is follower
- Nodes redirect all requests to the leader
- Candidate server in a leader election
  - Server with most votes wins election, becomes leader
  - Other nodes become followers
  - Each candidate sponsors its own election, and solicits votes
  - More than one candidate can be conducting an election at the same time

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.71

TERMS

- Raft divides time into TERMS of arbitrary length
- Terms are numbered with consecutive integers
- Terms start with an election (term # is incremented)
- If election results in a SPLIT VOTE, term ends, and a *new term* is started with an election
- There is only (1) Leader in any given term
- Terms act as a logical clock
- Each server stores current term number
- Terms are exchanged in communication

March 11, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L15.72

TERMS - 2

- If a larger term # is found, then ***all nodes*** update term # and defer to the term's **leader**
  - If **candidate** or **leader** finds its term is out of date, will immediately become a **follower** node
- If server receives request with stale term #, then request is rejected

RAFT METHODS

- Implemented as "RPCs", but can be implemented as TCP stream by marshalling data inputs/outputs
- **RequestVote()**
- Initiated by **candidates** during an election
- **AppendEntriesToLog()**
- Sent by **leaders** to **follower** nodes at regular intervals
- Used as a heartbeat to maintain leadership
- Provides log updates to nodes
- Performs consistency checks
- Commands are retried if no response after timeout
- Commands sent in parallel using multiple threads (performance)

RAFT ELECTIONS

- Every node has a ***randomized*** ElectionTimeout value
- If a node (**follower**) receives no heartbeat from the **leader** after the timeout, node expects the **leader** has gone offline
- **NEW ELECTION:**
- (1) The node ***begins a new election*** as **candidate**, sending RequestVote() to every node in the system
  - **Candidate** immediately votes for itself
  - RequestVote() sent in parallel to all nodes
- (2) Follower votes for **first** **candidate** a RequestVote() is received from ***only if the candidate's log is at least (or more) up-to-date***
  - *Inspect **candidate** provided last log index and log term values*
- (3) If **candidate** obtains a majority of the votes (*determined by calculating majority total from node directory*) **it wins the election!!!**

December 5, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L18.75

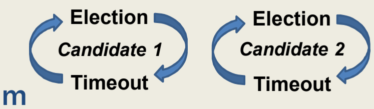
ELECTIONS - 2

- **Election outcomes**
- A – **Candidate** wins
- B – Another server establishes leadership
- C – There is no winner
- Servers vote for only one **candidate**
- Only (1) winner per election
- Only (1) **leader** per term
  - *"Election safety property"*
- New **leader** sends empty heartbeat to nodes to establish leadership

December 5, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L18.76

ELECTIONS - 3

- While a **candidate** waits for votes, it may receive an AppendEntries() call from another **leader**
  - If the **leader's** term >= **candidate's** term then the **candidate** concedes the election and returns to **Follower** state
- If multiple elections, then no one **candidate** may receive a majority vote. One election times out *first* based on a randomized-election-timeout value
  - Random timeout values help spread out the **candidates** to prevent endless looping



- **KEY IDEA**: by using random timeouts, when no majority vote occurs, a random node times out first and starts a new election before anyone else by incrementing the term #, and sending RequestVote()

ELECTIONS - 4

- Randomized timeout values should be reset every time
- Paper suggests a min timeout of 150ms, and max of 300ms
- Timeout should be "an order of magnitude" greater (10x) than the node-to-node communication latency
  - *I'm presently using 500 – 1000ms*
- Can experiment with different values

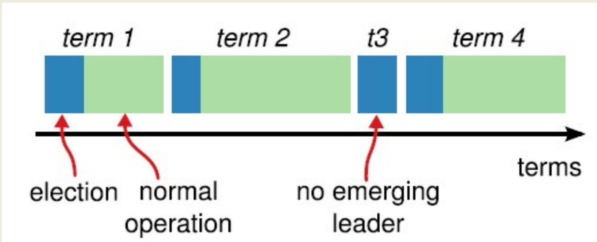
ELECTIONS - 5

- RAFT enforces leader logs to be up-to-date during an election
- Nodes ***ONLY*** vote for a candidate ***if*** :
- **Candidate** local term and log number >= **follower**
- Candidate's log ***must be*** at least as up-to-date as the majority of follower's log

- **MORE up-to-date log is defined as log with:**
- Higher term # in last log entry
- --- OR ---
- When term of last log entries match, log with more entires
- *E.g. longer log*

December 5, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L18.79

TYPICAL ELECTION SEQUENCE



- Term 1: normal election
- Term 2: normal election
- Term 3: SPLIT VOTE, no **leader** emerges, election times out
- Term 4: normal election

December 5, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L18.80

RAFT SAFETY

Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3

- Raft guarantees that each of these properties is always true

LOG REPLICATION

- Leader receives commands forwarded from followers
- Ways logs can diverge
- (a) Follower may be missing entries present on leader
- (b) Follower may have extra entries not present on the leader
- (c) Both A and B
- Because raft uses a "coordinator" node to achieve consensus the number of possible ways logs can diverge is limited
- Raft leaders FORCE followers logs to match its own
- Conflicting entries in follower logs are overwritten

LOG REPLICATION - 2

- FOR THE WHOLE SYSTEM THERE IS JUST ONE MONOTONICALLY INCREASING LOG INDEX
  - Akin to Lamport's Clocks

- Possible *follower* states at start of new term
- (a) Missing entries
- (b) Extra uncommitted entries
- (c) Both

December 5, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L18.83

RAFT - LOG REPLICATION ALGORITHM

- Leader:
1. Receives command(s)
2. Appends commands to local log (concurrent hash table)
3. Sends AppendEntries() to followers

- Leader tracks index of its highest committed log entry
- Provides this index to followers in AppendEntries() RPC

- Leader commit to state machine:
- (1) When log entries replicated at a majority of the followers, leader commits to its state machine (KV-store)

December 5, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L18.84

LOG REPLICATION ALGORITHM - 2

- **Synchronizing follower logs**
- (2) If **follower** rejects AppendEntries() then **leader** decrements its "follower-nextIndex" by one, and *retries* AppendEntries().
  - "follower-nextIndex" tracks which logs entries are sent to the follower for each AppendEntries() RPC call
- Loop continues until **leader** ***walks back*** its "follower-nextIndex" until it **matches** what is committed at the **follower**
  - **Follower** has a **commitIndex**
  - Tracks 1st phase of a "two-phase" commit
  - **Follower** has a **lastApplied** index
  - Tracks 2^{nd} phase of "two-phase" commit
- Once **leader** matches follower-nextIndex, the **follower** accepts the AppendEntries() RPC, and writes data to its log
  - Conflicting log entries are overwritten

LOG REPLICATION ALGORITHM - 3

- Leader based consensus algorithms require the leader to "eventually store" all committed log entries
- Raft handles follower node failure by retrying communication indefinitely
  - If crashed server restarts, the log will be resurrected, and the follower's state machine will be restored (kv-store)

COMMITTING LOG ENTRIES

- Each node keeps a **commitIndex** and **lastApplied** index variable
- **PHASE I**
- Leader: when log message replicated at a majority of follower logs (not state machines) ****- described next slide***
- Leader increments its commitIndex
- Followers set commitIndex to Min (leader-commitIndex , index of last new log entry)

If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

- **PHASE II**
- For any node (follower, leader):
- If commitIndex > lastApplied
  - Increment lastApplied by 1
  - commit log[lastApplied] to **state machine** (kv-store)

If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)

UPDATING COMMIT-INDEX OF LEADER

- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

- ***How leader determines when to update it's commitIndex***
- Use a ***majority consensus*** of what has been committed at follower logs
- **Leader** maintains follower state arrays:
- **nextIndex[]**: index of next log entry to send to follower
- **matchIndex[]**: index of highest log entry known to be replicated (to log) at follower
- Find N, such that N > $\text{commitIndex}_{\text{leader}}$
- **and** a majority of matchIndex[i] ≥ N *(from followers)*
- **and** $\text{log\_entry}_{\text{leader}}$[N].term == $\text{currentTerm}_{\text{leader}}$
- **then** set $\text{commitIndex}_{\text{leader}}$ = N

RAFT CLUSTER MEMBERSHIP – A3

- Cluster discovery performed at startup
- Use any method:
  - Static file, UDP discovery (kv-store), TCP discovery (kv-store)
- One membership is discovered, it can remain static/fixed
- Nodes can go offline, come back online
- One a common configuration is propagated across the system, it can not be changed without restarting
- RAFT specifies a configuration change protocol where the system does a “hand-off” between an old and new configuration (section 6 of the paper)

December 5, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L18.89

A3 RAFT SIMPLIFICATIONS

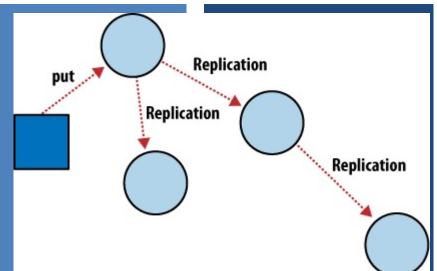
- RequestVote() can be single threaded
  - AppendEntries() probably should have one thread per <u>follower</u>
- TCP client catch exceptions:
  - IOExcpetion – newSocket()
  - IOException – getOutputStream()
  - IOException – getInputStream()
  - <u>Leader</u> should catch exceptions, and retry requests indefinitely
  - Use socket method .setSoTimeout() to set a socket timeout in MS
- Node directory should generate and track nodeIDs
  - E.g. 1, 2, 3, 4, … n
- Node directory should retrieve a node by ID, or IP/PORT

December 5, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L18.90

A3 RAFT SIMPLIFICATIONS - 2

- **Leader** election: if using a single thread for election **candidate** should retry RequestVote() up to 10 times for a **follower** then give-up and move to next **follower**
- Instead of pushing data to **followers** when put() or del() is received by **leader**, can wait until next scheduled heartbeat to **follower**

December 5, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L18.91



CONSISTENCY AND REPLICATION

L19.92

WHY REPLICATE DATA?

(1) Fault tolerance: continue working after one replica crashes

(2) Provide better protection against corrupted data

(3) Performance

(3a) Scaling up systems (***scalability***)

- Replicate server, load balance workload across replicas

(3b) For providing geographically close replicas

- Replicas at the edge
- **MOVE DATA TO THE COMPUTATION**
- Performance ***perceived*** at the edge increases
- **But what is the cost of localized replication?**

DATA REPLICATION COSTS

- Network bandwidth consumed maintaining replicas
  - Updates must be sent out and coordinated
- Maintaining consistency may be difficult
- All copies must be updated to ensure consistency
- **WHEN** and **HOW** updates need to be performed determines the prices of data replication…
- **Web caching example**
- Web browser caches local content to improve performance
- Doesn't know when content is "stale"
- **Solution:** Place server in charge of replication not browser
- Server invalidates and updates client cached copies
- Track how current copies are
- Degrades server performance → overhead from tracking, etc.

REPLICATION TRADEOFF EXAMPLE

- **Process P** accesses a local replica **N** times per second
- Replica is updated **M** times per second
- Updates involve complete refreshes of the data
- If **N << M** (very low access rate*)* many updates **M** are never accessed by **P**.
- Network communication overhead for most updates is useless.
- **TRADEOFFS:**
- Either move the replica away from **P**
  - *So the total number of accesses from multiple processes is higher*
- Or, apply a different strategy for updating the replica
  - *i.e. less frequent updates, possibly need based*
- **BALANCE TRADEOFF BETWEEN REPLICA ACCESS FREQUENCY AND COSTS OF REPLICATION** *(communication overhead)*

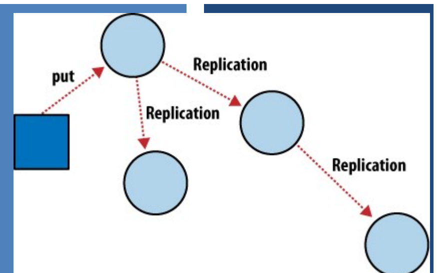
REPLICATION: SCALABILITY ISSUES

- **TIGHT CONSISTENCY**
- Reads must return same result
- Replication must occur after an update, before a read
- Provided by synchronous replication
- Update is performed across all copies as a single atomic operation (or transaction)
- **Assignment 2 replication is with tight consistency.**
- Keeping multiple copies consistent is subject to scalability problems
- May need global ordering of operations (e.g. Lamport clocks), or the use of a coordinator to assign order
- Global synchronization across a wide area network is time consuming (network latency)

REPLICATION SCALABILITY - 2

- Only solution is often to ***relax*** the consistency constraints
- Updates do not need to be executed as atomic operations
- Try to avoid instantaneous global synchronizations
- TRADEOFF: consistency
  - Not all copies may always be the same everywhere
- Whether consistency requirements can be relaxed depends on:
  - Access and update patterns
  - Use cases of the data
- Range of consistency models exist
- Implemented with distribution and consistency protocols

December 7, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L19.97

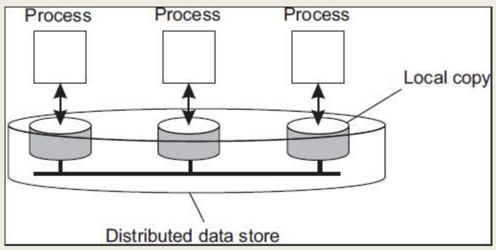


DATA CENTRIC CONSISTENCY MODELS

L19.98

DATA-CONSISTENCY MODELS

- Data consistency is discussed in the context of
  - Distributed shared memory
  - Distributed shared database
  - Distributed shared file system
- Generically referred to as a ***"data store"***
- Each process has a nearby replica:



DATA-CONSISTENCY MODELS

- **CONSISTENCY MODEL**
- Rules that must be followed to ensure consistency
- Represents a contract between processes and data store
- If processes agree to obey certain rules, store promises to work correctly

- No general rules for loosening consistency
- What can be tolerated is highly application dependent

- **Three types of inconsistencies**
- Data variation
- Staleness
- Ordering of update operations

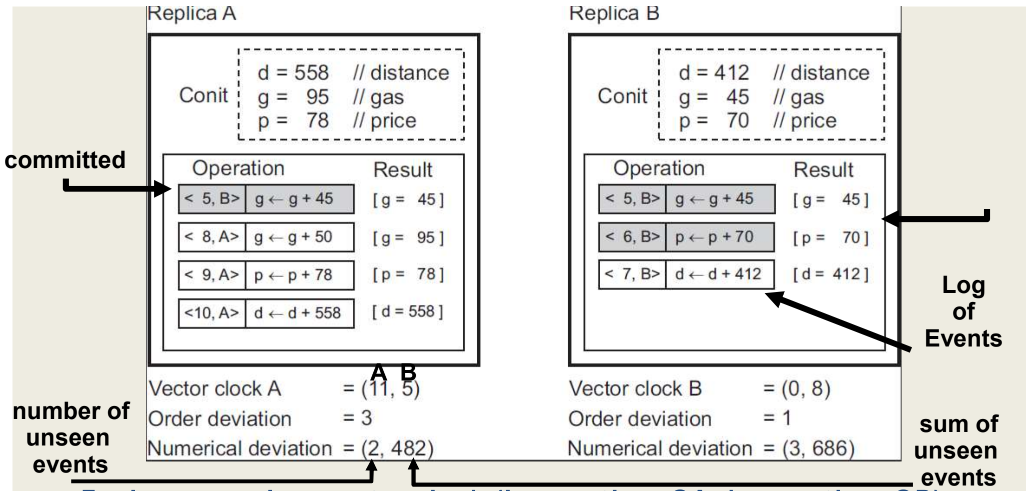
CONTINUOUS CONSISTENCY

- Ranges assigned to "what is allowed" for these deviations:
  - How much data variation?
  - How old/stale can the data be?
  - How much can ordering of update operations vary?
- Idea is to specify bounds for numeric deviation:
- **Relative numeric deviation**: 2% (percent)
- **Absolute numeric deviation**: .2 (implies a particular scale)
- **Numeric deviation**: may also refer to the number of updates applied to a replica
- **Staleness**: specifies bounds relative to time, e.g. how old?
- **Ordering of updates**: updates applied tentatively to local copy; may later be rolled back and applied in different order before becoming permanent

CONSISTENCY UNITS (CONIT)

- Abbreviated as "Conit"
- Specified the unit to measure consistency
- **Example**: Tracking fleet of rental cars
- Variables for a "conit":
- (g) gasoline consumed
- (p) price paid for gasoline
- (d) distance traveled
- Server keep conit consistently replicated

CONSISTENCY UNIT (CONIT)



- Each process has vector clock (known time @A, known time @B)

SEQUENTIAL CONSISTENCY

- Result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear ***in this sequence*** in the order specified by its program.

Sequentially Consistent

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)b	R(x)a

NOT Sequentially Consistent

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

- Exact order seen by processes **DOES NOT MATTER**
- As long as they all agree
- Processes here must see: R(x)b, then R(x)a

CAUSAL CONSISTENCY

- Writes that are potentially causally related ***must be seen*** by all processes ***in the same order***.
- ***Concurrent writes*** may be seen ***in a different order*** by different processes.
- Concurrent writes happen with no READS in between
  - Events can be seen as "concurrent events"
- **Which writes are concurrent?**

P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:	R(x)a				R(x)b	R(x)c

- **Note** how the reads after the concurrent write for P3 and P4 are ***in a different order.***
- This is ok with causal consistency

CAUSAL CONSISTENCY - 2

- **Which timing graphs uphold causal consistency?**
- (A)

P1: W(x)a			
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

- (B)

P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

- **Which writes are concurrent?**
- For (B), since R(x)a can influence W(x)b, the subsequent reads by P3 and P4 **must be in the same order** . . .

ENTRY CONSISTENCY

- Locks can be used to control access to data members
- Releasing a lock tells the distributed system that a variable needs to be synchronized / updated.
- A simple read without obtaining a lock may result in a stale value

P1:	L(x)	W(x)a	L(y)	W(y)b	U(x)	U(y)			
P2:							L(x) R(x)a		R(y) NIL
P3:								L(y) R(y)b	

- Here P2 does not obtain L(y) before reading y R(y)
  - P2 receives a stale/old value

CONSISTENCY VS. COHERENCE

- **Consistency models** define what to expect when processes concurrently operate on distributed data
- Data is consistent, if it adheres to the rules of the model

- **Coherence models**: describe what can be expected for only ***a single data item***
- Data item is replicated
- Data item is coherent when copies adhere to consistency model rules
- Coherence often uses **sequential consistency** applied to a single data item
- For concurrent writes, all processes eventually see the same order of updates

EVENTUAL CONSISTENCY

- If no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.
- System must reconcile differences between multiple distributed copies of data
- Servers must exchange data updates
- Servers must reconcile updates to agree on final state
  - Read repair: correction done when read finds inconsistency
  - Write repair: correct done on write operation
  - Asynchronous repair: correction done independently from read and write

EVENTUAL CONSISTENCY - 2

- Most processes mainly read from data store
  - Rarely update data
- How fast should updates be made to read-only processes?
- Example: Content Delivery Networks (video streaming)
  - Updates are propagated slowly
- Conflicts: write-write and read-write (most common)
- Often acceptable to propagate updates in a lazy manner when most processes perform only READ-ONLY access
- All replica gradually (eventually) become consistent

QUESTIONS

EXTRA SLIDES