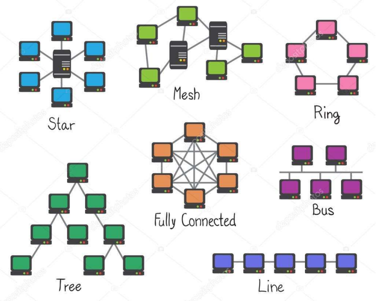
TCSS 558:
APPLIED DISTRIBUTED COMPUTING



Coordination

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

OBJECTIVES

- Assignment #2 Questions
- Assignment #3 Questions
- Feedback from 11/28

- Ch. 6 – Coordination
  - Distributed mutual exclusion
  - Election algorithms

- 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
- Raft Paper

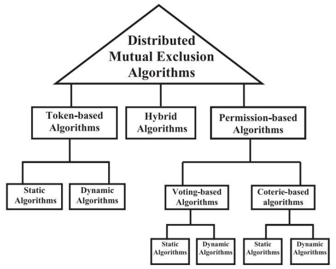
FEEDBACK FROM 11/28

- CENTRALIZED MUTUAL EXCLUSION

- In what node does the coordinator reside?
  - I interpret this question as, how do we select (or elect) a coordinator node?
  - Often election algorithms arbitrarily choose ***any*** node to be coordinator
  - We will cover election algorithms today in class
  - However, sometimes, it may be beneficial to elect a coordinator that has specific resources available (network capacity, memory, CPU capacity, access to special data)

FEEDBACK - 2

- **CENTRALIZED MUTUAL EXCLUSION**
- Does coordinator need continuous communication with the node using the shared resource?
  - The network link between the central coordinator, and the node accessing the share resource **must not be broken**
  - If the network link fails, the user may be done with the resource, but has no way of notifying the coordinator (or the distributed system)
  - In this case, it appears as if the node is still using the resource… ***potentially forever*** =(
- How does the coordinator know if a particular node has failed?
  - The centralized coordinator should probably "ping" nodes accessing the shared resource periodically. If the "pings" are not returned, then potentially the lock should be released



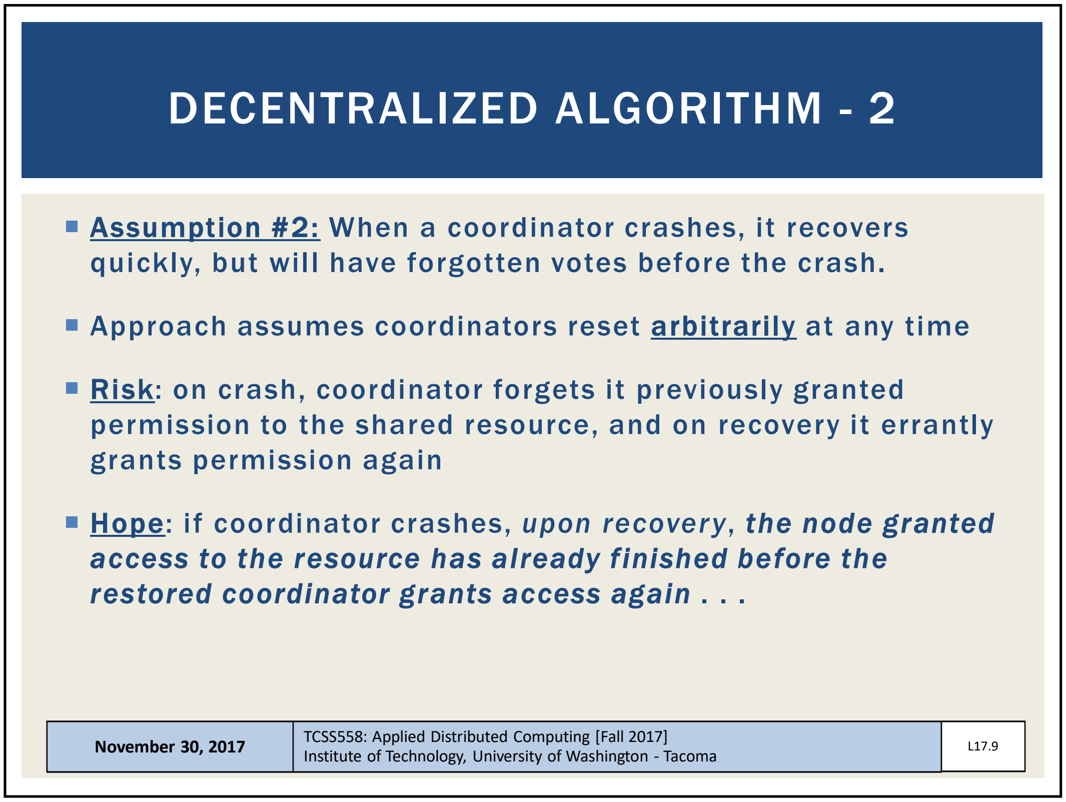
CH. 6.3: DISTRIBUTED MUTUAL EXCLUSION

DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

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

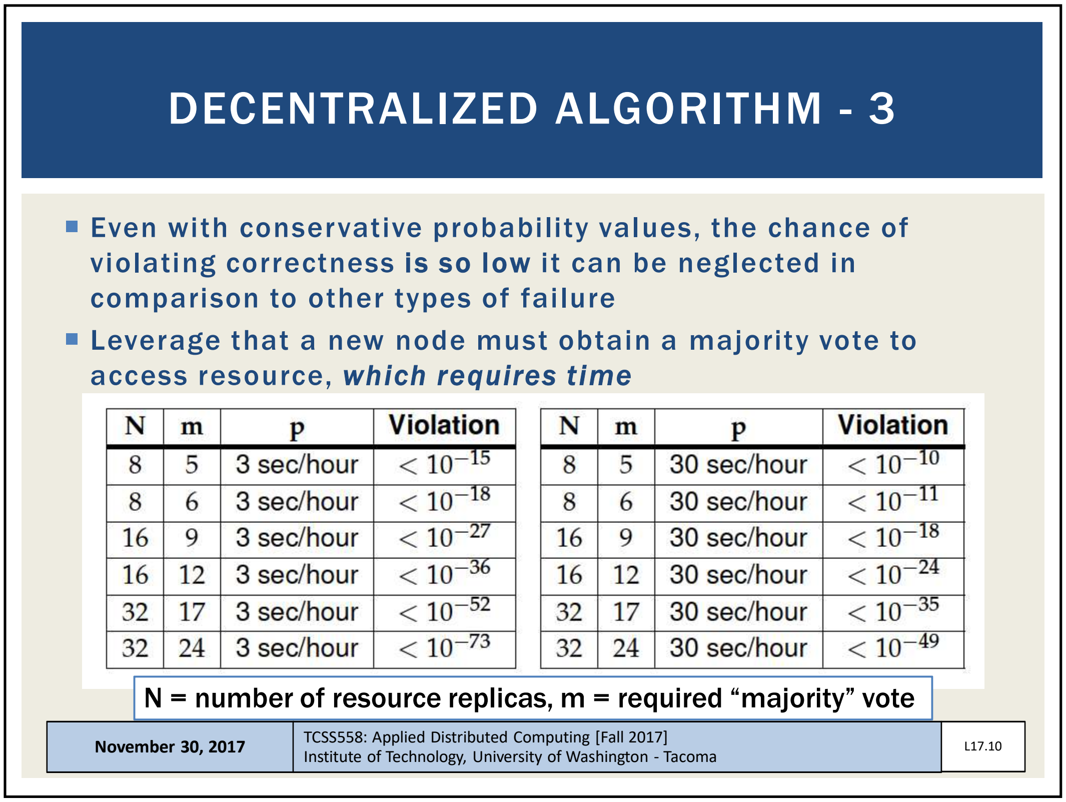
DECENTRALIZED ALGORITHM

- 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



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
- **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
- Leverage that a new node must obtain a majority vote to access resource, ***which requires time***

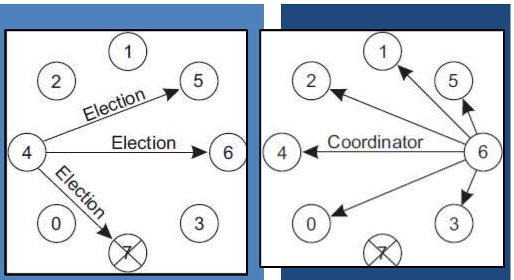
N	m	p	Violation
8	5	3 sec/hour	$< 10^{-15}$
8	6	3 sec/hour	$< 10^{-18}$
16	9	3 sec/hour	$< 10^{-27}$
16	12	3 sec/hour	$< 10^{-36}$
32	17	3 sec/hour	$< 10^{-52}$
32	24	3 sec/hour	$< 10^{-73}$

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]



CH. 6.4: ELECTION ALGORITHMS

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

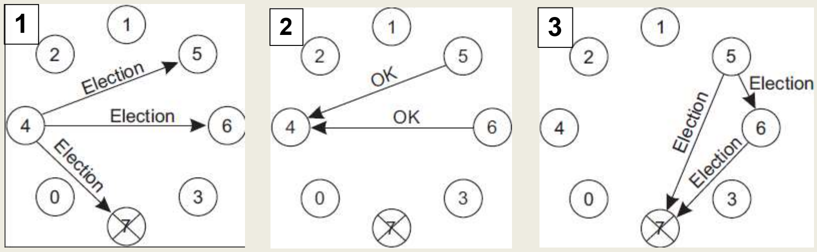
BULLY ALGORITHM

- 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.

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**

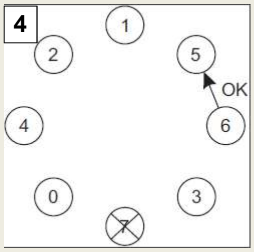
BULLY ALGORITHM - 3

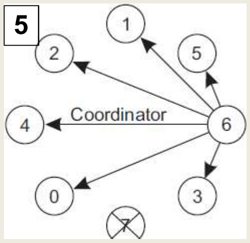


[1] Process 4 holds an election

[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

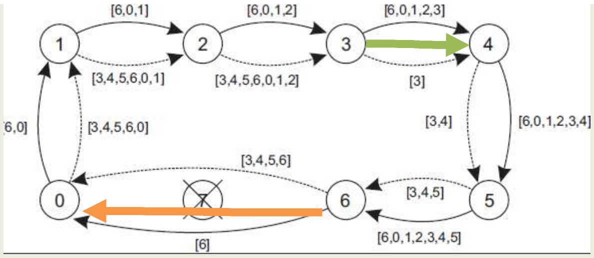
- Every node knows who is participating in the distributed system
  - Each node has a group membership directory
- First process to notice the leader is offline launches a new election
- GOAL: Find the highest number node that is running
  - Loop over the nodes until the highest numbered node is found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

RING: MULTIPLE ELECTION EXAMPLE



- 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 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**

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)

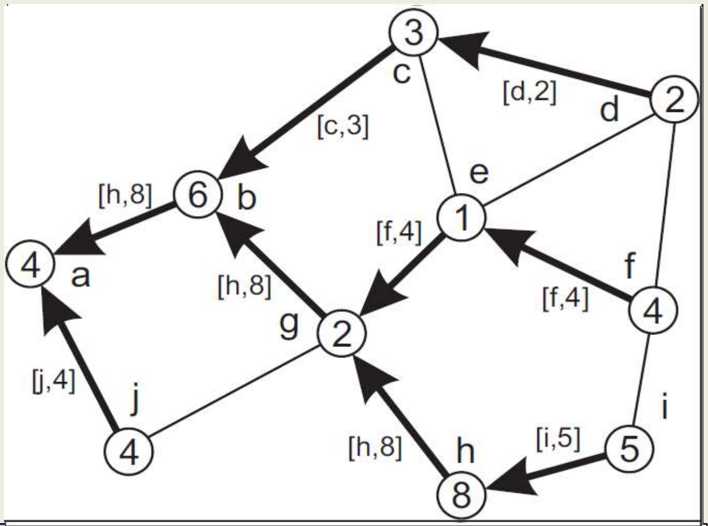
WIRELESS ELECTION - 2
SOURCE NODE: [A]

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*

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

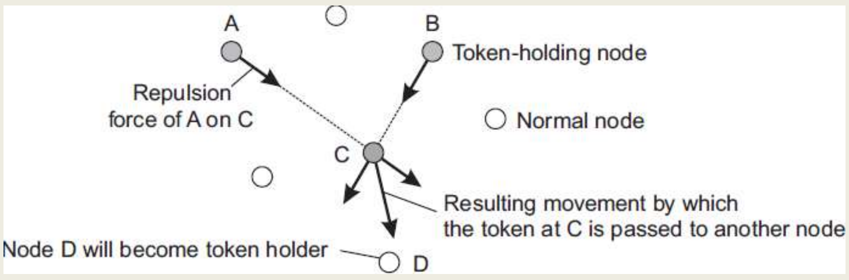
SUPER PEERS IN AN M-DIMENSIONAL SPACE

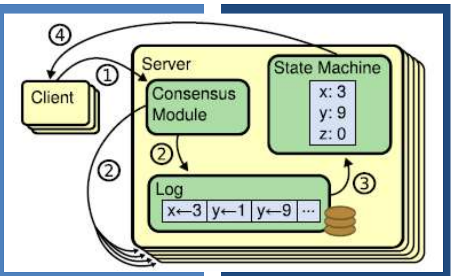
- Given an overlay network, the idea is to position superpeers throughout the network so they are evenly disbursed

- <u>**Use tokens:**</u>
- 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

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 <u>**threshold**</u>, token is moved away
- Once nodes hold token for awhile they become superpeers





RAFT CONSENSUS

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)

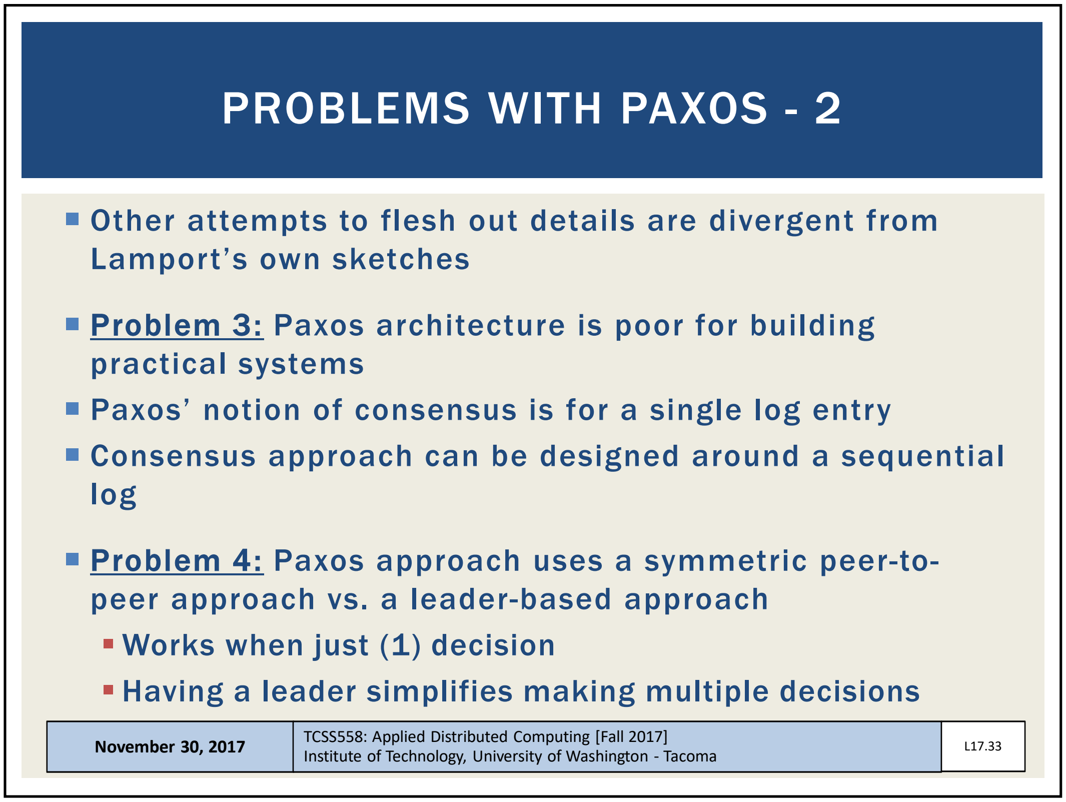
PAXOS DRAWBACKS

- **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

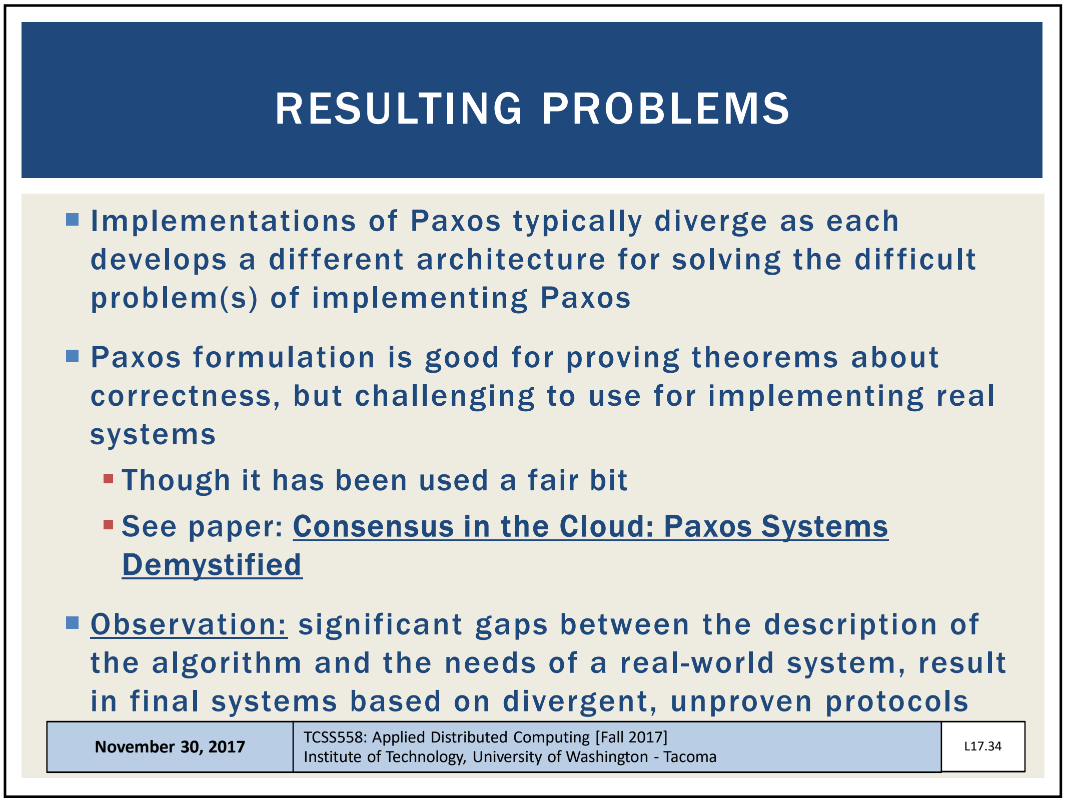
- **Problem 1:** Single Decree Paxos
- Two stages
- Lacks simple intuitive explanation
- Hard to understand why the "single-decree" protocol works
- Used for agreement on just one log entry

- **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



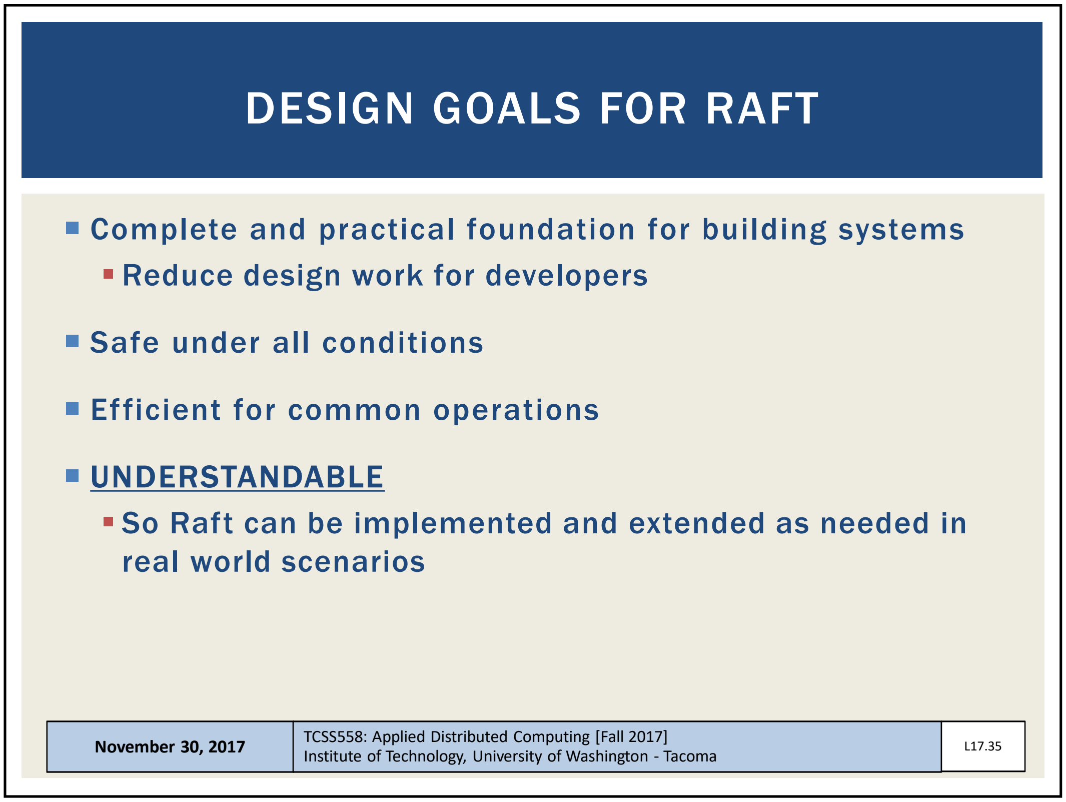
PROBLEMS WITH PAXOS - 2

- 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



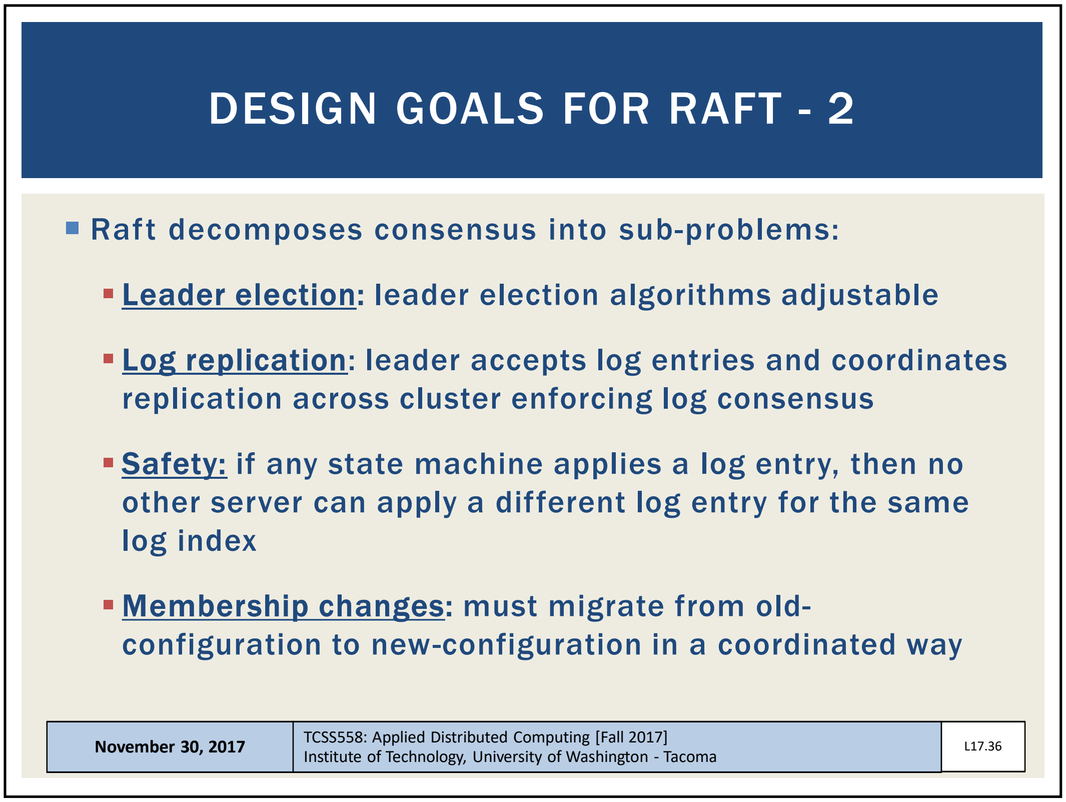
RESULTING PROBLEMS

- 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

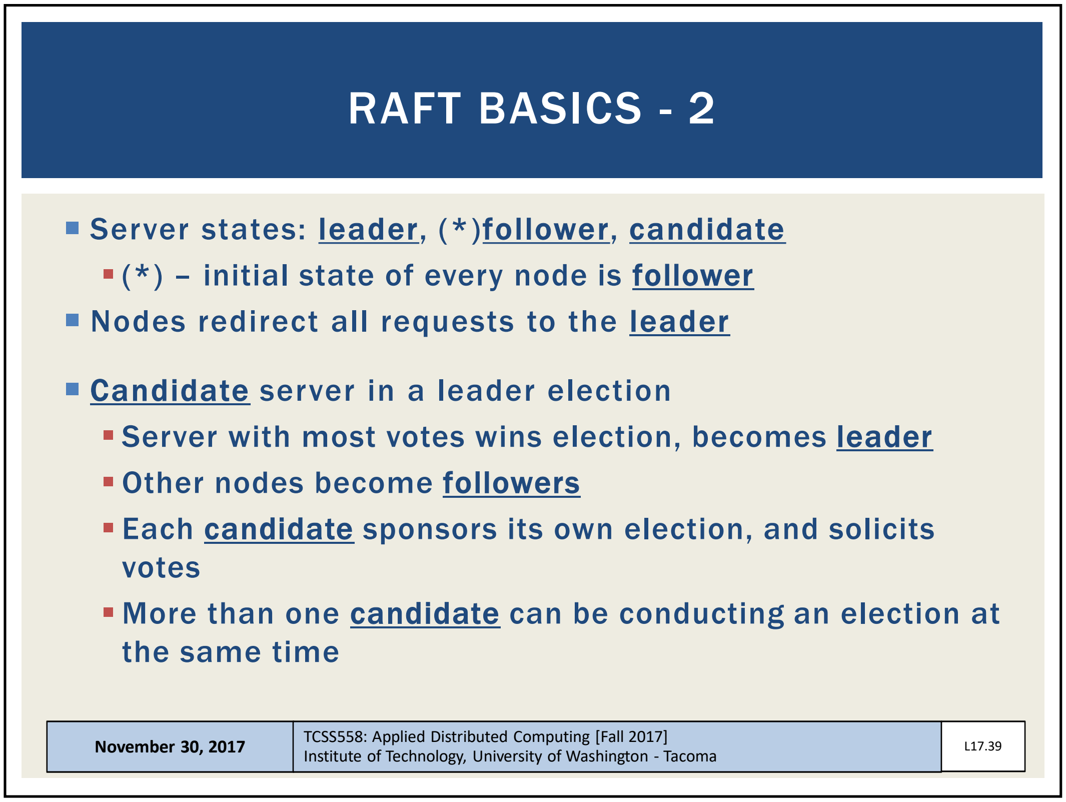
- Simplify the state space
- Reduce the number of states to consider
- Make system more coherent
- Eliminate non-determinisim
- LOGS not allowed to have holes
- Limit ways logs can be inconsistent

RAFT ALGORITHM BASICS

- Begins by electing a <u>leader</u>
- <u>Leader</u> manages log replication

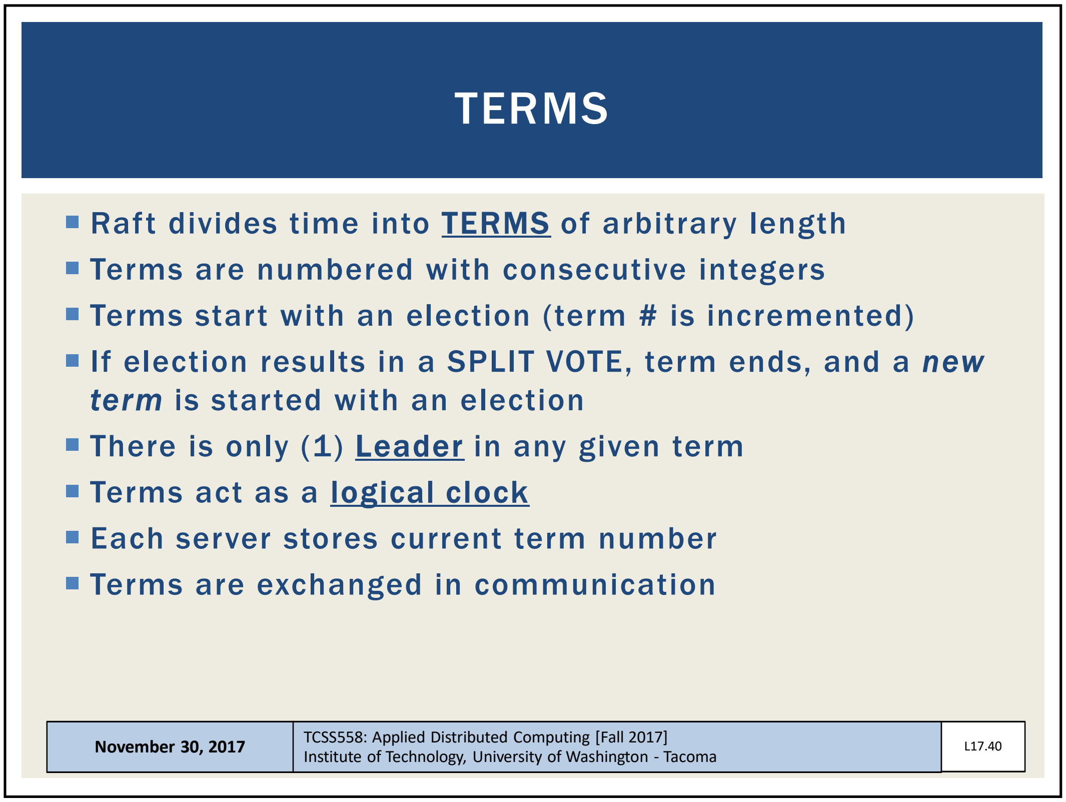
- <u>LEADER ACTIVITIES</u>
  - Accepts log entries from other nodes
  - Replicates them on other servers
  - Tells nodes when safe to apply log entries to their state machines (KV store)

  - <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



TERMS

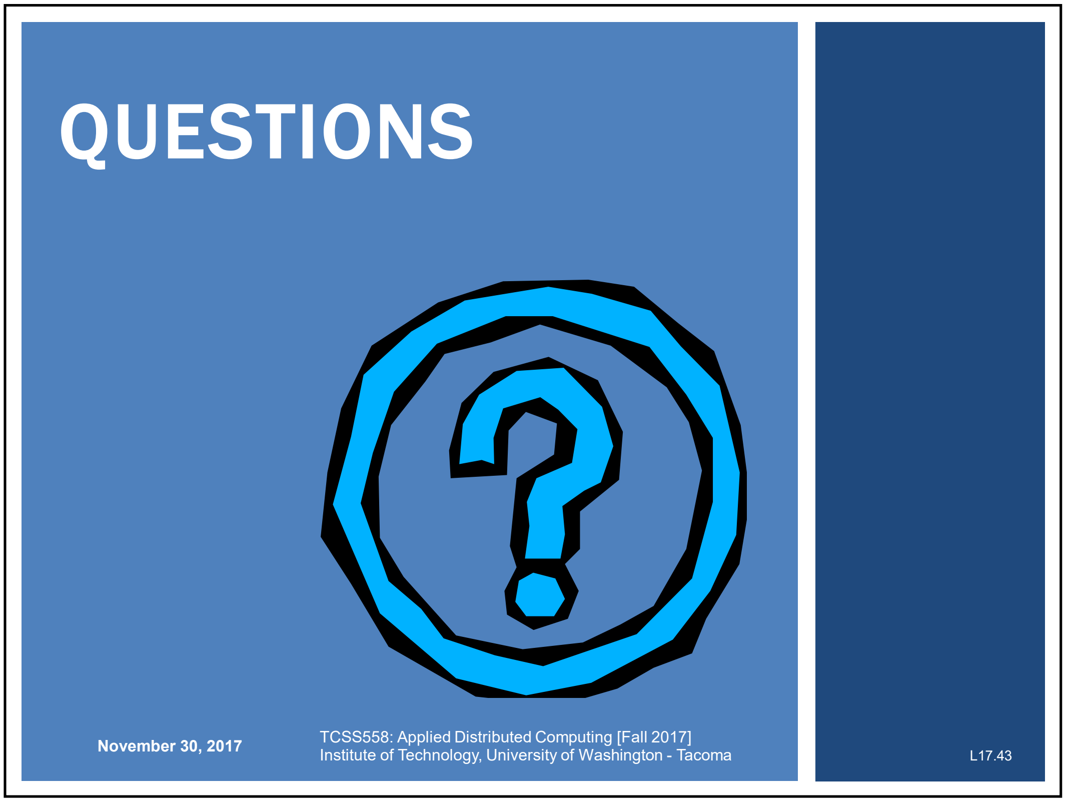
- Raft divides time into **TERMS** of arbitrary length
- Terms are numbered with consecutive integers
- Terms start with an election (term # is incremented)
- If election results in a SPLIT VOTE, term ends, and a *new term* is started with an election
- There is only (1) **Leader** in any given term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

TERMS - 2

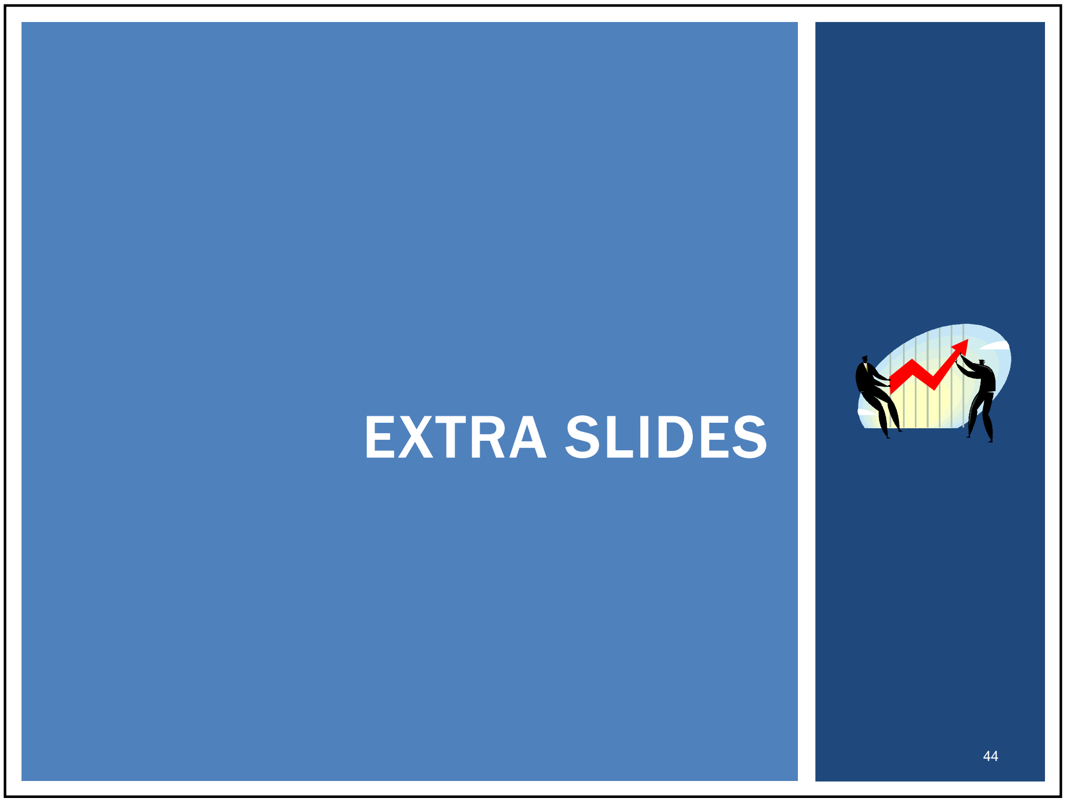
- If a larger term # is found, then ***all nodes*** update term # and defer to the term’s **leader**
  - If **candidate** or **leader** finds its term is out of date, will immediately become a **follower** node
- If server receives request with stale term #, then request is rejected

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)



QUESTIONS



EXTRA SLIDES