SYNCHRONIZATION

CS435 Distributed Systems

Basit Qureshi PhD, FHEA, SMIEEE, MACM
https://www.drbasit.org/

TOPICS

- Dist. Mutual Exclusion
  - Centralized Algorithm
  - Token Ring Algorithm
- Dist. Mutual Exclusion Algorithms
  - Lamport
  - Ricart & Agarwala
- Leader Election Algorithms
  - Bully
  - Ring
- Concensus Algorithms
- Raft

© 2024 - Dr. Basit Qureshi

DISTRIBUTED MUTUAL EXCLUSION

DIST MUTUAL EXCLUSION

"A distributed system ensures that only **one process or node** can access a shared resource or **critical section** at any given time".

- Examples:
  - Modify a shared file
  - Update a database field
  - Modify replication messages
- Easy to handle for **atomic** requests
  - One message, one server
  - One system: Hardware compatibility, Semaphores, Messages, Condition variables
- Challenging if
  - Multiple messages on multiple servers
  - Need synchronization and coordination

© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION

GOAL:

- Distributed Mutual Exclusion ensures that only one process is granted permission to access the resource at a time, while others are blocked or delayed until the resource becomes available.

AIM:

- **Safety**: Ensuring that only one process accesses the critical section at a time
- **Liveness**: Ensuring that processes eventually gain access to the critical section, even in the presence of failures, delays, or network partitions.
- **Efficiency**: [Optional] Minimizing overhead and maximizing resource utilization while maintaining safety and liveness properties.

© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION

HOW:

- Process identification: Every process has a unique Identifier (e.g., `address.process_id`)
- Reliable communication: Network messages are reliable
- Live processes: The processes in the system are responsive & do not die.
- Resource identification – Agree on resource identification
  - Pass the identifier with each request
  - e.g., `lock("printer")`, `lock("table:employees")`, `lock("table:employees;row:15")`, `lock("shared_file.txt")`
  - We'll just use `request(R)` to request exclusive access to resource `R`

© 2024 - Dr. Basit Qureshi

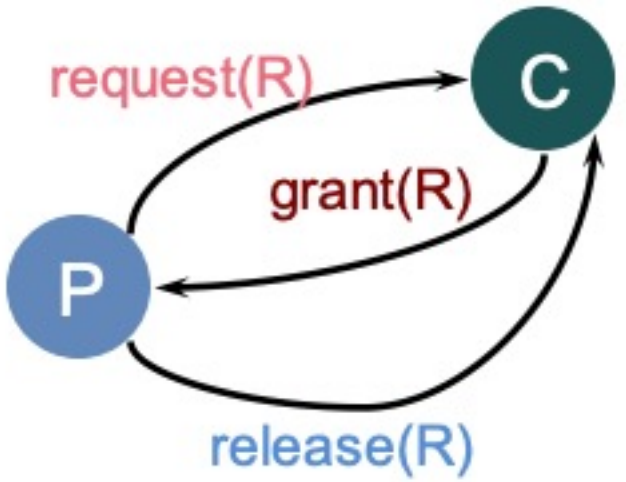
DIST MUTUAL EXCLUSION

- Algorithms
  - **Centralized**: A coordinator is responsible for allowing access to a shared resource
  - **Token-based:** Access if a token was granted
  - **Contention-based**: Via Distributed agreement

© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION

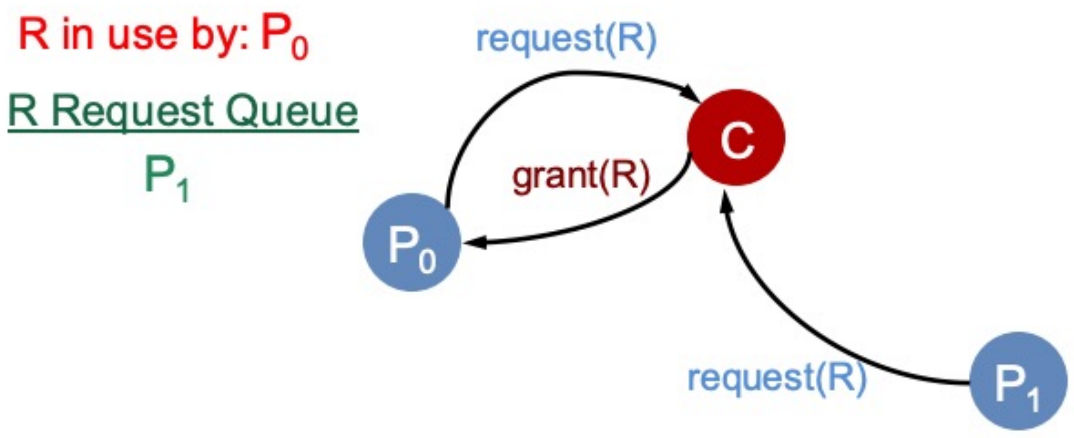
- **Centralized Algorithms**: Similar to a single processor system:
  - Process P Request(R) access to resource R from Coordinator C
  - Wait for response
  - Receive Access
  - Access resource
  - Release(R)



© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION

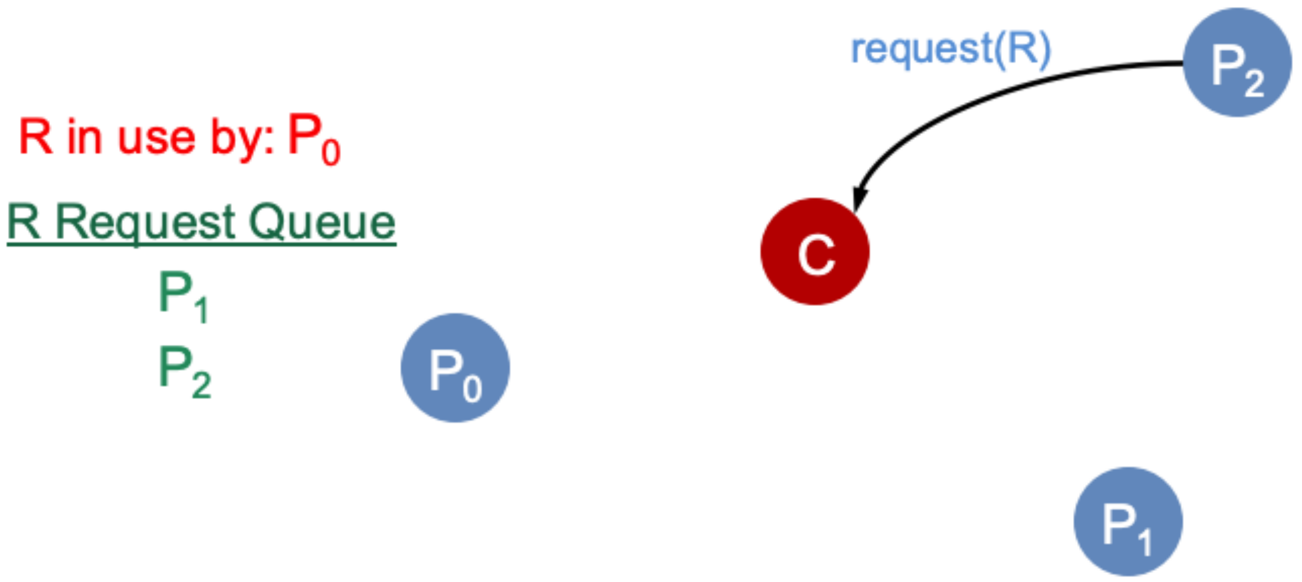
- **Centralized Algorithms**: If another Process tries to access:
  - Maintain a FIFO queue at coordinator
  - Coordinator: Donot reply until resource available



© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION

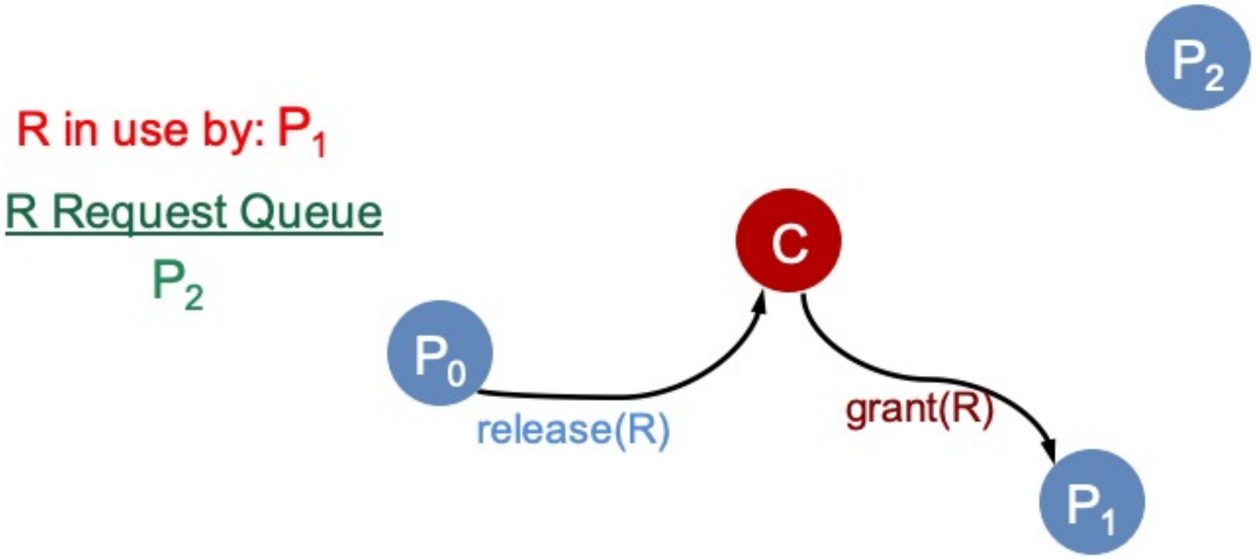
- **Centralized Algorithms**: If another Process tries to access:
  - Maintain a FIFO queue at coordinator
  - Coordinator: Donot reply until resource available



© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION

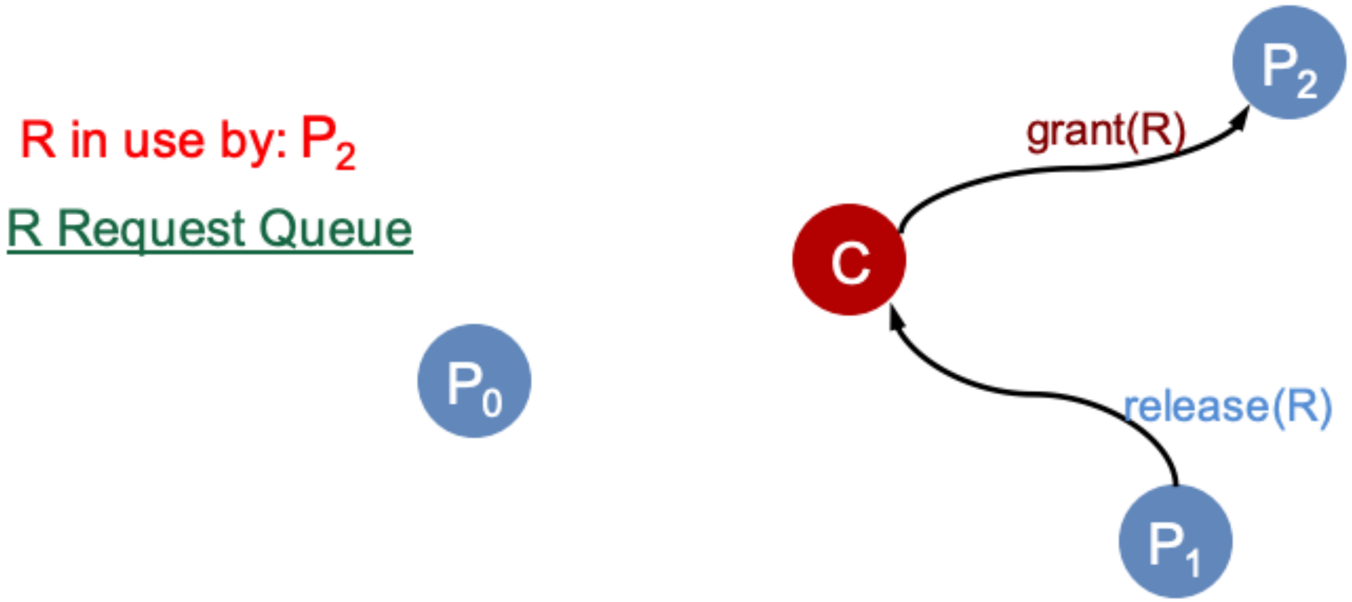
- **Centralized Algorithms**: If another Process tries to access:
  - Maintain a FIFO queue at coordinator
  - Coordinator: Donot reply until resource available



© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION

- **Centralized Algorithms**: If another Process tries to access:
  - Maintain a FIFO queue at coordinator
  - Coordinator: Donot reply until resource available



© 2024 - Dr. Basit Qureshi

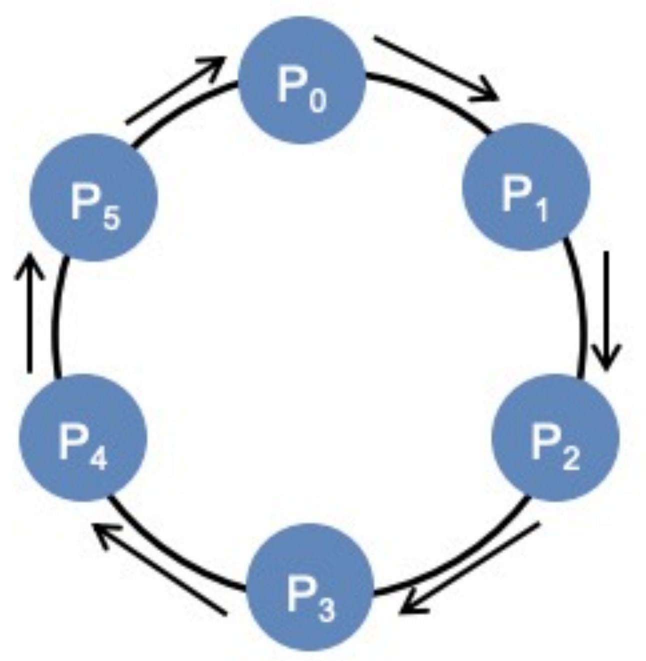
DIST MUTUAL EXCLUSION

- **Centralized Algorithms**:
- The Good
  - Easy to implement
  - FIFO Queue takes order into consideration
  - Processes do not need to communicate to other processes; just the coordinator
  - Efficient: 2 message to enter, 1 message to exit
- The Bad
  - Single point of failure: Coordinator crashes!
  - A crashed coordinator blocks access to resource
  - Coordinator can become a bottleneck!

© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION

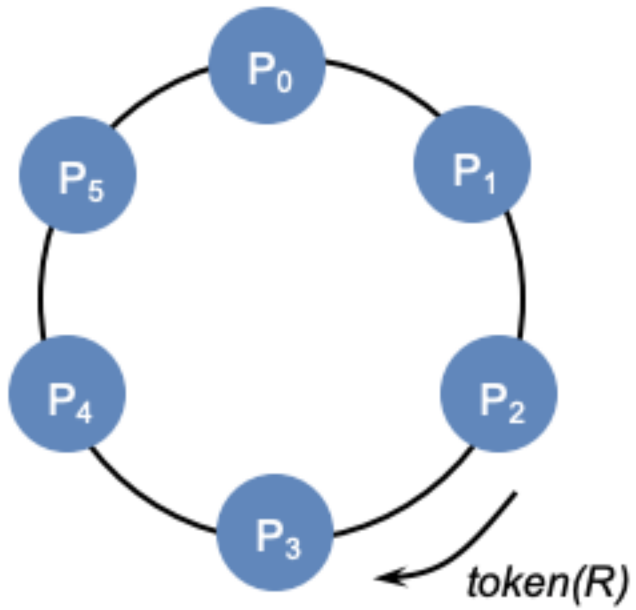
- **Token-Ring Algorithms**:
- Processes known each other in a group
  - Processes can be assigned a unique process IDs
  - Construct logical ring in software
  - Process communicates with its neighbor and not with the coordinator directly



© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION

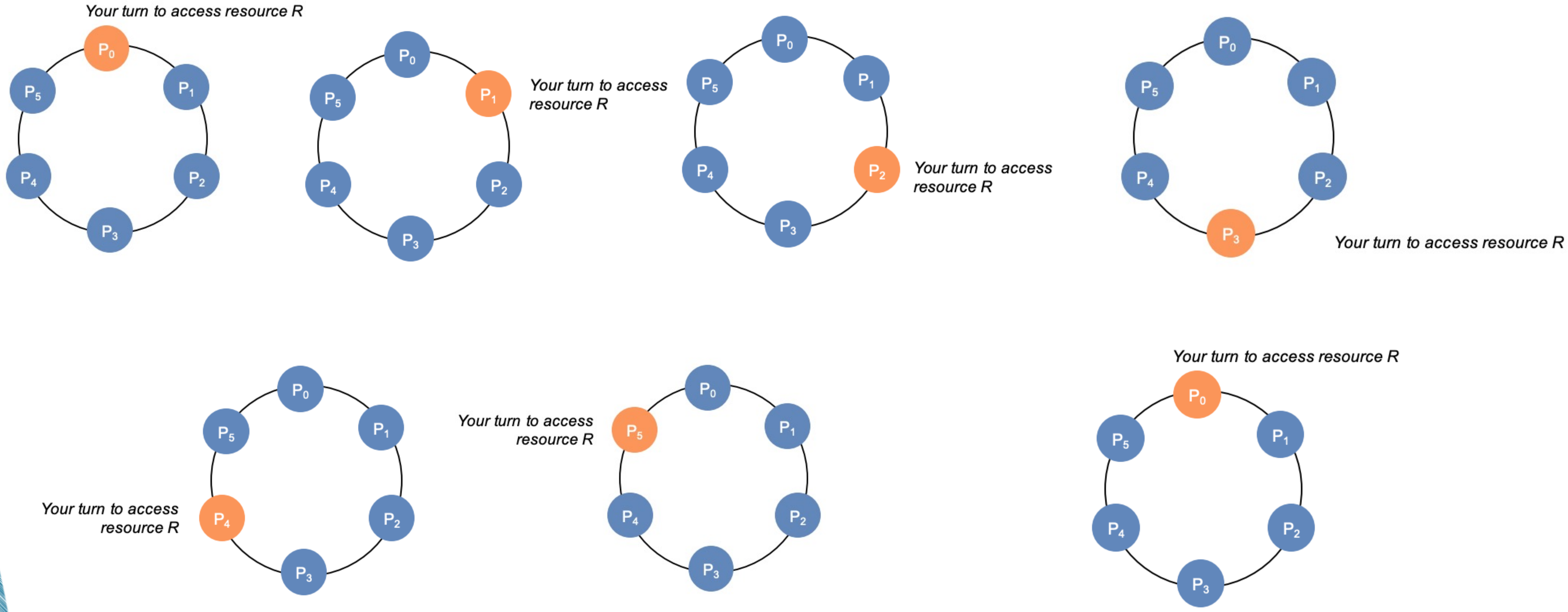
- **Token-Ring Algorithms**:
- Initialization
  - Process 0 creates a token for resource R
- Token circulates around ring from **P_i to $P_{(i+1)}$mod N**
  - When process acquires token
    - Checks to see if it needs the resource (**the lock**)
    - **No**: send the token to its neighbor
    - **Yes**: access resource & hold token until done



© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION

- **Token-Ring Algorithms**:



© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION

- **Token-Ring Algorithms**:
- The Good
  - **Saftey**: Only One process at a time [Mutual Exclusion is guaranteed]
  - **Liveness**: Order is defined, but not always First-Come-First-Server (FCFS)
  - **Delay**: Request = 0…N-1 messages; Release = 1 message
- The Bad
  - Constant activity
  - Process dies: Token is lost! Needs to be re-generated
    - Detecting loss can be challenging (really lost or someone is holding it)
  - Communication error: What if no communication with neighbor

© 2024 - Dr. Basit Qureshi

DIST MUTUAL EXCLUSION ALGORITHMS

LAMPORT MUTUAL EXCLUSION

- Proposed by Leslie Lamport in 1978, is one of the pioneering algorithms for achieving mutual exclusion in distributed systems
- Uses Lamport's logical clocks and message passing to coordinate access to a shared resource among multiple processes.

© 2024 - Dr. Basit Qureshi

LAMPORT MUTUAL EXCLUSION

- Messages are sent reliably and in single-source FIFO order
  - Each message is time stamped with **totally ordered (i.e., unique)** Lamport timestamps
  - Ensures that each timestamp is unique
  - Every node can make the same decision by comparing timestamps
- Each process maintains a request queue
  - Queue contains **mutual exclusion requests**
  - Queues are sorted by message timestamps

© 2024 - Dr. Basit Qureshi

LAMPORT MUTUAL EXCLUSION

Step 1: Request a resource R:

- Process Pi sends `Request(R, i, Ti)` to all nodes
  - It also places the same request onto its own queue
- When a process Pj receives a request:
  - It returns a timestamped `Reply(Tj)`
  - Places the request on its request queue
- Every process will have an identical queue
  - Same contents in the same order

Process	Time stamp
P_4	1021
P_8	1022
P_1	3944
P_6	8201
P_{12}	9638

Sample request queue for R
Identical at each process

LAMPORT MUTUAL EXCLUSION

Step 2: Use the resource R:

- Pi can access the resource if
  - Pi has received `Reply` messages from every process Pj where Tj > Ti
  - Pi ’s request has the earliest timestamp in its queue

Process	Time stamp
P_4	1021
P_8	1022
P_1	3944
P_6	8201
P_{12}	9638

Sample request queue for R
Identical at each process

i.e. If your request is at the head of the queue AND you received Replies for that request … then you can access the critical section

© 2024 - Dr. Basit Qureshi

LAMPORT MUTUAL EXCLUSION

`Step 3: Release the resource R:`

- Process Pi removes its request from its queue
- Sends `Release(Ti)` to all nodes
- Each process now checks if its request is the earliest in its queue
- If so, that process now has the `lock` on the resource

Process	Time stamp
P_4	1021
P_8	1022
P_1	3944
P_6	8201
P_{12}	9638

Sample request queue for R
Identical at each process

© 2024 - Dr. Basit Qureshi

LAMPORT MUTUAL EXCLUSION

Assessment

- **Safety**: Replicated queues – same process on top
- **Liveness**: Sorted queue & Lamport timestamps ensure First come first serve
- **Delay/Bandwidth:**
  - *Request* = **2(N-1) messages: (N-1) Request msgs + (N-1) Reply msgs**
  - *Release* = **(N-1) Release msgs**
- **Problems**
  - N points of failure
  - **A lot of messaging traffic**: Requests & releases are sent to the entire group

© 2024 - Dr. Basit Qureshi

RICART & AGARWALA MUTUAL EXCLUSION

Designed to **reduce message overhead** compared to Lamport's algorithm

Basic Idea:

- Allow processes to grant permission to enter the critical section directly
- No need to consult a central authority

© 2024 - Dr. Basit Qureshi

RICART & AGARWALA MUTUAL EXCLUSION

When a process wants to enter critical section:

1. Compose a **`Request`**(**`R, i, Ti)`** message containing:
   - **`R`**: Name of resource
   - **`i`**: Process Identifier(machine ID, process ID)
   - **`Ti`**: Timestamp (totally-ordered Lamport)
2. Reliably **`multicast`** request to all processes in group
3. **`Wait`** until everyone gives permission (sends a **`Reply`**)
4. **`Enter`** critical section / use resource

© 2024 - Dr. Basit Qureshi

RICART & AGARWALA MUTUAL EXCLUSION

When process receives a request:

- If receiver not interested: **send Reply to sender**
- If receiver is using the resource: **do not reply**; **add request to queue**
- If receiver just sent a request as well: (**potential race condition**)
  - **Compare timestamps** on received & sent messages: earliest timestamp wins
  - If receiver is the **loser**: `send Reply`
  - If receiver is the **winner**: `do not reply`
  - Queue the request
    - When **done** with resource: `send Reply to all` queued requests

© 2024 - Dr. Basit Qureshi

RICART & AGARWALA MUTUAL EXCLUSION

Assessment

- **Safety**: Two competing processes will not send a REPLY to each other
  - Timestamps in the requests are unique
  - one will be earlier than the other
- **Liveness**: Lamport timestamps ensure First come first serve
- **Delay/Bandwidth:**
  - *Request* = **2(N-1) messages: (N-1) Request msgs + (N-1) Reply msgs**
  - *Release* = **0…(N-1) Reply msgs to queued requests**
- **Problems**
  - N points of failure
  - **A lot of messaging traffic**: Requests & releases are sent to the entire group

LAMPORT VS RICART & AGARWALA MUTUAL EXCLUSION

Lamport

- Everyone replies … always – no hold-back
- 3(N-1) messages Request → Reply → Release
- Process is granted the resource if its request is the earliest in its queue

Ricart & Agarwala

- If you are in the critical section (or won a tie)
  - Don't respond with a Reply until you are done with the critical section
- 2(N-1) messages
  - Request → ACK
- Process is granted the resource if it gets ACKs from everyone

Other algorithms

- Suzuki-Kasami
- Maekawa
- Dijkstra's Token Ring Algorithm
- Raynal's Algorithm

© 2024 - Dr. Basit Qureshi

LEADER ELECTION ALGORITHMS

BULLY ALGORITHM

- GOAL: Select the process with the largest ID as a leader

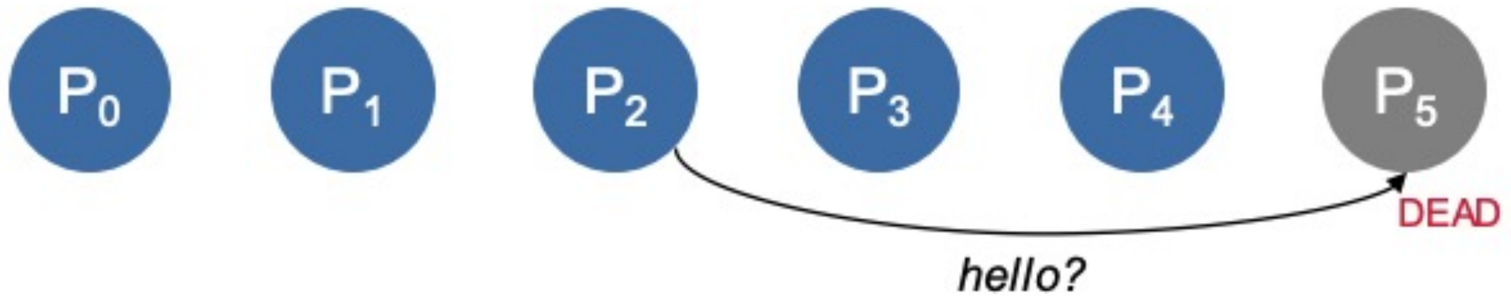
Bully Algorithm

- Holding an election: when process Pi detects a dead leader:
  - Send **election** message to all processes with higher IDs
    - If nobody responds, **Pi wins** and takes over
    - If any process responds, P's job is done
  - Optional: Let all nodes with lower IDs know an election is taking place

- If a process receives an **election** message
  - Send an **OK** message back
  - Hold an election (unless it is already holding one)

© 2024 - Dr. Basit Qureshi

BULLY ALGORITHM



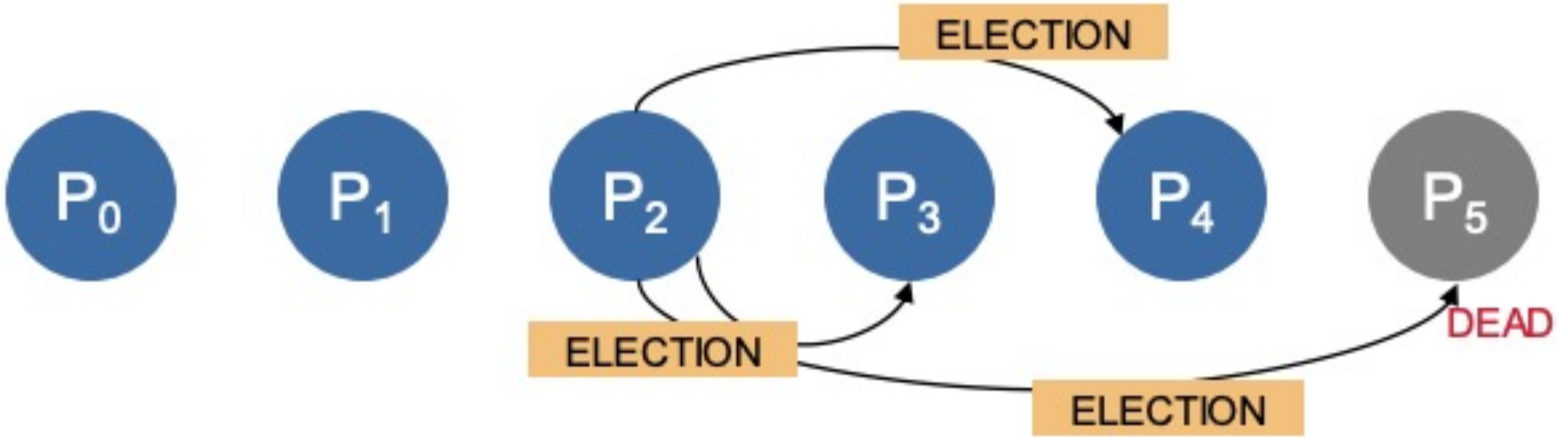
Rule: highest # process is the leader

Suppose P_5 dies

P_2 detects P_5 is not responding

© 2024 - Dr. Basit Qureshi

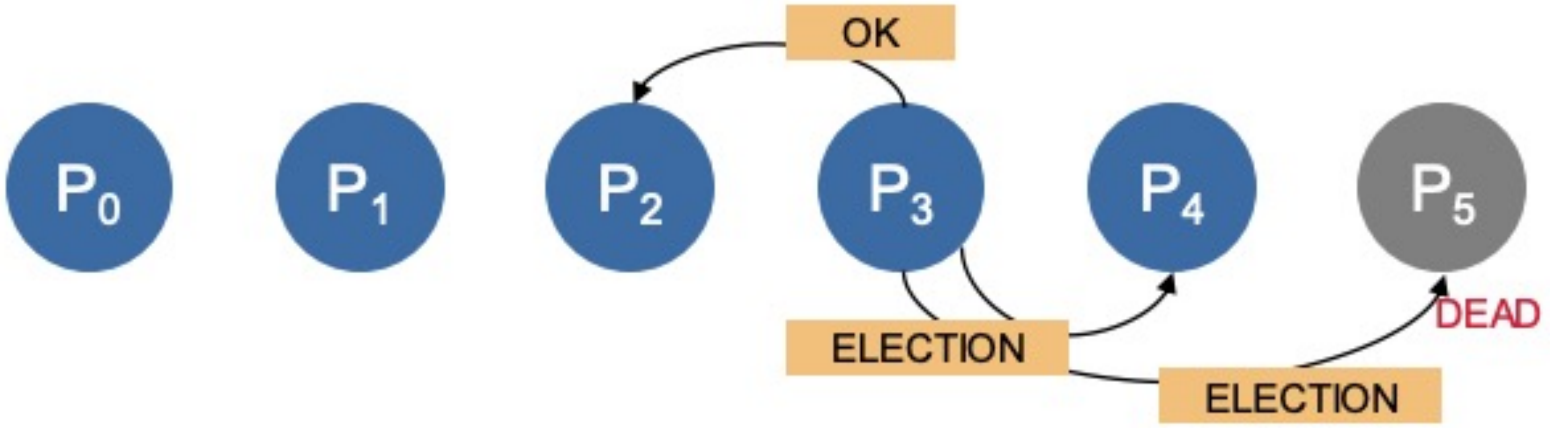
BULLY ALGORITHM



P_2 starts an election

Contacts all higher-numbered systems

BULLY ALGORITHM



Everyone who receives an ***election*** message responds

… and holds their own election, contacting higher # processes

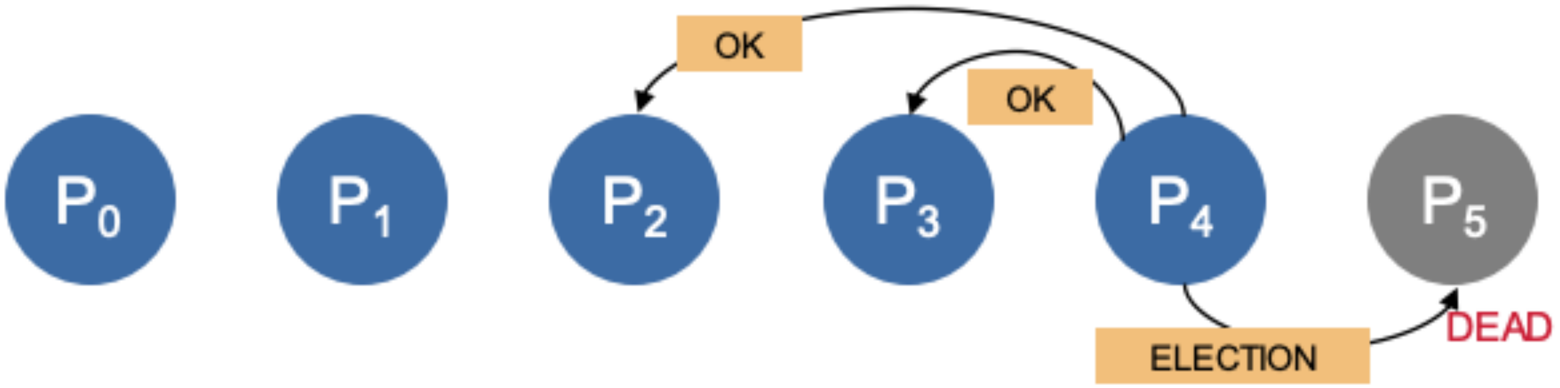
Example: P_3 receives the message from P_2

Responds to P_2

Sends ***election*** messages to P_4 and P_5

© 2024 - Dr. Basit Qureshi

BULLY ALGORITHM

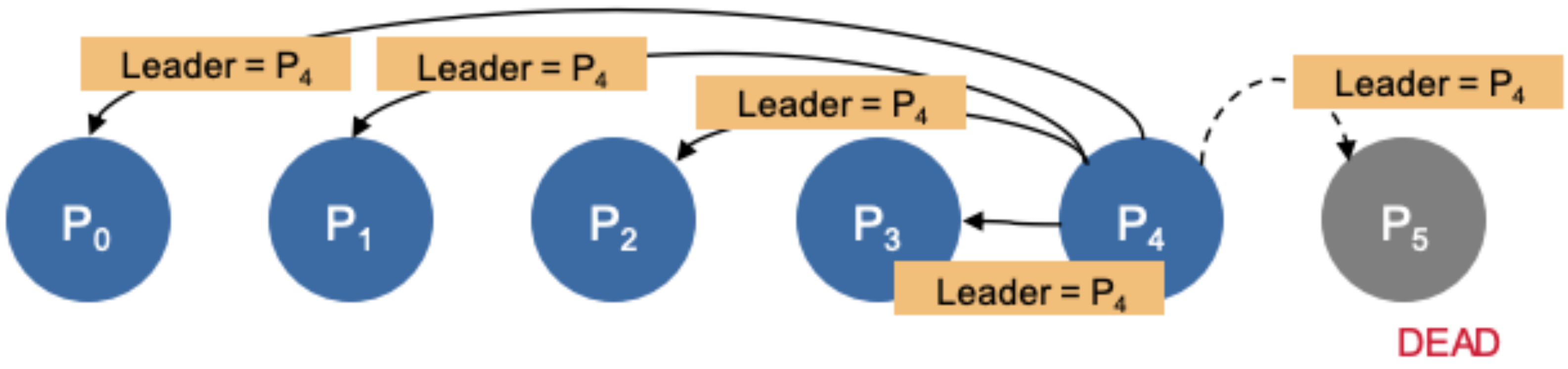


P_4 responds to P_3 and P_2's messages

… and holds an election

© 2024 - Dr. Basit Qureshi

BULLY ALGORITHM



Nobody responds to P_4

After a timeout, P_4 declares itself the leader

RING ELECTION ALGORITHM

- GOAL: Select the process with the largest ID as a leader

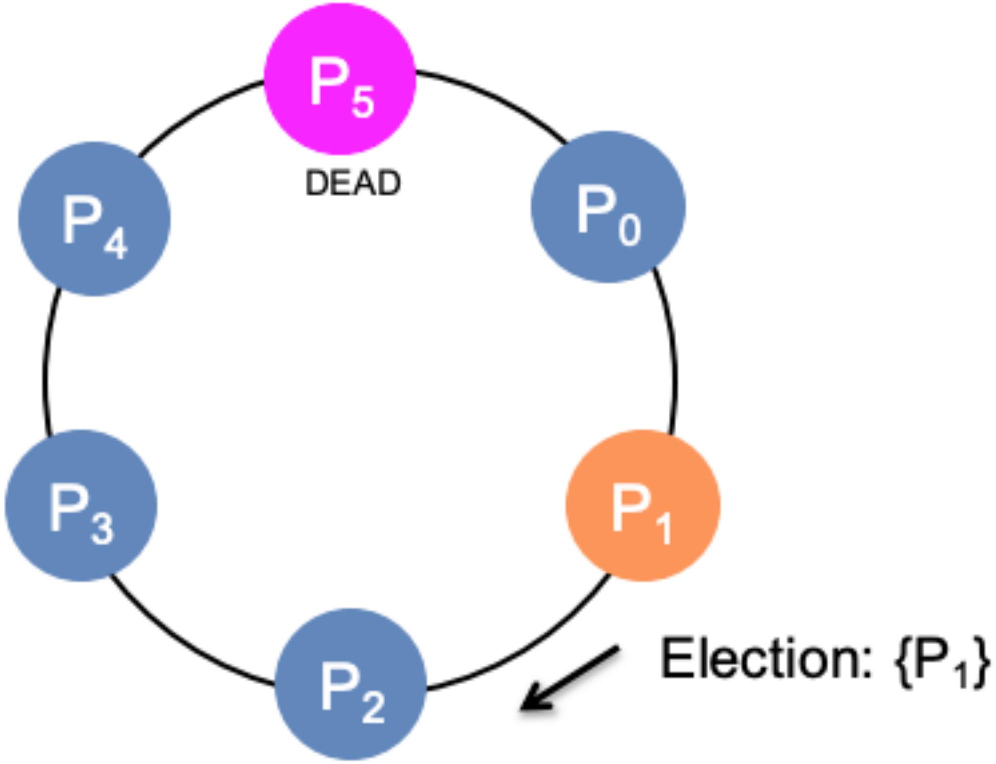
Ring Election Algorithm

- Initiate the election by **sending** an "`election`" message to its neighbor with the next highest priority.
- Upon receiving an election message, compare the priority value in the message with its own.
  - If priority is **higher** than its own, it forwards the message to its neighbor.
  - If priority is **lower or equal**, it discards the message.
- The election message continues around the ring until it reaches the highest priority process.
- The new leader broadcasts a "`leader`" message to inform all other processes of its election.

© 2024 - Dr. Basit Qureshi

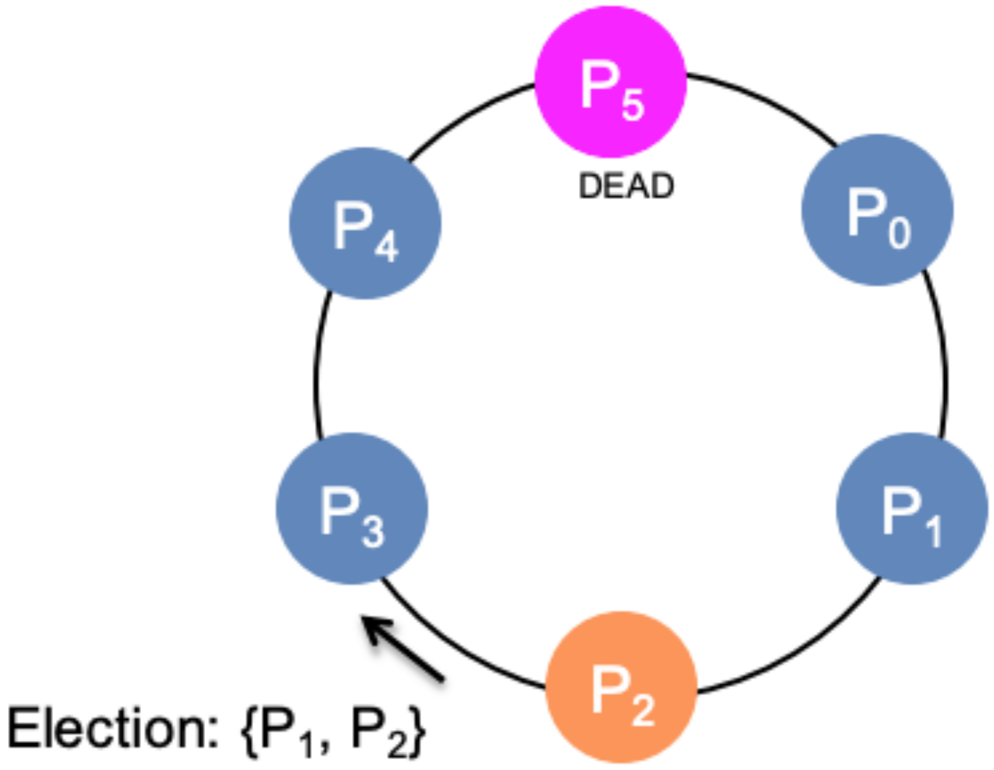
RING ELECTION ALGORITHM

Ring Election Algorithm



RING ELECTION ALGORITHM

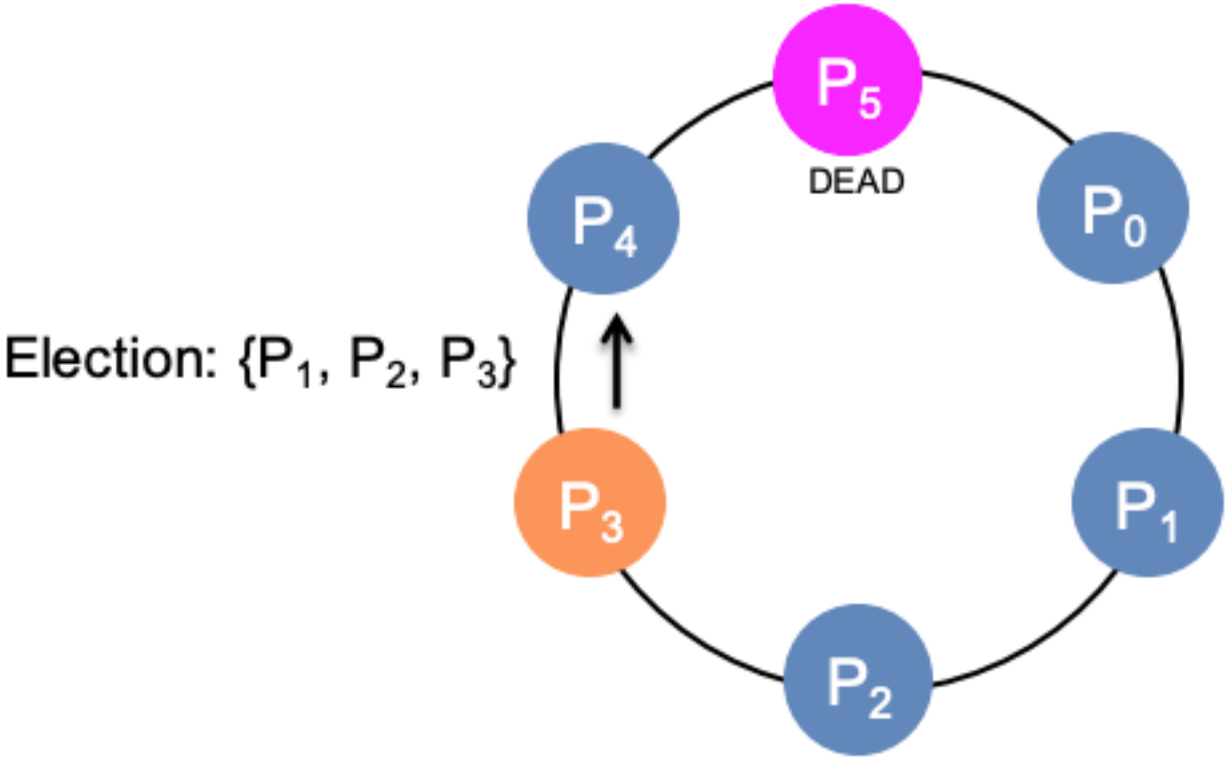
Ring Election Algorithm



© 2024 - Dr. Basit Qureshi

RING ELECTION ALGORITHM

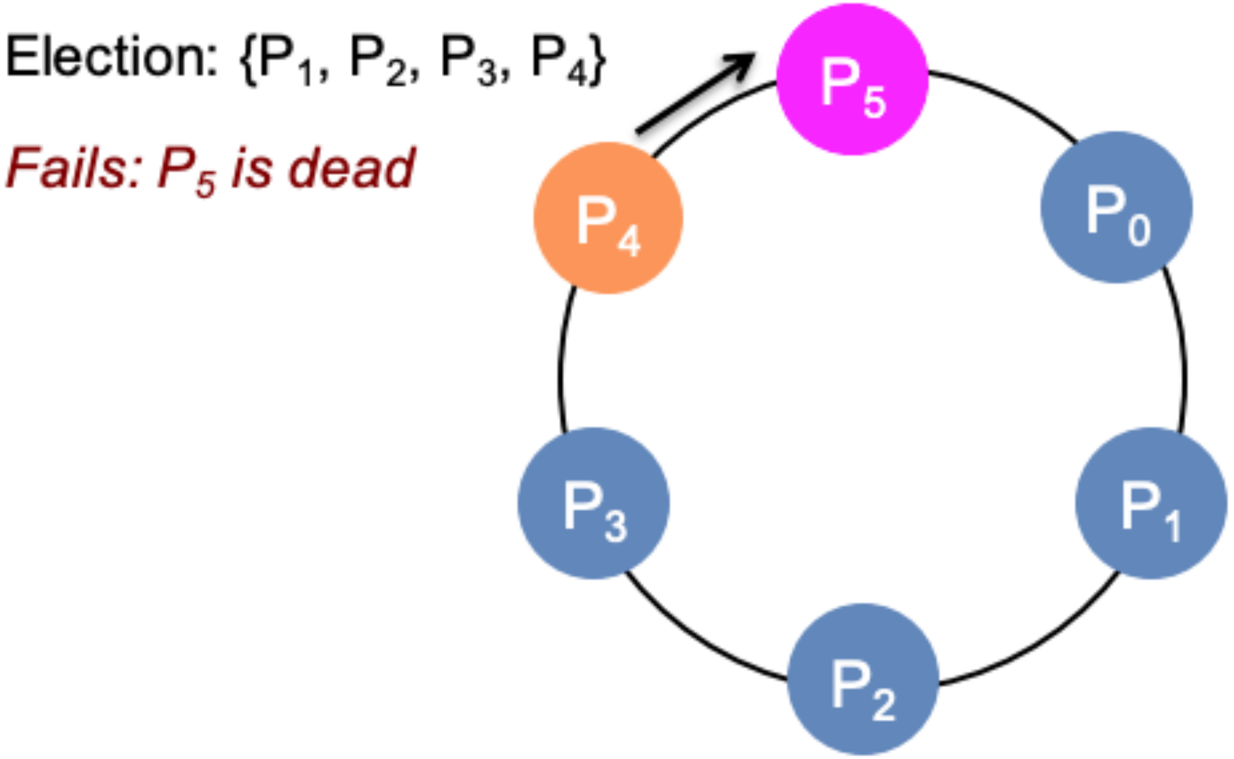
Ring Election Algorithm



© 2024 - Dr. Basit Qureshi

RING ELECTION ALGORITHM

Ring Election Algorithm



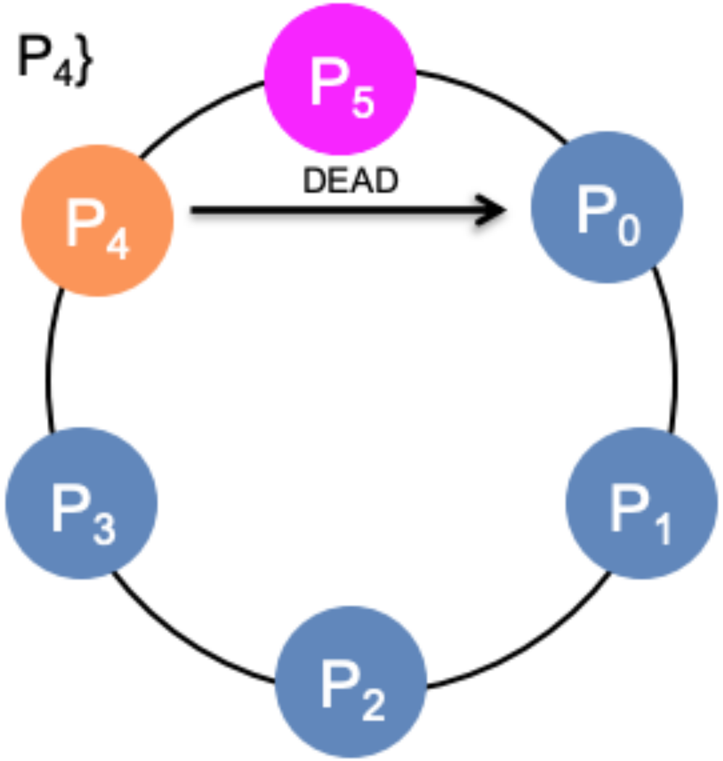
© 2024 - Dr. Basit Qureshi

RING ELECTION ALGORITHM

Ring Election Algorithm

Election: $\{P_1, P_2, P_3, P_4\}$

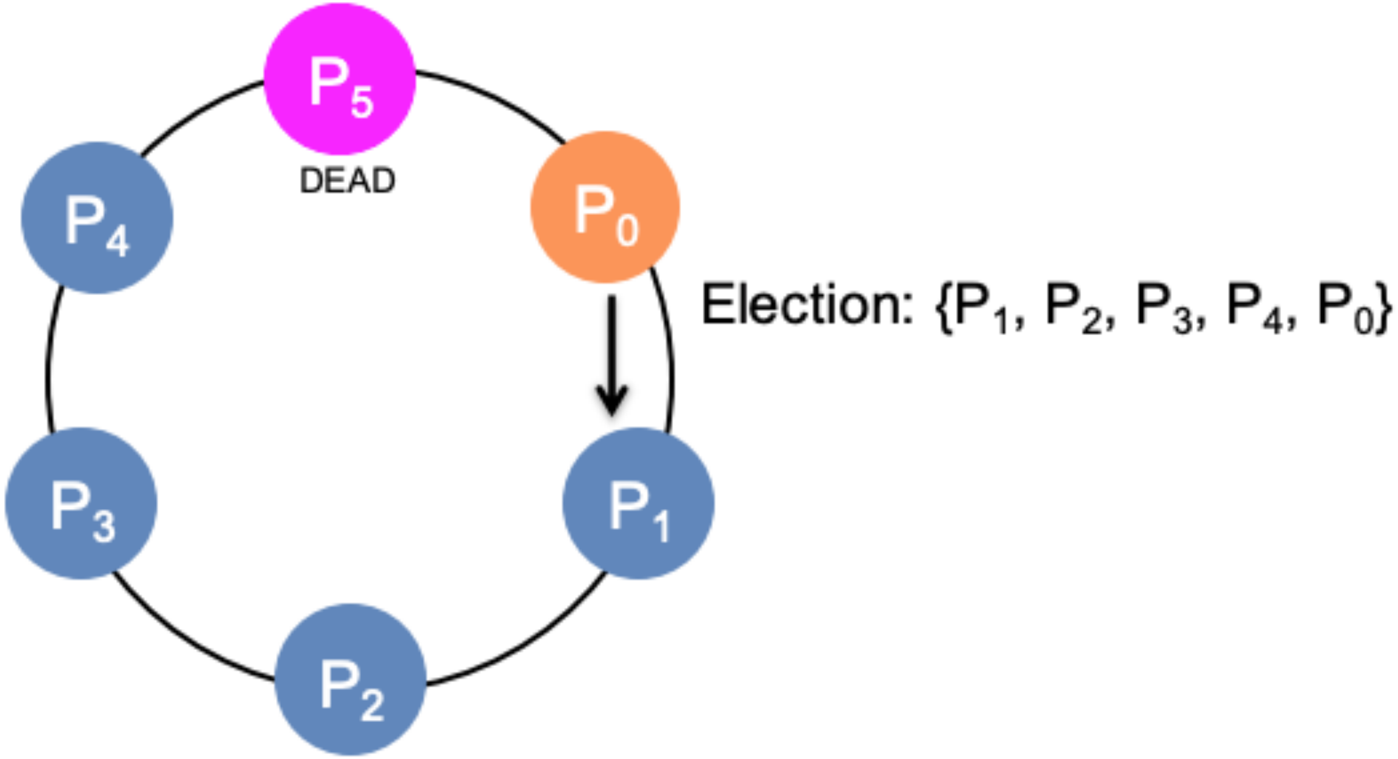
Skip to P_0



© 2024 - Dr. Basit Qureshi

RING ELECTION ALGORITHM

Ring Election Algorithm



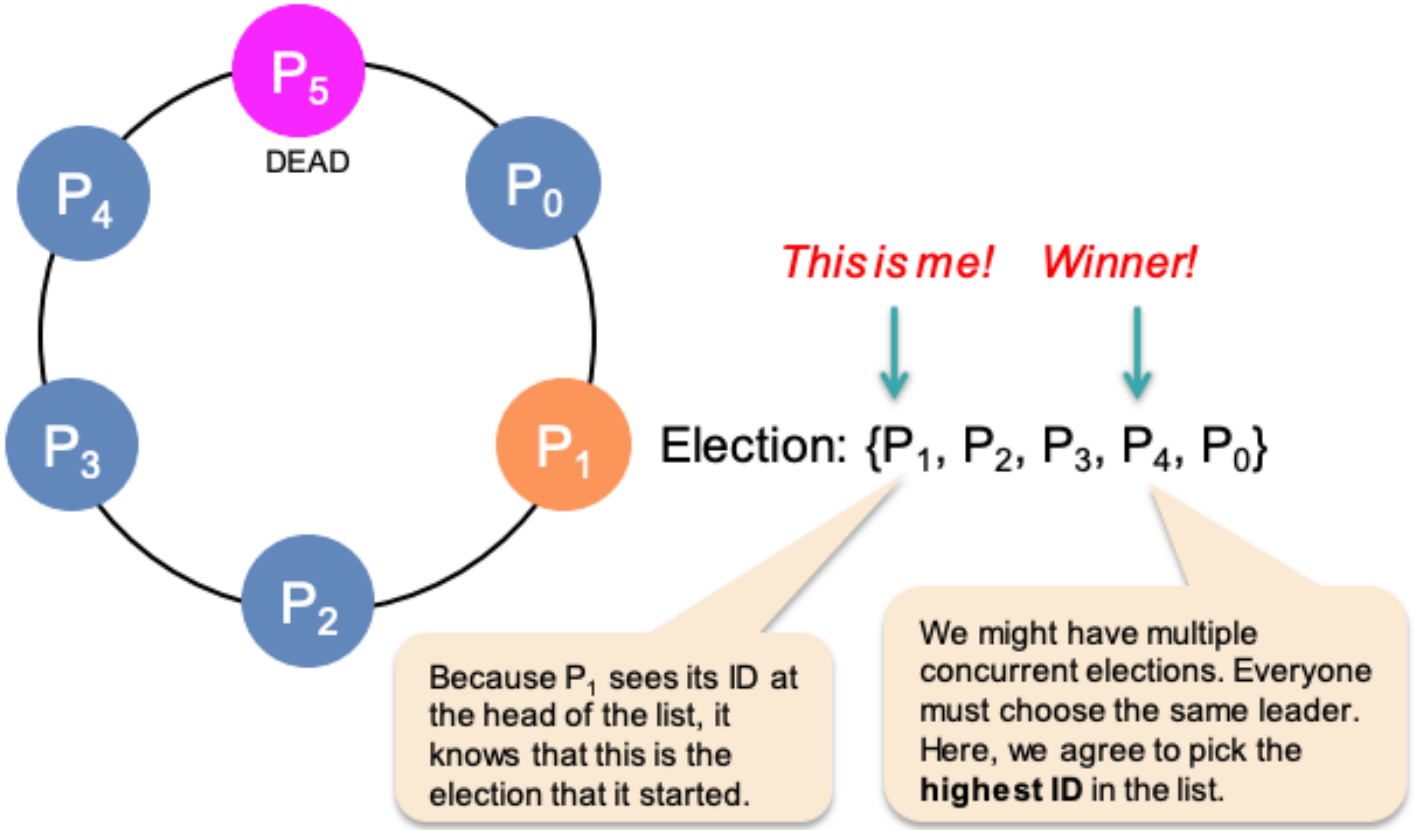
© 2024 - Dr. Basit Qureshi

RING ELECTION ALGORITHM

Ring Election Algorithm

P_2 receives the election message that it initiated

P_2 now picks a leader (e.g., highest ID)

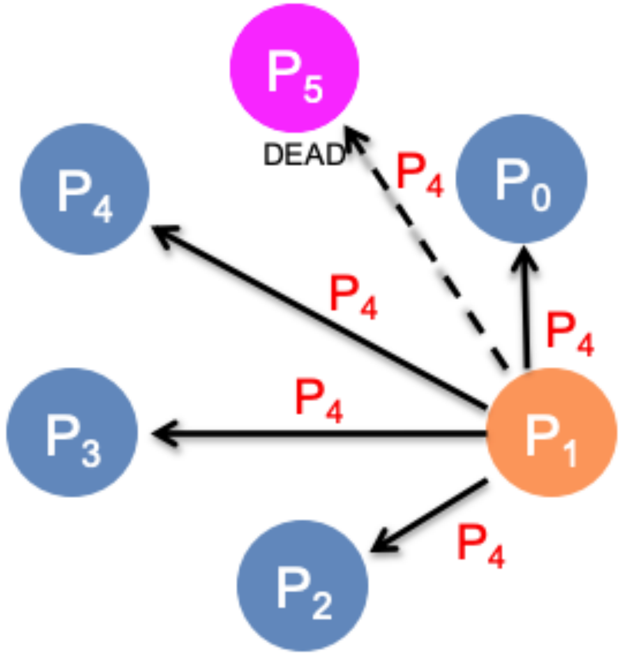


© 2024 - Dr. Basit Qureshi

RING ELECTION ALGORITHM

Ring Election Algorithm

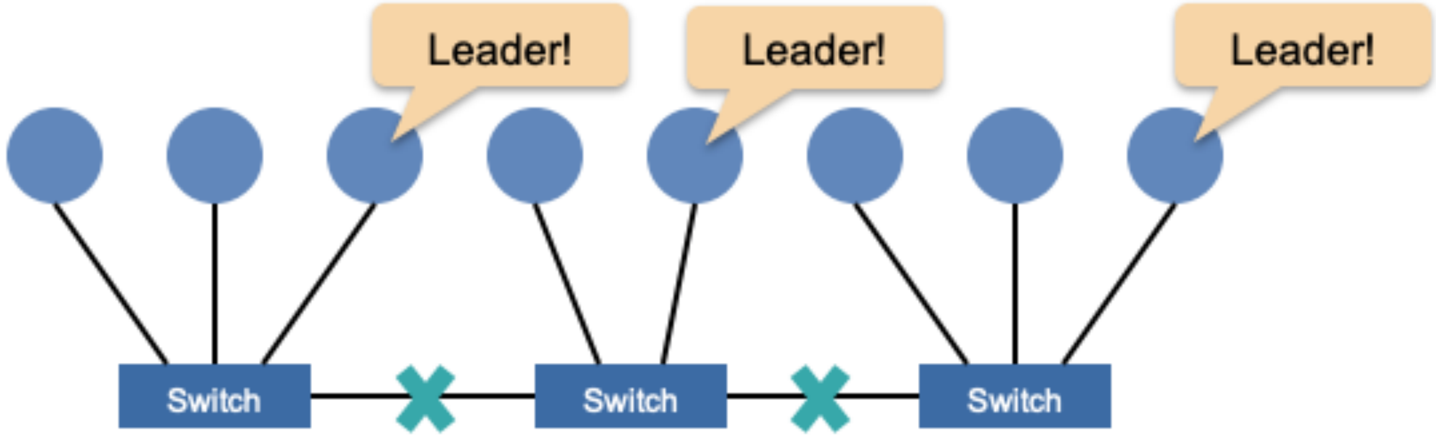
P_1 announces that P_4 is the new leader to the group



Many other election algorithms that target other topologies: mesh, torus, hypercube, trees, …

ELECTIONS & NETWORK PARTITIONS

- Network partitions (segmentation)
  - Multiple nodes may decide they're the leader
  - Multiple groups, each with a leader & diverging data among them → split brain



- Insist on a majority → if no majority, the system will not function.
- **Quorum** = minimum # of participants required for a system to function)

CONSENSUS ALGORITHMS

CONSENSUS

- In decentralized systems, with no central authority, achieving consensus is crucial for ensuring the integrity and consistency of the shared data
  - 1. If a node hasn't received a message for some time, assume it is **DEAD**
  - 2. When nodes suspect the current leader has failed: **HOLD ELECTION**
  - 3. One or more nodes becomes a **CANDIDATE**
  - 4. Other nodes **VOTE** on whether they accept the candidate as their new leader.
  - 5. Election, the new **LEADER**:
    - If a quorum of nodes vote in favor of the candidate, it becomes the new leader.
    - If a majority quorum is used, this vote can succeed as long as a majority of nodes (2 out of 3, or 3 out of 5, etc.) are working and able to communicate.

- Challenge: **How do we get unanimous agreement on a given value**?

© 2024 - Dr. Basit Qureshi

CONSENSUS

Why consensus is needed?

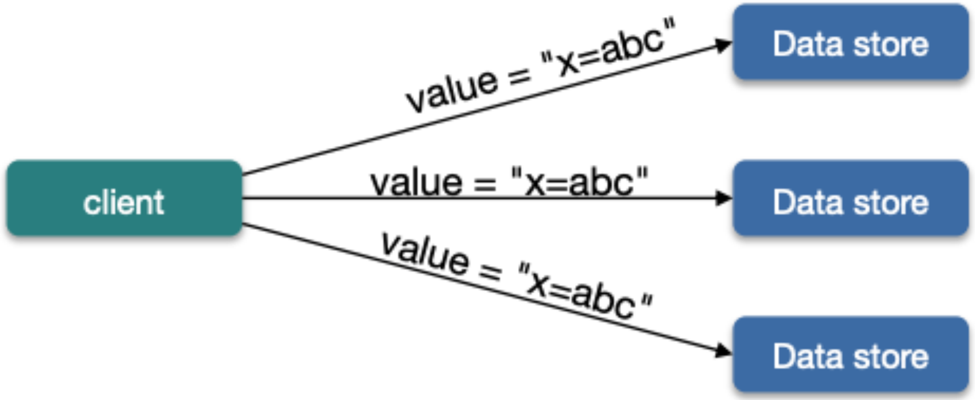
- Mutual Exclusion:
  - Choose which process can access a resource from all who want it
  - Agree on who gets a resource or who becomes a coordinator
- Election algorithms
  - Choose one process from the set of willing processes
- Uses:
  - **Blockchain Technology**: enable nodes to agree on the validity and ordering of transactions.
  - **Cryptocurrencies**: Bitcoin, Ethereum etc, rely on consensus algorithms to validate and confirm transactions, preventing double-spending and ensuring the integrity of the currency.
  - **Internet of Things (IoT)**: reach agreement on the state of sensor data

© 2024 - Dr. Basit Qureshi

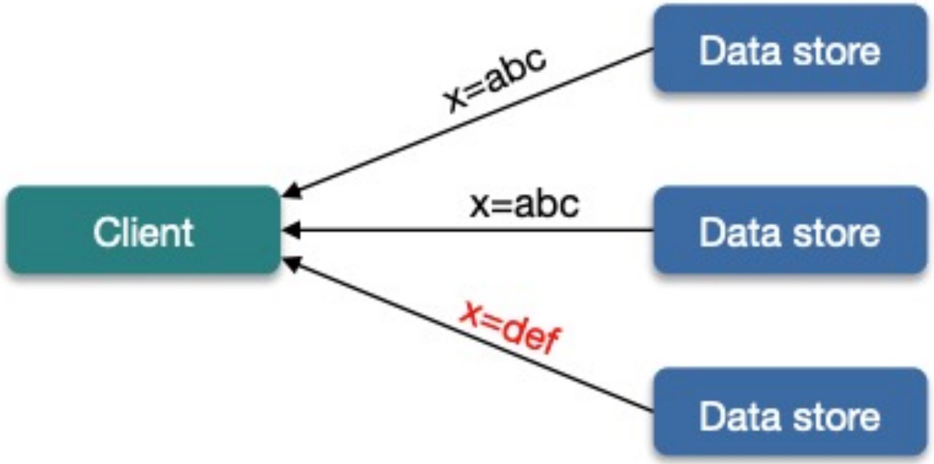
CONSENSUS

Why consensus is needed?

- Single Client



Easy if only one client sends request at a time



We rely on a **quorum** (majority) for reads & writes

If we have to write to a majority of servers for the write to succeed **and** we have to read from a majority of servers for the read to succeed **then** we can be certain that at least one server has the latest version of data.

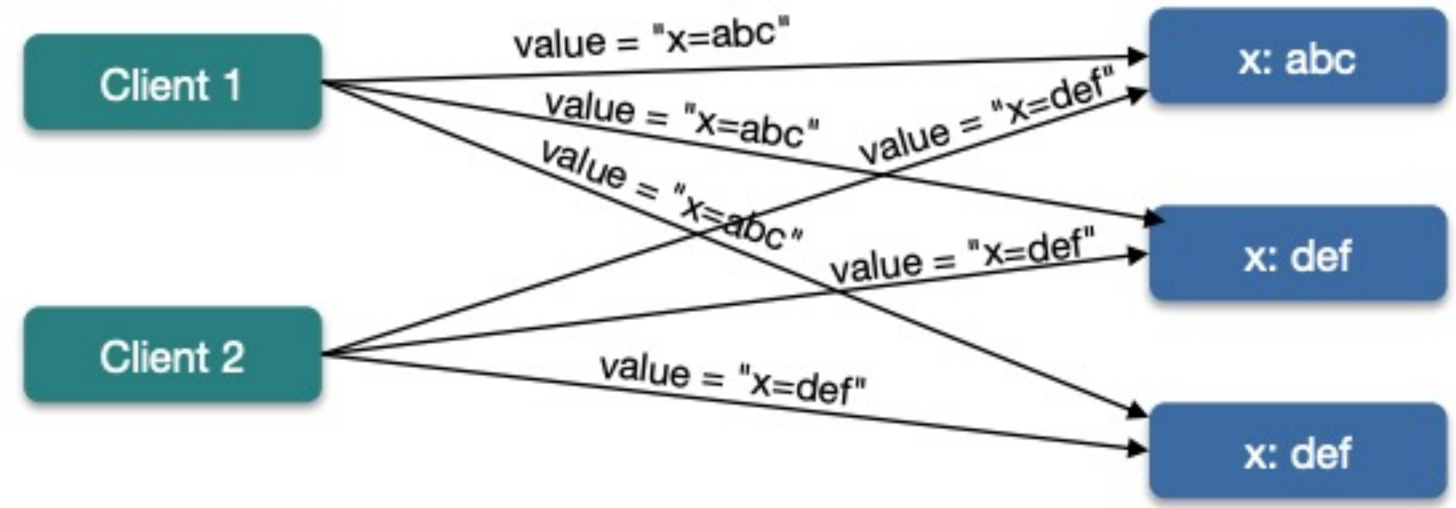
No quorum = failed read!

© 2024 - Dr. Basit Qureshi

CONSENSUS

Why consensus is needed?

- Multiple clients



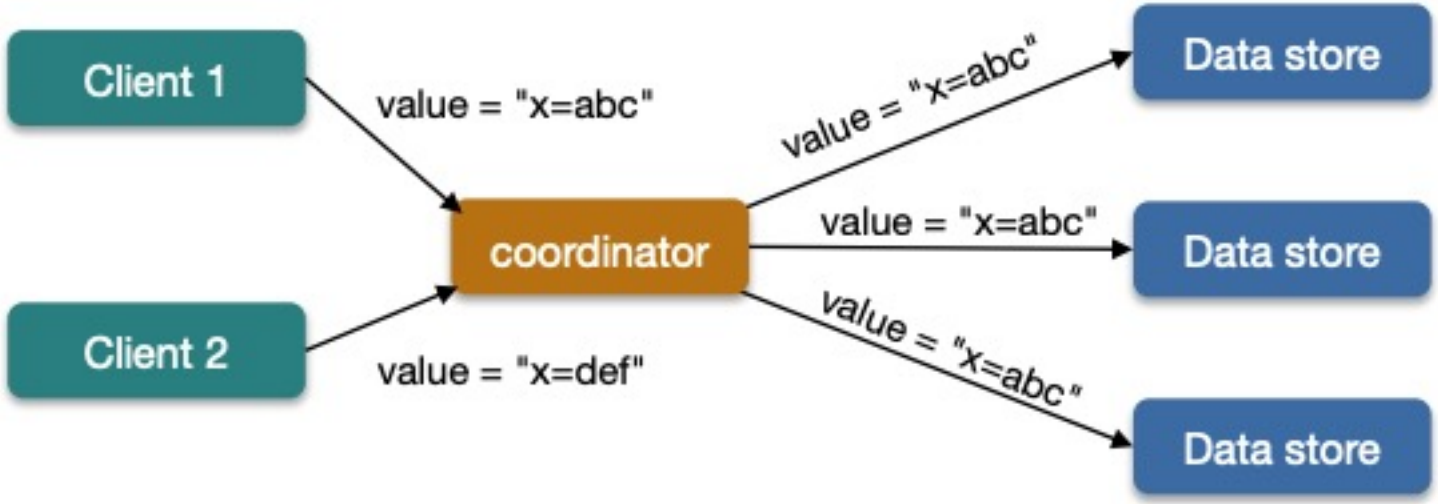
We risk inconsistent updates

© 2024 - Dr. Basit Qureshi

CONSENSUS

Why consensus is needed?

- Multiple clients -> use Coordinator?



Coordinator (or sequence # generator) processes requests one at a time

But now we have a single point of failure!

© 2024 - Dr. Basit Qureshi

CONSENSUS

Why consensus is needed?

- Multiple clients -> use Coordinator?
- Without consensus
  - Processors may fail (some may need stable storage)
  - Messages may be lost, out of order, or duplicated
  - If delivered, messages are not corrupted

Quorum: majority (>50%) agreement is the key part:

It avoids split-brain: you cannot have two majorities doing their own thing

It ensures continuity: if members die and others come up, **there will be one member in common** with the old group that still holds the information.

© 2024 - Dr. Basit Qureshi

CONSENSUS

Consensus GOAL

- **AGREE** on one result among a group of participants

Consensus Requirements

- **Validity**: Only proposed values may be selected (you can't make stuff up)
- **Uniform agreement**: No two nodes may select different values (you agree with everyone else)
- **Integrity**: A node can select only a single value (you cannot change your mind)
- **Termination**: Every node will eventually decide on a value (you come to a decision)

© 2024 - Dr. Basit Qureshi

CONSENSUS

The FLP Impossibility

Consensus protocols with asynchronous communication & faulty processes, "**Every protocol for this problem has the possibility of nontermination, even with only one faulty process**"

- *Impossibility of distributed consensus with one faulty process by Fischer, Lynch and Patterson*

What does it mean?

- We cannot achieve consensus in bounded time But we can with partially synchronous networks
  - Partially synchronous = network with a bounded time for message delivery but we don't know ahead of time what that bound is
- We can either wait long enough for messaging traffic so the protocol can complete or else terminate

© 2024 - Dr. Basit Qureshi

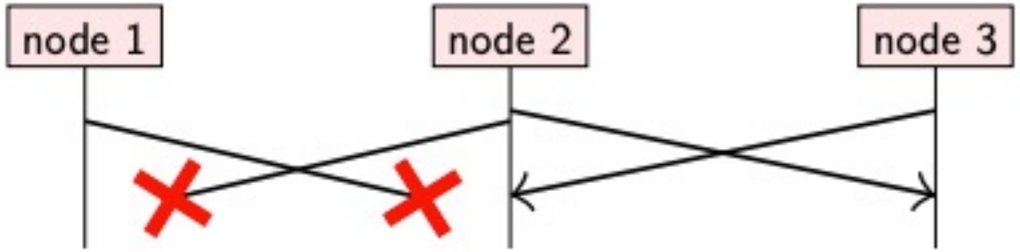
CONSENSUS

Common Consensus Algorithms

- Guarantee a leaders term
- In a partially synchronous system, a timeout-based failure detector may be inaccurate: it may suspect a node has having crashed when in fact the node is functioning fine, for example due to a spike in network latency

Cannot prevent having multiple leaders from different terms.

Example: node 1 is leader in term t, but due to a network partition it can no longer communicate with nodes 2 and 3:



Now we have two leaders !??

Nodes 2 and 3 may elect a new leader in term $t + 1$.

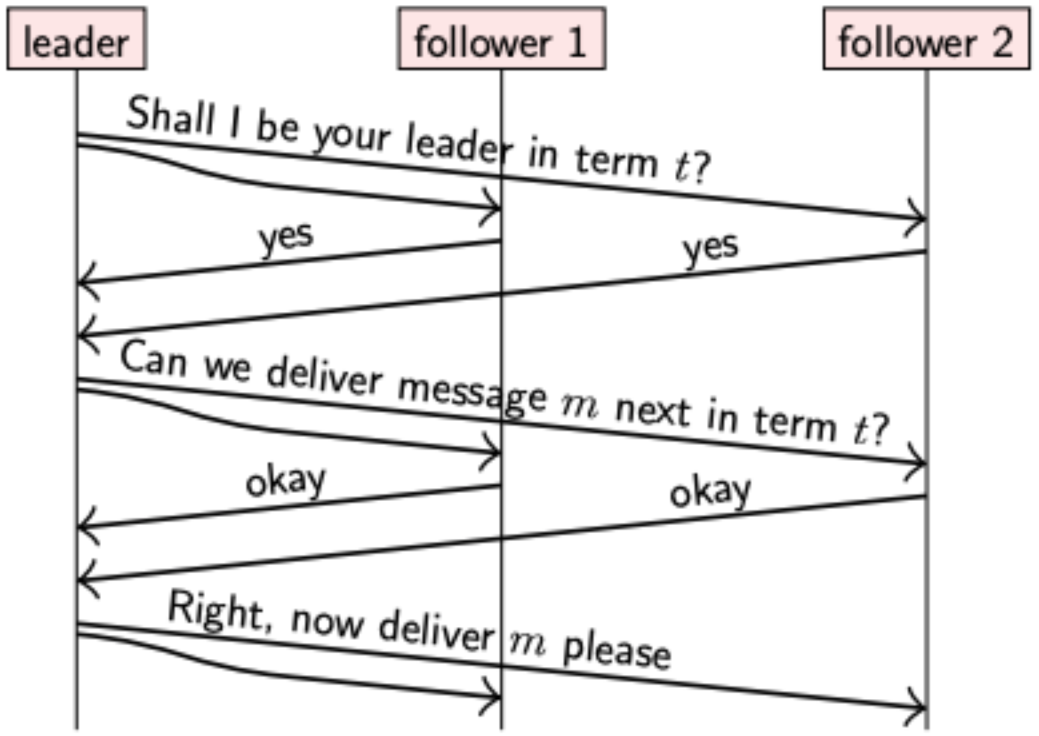
Node 1 may not even know that a new leader has been elected!

CONSENSUS

Common Consensus Algorithms

- Solution?
- Even after a node has been elected leader, it must act carefully

For every decision (message to deliver), the leader must first get acknowledgements from a quorum.



© 2024 - Dr. Basit Qureshi

CONSENSUS

Common Consensus Algorithms

- Paxos: single-value consensus
  - Multi-Paxos: generalization to total order broadcast
- Raft: FIFO-total order broadcast by default

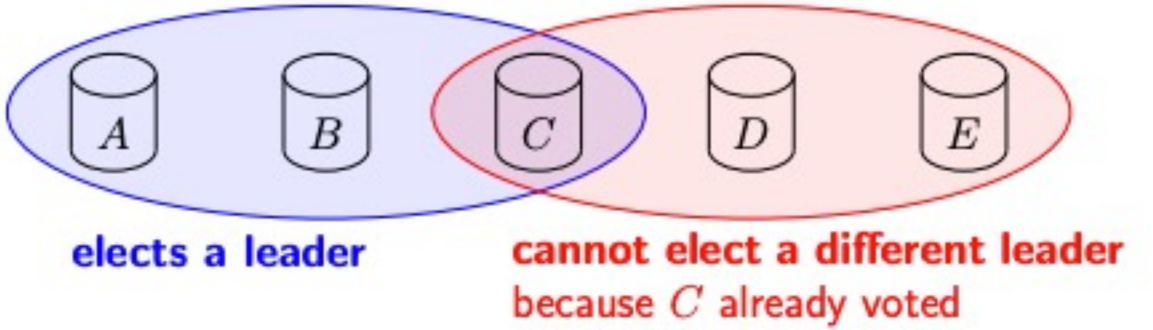
Paxos, Raft, etc. assume a partially synchronous, crash-recovery system model

© 2024 - Dr. Basit Qureshi

CONSENSUS

Common Consensus Algorithms

- Multi-Paxos, Raft, etc. use a leader to sequence messages.
  - Use a failure detector (timeout) to determine suspected crash or unavailability of leader.
  - On suspected leader crash, elect a new one.
  - Prevent two leaders at the same time ("split-brain")
- Ensure ≤ 1 leader per term:
  - Term is incremented every time a leader election is started
  - A node can only vote once per term
  - Require a quorum of nodes to elect a leader in a term



© 2024 - Dr. Basit Qureshi

CONSENSUS

Common Consensus Algorithms

- Paxos by Lamport (1989)
  - Robust but complex to understand
- Multi-Paxos by Lamport (2001)
  - Extended to work with multiple instances
- Fast-Paxos by Lamport (2005)
  - An optimization of the original Paxos algorithm, aimed at reducing latency and improving efficiency
- Raft (2014)
  - Specifically designed with understandability and ease of implementation in mind
- Paxos vs Raft (2020)
  - Heidi Howard and Richard Mortier

Heidi Howard, Richard Mortier, "Paxos vs Raft: have we reached consensus on distributed consensus?" Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, April 2020. https://doi.org/10.1145/3380787.3393681

RAFT: A CONSENSUS ALGORITHM FOR REPLICATED LOGS

Diego Ongaro and John Ousterhout

Stanford University

RAFT slides based on those from Diego Ongaro and John Ousterhout.

Raft Overview

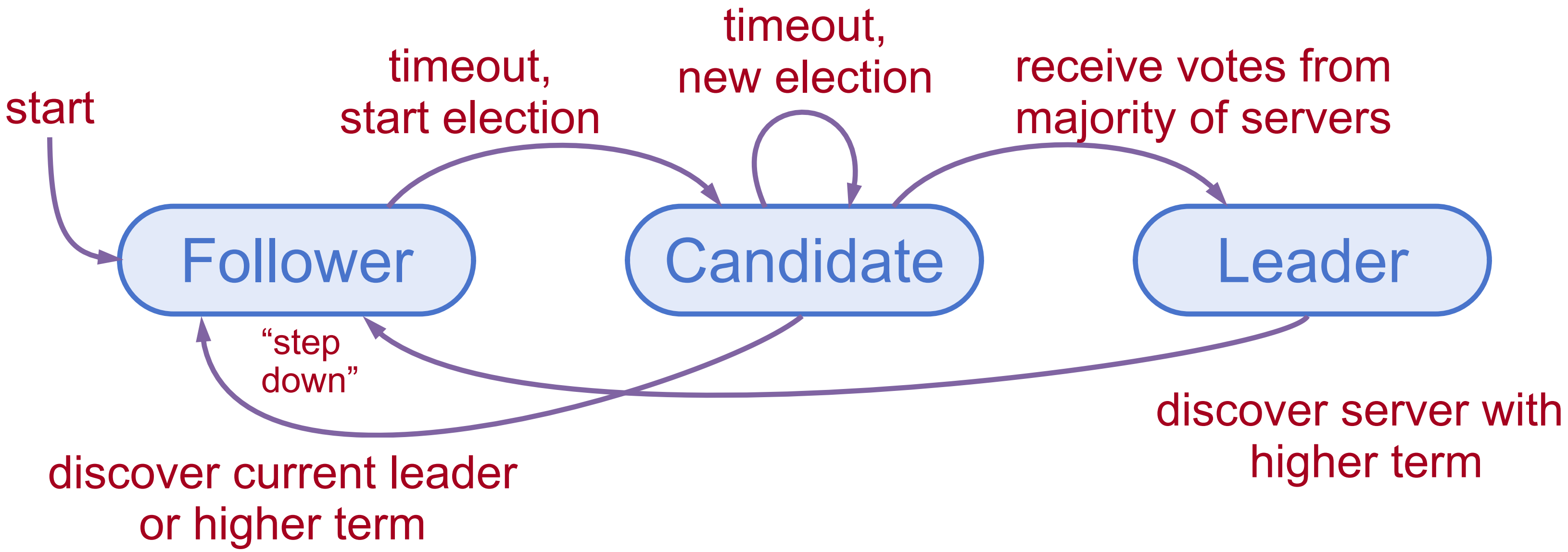
1. Leader election
2. Normal operation (basic log replication)
3. Safety and consistency after leader changes
4. Neutralizing old leaders
5. Client interactions
6. Reconfiguration

Server States

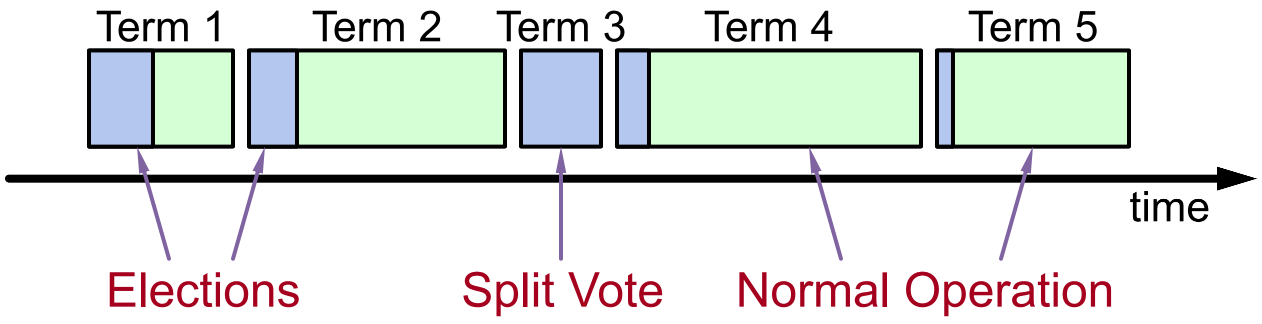
- At any given time, each server is either:
  - Leader: handles all client interactions, log replication
  - Follower: completely passive
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers

Liveness Validation

- Servers start as followers
- Leaders send heartbeats (empty **AppendEntries** RPCs) to maintain authority
- If electionTimeout elapses with no RPCs (100-500ms), follower assumes leader has crashed and starts new election (**RequestVotes** RPC)



Terms (aka epochs)



- Time divided into terms
  - Election (either failed or resulted in 1 leader)
  - Normal operation under a single leader
- Each server maintains current term value
- Key role of terms: identify obsolete information

Elections

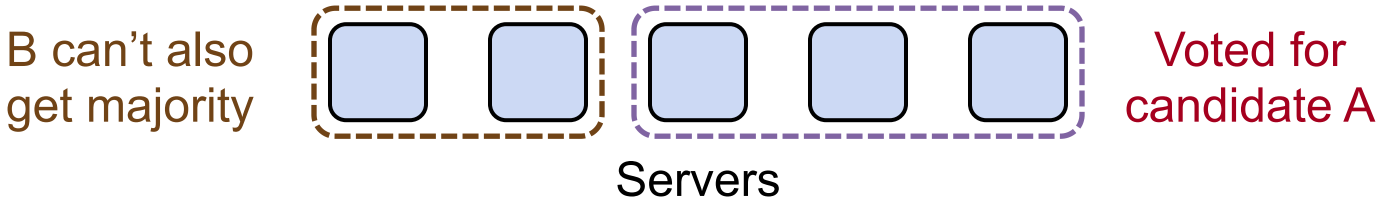
- **Start election:**
  - Increment current term, change to candidate state, vote for self

- **Send RequestVote to all other servers, retry until either:**

  1. Receive votes from majority of servers:
     - Become leader
     - Send **AppendEntries** heartbeats to all other servers
  2. Receive RPC from valid leader:
     - Return to follower state
  3. No-one wins election (election timeout elapses):
     - Increment term, start new election

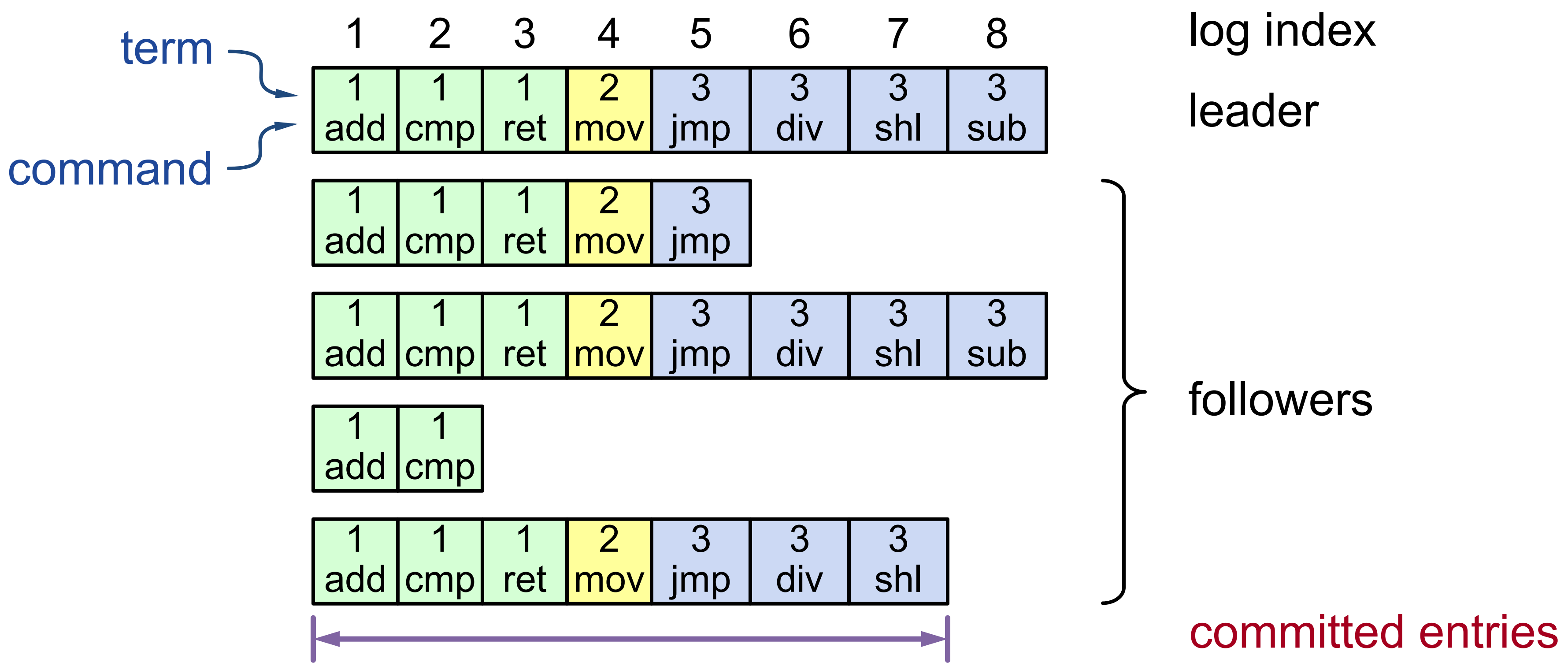
Elections

- **Safety: allow at most one winner per term**
  - Each server votes only once per term (persists on disk)
  - Two different candidates can't get majorities in same term



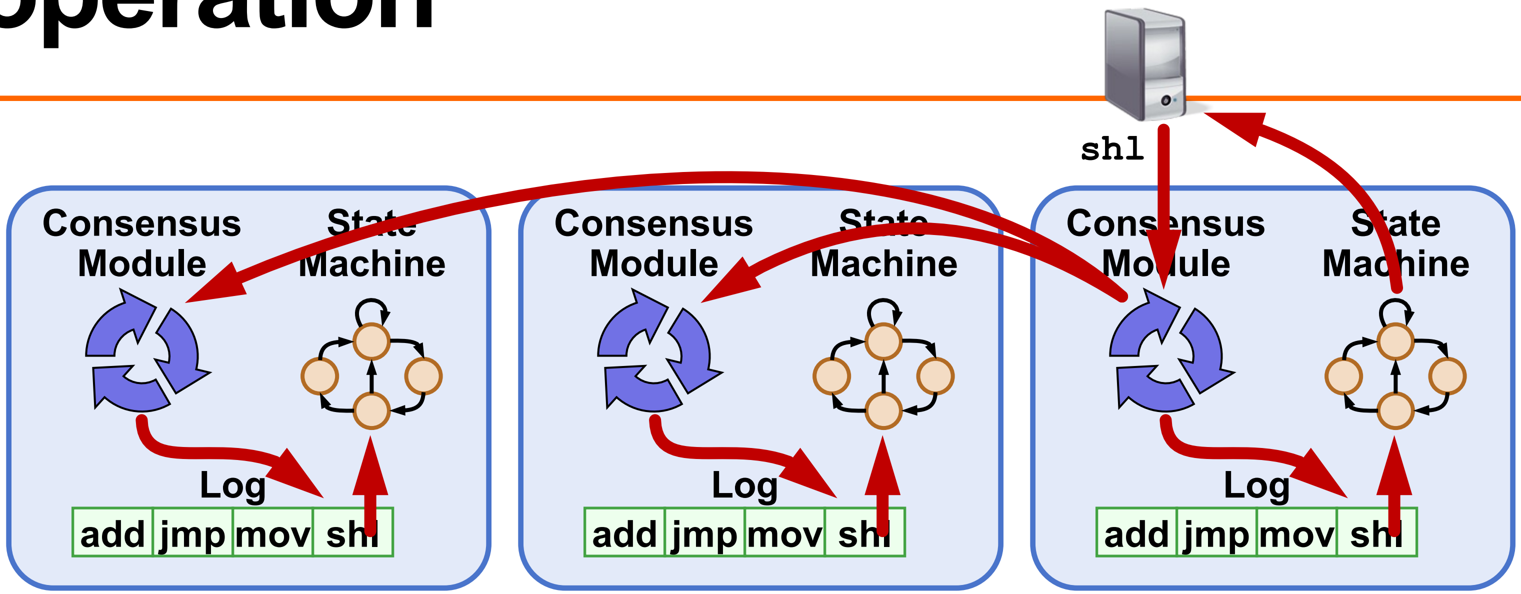
- **Liveness: some candidate must eventually win**
  - Each choose election timeouts randomly in [T, 2T]
  - One usually initiates and wins election before others start
  - Works well if T >> network RTT

Log Structure



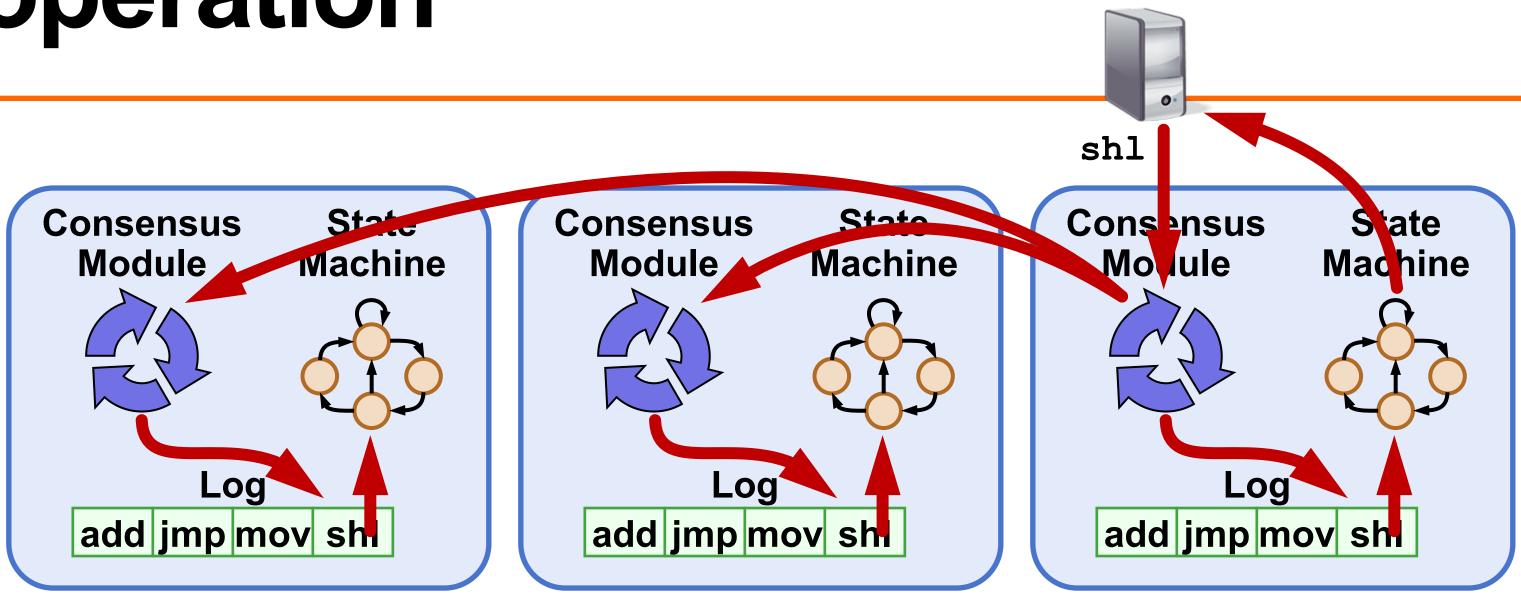
- Log entry = < index, term, command >
- Log stored on stable storage (disk); survives crashes
- Entry committed if known to be stored on majority of servers
  - Durable / stable, will eventually be executed by state machines

Normal operation



- Client sends command to leader
- Leader appends command to its log
- Leader sends **AppendEntries** RPCs to followers
- Once new entry committed:
  - Leader passes command to its state machine, sends result to client
  - Leader piggybacks commitment to followers in later **AppendEntries**
  - Followers pass committed commands to their state machines

Normal operation



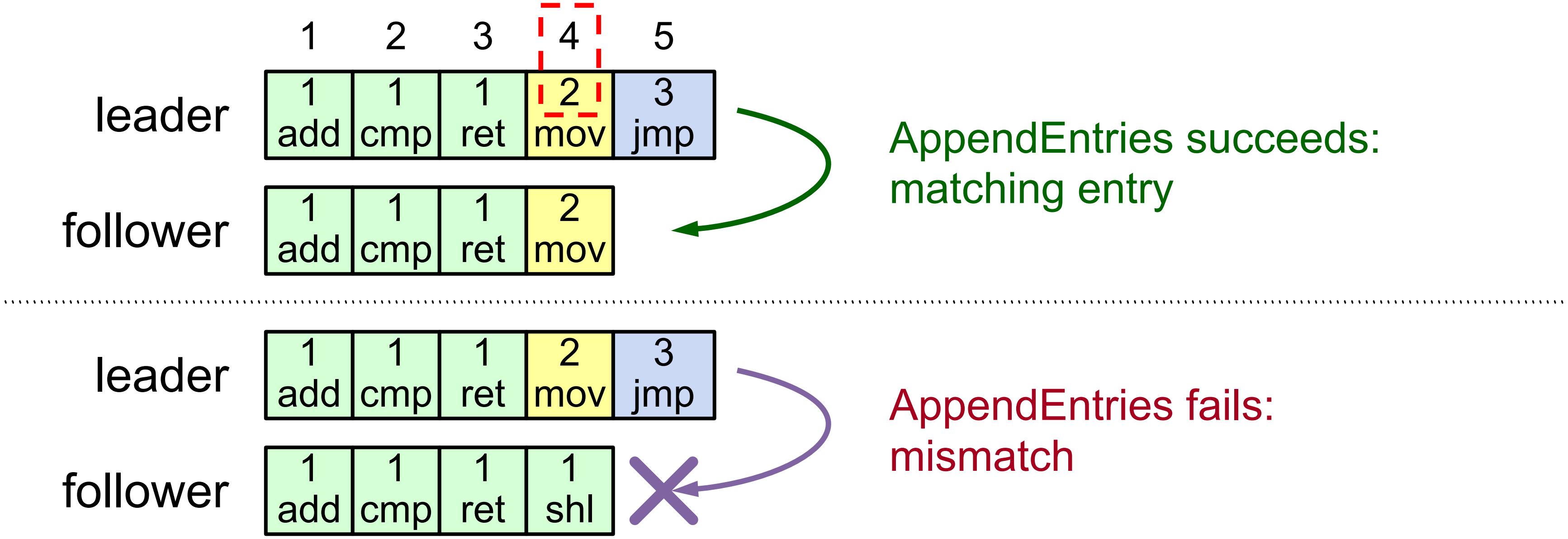
- Crashed / slow followers?
  - Leader retries RPCs until they succeed
- Performance is optimal in common case:
  - One successful RPC to any majority of servers

Log Operation: Highly Coherent

	1	2	3	4	5	6
server1	1 add	1 cmp	1 ret	2 mov	3 jmp	3 div
server2	1 add	1 cmp	1 ret	2 mov	3 jmp	4 sub

- If log entries on different server have same index and term:
  - Store the same command
  - Logs are identical in all preceding entries
- If given entry is committed, all preceding also committed

Log Operation: Consistency Check

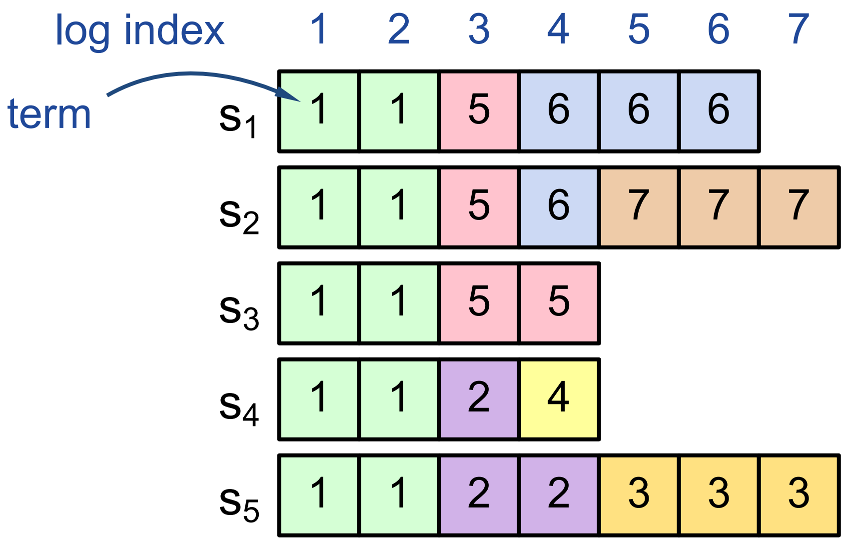


- AppendEntries has <index,term> of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects
- Implements an induction step, ensures coherency

Leader Changes

- New leader's log is truth, no special steps, start normal operation
  - Will eventually make follower's logs identical to leader's
  - Old leader may have left entries partially replicated

- Multiple crashes can leave many extraneous log entries



Safety Requirement

Once log entry applied to a state machine, no other state machine must apply a different value for that log entry

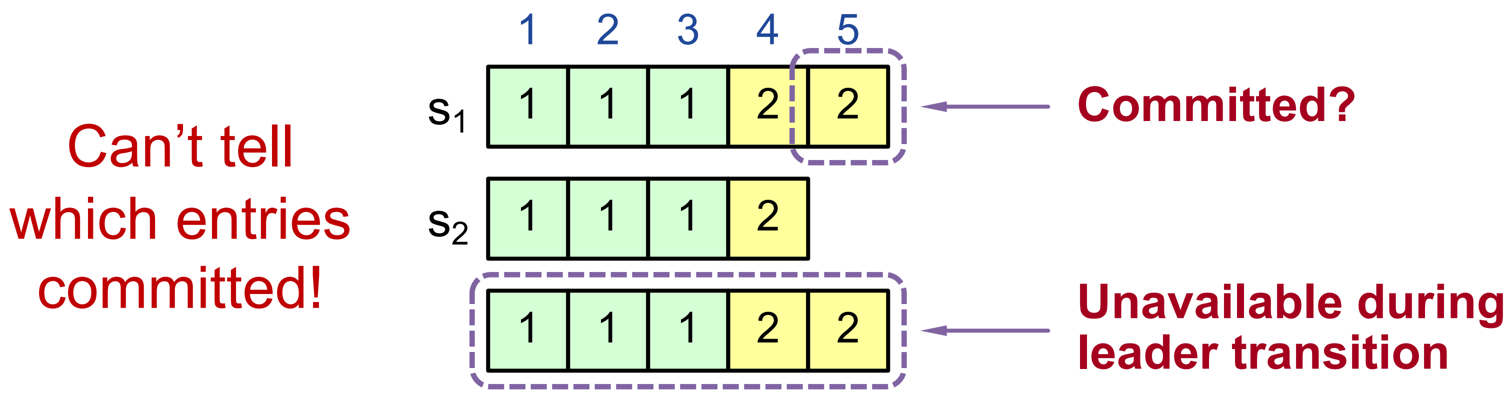
- Raft safety property: If leader has decided log entry is committed, entry will be present in logs of all future leaders
- Why does this guarantee higher-level goal?
  1. Leaders never overwrite entries in their logs
  2. Only entries in leader's log can be committed
  3. Entries must be committed before applying to state machine

Committed → Present in future leaders' logs

Restrictions on commitment

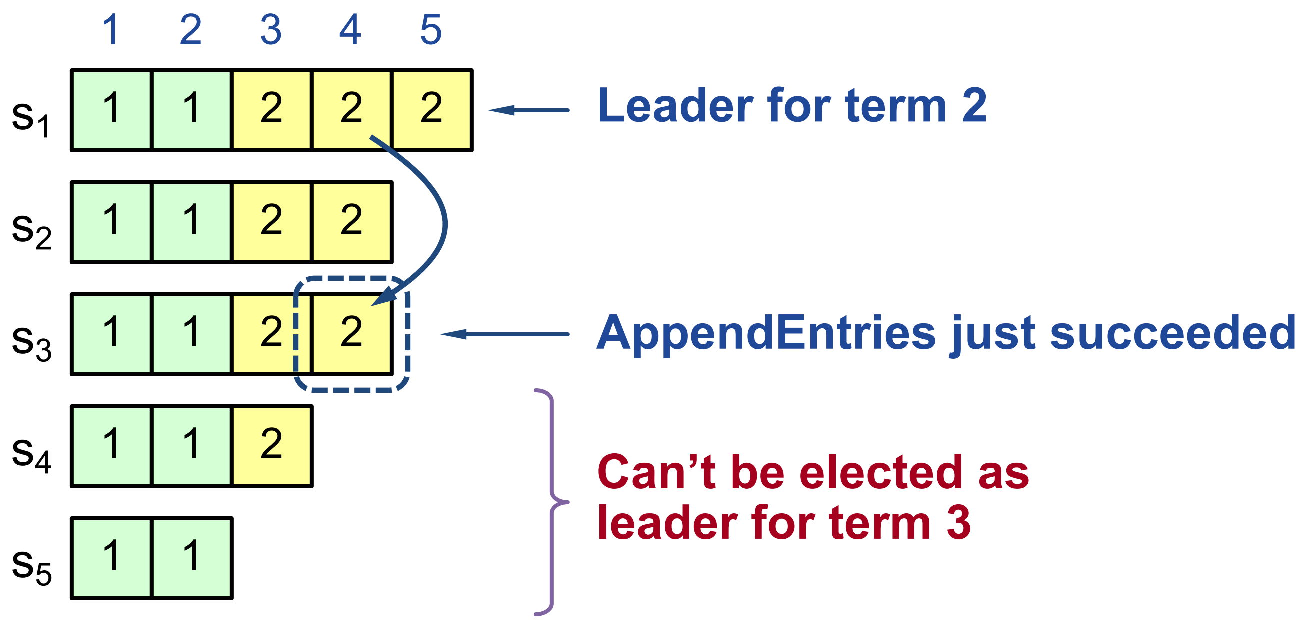
Restrictions on leader election

Picking the Best Leader



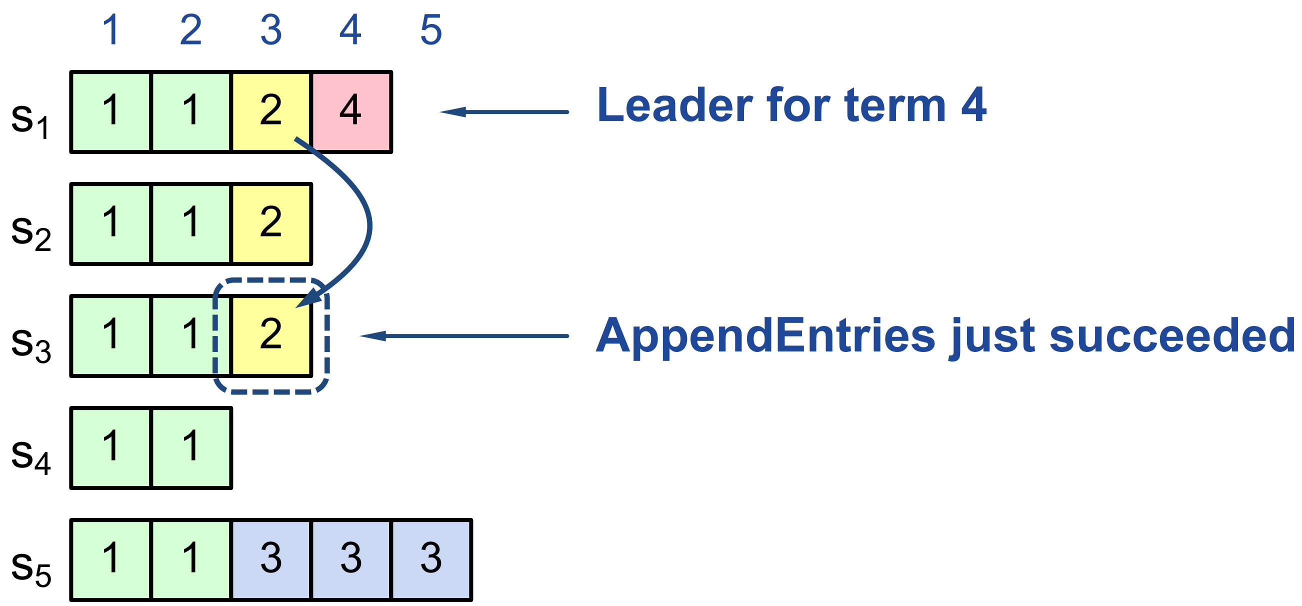
- Elect candidate most likely to contain all committed entries
  - In RequestVote, candidates incl. index + term of last log entry
  - Voter V denies vote if its log is "more complete":
    (newer term) or (entry in higher index of same term)
  - Leader will have "most complete" log among electing majority

Committing Entry from Current Term



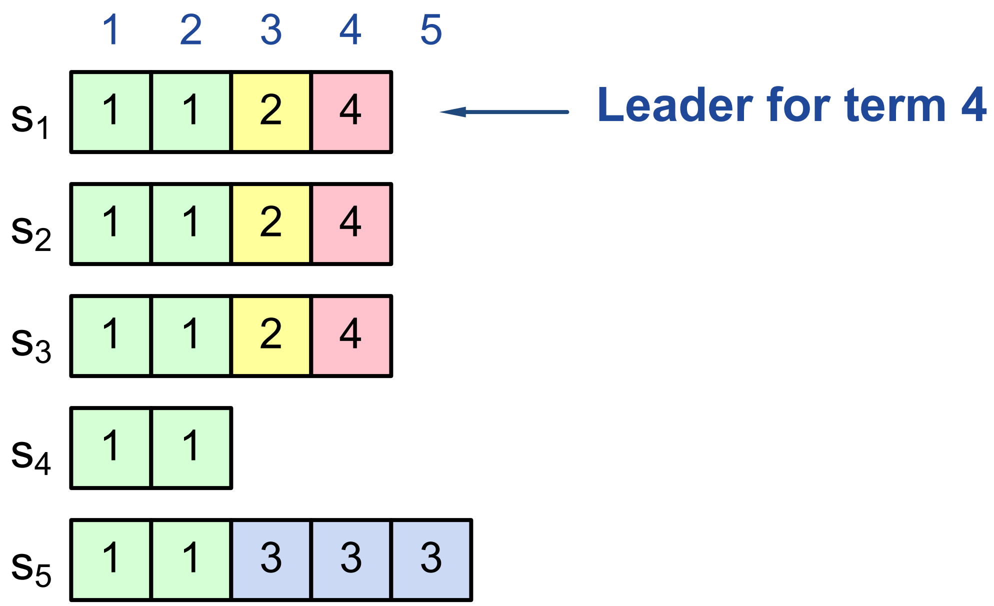
- **Case #1:** Leader decides entry in current term is committed
- Safe: leader for term 3 must contain entry 4

Committing Entry from Earlier Term



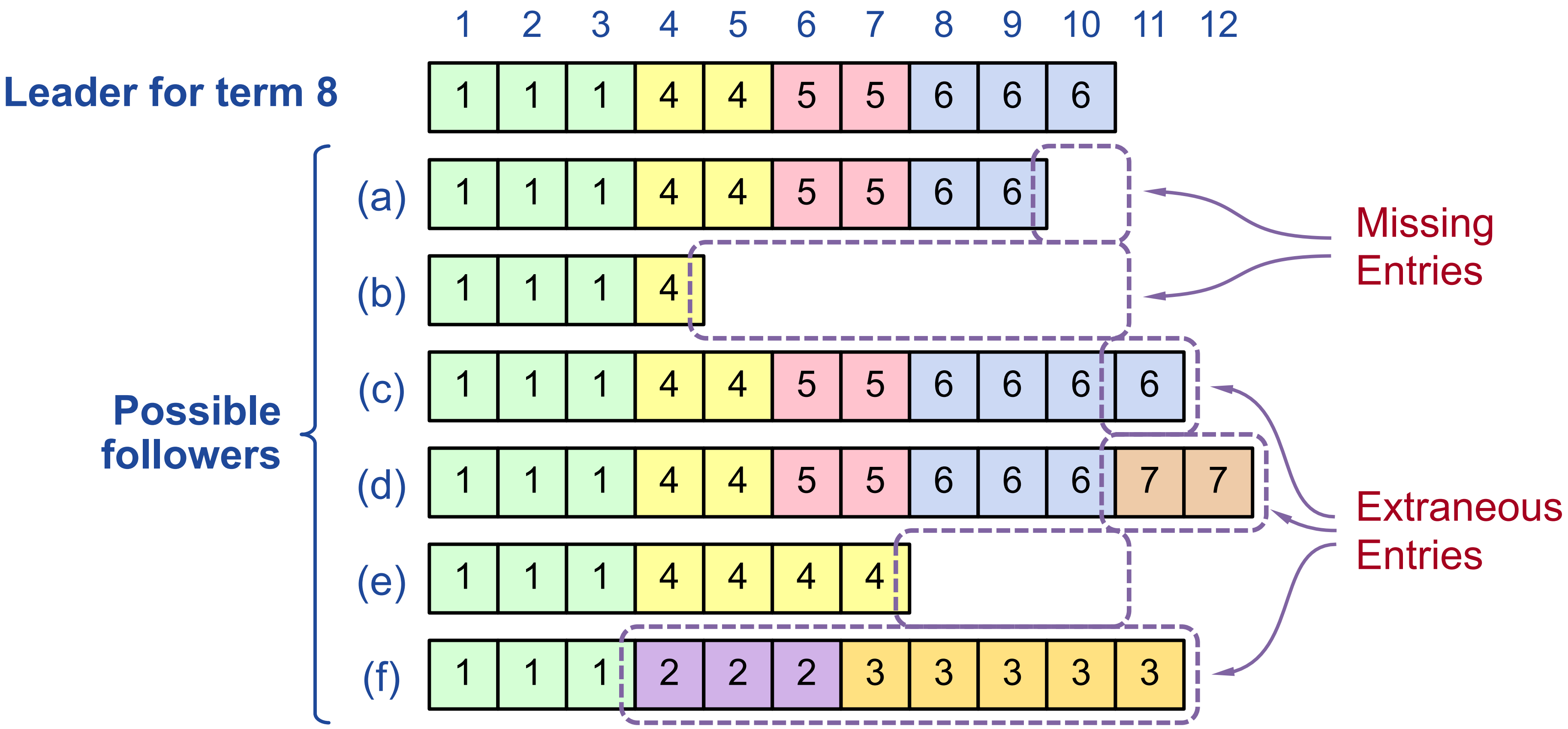
- **Case #2:** Leader trying to finish committing entry from earlier
- Entry 3 not safely committed:
  - s_5 can be elected as leader for term 5 (how?)
  - If elected, it will overwrite entry 3 on s_1, s_2, and s_3

New Commitment Rules



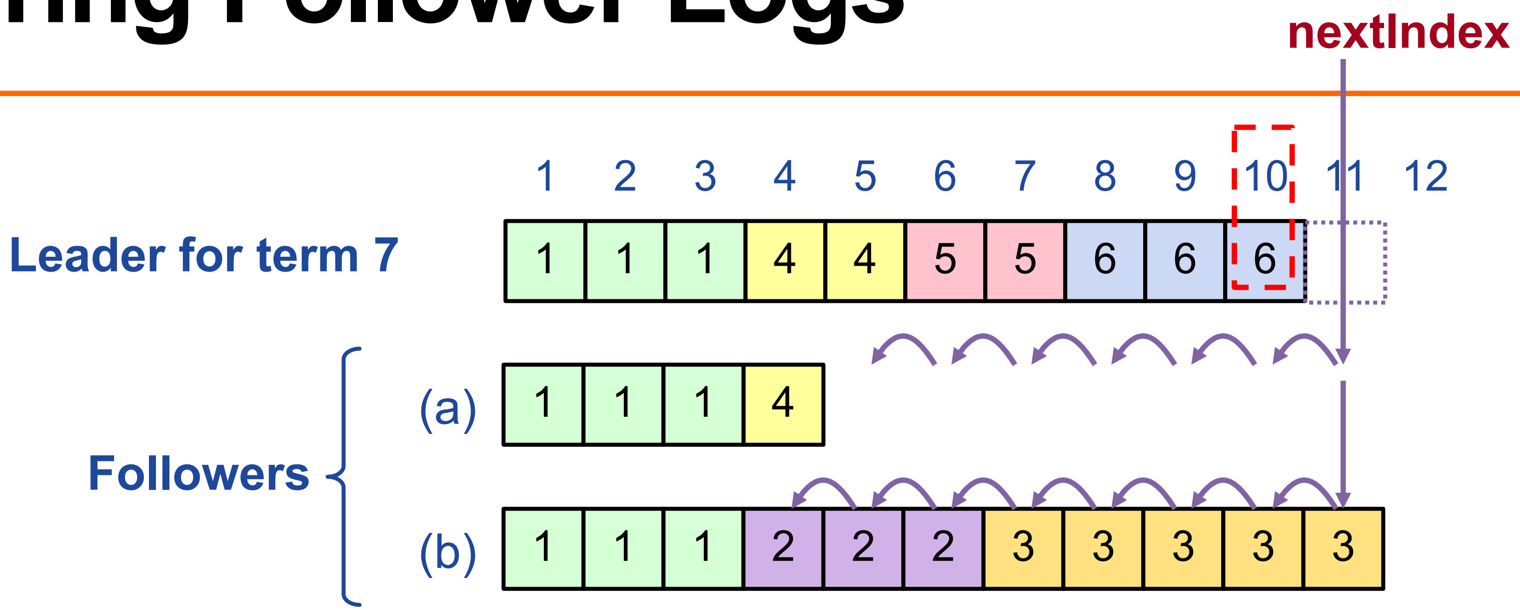
- **For leader to decide entry is committed:**
  1. Entry stored on a majority
  2. ≥ 1 new entry from leader's term also on majority
- Example; Once e4 committed, s_5 cannot be elected leader for term 5, and e3 and e4 both safe

Challenge: Log Inconsistencies



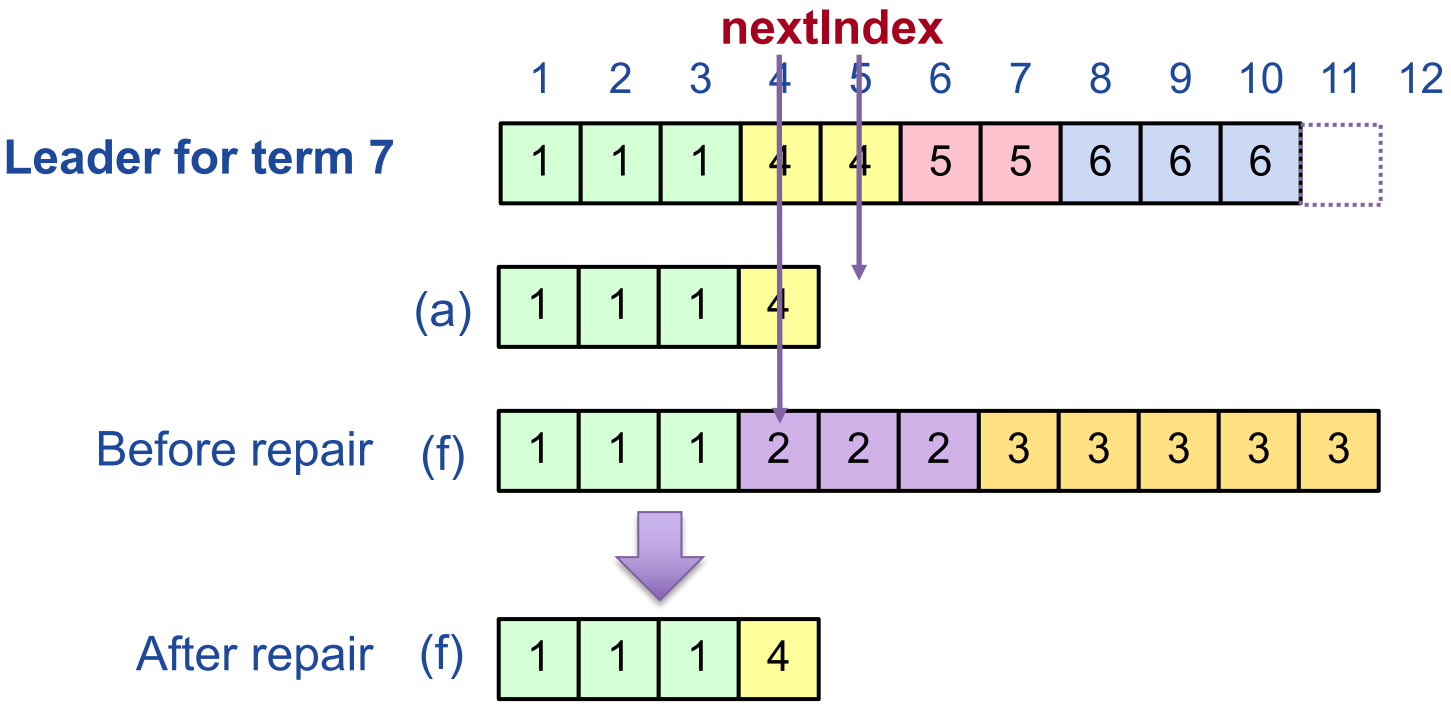
Leader changes can result in log inconsistencies

Repairing Follower Logs



- **New leader must make follower logs consistent with its own**
  - Delete extraneous entries
  - Fill in missing entries
- **Leader keeps nextIndex for each follower:**
  - Index of next log entry to send to that follower
  - Initialized to (1 + leader's last index)
- If AppendEntries consistency check fails, decrement nextIndex, try again

Repairing Follower Logs



Neutralizing Old Leaders

Leader temporarily disconnected

→ other servers elect new leader

→ old leader reconnected

→ old leader attempts to commit log entries

- **Terms used to detect stale leaders (and candidates)**
  - Every RPC contains term of sender
  - Sender's term < receiver:
    - Receiver: Rejects RPC (via ACK which sender processes…)
  - Receiver's term < sender:
    - Receiver reverts to follower, updates term, processes RPC
- **Election updates terms of majority of servers**
  - Deposed server cannot commit new log entries

Client Protocol

- **Send commands to leader**
  - If leader unknown, contact any server, which redirects client to leader
- **Leader only responds after command logged, committed, and executed by leader**
- **If request times out (e.g., leader crashes):**
  - Client reissues command to new leader (after possible redirect)
- **Ensure exactly-once semantics even with leader failures**
  - E.g., Leader can execute command then crash before responding
  - Client should embed unique ID in each command
  - This client ID included in log entry
  - Before accepting request, leader checks log for entry with same id

RAFT

- An excellent visual representation of RAFT
- https://thesecretlivesofdata.com/raft/

© 2024 - Dr. Basit Qureshi