Leader Election

K V SUBRAMANIAM

Leader Election

- In many distributed applications, particularly the centralized solutions, some process needs to be declared the central coordinator
- Electing the leader also may be necessary when the central coordinator crashes
- Election algorithms allow processes to elect a unique leader in a decentralized manner

When and how to elect?

- During the system initiation
- When the existing leader fails.
  - If there’s no response from the current leader for a predetermined time

- The process has to completely distributed
- Why?
  - Failure of leader is detected independently by any other node
  - Starts the election independently

Liveness and Safety

- Safety
  - Ensures that only one leader is elected at a time
  - Guarantees
    - No two nodes believe they are the leader
    - Prevents split-brain scenarios
    - Ensures consistency in decision making
- Liveness
  - Ensures that *leader is eventually elected,* even in the presence of delays and failures
  - Guarantees
    - System makes progress
    - New leader is elected after a timeout/failure
    - Prevents indefinite waiting or deadlock

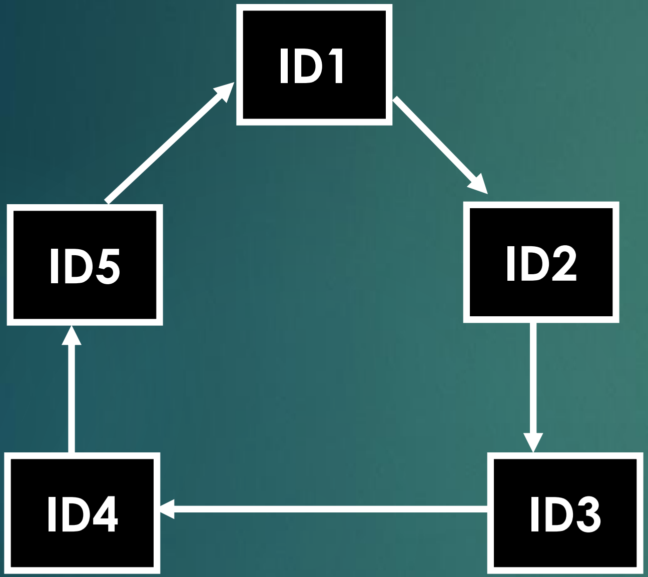
Safety in Kubernetes

- Leader election used to ensure only one instance of controllers e.g, scheduler
- Mechanism
  - Stores *lease objects* in the API server
  - Each candidate tries to acquire the lease
  - Only one pod (or process) can hold lease at a time
  - Example
    - If two pods try to acquire a lease, one gets it
    - Other backs off and monitors the lease
  - Lease updates are atomic. API server ensures consistency

Liveness in Kubernetes

- Mechanism
  - Lease has a renewal lifetime
  - On failure of renew lease in time
    - Others detect expiry
    - New leader is elected by acquiring the lease
  - Each candidate tries to acquire the lease
  - Only one pod (or process) can hold lease at a time
  - Example
    - If two pods try to acquire a lease, one gets it
    - Other backs off and monitors the lease
  - Lease updates are atomic. API server ensures consistency

Leader Election in a Ring



- Each process has unique ID; can receive messages from left, and send messages to the right
- Goal: agree on who is the leader (initially everyone knows only its own ID)
- Idea:
  - initially send your own ID to the right. When you receive an ID from left, if it is higher than what you have seen so far, send it to right.
  - If your own ID is received from left, you have the highest ID and are the leader

Raft

Consensus algorithm

Overall system eliability

In presence of faulty processes

Coordinating processes need to agree

Key Concepts

Leader- sends heartbeat to followers

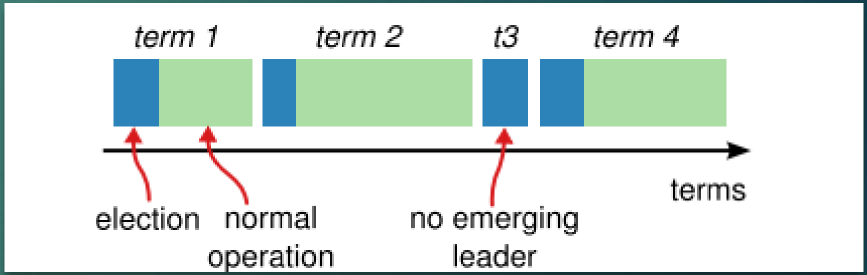
Followers

Candidate – when leader is not available

Term – period of time for which the new leader needs to be elected.

What is a term?

- Each server stores it's *current term*
  - Think of it as a logical clock
- Increases monotonically with time
- Exchange current term when servers communicate
- If (current term < received current term)
  - Update current term to higher value received
- If leaders current term is out of date
  - Leader becomes follower
- If server receives request from stale term
  - Reject the request



Raft Leader Election - etcd

Follower Timeout

- A follower doesn't hear from a leader within the election timeout.

Becomes Candidate

- It increments its current term and starts an election.
- Marks itself as a canidate

Requests Votes

- Sends RequestVote RPCs to other nodes.

Voting

- Each node can vote for one candidate per term. FCFS
- If a candidate receives a majority of votes, it becomes the leader.

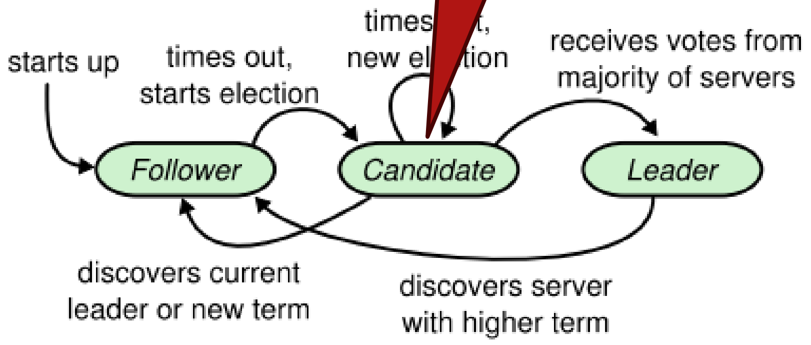
Leader Heartbeats

- The new leader sends AppendEntries (heartbeats) to maintain authority.
- While waiting as candidate it receives AppendEntries from >= current term value, it becomes a follower.

Followers Reset Timers

- On receiving heartbeats, followers reset their election timers.

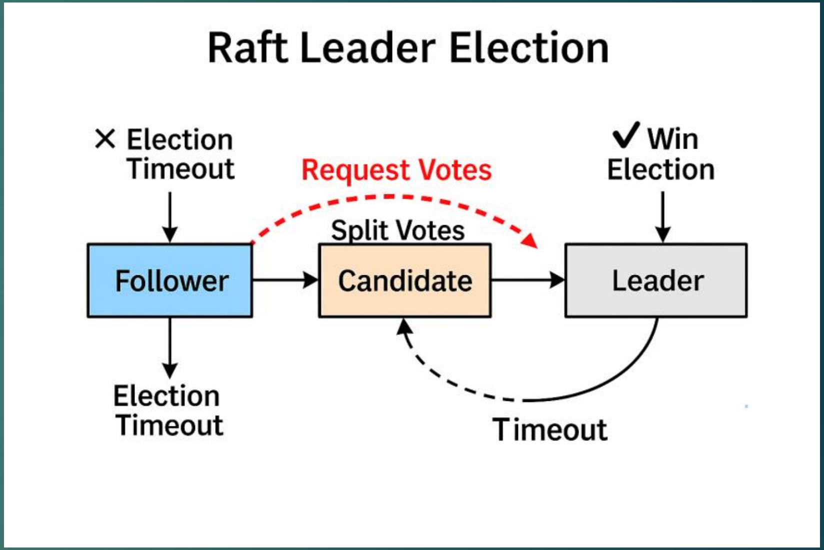
Remains candidate till
- Wins election
- Other server wins election
- Timeout with no winner


starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

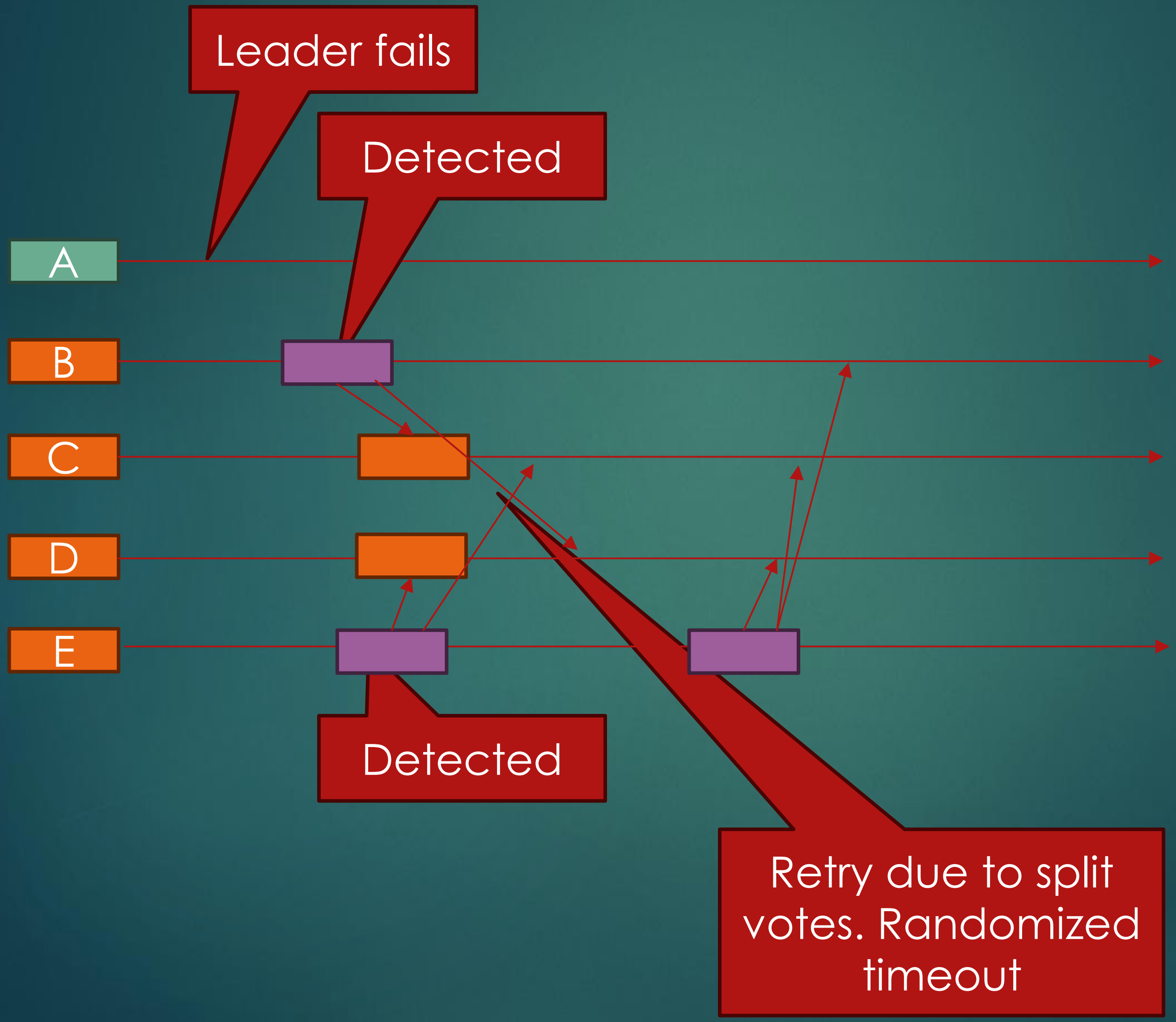
Courtesy: https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf

Handling split vote

- Why
  - Due to network delays
  - Simultaneous timeouts
  - Partitioned clusters
- Handling
  - Randomize timeout
  - No majority no leader
  - Retry election
  - Term increment
    - Votes granted to candidate with up to date logs and higher terms



Handling split vote



Cluster Coordination Service: Zookeeper

Examples: Cluster Coordination Problems

- Cluster membership
  - Node A thinks B is down; node B thinks A is down
  - Node A tries to recover B; B tries to recover A
- Distributed locking
  - Multiple nodes can try to access same resource
- Leader election
  - Nodes elect a "leader" that will perform certain critical functions
  - For example, generate a unique transaction ID
  - How can we ensure that all nodes end up with same leader?

Cluster Coordination

- Simple if
  - No node crashes
  - No network congestion
  - …
- These are not realistic assumptions

Zookeeper Overview

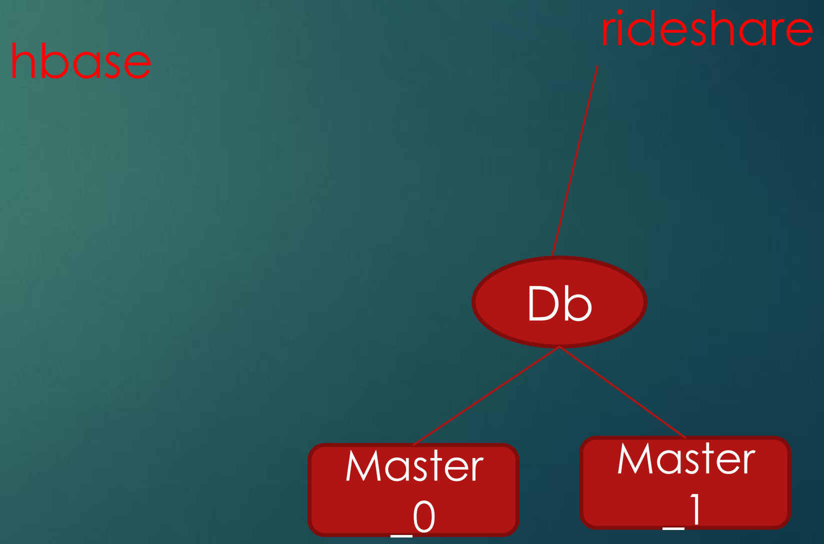
- Zookeeper provides access to a shared data structure
- Nodes performing a distributed computation (clients) can Read / write this data structure
- This data structure is guaranteed to be consistent across all clients
- Clients can use this data structure to coordinate their actions
- Examples will be given

Zookeeper architecture

- Zookeeper is a distributed service
- Clients can read data structure from any node
- Writes go to the leader which coordinates updates
  - Majority of nodes must accept write

Zookeeper Shared Data Structure

- Hierarchical arrangement of znodes
  - Naming: full path name, like a file
  - znodes can hold data
- Znodes are of various types
  - Persistent - configuration information like folder names, port numbers etc.
  - Ephemeral: destroyed if client crashes
  - Sequence: Add unique sequence number to pay
- Znode operationS
  - CRUD
  - WATCH: BE NOTIFIED ON CHANGE



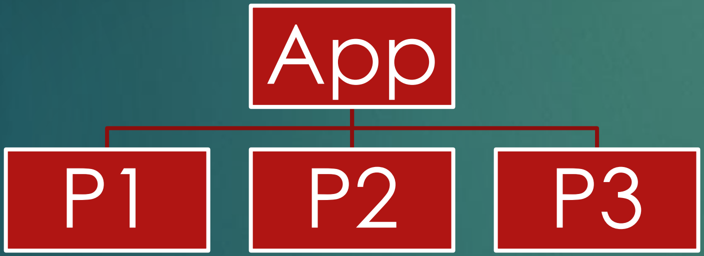
Class Exercise (10 mins)

- Suppose there is a CPU with 2 cores
- P1, P2 are running on the cores
- Both processes are updating a common variable *count* which counts some event
- Is there a problem?
- How can be problem be solved?

Class Exercise (Solution)

- Problem: race condition
  - P1 and P2 read *count* at the same time (say 15)
  - P1 and P2 both update it to 16
  - Correct answer should be 17
- Solution: a *lock* (similar to a gatekeeper)
  - Forces P1 and P2 to access the variable in turn
  - If P1 has locked the variable, P2 has to wait
  - So they cannot read *count* at the same time

Distributed locking



- Get lock on resource /app1/p
- Locking
  - Call Create() on /app1/p1 with Ephemeral sequence node
  - Increments sequence number in node name
  - Returns name of node created
  - Call get_children()
  - If node name same as create, got lock
  - Otherwise watch lock with next lowest number
- Unlock: delete created node. Will wake up another client

Class Exercise (5 mins)

- Suppose there are 3 processes
- P1 gets a lock on a shared resource R
- P2 tries to get the lock and is suspended
- P3 tries to get a lock and is suspended
- P1 releases the lock
- Some process (which one?) gets the lock

Zookeeper Recipes

- Distributed locks is an example of a zookeeper recipe
- Many other recipes

Videos

- https://www.youtube.com/watch?v=gifeThkqHjg
  - Apache Hadoop Zookeeper
- https://www.youtube.com/watch?v=gZj16chk0Ss
  - 064 Zookeeper Explained
- Original RAFT paper
  - Ongaro and Ousterhout, "In search of an understandable consensus algorithm", USENIX, 2014