Service Based Architecture: Kubernetes Implementation

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs434/

04/26/2021
Admin

Projects update

- Week 4 (Apr 23 - Apr 29): Prototyping; Mid-point checkpoint; meet w/ the instructor
  - Offline hours:
    - Tuesday 10:30—11:30 am
    - Wednesday: 5:15 – 6:15 pm after class
    - Thursday: 1-2 pm
    - Friday: 11-12

- Week 5 (Apr 30 - May 6): Refinement; iterations
- Week 6 (May 7 - May 13): Final implementation, final report (6-8 pages)
Recap: Kubernetes Foundation

- **Containers**

```python
def get_random_bytes():
    r = requests.get("http://rng/32")
    return r.content

def hash_bytes(data):
    r = requests.post("http://hasher/",
                      data=data,
                      headers={"Content-Type": "application/octet-stream"})
    hex_hash = r.text
    return hex_hash
```

```ini
FROM python:alpine
RUN pip install redis
RUN pip install requests
COPY worker.py /
CMD ["python", "worker.py"]
```
Recap: Kubernetes Basic Service Model

- **Pod:** models an application-specific "logical host"
  - A pod contains one or more application containers which are relatively tightly coupled
  - Pod’s contents are always co-located and co-scheduled, and run in a shared storage and network context

- **Deployment:** provides a declarative specification about a desired state
  - Deployment controller changes the actual state to the desired state at a controlled rate
    - `kubectl create deployment nginx --image=nginx`
    - `kubectl scale deployment nginx --replicas=3`

- **Service:** provides a stable address for a pod (or a bunch of pods)
  - *ClusterIP, NodePort, LoadBalancer, ExternalName*
    - `kubectl expose deployment nginx --port 8888`
    - `kubectl expose deploy/webui --type=NodePort --port=80 // assume allocate port 31207`
    - `http://0.0.0.0:31207/index.html`
Recap: Node, Container, Pod, Service (IP addr)
Discussion

- What are the functions needed to realize Kubernetes?

  Our focus
  
  - Configuration and coordination data store, keeping track of state, e.g.,
    - cluster resource availability (e.g., nodes status)
    - user requested target state (create, delete, …)
    - container states realizing user requests (container running state, …)
  
  - Controller-manager
    - feedback control loop between current state and user request target state
  
  - Scheduling
    - pod -> node binding
  
  - Data path setup (container/packet run-time set up)
    - L3 routing setup, L4/L7 set up (load balancing, transformation of packets) …
Recap: Kubernetes Implementation Architecture
Outline

- Admin and recap
- Microservice oriented architecture - Kubernetes
  - Overview
  - Basic model
  - Kubernetes implementation
    - Reliable coordination/configuration data store
Overview

- Kubernetes coordination/configuration data store is based on etcd, a strongly consistent, distributed key-value store
- etcd is based on RAFT, a consensus protocol
- Raft designed in 2013 (Diego Ongaro, John Ousterhout)
  - Later raft slides based on talk by Ongaro and Ousterhout
- We will focus on core ideas. Additional details see references
  - [https://blog.containership.io/etcd/](https://blog.containership.io/etcd/)
  - [https://github.com/etcd-io/etcd](https://github.com/etcd-io/etcd)
  - [https://etcd.io/docs/v3.4.0/learning/](https://etcd.io/docs/v3.4.0/learning/)
Motivation: Why is Data Store Hard to Design?

- Design 1: a single server provides a data store (e.g., receiving user requests, configurations, node status report, controller state control)
  - Exercise: what may go wrong?

```
create deployment nginx --image=nginx
```

```
delete deploy/nginx
```
Motivation: Why is Data Store Hard to Design?

- Design 2: use multiple servers (say 2). The servers run a leader election protocol (e.g., VRRP/OSPF designated-router leader election, with node assigned priority), and a client picks the highest live server?

```
create deployment nginx --image=nginx
```

Old leader

New leader

New leader has no old state
**Motivation: Why is Data Store Hard to Design?**

- Design 2: use multiple servers (say 2). The servers run a leader election protocol (e.g., VRRP/OSPF designated-router leader election, with node assigned priority), and a client picks the highest live server?

```
create deployment nginx --image=nginx
```

- Leader for partition 1

- Leader for partition 2
Replicated operations log => replicated state machine => a server has state when old leader crashes and this server becomes the new leader.

Consensus replication protocol
- Time consistency => a client switching from crashed leader to new leader will not cause inconsistency
- Spatial consistency => System state will not diverge if network partitioned

Note: Assumed failure model: fail-stop (not Byzantine), delayed/lost messages
Note: Alternative Design

Two general approaches:

- Asymmetric, leader-based protocol (our focus)
  - At any given time, one server is in charge, others accept its decisions
  - Clients communicate with the leader
  - Example: Raft (etcd -> Kubernetes), Zab (ZooKeeper -> Kafka)

- Symmetric, leader-less (alternative)
  - All servers have equal roles
  - Clients can contact any server
  - Example: Paxos

“The dirty little secret of the NSDI community is that at most five people really, truly understand every part of Paxos ;-).” —NSDI reviewer
## Consensus Protocol and Network Systems

<table>
<thead>
<tr>
<th>System</th>
<th>Protocol</th>
<th>Implementation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google GFS</td>
<td>Multi-Paxos</td>
<td>Chubby</td>
<td>Lock Service</td>
</tr>
<tr>
<td>Google Spanner</td>
<td>Multi-Paxos</td>
<td>Chubby</td>
<td></td>
</tr>
<tr>
<td>Google Borg</td>
<td>Multi-Paxos</td>
<td>Chubby</td>
<td>Configuration, Master election</td>
</tr>
<tr>
<td>Apache HDFS</td>
<td>Zab</td>
<td>ZooKeeper</td>
<td>Failure detection, Active NameNode election</td>
</tr>
<tr>
<td>Apache Giraph</td>
<td>Zab</td>
<td>ZooKeeper</td>
<td>Coordination, Configuration, Aggregators</td>
</tr>
<tr>
<td>Apache Hama</td>
<td>Zab</td>
<td>ZooKeeper</td>
<td>Coordination</td>
</tr>
<tr>
<td>CoreOS</td>
<td>Raft</td>
<td>etcd</td>
<td>Service Discovery</td>
</tr>
<tr>
<td>OpenStack</td>
<td>Zab</td>
<td>ZooKeeper</td>
<td>Service Discovery</td>
</tr>
<tr>
<td>Apache Kafka</td>
<td>Zab</td>
<td>ZooKeeper</td>
<td>Coordination, Configuration</td>
</tr>
<tr>
<td>Apache BookKeeper</td>
<td>Zab</td>
<td>ZooKeeper</td>
<td>Coordination, Configuration</td>
</tr>
</tbody>
</table>
Raft Components

- Basic leader election
- Basic log replication (normal operation)
- Revised leader election and log replication
- Neutralizing old leaders
- Client interactions
  - Implementing linearizeable semantics

Configuration changes
  - Adding and removing servers
Raft Protocol Specification

Followers
- Respond to RPCs from candidates and leaders.
- Convert to candidate if election timeout elapses without either:
  - Receiving valid AppendEntries RPC, or
  - Granting vote to candidate

Candidates
- Increment currentTerm, vote for self
- Reset election timeout
- Send RequestVote RPCs to all other servers, wait for either:
  - Votes received from majority of servers: become leader
  - AppendEntries RPC received from new leader: step down
- Election timeout elapses without election resolution: increment term, start new election
- Discover higher term: step down

Leaders
- Initialize nextIndex for each to last log index + 1
- Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts
- Accept commands from clients, append new entries to local log
- Whenever last log index ≥ nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful
- If AppendEntries fails because of log inconsistency, decrement nextIndex and retry
- Mark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of servers
- Step down if currentTerm changes

Persistent State
Each server persists the following to stable storage synchronously before responding to RPCs:
- currentTerm: latest term server has seen (initialized to 0 on first boot)
- votedFor: candidateId that received vote in current term (or null if none)
- log: log entries

Log Entry
- term: term when entry was received by leader
- index: position of entry in the log
- command: command for state machine

RequestVote RPC
Invoked by candidates to gather votes.
- Arguments:
  - candidateId: candidate requesting vote
  - term: candidate's term
  - lastLogIndex: index of candidate's last log entry
  - lastLogTerm: term of candidate's last log entry
- Results:
  - term: currentTerm, for candidate to update itself
  - voteGranted: true means candidate received vote
- Implementation:
  1. If term > currentTerm, currentTerm ← term (step down if leader or candidate)
  2. If term == currentTerm, votedFor is null or candidateId, and candidate's log is at least as complete as local log, grant vote and reset election timeout

AppendEntries RPC
Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat.
- Arguments:
  - term: leader's term
  - leaderId: so follower can redirect clients
  - prevLogIndex: index of log entry immediately preceding new ones
  - prevLogTerm: term of prevLogIndex entry
  - entries: log entries to store (empty for heartbeat)
  - commitIndex: last entry known to be committed
- Results:
  - term: currentTerm, for leader to update itself
  - success: true if follower contained entry matching prevLogIndex and prevLogTerm
- Implementation:
  1. Return if term < currentTerm
  2. If term > currentTerm, currentTerm ← term
  3. If candidate or leader, step down
  4. Reset election timeout
  5. Return failure if log doesn’t contain an entry at prevLogIndex whose term matches prevLogTerm
  6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry
  7. Append any new entries not already in the log
  8. Advance state machine with newly committed entries
Basic Operations

- Client sends commands to leader
- Leader appends command to its log
- Leader sends AppendEntries commands to followers (called commit)
- Leader executes command to its state machine, returns result to client

create deployment nginx --image=nginx

delete deploy/nginx
Detecting Failures of Basic Operations

- Problem: leader can crash, need detection

- Q: How may a client detect leader crash?
  - A: Client request should have time out (what does a client do after detecting leader crash?)

- Q: How can a server detect leader crash?
  - A: Leader sends heartbeats to maintain authority
  - If no heartbeats (typically 100-500ms), assume leader crashes (what does a server do after detecting leader crash?)
Fixing Basic Failure: Basic Election

- Server States and State Transitions

- Follower
  - start
  - timeout, start election
  - loss election

- Candidate
  - timeout, new election
  - "step down"

- Leader
  - win election

"timeout, new election"
Basic Election Requirements

- **Safety** requirement: allow at most one winner per election
- **Liveness** requirement: some candidate must eventually win
- **Exercise**: how about assigning unique ID to servers and picking the highest ID server as next leader?
**Example Server States**

<table>
<thead>
<tr>
<th>Server ID</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
<td>div</td>
<td>shl</td>
<td>sub</td>
</tr>
<tr>
<td>1</td>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
<td>div</td>
<td>shl</td>
<td>sub</td>
</tr>
<tr>
<td>4</td>
<td>add</td>
<td>cmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Highest ID server may not have the most up-to-date state
Raft Basic Election Requirements and Mechanisms

- **Safety** requirement: allow at most one winner per election
  - Mechanism: Winner must get majority vote, and each server gives out only one vote per term (persist on disk)
    - Each election is identified by a number (term)

- **Liveness** requirement: some candidate must eventually win
  - Mechanism: To avoid livelock (persistent split votes), choose election timeouts randomly in \([T, 2T]\)
    - Works well if \(T \gg\) broadcast time, one server usually times out and wins election before others wake up
Basic Election: Big Picture

- Time divided into terms:
  - Election
  - Normal operation under a single leader
- At most one leader per term
  - Some terms have no leader (failed election)
- Each server maintains current term value
Basic Election: Details

- Increment current term
- Change to Candidate state
- Vote for self
- Send RequestVote RPCs to all other servers, retry until one of the following:
  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

Need refinement to basic election. See later.
Basic Log Operations + Basic Election (BLO+BE)

- Log entry = index, term, command
- Log stored on stable storage (disk); survives crashes

Exercise: what are
- BLO commit design options?
- BE election-rule design options?
Leader Commit Complexity: Leader Changes

Due to leader change, at beginning of new leader’s term:
- Old leader may have left entries partially replicated
- Multiple crashes can leave many extraneous log entries:

<table>
<thead>
<tr>
<th>log index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>leader for term 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>possible followers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>(c)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>(d)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>(e)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(f)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Leader changes can result in log complexities:
- Missing Entries
- Extraneous Entries
Raft Design Decision on Leader Commit w/ Leader Changes

- Leader’s log is “the truth”
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders
  - No special steps by a new leader to revise leader log

- Leader will eventually make follower’s logs identical to leader’s
Q: Can the leader (s1) commit Log[4]?

A: If only simple majority leader election, if s1 crashes right after commit, S5 may become the next leader, but S5 misses Log[4].

Implication: If the next leader is guaranteed to be one of those with Log[4] already replicated, then it is safe.
Leader Election Rule

- During elections, choose candidate with log from newer terms
  - Candidates include log info in RequestVote RPCs (index & term of last log entry)
  - Voting server V denies vote if its log is “more complete”:
    \[(\text{lastTerm}_V > \text{lastTerm}_C) \lor (\text{lastTerm}_V == \text{lastTerm}_C) \land (\text{lastIndex}_V > \text{lastIndex}_C)\]
  - Leader will likely have “most complete” log among electing majority
Leader Election: Example

Since any new leader will have Log[4], it is OK to commit it.
Leader Commit: Continue

Q: Assume revised leader election, can the leader (s1) commit Log[3] in this case?

A: The revised leader election may elect s5, again causing problem:
If elected, it will overwrite entry 3 on s1, s2, and s3!

Raft solution: At least one new entry from leader’s term must also be stored on majority of servers.
Raft Commitment Rules

- For a leader to decide an entry is committed:
  - Must be stored on a majority of servers
  - At least one new entry from leader's term must also be stored on majority of servers

- Once entry 4 stored on majority:
  - $s_5$ cannot be elected leader for term 5
  - Entries 3 and 4 both safe
Summary: Raft Safety

- Raft safety property:
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders.
  - No special steps by a new leader to revise leader log.

- Solution
  - Use a combination of election rules and commitment rules to achieve safety.
Neutralizing Old Leaders

- Deposed leader may not be dead:
  - Temporarily disconnected from network
  - Other servers elect a new leader
  - Old leader becomes reconnected, attempts to commit log entries

- Terms used to detect stale leaders (and candidates)
  - Every RPC contains term of sender
  - If sender's term is older, RPC is rejected, sender reverts to follower and updates its term
  - If receiver's term is older, it reverts to follower, updates its term, then processes RPC normally

- Election updates terms of majority of servers
  - Deposed server cannot commit new log entries
**Updating Follower’s Logs**

Leader changes can result in log inconsistencies:

- **Possible followers**: (a), (b), (c), (d), (e), (f)

**Implication**: New leader must make follower logs consistent with its own
- Delete extraneous entries
- Fill in missing entries
Log Consistency Invariant

High level of coherency between logs:

- If log entries on different servers have same index and term:
  - They store the same command
  - The logs are identical in all preceding entries

- If a given entry is committed, all preceding entries are also committed
AppendEntries Consistency Check

- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request (see later for how this is used)
- Implements an induction step, ensures coherency

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>12</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
</tr>
</tbody>
</table>

leader
```

Follower

```plaintext
add cmp ret mov
```

AppendEntries succeeds: matching entry

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
</tr>
</tbody>
</table>

leader

```plaintext
add cmp ret mov
```

follower

```
add cmp ret shl
```

AppendEntries fails: mismatch
Repairing Follower Logs

- Leader keeps nextIndex for each follower:
  - Index of next log entry to send to that follower
  - Initialized to \((1 + \text{leader's last index})\)
- When AppendEntries consistency check fails, decrement nextIndex and try again:

```
log index   1  2  3  4  5  6  7  8  9  10 11 12
leader for term 7 1 1 1 4 4 5 5 6 6 6 6 6
(a) followers 1 1 1 4
(b) followers 1 1 1 2 2 2 3 3 3 3 3 3
```
When follower overwrites inconsistent entry, it deletes all subsequent entries:

<table>
<thead>
<tr>
<th>log index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>leader for term 7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>follower (before)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>follower (after)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Introducing Clients:
Raft as Client/Server Protocol

- **Client sends commands to leader**
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirect to leader
  - If request times out (e.g., leader crash):
    - Client reissues command to some other server
    - Eventually redirected to new leader
    - Retry request with new leader

- **Leader response**
  - Leader does not respond until command has been logged, committed, and executed by leader’s state machine
Introducing Clients: Raft as Client/Server Protocol

- Problem: a leader crashes after executing command, but before responding
  - Must not execute command twice

- Solution: client embeds a unique id in each command
  - Server includes id in log entry
  - Before accepting command, leader checks its log for entry with that id
  - If id found in log, ignore new command, return response from old command

- Result: exactly-once semantics as long as client doesn’t crash
Basic Raft Protocol Summary

<table>
<thead>
<tr>
<th>Raft Protocol Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Followers</strong></td>
</tr>
<tr>
<td>• Respond to RPC's from candidates and leaders.</td>
</tr>
<tr>
<td>• Convert to candidate if election timeout elapses without either:</td>
</tr>
<tr>
<td>• Receiving valid AppendEntries RPC, or</td>
</tr>
<tr>
<td>• Granting vote to candidate</td>
</tr>
<tr>
<td><strong>Candidates</strong></td>
</tr>
<tr>
<td>• Increment currentTerm, vote for self</td>
</tr>
<tr>
<td>• Reset election timeout</td>
</tr>
<tr>
<td>• Send RequestVote RPCs to all other servers, wait for either:</td>
</tr>
<tr>
<td>• Votes received from majority of servers: become leader</td>
</tr>
<tr>
<td>• AppendEntries RPC received from new leader: step down</td>
</tr>
<tr>
<td>• Election timeout elapses without election resolution: increment term, start new election</td>
</tr>
<tr>
<td>• Discover higher term: step down</td>
</tr>
<tr>
<td><strong>Leaders</strong></td>
</tr>
<tr>
<td>• Initialize nextIndex for each to last log index + 1</td>
</tr>
<tr>
<td>• Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts</td>
</tr>
<tr>
<td>• Accept commands from clients, append new entries to local log</td>
</tr>
<tr>
<td>• Whenever last log index ≥ nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful</td>
</tr>
<tr>
<td>• If AppendEntries fails because of log inconsistency, decrement nextIndex and retry</td>
</tr>
<tr>
<td>• Mark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of servers</td>
</tr>
<tr>
<td>• Step down if currentTerm changes</td>
</tr>
<tr>
<td><strong>Persistent State</strong></td>
</tr>
<tr>
<td>Each server persists the following to stable storage synchronously before responding to RPCs:</td>
</tr>
<tr>
<td>currentTerm</td>
</tr>
<tr>
<td>votedFor</td>
</tr>
<tr>
<td>log[]</td>
</tr>
<tr>
<td><strong>Log Entry</strong></td>
</tr>
<tr>
<td>term</td>
</tr>
<tr>
<td>index</td>
</tr>
</tbody>
</table>

### RequestVote RPC

Invoked by candidates to gather votes.

**Arguments:**
- **candidateId** candidate requesting vote
- **term** candidate's term
- **lastLogIndex** index of candidate's last log entry
- **lastLogTerm** term of candidate's last log entry

**Results:**
- **term** currentTerm, for candidate to update itself (step down if leader or candidate)
- **voteGranted** true means candidate received vote

**Implementation:**
1. If term > currentTerm, currentTerm ← term (step down if leader or candidate)
2. If term == currentTerm, votedFor is null or candidateId, and candidate's log is at least as complete as local log, grant vote and reset election timeout

### AppendEntries RPC

Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat.

**Arguments:**
- **term** leader's term
- **leaderId** so follower can redirect clients
- **prevLogIndex** index of log entry immediately preceding new ones
- **prevLogTerm** term of prevLogIndex entry
- **entries[]** log entries to store (empty for heartbeat)
- **commitIndex** last entry known to be committed

**Results:**
- **term** currentTerm, for leader to update itself
- **success** true if follower contained entry matching prevLogIndex and prevLogTerm

**Implementation:**
1. Return if term < currentTerm
2. If term > currentTerm, currentTerm ← term
3. If candidate or leader, step down
4. Reset election timeout
5. Return failure if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry
7. Append any new entries not already in the log
8. Advance state machine with newly committed entries
Configuration Changes

- System configuration:
  - ID, address for each server
  - Determines what constitutes a majority

- Consensus mechanism must support changes in the configuration:
  - Replace failed machine
  - Change degree of replication
Cannot switch directly from one configuration to another: conflicting majorities could arise
Raft uses a 2-phase approach:

- Intermediate phase uses joint consensus (need majority of both old and new configurations for elections, commitment)
- Configuration change is just a log entry; applied immediately on receipt (committed or not)
- Once joint consensus is committed, begin replicating log entry for final configuration
Joint Consensus, cont’d

- **Additional details:**
  - Any server from either configuration can serve as leader
  - If current leader is not in $C_{\text{new}}$, must step down once $C_{\text{new}}$ is committed.

![Diagram showing the transition between old and new configurations]

- $C_{\text{old}}$ can make unilateral decisions
- $C_{\text{old} + \text{new}}$ can make unilateral decisions
- $C_{\text{new}}$ can make unilateral decisions
- $C_{\text{old}}$ steps down here
- Leader not in $C_{\text{new}}$ steps down here

48
Raft Summary

1. Basic leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. Basic log replication (Normal operation)
3. Revised leader election and log replication for safety and consistency
4. Neutralizing old leaders
5. Client interactions
   - Implementing linearizeable semantics
6. Configuration changes:
   - Adding and removing servers