Raft: A Consensus Algorithm for Replicated Logs

Diego Ongaro and John Ousterhout
Stanford University

- Replicated log => replicated state machine
  - All servers execute same commands in same order

- Consensus module ensures proper log replication

- System makes progress as long as any majority of servers are up

- Failure model: fail-stop (not Byzantine), delayed/lost messages
Two general approaches to consensus:

- **Symmetric, leader-less:**
  - All servers have equal roles
  - Clients can contact any server

- **Asymmetric, leader-based:**
  - At any given time, one server is in charge, others accept its decisions
  - Clients communicate with the leader

- **Raft uses a leader:**
  - Decomposes the problem (normal operation, leader changes)
  - Simplifies normal operation (no conflicts)
  - More efficient than leader-less approaches

Raft Overview

1. **Leader election:**
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader

2. **Normal operation (basic log replication)**

3. **Safety and consistency after leader changes**

4. **Neutralizing old leaders**

5. **Client interactions**
   - Implementing linearizeable semantics

6. **Configuration changes:**
   - Adding and removing servers
Server States

- **At any given time, each server is either:**
  - **Leader:** handles all client interactions, log replication
    - At most 1 viable leader at a time
  - **Follower:** completely passive (issues no RPCs, responds to incoming RPCs)
  - **Candidate:** used to elect a new leader

- **Normal operation: 1 leader, N-1 followers**

  ![Server State Diagram]

- **Time divided into terms:**
  - Election
  - Normal operation under a single leader

- **At most 1 leader per term**
- **Some terms have no leader (failed election)**
- **Each server maintains current term value**
- **Key role of terms: identify obsolete information**
Raft Protocol Summary

**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
- 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 currentTerm, start new election
  - Discover higher term, step down

**Candidates**
- Initialize all candidates 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, discard 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

**Leaders**
- 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
  - candidate that received vote in current term (or null if none)
  - log
  - log entries

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

**RequestVote RPC**
- Invoked by candidates to gather votes.
- Arguments:
  - candidateId
  - term
  - log
  - log entries
- Results:
  - term
  - voteGranted
- Implementation:
  1. If term < currentTerm, return failure
  2. If term == currentTerm, voteGrant is null or candidateId, and candidate's log is at least as complete as local log

**AppendEntries RPC**
- Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat.
- Arguments:
  - term
  - leaderId
  - prevLogIndex
  - prevLogTerm
  - entries
  - committed
- Results:
  - term
  - success
- Implementation:
  1. Return if term < currentTerm
  2. If term == currentTerm, return success
  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 new entries not already in the log
  8. Advance state machine with newly committed entries

Heartbeats and Timeouts

- Servers start up as followers
- Followers expect to receive RPCs from leaders or candidates
- Leaders must send **heartbeats** (empty AppendEntries RPCs) to maintain authority
- If **electionTimeout** elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500ms
Election Basics

- Increment current term
- Change to Candidate state
- Vote for self
- Send RequestVote RPCs 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, cont’d

- **Safety**: allow at most one winner per term
  - Each server gives out only one vote per term (persist on disk)
  - Two different candidates can’t accumulate majorities in same term
    - B can’t also get majority
    - Servers
    - Voted for candidate A

- **Liveness**: some candidate must eventually win
  - Choose election timeouts randomly in [T, 2T]
  - One server usually times out and wins election before others wake up
  - Works well if T >> broadcast time
**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, 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, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers pass committed commands to their state machines
- Crashed/slow followers?
  - Leader retries RPCs until they succeed
- Performance is optimal in common case:
  - One successful RPC to any majority of servers
Log Consistency

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
- Implements an induction step, ensures coherency
**Leader Changes**

- **At beginning of new leader’s term:**
  - Old leader may have left entries partially replicated
  - No special steps by new leader: just start normal operation
  - Leader’s log is “the truth”
  - Will eventually make follower’s logs identical to leader’s
  - Multiple crashes can leave many extraneous log entries:

```
<table>
<thead>
<tr>
<th>term</th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
<th>s5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
```

---

**Safety Requirement**

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

- **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

- **This guarantees the safety requirement**
  - Leaders never overwrite entries in their logs
  - Only entries in the leader’s log can be committed
  - 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

- Can’t tell which entries are committed!

  ![Log Entries Diagram]

- During elections, choose candidate with log most likely to contain all committed entries
  - 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 have “most complete” log among electing majority

Committing Entry from Current Term

- Case #1/2: Leader decides entry in current term is committed

  ![Committing Entry Diagram]

- Safe: leader for term 3 must contain entry 4
Committing Entry from Earlier Term

- Case #2/2: Leader is trying to finish committing entry from an earlier term

  1 2 3 4 5 6
  s_1 1 1 2 4  
  s_2 1 1 2  
  s_3 1 1 2  
  s_4 1 1  
  s_5 1 1 3 3 3

  Leader for term 4
  AppendEntries just succeeded

- Entry 3 *not safely committed*:
  - s_5 can be elected as leader for term 5
  - If elected, it will overwrite entry 3 on s_1, s_2, and s_3!

New 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 committed:
  - s_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe

Combination of election rules and commitment rules makes Raft safe
Log Inconsistencies

Leader changes can result in log inconsistencies:

<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>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>
<td>6</td>
</tr>
</tbody>
</table>

(a) | 1 | 1 | 1 | 4 | 4 | 5 | 5 | 6 | 6 | 6 | 6 |
(b) | 1 | 1 | 1 | 4 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
(c) | 1 | 1 | 1 | 4 | 4 | 5 | 5 | 6 | 6 | 6 | 6 | 6 |
(d) | 1 | 1 | 1 | 4 | 4 | 5 | 5 | 6 | 6 | 6 | 7 | 7 |
(e) | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
(f) | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |

Possible followers:
- Missing Entries
- Extraneous Entries

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)
- When AppendEntries consistency check fails, decrement nextIndex and try again:
Repairing Logs, cont’d

- When follower overwrites inconsistent entry, it deletes all subsequent entries:

```
log index 1 2 3 4 5 6 7 8 9 10 11
leader for term 7 1 1 1 4 4 5 5 6 6 6
follower (before) 1 1 1 2 2 2 3 3 3 3 3
follower (after) 1 1 1 4
```

- 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
Client Protocol

- Send commands to leader
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirect to leader

- Leader does not respond until command has been logged, committed, and executed by leader’s state machine

- 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

Client Protocol, cont’d

- What if 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
**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

---

**Configuration Changes, cont’d**

Cannot switch directly from one configuration to another: **conflicting majorities** could arise

---

![Diagram showing majority of configurations](image-url)
Joint Consensus

- 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 $C_{new}$, must step down once $C_{new}$ is committed.
**Raft Summary**

1. Leader election
2. Normal operation
3. Safety and consistency
4. Neutralize old leaders
5. Client protocol
6. Configuration changes

---

**Implementing Replicated Logs with Paxos**

John Ousterhout and Diego Ongaro
Stanford University

Note: this material borrows heavily from slides by Lorenzo Alvisi, Ali Ghodsi, and David Mazières
Goal: Replicated Log

- Replicated log => replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication
- System makes progress as long as any majority of servers are up
- Failure model: fail-stop (not Byzantine), delayed/lost messages

The Paxos Approach

Decompose the problem:

- Basic Paxos (“single decree”):
  - One or more servers propose values
  - System must agree on a single value as chosen
  - Only one value is ever chosen
- Multi-Paxos:
  - Combine several instances of Basic Paxos to agree on a series of values forming the log
Requirements for Basic Paxos

- **Safety:**
  - Only a single value may be chosen
  - A server never learns that a value has been chosen unless it really has been

- **Liveness (as long as majority of servers up and communicating with reasonable timeliness):**
  - Some proposed value is eventually chosen
  - If a value is chosen, servers eventually learn about it

The term “consensus problem” typically refers to this single-value formulation

Paxos Components

- **Proposers:**
  - Active: put forth particular values to be chosen
  - Handle client requests

- **Acceptors:**
  - Passive: respond to messages from proposers
  - Responses represent votes that form consensus
  - Store chosen value, state of the decision process
  - Want to know which value was chosen

For this presentation:
- Each Paxos server contains both components
Strawman: Single Acceptor

- Simple (incorrect) approach: a single acceptor chooses value
- What if acceptor crashes after choosing?
- Solution: quorum
  - Multiple acceptors (3, 5, ...)
  - Value v is chosen if accepted by majority of acceptors
  - If one acceptor crashes, chosen value still available

Problem: Split Votes

- Acceptor accepts only first value it receives?
- If simultaneous proposals, no value might be chosen

Acceptors must sometimes accept multiple (different) values
Problem: Conflicting Choices

- Acceptor accepts every value it receives?
- Could choose multiple values

Once a value has been chosen, future proposals must propose/choose that same value (2-phase protocol)

Conflicting Choices, cont’d

- $s_5$ needn’t propose red (it hasn’t been chosen yet)
- $s_1$’s proposal must be aborted ($s_3$ must reject it)

Must order proposals, reject old ones
Proposal Numbers

- Each proposal has a unique number
  - Higher numbers take priority over lower numbers
  - It must be possible for a proposer to choose a new proposal number higher than anything it has seen/used before

- One simple approach:

<table>
<thead>
<tr>
<th>Proposal Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round Number</td>
</tr>
<tr>
<td>Server Id</td>
</tr>
</tbody>
</table>

  - Each server stores maxRound: the largest Round Number it has seen so far
  - To generate a new proposal number:
    - Increment maxRound
    - Concatenate with Server Id
  - Proposers must persist maxRound on disk: must not reuse proposal numbers after crash/restart

Basic Paxos

Two-phase approach:

- **Phase 1:** broadcast Prepare RPCs
  - Find out about any chosen values
  - Block older proposals that have not yet completed

- **Phase 2:** broadcast Accept RPCs
  - Ask acceptors to accept a specific value
Basic Paxos

**Proposers**
1) Choose new proposal number \( n \)
2) Broadcast \( \text{Prepare}(n) \) to all servers

**Acceptors**
3) Respond to \( \text{Prepare}(n) \):
   - If \( n > \text{minProposal} \) then \( \text{minProposal} = n \)
   - Return(\( \text{acceptedProposal} \), \( \text{acceptedValue} \))
4) When responses received from majority:
   - If any \( \text{acceptedValues} \) returned, replace value with \( \text{acceptedValue} \) for highest \( \text{acceptedProposal} \)
5) Broadcast \( \text{Accept}(n, \text{value}) \) to all servers
6) When responses received from majority:
   - Any rejections (result > \( n \))? goto (1)
   - Otherwise, \text{value is chosen} \\

Acceptors must record \( \text{minProposal} \), \( \text{acceptedProposal} \), and \( \text{acceptedValue} \) on stable storage (disk)

### Basic Paxos Examples

Three possibilities when later proposal prepares:

1. **Previous value already chosen:**
   - New proposer will find it and use it

   ![Diagram showing time and values]

   - \( X \) \( S_1 \) \( P \ 3.1 \) \( A \ 3.1 \ X \)
   - \( S_2 \) \( P \ 3.1 \) \( A \ 3.1 \ X \)
   - \( S_3 \) \( P \ 3.1 \) \( A \ 3.1 \ X \) \( P \ 4.5 \) \( A \ 4.5 \ X \)
   - \( S_4 \) \( P \ 4.5 \) \( A \ 4.5 \ X \)
   - \( S_5 \) \( P \ 4.5 \) \( A \ 4.5 \ X \)

   "Prepare proposal 3.1 (from \( s_i \))"
   "Accept proposal 4.5 with value \( X \) (from \( s_5 \))"
Basic Paxos Examples, cont’d

Three possibilities when later proposal prepares:

2. Previous value not chosen, but new proposer sees it:
   - New proposer will use existing value
   - Both proposers can succeed

3. Previous value not chosen, new proposer doesn’t see it:
   - New proposer chooses its own value
   - Older proposal blocked
Liveness

- Competing proposers can livelock:

  - One solution: randomized delay before restarting
    - Give other proposers a chance to finish choosing
  - Multi-Paxos will use leader election instead

Other Notes

- Only proposer knows which value has been chosen
- If other servers want to know, must execute Paxos with their own proposal
Multi-Paxos

- Separate instance of Basic Paxos for each entry in the log:
  - Add `index` argument to Prepare and Accept (selects entry in log)

1. Client sends command to server
2. Server uses Paxos to choose command as value for a log entry
3. Server waits for previous log entries to be applied, then applies new command to state machine
4. Server returns result from state machine to client

Multi-Paxos Issues

- Which log entry to use for a given client request?
- Performance optimizations:
  - Use leader to reduce proposer conflicts
  - Eliminate most Prepare requests
- Ensuring full replication
- Client protocol
- Configuration changes

Note: Multi-Paxos not specified precisely in literature
Selecting Log Entries

- When request arrives from client:
  - Find first log entry not known to be chosen
  - Run Basic Paxos to propose client’s command for this index
  - Prepare returns acceptedValue?
    - Yes: finish choosing acceptedValue, start again
    - No: choose client’s command

Selecting Log Entries, cont’d

- Servers can handle multiple client requests concurrently:
  - Select different log entries for each
- Must apply commands to state machine in log order
Improving Efficiency

- Using Basic Paxos is inefficient:
  - With multiple concurrent proposers, *conflicts* and restarts are likely (higher load → more conflicts)
  - 2 rounds of RPCs for each value chosen (Prepare, Accept)

Solution:

1. **Pick a leader**
   - At any given time, only one server acts as Proposer
2. **Eliminate most Prepare RPCs**
   - Prepare once for the entire log (not once per entry)
   - Most log entries can be chosen in a single round of RPCs

---

Leader Election

One simple approach from Lamport:

- Let the server with highest ID act as leader
- Each server sends a heartbeat message to every other server every T ms
- If a server hasn’t received heartbeat from server with higher ID in last 2T ms, it acts as leader:
  - Accepts requests from clients
  - Acts as proposer and acceptor
- If server not leader:
  - Rejects client requests (redirect to leader)
  - Acts only as acceptor
Eliminating Prepares

- Why is Prepare needed?
  - Block old proposals
    - Make proposal numbers refer to the entire log, not just one entry
  - Find out about (possibly) chosen values
    - Return highest proposal accepted for current entry
    - Also return noMoreAccepted: no proposals accepted for any log entry beyond current one
- If acceptor responds to Prepare with noMoreAccepted, skip future Prepares with that acceptor (until Accept rejected)
- Once leader receives noMoreAccepted from majority of acceptors, no need for Prepare RPCs
  - Only 1 round of RPCs needed per log entry (Accepts)

Full Disclosure

- So far, information flow is incomplete:
  - Log entries not fully replicated (majority only)
    Goal: full replication
  - Only proposer knows when entry is chosen
    Goal: all servers know about chosen entries
- Solution part 1/4: keep retrying Accept RPCs until all acceptors respond (in background)
  - Fully replicates most entries
- Solution part 2/4: track chosen entries
  - Mark entries that are known to be chosen: acceptedProposal[i] = ∞
  - Each server maintains firstUnchosenIndex: index of earliest log entry not marked as chosen
• **Solution part 3/4:** proposer tells acceptors about chosen entries
  - Proposer includes its firstUnchosenIndex in Accept RPCs.
  - Acceptor marks all entries i chosen if:
    - $i < \text{request.firstUnchosenIndex}$
    - $\text{acceptedProposal}[i] = \text{request.proposal}$
  - Result: acceptors know about *most* chosen 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>
</tr>
</thead>
<tbody>
<tr>
<td>acceptedProposal</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>2.5</td>
<td>$\infty$</td>
<td>3.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... Accept(proposal = 3.4, index=8, value = v, firstUnchosenIndex = 7) ...

<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>
</tr>
</thead>
<tbody>
<tr>
<td>acceptedProposal</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>2.5</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>3.4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Still don’t have complete information

---

• **Solution part 4/4:** entries from old leaders
  - Acceptor returns its firstUnchosenIndex in Accept replies
  - If proposer’s firstUnchosenIndex > firstUnchosenIndex from response, then proposer sends **Success** RPC (in background)

• **Success(index, v):** notifies acceptor of chosen entry:
  - $\text{acceptedValue}[\text{index}] = v$
  - $\text{acceptedProposal}[\text{index}] = \infty$
  - return firstUnchosenIndex
  - Proposer sends additional Success RPCs, if needed
Client Protocol

- Send commands to leader
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirect to leader

- Leader does not respond until command has been chosen for log entry and executed by leader’s state machine

- 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

Client Protocol, cont’d

- What if 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
  - State machine records most recent command executed for each client
  - Before executing command, state machine checks to see if command already executed, if so:
    - Ignore new command
    - Return response from old command

- Result: exactly-once semantics as long as client doesn’t crash
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

Configuration Changes, cont’d

- **Safety requirement:**
  - During configuration changes, it must not be possible for different majorities to choose different values for the same log entry:

  ![Diagram showing configuration changes](attachment:configuration_changes_diagram.png)
**Configuration Changes, cont’d**

- **Paxos solution: use the log to manage configuration changes:**
  - Configuration is stored as a log entry
  - Replicated just like any other log entry
  - Configuration for choosing entry \( i \) determined by entry \( i-\alpha \).

  \[
  \begin{array}{cccccccccc}
  1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
  \hline
  C_0 & & C_1 & C_2 & & & & & & \\
  \end{array}
  \]

  Suppose \( \alpha = 3 \):

- **Notes:**
  - \( \alpha \) limits concurrency: can’t choose entry \( i+\alpha \) until entry \( i \) chosen
  - Issue no-op commands if needed to complete change quickly

---

**Paxos Summary**

- **Basic Paxos:**
  - Prepare phase
  - Accept phase

- **Multi-Paxos:**
  - Choosing log entries
  - Leader election
  - Eliminating most Prepare requests
  - Full information propagation

- **Client protocol**
- **Configuration changes**