High-Level Programming for Programmable Networks: Consistent, Replicated Data Store using Raft
Consistent Network Updates

Yang (Richard) Yang
Computer Science Department
Yale University
208A Watson
Email: yry@cs.yale.edu

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

Acknowledgements: RAFT slides are based on RAFT usability slides linked on the Schedule page.
Outline

- Admin and recap
- Controller software framework (network OS) supporting programmable networks
  - architecture
  - data model and operations: OpenDaylight as an example
  - distributed data store
    - overview
    - basic Paxos
    - multi-Paxos
    - Raft
  - south-bound: consistent network updates
Admin

- PS1 to be posted by Thursday
  - A total of 3 assignments

- Please start to make team and talk to me on potential projects
Recap: Replicated Log for Reliable Data Store

- Replicated operations log $\Rightarrow$ replicated state machine
  - All servers execute same commands in same order
- System makes progress as long as any majority of servers are up
## Recap: Basic Paxos Protocol

<table>
<thead>
<tr>
<th>Proposers</th>
<th>Acceptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Choose new proposal number n</td>
<td>3) Respond to Prepare(n):</td>
</tr>
<tr>
<td>2) Broadcast Prepare(n) to all servers</td>
<td>o If n &gt; minProposal then minProposal = n</td>
</tr>
<tr>
<td></td>
<td>o Return(acceptedProposal, acceptedValue)</td>
</tr>
<tr>
<td>4) When responses received from majority:</td>
<td>6) Respond to Accept(n, value):</td>
</tr>
<tr>
<td>o If any acceptedValues returned, replace value with acceptedValue for highest acceptedProposal</td>
<td>o If n ≥ minProposal then acceptedProposal = minProposal = n</td>
</tr>
<tr>
<td></td>
<td>acceptedValue = value</td>
</tr>
<tr>
<td>5) Broadcast Accept(n, value) to all servers</td>
<td>o Return(minProposal)</td>
</tr>
<tr>
<td>6) When responses received from majority:</td>
<td>o Any rejections (result &gt; n)? goto (1)</td>
</tr>
<tr>
<td>o Otherwise, value is chosen</td>
<td>o Otherwise, value is chosen</td>
</tr>
</tbody>
</table>

Acceptors must record minProposal, acceptedProposal, and acceptedValue on stable storage (disk)
Recap: 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
Recap: MultiPaxos Entry Selection

- 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
Recap: Improving and Extending Multi-Paxos

- Improving efficiency
  - With multiple concurrent proposers, **conflicts** and restarts are likely (higher load → more conflicts)
  - 2 rounds of RPCs for each value chosen (Prepare, Accept)

- Handling other case, e.g., configuration change
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
Safety requirement:

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

Choose \( v_1 \) using old configuration

Choose \( v_2 \) using new configuration

Old Configuration

New Configuration
Paxos solution: use the log to manage configuration changes:

- Configuration is stored as a log entry
- Replicated just like any other log entry
- New configuration takes effect after $\alpha$ entries.

Suppose $\alpha = 3$:

```
1  2  3  4  5  6  7  8  9  10
C1  C2
```

Use $C_0$, Use $C_1$, Use $C_2$

Note:

- $\alpha$ limits concurrency: cannot move beyond $\alpha$
- Issue no-op commands if needed to complete change quickly
Outline

- Admin and recap
- Network OS supporting programmable networks
  - overview
  - data store
  - distributed network OS
    - Overview
    - Paxos
    - Raft
Raft Overview

- Designed in 2013
- Basic design ideas
  - leader-less => leader based
  - try to decompose the problem (normal operation, leader changes)
## Raft Usage

<table>
<thead>
<tr>
<th>Name</th>
<th>Primary Authors</th>
<th>Language</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>RethinkDB/clustering</td>
<td>Blake Mizerany, Xiang Li and Yicheng Qin</td>
<td>C++</td>
<td>AGPL</td>
</tr>
<tr>
<td>etcd/raft</td>
<td>Ben Johnson (Sky) and Xiang Li (CMU, CoreOS)</td>
<td>Go</td>
<td>Apache 2.0</td>
</tr>
<tr>
<td>LogCabin</td>
<td>Diego Ongaro (Stanford)</td>
<td>C++</td>
<td>ISC</td>
</tr>
<tr>
<td>go-raft</td>
<td>Armon Dadgar (hashicorp)</td>
<td>Go</td>
<td>MIT</td>
</tr>
<tr>
<td>hoverbear/raft</td>
<td>Andrew Hobden, Dan Burkert</td>
<td>Rust</td>
<td>MIT</td>
</tr>
<tr>
<td>ckit</td>
<td>Pablo Medina</td>
<td>Scala</td>
<td>Apache2</td>
</tr>
<tr>
<td>verd/raft</td>
<td>James Wilcox, Doug Woos, Pavel Panchekha, Zach Tatlock, Xi Wang, Mike Ernst, and Tom Anderson (University of Washington)</td>
<td>Coq</td>
<td>BSD</td>
</tr>
<tr>
<td>OpenDaylight</td>
<td>Moiz Raja, Kamal Rameshan, Robert Varga (Cisco), Tom Pantelis (Brocade)</td>
<td>Java</td>
<td>Eclipse</td>
</tr>
<tr>
<td>zraft/lib</td>
<td>Gunin Alexander</td>
<td>Erlang</td>
<td>Apache2</td>
</tr>
<tr>
<td>kanaka/raft.js</td>
<td>Joel Martin</td>
<td>Javascript</td>
<td>MPL-2.0</td>
</tr>
<tr>
<td>akka-raft</td>
<td>Konrad Malawski</td>
<td>Scala</td>
<td>Apache2</td>
</tr>
<tr>
<td>rafter</td>
<td>Andrew Stone (Basho)</td>
<td>Erlang</td>
<td>Apache2</td>
</tr>
<tr>
<td>floss</td>
<td>Alexander Flatter</td>
<td>Ruby</td>
<td>MIT</td>
</tr>
<tr>
<td>willem/raft</td>
<td>Willem Hendrik Thiart</td>
<td>C</td>
<td>BSD</td>
</tr>
</tbody>
</table>

...
### 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.

#### 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 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

#### RequestVote RPC
- Invoked by candidates to gather votes.
- **Arguments:**
  - **candidateId**
  - **term**
  - **lastLogIndex**
  - **lastLogTerm**
- **Results:**
  - **term**
  - **voteGranted**
- **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**
  - **leaderId**
  - **prevLogIndex**
  - **prevLogTerm**
  - **entries[]**
  - **commitIndex**
- **Results:**
  - **term**
  - **success**
- **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

#### Persistent State
Each server persists the following to stable storage synchronously before responding to RPCs:
- **currentTerm**
- **votedFor**
- **log[]**

#### Log Entry
- **term**
- **index**
- **command**
Raft Components

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

- Admin and recap
- Network OS supporting programmable networks
  - overview
  - data store
  - distributed data store
    - overview
    - paxos
    - raft
      - basic operation: leader based operation
Basic Leader Based Protocol

- Client sends commands to leader

- Leader responds
  - Leader commits and executes command
  - Leader sends results back to client

Discussion: what can go wrong?
Basic Election

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

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

- **Follower**
  - start
  - timeout, start election
  - discover current server or higher term

- **Candidate**
  - timeout, new election
  - receive enough votes

- **Leader**
  - "step down"
Safety: allow at most one winner per election

- How:
  - Each election is identified by a number (term)
  - Each server gives out only one vote per term (persist on disk) and winner must get majority vote.

B can’t also get majority

Voted for candidate A

Servers

Liveness: some candidate must eventually win

- 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 (Normal) Log Operation (BLO1)

- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries to followers
- Leader executes command to its state machine, returns result to client
Basic (Normal) Log Operation (BLO1)

Log entry = index, term, command

Log stored on stable storage (disk); survives crashes

Discussion: what can go wrong w/ BLO1?
BLO1: Revise

- **Problem:**
  - Leader can crash right after execution

- **Solution**
  - Execute and return results only after **commitment** (availability if majority of servers survive)

**Discussion:** what is the simplest commitment rule?
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:

Old leader's 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>1</td>
<td>1</td>
<td>1</td>
<td>4</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>
</tbody>
</table>

Possible followers:

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

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
**Leader Commit**

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”:
    \[(\text{lastTerm}_V > \text{lastTerm}_C)\]
    \[\text{||}\]
    \[(\text{lastTerm}_V == \text{lastTerm}_C)\]
    \[\text{&&} (\text{lastIndex}_V > \text{lastIndex}_C)\]
  - 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

Leader for term 4
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.

<table>
<thead>
<tr>
<th>Committed</th>
<th>Present in future leaders’ logs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditions on commitment</td>
<td>Conditions on leader election</td>
</tr>
</tbody>
</table>
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
Leader changes can result in log inconsistencies:

Possible followers:

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

log index
leader for term 8

1 2 3 4 5 6 7 8 9 10 11 12

1 1 1 4 4 5 5 6 6 6 6

Implication: New leader must make follower logs consistent with its own

- Delete extraneous entries
- Fill in missing entries
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

![Diagram of AppendEntries Consistency Check]

**AppendEntries succeeds:**
- Matching entry

**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:**

<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 7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>followers</td>
<td>(a)</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>
<tr>
<td></td>
<td>(b)</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>
</tbody>
</table>

(nextIndex)
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>

nextIndex
Getting-it-together: 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
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
Raft Summary

1. Basic leader election:
   o Select one of the servers to act as leader
   o 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
   o Implementing linearizeable semantics
6. Configuration changes:
   o Adding and removing servers
Outline

- Admin and recap
- **Controller software framework (network OS) supporting programmable networks**
  - architecture
  - data model and operations: OpenDaylight as an example
  - distributed data store: consistent data store
  - south-bound: consistent network updates
Logical Data Store

Recap: Big Picture

Key goal: provide applications with high-level views and make the views highly available (e.g., 99.99%), scalable.

Key component - data store:
- Data model
- Data operation model
- Data store availability

logically centralized data store

Program

Network View

Service/Policy

NE Datapath

NE Datapath
Example

- Workflow when a link weight is changed
Example Data Store Transaction: Changing a Flow’s Route

- Link weight change => path for flow \( a \rightarrow d \) change:
  - from \( a \rightarrow b \rightarrow c \rightarrow d \)
  - to \( a \rightarrow e \rightarrow f \rightarrow d \)

- Discussion: what are the operations (ops) at local devices?
- Discussion: what order should we carry out the ops, why?
Example Data Store Transaction: Updating a Set of Flows

- Assume each link has capacity 10
- \( F_i: b \): means that flow \( i \) needs \( b \) amount of bw

Discussion: what is a bad sequence to carry out the ops in the southbound transaction?

Example from Dionysus
Prepare for Next Class