HY590.45
Modern Topics in
Scalable Storage Systems

Kostas Magoutis
magoutis@csd.uoc.gr
http://www.csd.uoc.gr/~hy590-45
Raft

- Consensus algorithm for log replication
- Easier to understand compared to Multi-Paxos
Replicated state machine architecture

Raft
Server states

Raft uses the voting process to prevent a candidate from winning an election unless its log contains all committed entries. A candidate must contact a majority of the cluster in order to be elected, which means that every committed entry must be present in at least one of those servers. If the candidate’s log is at least as up-to-date as any other log in that majority (where “up-to-date” is defined precisely below), then it will hold all the committed entries.

Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs. If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.
Terms (epochs)

Raft
Log entries

Raft
Possible states of followers

<table>
<thead>
<tr>
<th></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>(a)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>(b)</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>
<td></td>
</tr>
<tr>
<td>(c)</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>
<tr>
<td>(d)</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>7</td>
</tr>
<tr>
<td>(e)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(f)</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>
<td>3</td>
</tr>
</tbody>
</table>

log index
leader for term 8

Raft
When is an entry committed?

Leader S1, term 2

Leader S1, term 4

Leader S5, term 3

Leader S5, term 5

Leader S1, term 4
Properties

**Election Safety:** at most one leader can be elected in a given term. §5.2

**Leader Append-Only:** a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

**Log Matching:** if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

**Leader Completeness:** if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

**State Machine Safety:** if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3
Reconfiguration

Server 1
Server 2
Server 3
Server 4
Server 5

problem: two disjoint majorities

Raft
Joint consensus
Log compaction - snapshots

1 2 3 4 5 6 7
1 x⇐3 1 y⇐1 1 y⇐9 2 x⇐2 3 x⇐0 3 y⇐7 3 x⇐5

log index
before

3 y⇐7 3 x⇐5

snapshot
last included index: 5
last included term: 3
state machine state:
x ⇐ 0
y ⇐ 9

committed entries

after

Raft
Time to detect and replace crashed leader

Timing requirement

\[
\text{broadcastTime} \ll \text{electionTimeout} \ll \text{MTBF}
\]

Raft
# Server behavior

## State

**Persistent state on all servers:**
(Updated on stable storage before responding to RPCs)

- **currentTerm**
  - latest term server has seen (initialized to 0 on first boot, increases monotonically)
- **votedFor**
  - candidateID that received vote in current term (or null if none)
- **log[]**
  - log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

**Volatile state on all servers:**

- **commitIndex**
  - index of highest log entry known to be committed (initialized to 0, increases monotonically)
- **lastApplied**
  - index of highest log entry applied to state machine (initialized to 0, increases monotonically)

**Volatile state on leaders:**
(Reinitialized after election)

- **nextIndex[]**
  - for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
- **matchIndex[]**
  - for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

---

Raft
E lecting a leader

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

**Arguments:**

- **term**: candidate’s term
- **candidateId**: candidate requesting vote
- **lastLogIndex**: index of candidate’s last log entry (§5.4)
- **lastLogTerm**: term of candidate’s last log entry (§5.4)

**Results:**

- **term**: currentTerm, for candidate to update itself
- **voteGranted**: true means candidate received vote

**Receiver implementation:**

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate’s log is at least as up-to-date as receiver’s log, grant vote (§5.2, §5.4)

Raft
# Servers

## Rules for Servers

### All Servers:
- If `commitIndex > lastApplied`: increment `lastApplied`, apply `log[lastApplied]` to state machine (§5.3)
- If RPC request or response contains `term T > currentTerm`:
  - set `currentTerm = T`, convert to follower (§5.1)

### Followers (§5.2):
- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving `AppendEntries` RPC from current leader or granting vote to candidate:
  - convert to candidate

### Candidates (§5.2):
- On conversion to candidate, start election:
  - Increment `currentTerm`
  - Vote for self
  - Reset election timer
  - Send `RequestVote` RPCs to all other servers
  - If votes received from majority of servers: become leader
  - If `AppendEntries` RPC received from new leader: convert to follower
  - If election timeout elapses: start new election

### Leaders:
- Upon election: send initial empty `AppendEntries` RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If `last log index ≥ nextIndex` for a follower: send `AppendEntries` RPC with log entries starting at `nextIndex`
  - If successful: update `nextIndex` and `matchIndex` for follower (§5.3)
  - If `AppendEntries` fails because of log inconsistency:
    - decrement `nextIndex` and retry (§5.3)
  - If there exists an `N` such that `N > commitIndex`, a majority of `matchIndex[i] ≥ N`, and `log[N].term == currentTerm`:
    - set `commitIndex = N` (§5.3, §5.4).
Replicate log entries and heartbeat

---

### AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

**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; may send more than one for efficiency)
- **leaderCommit:** leader’s commitIndex

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

**Receiver implementation:**
1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn’t contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

---

Raft