CSE 124: REPLICATED STATE MACHINES

George Porter
November 8 and 10, 2017
ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license

• These slides incorporate material from:
  • Tanenbaum and Van Steen, Dist. Systems: Principles and Paradigms
  • Michael Freedman, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
  • Diego Ongaro and John Ousterhout, RAFT project (https://raft.github.io/)
ANNOUNCEMENTS

HW 4 out

Project 2 out
Outline

1. Replicated state machine overview
2. Leaders
3. Normal operation
4. Client interactions
RECALL: PRIMARY-BACKUP

• **Mechanism**: Replicate and separate servers

• **Goal #1**: Provide a highly reliable service

• **Goal #2**: Servers should behave just like a single, more reliable server
STATE MACHINE REPLICATION

- Any server is essentially a *state machine*
- Operations **transition** between states

- Need an op to be executed on all replicas, or none at all
  - *i.e.*, we need **distributed** all-or-nothing atomicity
- If op is deterministic, replicas will end in same state
Expect success as replicas are all identical (unlike distributedtxn)

1. C $\rightarrow$ P: “request <op>”
2. P $\rightarrow$ A, B: “prepare <op>”
3. A, B $\rightarrow$ P: “prepared” or “error”
4. P $\rightarrow$ C: “result exec<op>” or “failed”
5. P $\rightarrow$ A, B: “commit <op>”

“Okay” (i.e., op is stable) if written to > ½ backups
• Basic fault-tolerant Replicated State Machine (RSM) approach

1. Consensus protocol to elect leader

2. 2PC to replicate operations from leader

3. All replicas execute ops once committed
Outline

1. Replicated state machine overview
2. Leaders
3. Normal operation
4. Client interactions
WHY BOTHER WITH A LEADER?

Not necessary, but ...

- Decomposition: normal operation vs. leader changes
- Simplifies normal operation (no conflicts)
- More efficient than leader-less approaches
- Obvious place to handle non-determinism

Image courtesy of Reuters
GOAL: REPLICATED LOG

- Replicated log => replicated state machine
- All servers execute same commands in same order
- Consensus module ensures proper log replication
GENERAL SOLUTION

1. Leader election
2. Normal operation (basic log replication)
3. Safety and consistency after leader changes
4. Neutralizing old leaders
5. Client interactions
6. Reconfiguration
CONCEPTS COVERED IN CSE 124

1. Leader election
2. Normal operation (basic log replication)
3. Safety and consistency after leader changes
4. Neutralizing old leaders
5. Client interactions
6. Reconfiguration
At any given time, each server is either:

- **Leader**: handles all client interactions, log replication
- **Follower**: completely passive, answers to leader

**Normal operation**: 1 leader, N-1 followers
LEADER TERMS (AKA EPOCHS)

- Time divided into terms
  - Election (either failed or resulted in 1 leader)
  - Normal operation under a single leader
- Each server maintains current term value
- Key role of terms: identify obsolete information

We will always be in “Term 1”!
LEADER ELECTION

- **Safety**: allow at most one winner per term
- **Liveness**: some candidate must eventually win

*Image courtesy Laurel L. Russwurm; used with permission*
Outline

1. Replicated state machine overview
2. Leaders
3. Normal operation
4. Client interactions
**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 / stable, 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, sends result to client
  - Leader piggybacks commitment to followers in later AppendEntries
  - 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
getting followers up-to-date

appendEntries RPC at position i:

- Success if follower adds one entry to its log at i
- Fails if the follower would have a “gap” in the index
  - Leader tries again, at position i-1
  - Then again at i-2, etc., until the call succeeds
  - This is an *induction step* to ensure that all followers eventually catch up

AppendEntries RPC call fills in “missing” entries
SAFETY PROPERTY

• If leader decides that an index is committed
  • Then must be committed in any follower that takes
    over for the leader in case the leader fails
    • (Won’t actually exercise this outcome in our project)

• The leader includes the highest index that it has
  committed in its appendEntries RPC call
Outline

1. Replicated state machine overview
2. Leaders
3. Normal operation
4. Client interactions
• Send commands to leader

  • If leader unknown, contact servers in order, inquiring if they are the leader or not

• Leader only responds after command logged, committed, and executed by leader

• If request times out (e.g., leader crashes):
  • Client reissues command to new leader (after possible redirect)
CLIENT INTERACTIONS (2/2)

- Ensure **exactly-once** semantics even with leader failures
  - E.g., Leader can execute command then crash before responding
  - Client should embed unique ID in each command
  - This client ID included in log entry
  - Before accepting request, leader checks log for entry with same id
- We’ll use **per-object (file) versions** to ensure exactly-once semantics
UC San Diego