CS434/534: Topics in Networked (Networking) Systems

Distributed Network OS (cont.);
From Data to Function Store

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

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

- Admin and recap
- High-level datapath programming
  - blackbox (trace-tree)
  - whitebox
- Network OS supporting programmable networks
  - overview
  - OpenDaylight
  - distributed network OS (Paxos, RAFT)
  - from data store to function store
Admin

- PS1 small bug to be fixed
Recap: A Common Approach of Designing Distributed Data Store: Replicated Log

- Replicated log => replicated state machine
  - All servers execute same commands in same order
  - If each data item has a version number, then (data-item, version) will be the same if you read from any server
- Consensus module ensures proper log replication
- System makes progress as long as any majority of servers are up
Two general approaches to consensus:

- **Symmetric, leader-less:**
  - All servers have equal roles
  - Clients can contact any server
  - Basic Paxos is an example

- **Asymmetric, leader-based:**
  - At any given time, one server is in charge, others accept its decisions
  - Clients communicate with the leader
  - Raft is an example
## Recap: Basic Paxos

### Proposers

1. Choose new proposal number \( n \)

2. Broadcast \( \text{Prepare}(n) \) to all servers

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, value is chosen

### Acceptors

3. Respond to \( \text{Prepare}(n) \):
   - If \( n > \text{minProposal} \) then \( \text{minProposal} = n \)
   - Return(\( \text{acceptedProposal}, \text{acceptedValue} \))

6. Respond to \( \text{Accept}(n, \text{value}) \):
   - If \( n \geq \text{minProposal} \) then
     - \( \text{acceptedProposal} = \text{minProposal} = n \)
     - \( \text{acceptedValue} = \text{value} \)
   - Return(\( \text{minProposal} \))

Acceptors must record \( \text{minProposal}, \text{acceptedProposal} \), and \( \text{acceptedValue} \) on stable storage (disk)
Recap: Raft Big Picture

- Time divided into terms:
  - Election
  - Operation under a single leader

- Basic leader election
  - Heartbeats to detect leader failure/disconnection
  - Need to get majority of votes to become new leader
Recap: Basic Log Operation

- **Command**
  - 1 add
  - 1 cmp
  - 1 ret
  - 2 mov
  - 3 jmp
  - 3 div
  - 3 shl
  - 3 sub

- **Log Index**
  -_leader_

- **Leaders**
  - 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

- **Followers**
  - committed entries
Complexity: Leader Change

- At beginning of new leader’s term:
  - Old leader may have left entries partially replicated

Safety:
- Once a log entry has been applied (committed) to a state machine, no other state machine can apply a different value for that log entry.

Liveness and design decision:
- Leader uses its own log to proceed (no complex leader log fix for a new leader)
- Eventually followers’ log becomes identical to the leader’s, for live leader.
**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”:
    \[(\text{lastTerm}_V > \text{lastTerm}_C) \lor (\text{lastTerm}_V == \text{lastTerm}_C) \land (\text{lastIndex}_V > \text{lastIndex}_C)\]
  - Leader will likely have “most complete” log among electing majority
Leader Election: Example

Since any new leader will have \( \text{Log}[4] \), it is OK to commit it.
Q: 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.
Leader Commit: Continue

- 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: Leader Election + Commit

- Use a combination of election rules and commitment rules to achieve safety

- Committed
- Present in future leaders’ logs

Conditions on commitment

Conditions on leader election
**Follower's 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>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)**
  - $1 \ 1 \ 1 \ 4 \ 4 \ 5 \ 5 \ 6 \ 6$

- **(b)**
  - $1 \ 1 \ 1 \ 4$

- **(c)**
  - $1 \ 1 \ 1 \ 4 \ 4 \ 5 \ 5 \ 6 \ 6 \ 6 \ 6 \ 6$

- **(d)**
  - $1 \ 1 \ 1 \ 4 \ 4 \ 5 \ 5 \ 6 \ 6 \ 6 \ 7 \ 7$

- **(e)**
  - $1 \ 1 \ 1 \ 4 \ 4 \ 4 \ 4$

- **(f)**
  - $1 \ 1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 3 \ 3 \ 3 \ 3 \ 3$

**Leader changes can result in log inconsistencies:**

- **Missing Entries**
  - (b), (c), (d), (e), (f)

- **Extraneous Entries**
  - (d), (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.

**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 + leader’s last index)`
- When `AppendEntries` consistency check fails, decrement `nextIndex` and try again:

![Diagram showing logs and leader/follower consistency]

- `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
- `followers`:
  - (a) 1 1 1 4
  - (b) 1 1 1 2 2 2 3 3 3 3 3 3
- `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>
Client Protocol

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

- Leader response
  - 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
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
Next 4 Slides Not Covered in Class
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.

- $C_{\text{old}}$
- $C_{\text{new}}$

Server 1

Server 2

Server 3

Server 4

Server 5

**Majority of $C_{\text{old}}$**

**Majority of $C_{\text{new}}$**
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

- $C_{old}$ can make unilateral decisions
- $C_{new}$ can make unilateral decisions

- Timeline:
  - $C_{old}$ entry committed
  - $C_{old+new}$ entry committed
  - $C_{new}$ entry committed
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.
End of Slides Not Covered in Class
**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
ONOS Architecture: Big Picture

Northbound - Application Intent Framework
(policy enforcement, conflict resolution)

Distributed Core
 scalability, availability, performance, persistence

Southbound
(discover, observe, program, configure)

OpenFlow
(pluggable, extensible)

Intent-based L2 forwarding
Intent calendaring
SDN-IP peering

Source: ON.LAB
ONOS Architecture: Some Key Abstractions

https://wiki.onosproject.org/display/ONOS/System+Components
Discussions

- Programming tasks of a data store based programming model
Data Store Programming Complexity

- Issue 1: Programmers need to manually record data access and register subscriptions (what data to monitor?)
Issue 2: Programmers need to handle cleanup of any changes made in previous execution.
Data Store Programming Complexity

Issue 3: Programmers need to make sure that no data-change loops are triggered, to avoid instability

A:  
if (remBw > 10)  
remBw -= 10

B:  
if (remBw > 20)  
remBw -= 20
Outline

- Admin and recap
- High-level datapath programming
  - blackbox (trace-tree)
  - whitebox
- Network OS supporting programmable networks
  - overview
  - OpenDaylight
  - distributed network OS (Paxos, RAFT)
  - from data store to function store
From Data Store to Function Store

- Users specify list of functions to execute
- Users specify acceptable execution sequences (Issue 3)
- System automatically detects invalidated executions after data changes, clean up, and re-exec (Issues 1,2)
- System schedules optimal recovery
APIs

- **Function store API**
  - add(func, attr, [req]): returns a handler
  - remove(handler)
  - specify the relationship between two functions via precedence (e.g., f1 -> f2 means f1 exe before f2)
    - A valid execution is one all precedence conditions are satisfied

- **Data access API**
  - read(xpath): wrapper for NOS data store read
  - update(xpath, op, val): wrapper for NOS data store update
  - test(xpath, boolean expression)
Issue to Think About

- Is the function store (λ-programming model) complete (i.e., compared w/ pure data store)?
Pick any topological order of the functions \{f_1, f_2, \ldots, f_n\}, and execute them in order; collect the read data of each function. If any data changes, re-exec all functions.
Assume exec’d f1-f6. Then a changes. Which functions must re-execute? Which do not need re-exec?