PA3 Logistics

Deadline - Today, 5 pm

Submission - A report analyzing the measurements conducted in the experiments as per the Rubric

Put the GitHub link to your code in the report.
Raft

- A consensus algorithm
- Built with the goal to be understandable
- Well defined RPCs

**Figure 1:** Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.
## Raft

<table>
<thead>
<tr>
<th>State</th>
<th>AppendEntries RPC</th>
<th>RequestVote RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Persistent state on all servers:</strong> (Updated on stable storage before responding to RPCs)</td>
<td>Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).</td>
<td>Invoked by candidates to gather votes (§5.2).</td>
</tr>
<tr>
<td>currentTerm</td>
<td>latest term server has seen (initialized to 0 on first boot, increases monotonically)</td>
<td><strong>Arguments:</strong></td>
</tr>
<tr>
<td>votedFor</td>
<td>candidateId that received vote in current term (or null if none)</td>
<td>term</td>
</tr>
<tr>
<td>log[]</td>
<td>log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)</td>
<td>candidateId</td>
</tr>
<tr>
<td><strong>Volatile state on all servers:</strong></td>
<td><strong>prevLogIndex</strong></td>
<td>lastLogIndex</td>
</tr>
<tr>
<td>commitIndex</td>
<td>index of highest log entry known to be committed (initialized to 0, increases monotonically)</td>
<td>lastLogTerm</td>
</tr>
<tr>
<td>lastApplied</td>
<td>index of highest log entry applied to state machine (initialized to 0, increases monotonically)</td>
<td><strong>Results:</strong></td>
</tr>
<tr>
<td><strong>Volatile state on leaders:</strong> (Reinitialized after election)</td>
<td><strong>prevLogTerm</strong></td>
<td><strong>term</strong></td>
</tr>
<tr>
<td>nextIndex[]</td>
<td>for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)</td>
<td><strong>voteGranted</strong></td>
</tr>
<tr>
<td>matchIndex[]</td>
<td>for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)</td>
<td><strong>Receiver implementation:</strong></td>
</tr>
</tbody>
</table>

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 - Log Replication

1. An entry is executed after it is committed.
2. An entry is committed after it has been replicated to a majority of nodes
Log Replication

Example (Raft Thought Question) -

- S1: 3.1 4.1
- S2: 3.1 3.2
- S3: 3.1 5.1
- S4: 3.1 5.1 5.2
- S5: 3.1

Could any replica have already executed the operation in log entry 5.1?

Answer: No, operation 5.1 has not been replicated to a majority of nodes. This implies that it has not been committed and so a replica cannot execute it.
Raft - Log Replication

Log Matching Property:

1. If two entries in different logs have the same index and term, then they store the same command.
2. If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.
Log Matching - Property 1

If two entries in different logs have the same index and term, then they store the same command.

- A leader in any given term only makes one entry per index and log entries never change index
Log Matching - Property 2

If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

- Leader ensures that followers have the same log entries as itself.
- This includes
  a) Checking the previous log entry in a follower’s log before appending a new entry
  b) Asking the follower to duplicate the log entries on the leader in case there is a difference
Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.
Log Replication - Safety

So far,

1. How leader is elected
2. How logs are replicated

To ensure consistency we need two more things -

1. Election Restriction
2. Handling entries from previous terms
Election Restriction

Followers Vote for only that node that has all the committed entries.

Example (Raft Thought Question) -

- S1: 3.1 4.1
- S2: 3.1 3.2
- S3: 3.1 5.1
- S4: 3.1 5.1 5.2
- S5: 3.1

Could operation 3.2 be committed in the future?
Answer

For 3.2 to be committed, S2 has become a leader.

Can S2, become a leader? - No, a candidate will only get votes from replicas if its log is as up to date as the replica. Since a majority of nodes have higher commands with higher term at index 2, S2 cannot get a majority of votes.

So, 3.2 cannot be committed in the future.
Handling entries from previous terms

Entries from previous term cannot be committed unless a new entry in current term is committed by the leader.

Example (Raft Thought Question) -

- S1: 3.1 4.1
- S2: 3.1 3.2
- S3: 3.1 5.1
- S4: 3.1 5.1 5.2
- S5: 3.1

Could operation 4.1 be committed in the future?
Answer

Yes, S1 can become a leader with votes from itself, S2 and S5.

It can then replicate 4.2 to a majority of nodes. But to commit 4.2, S1 will have to commit an entry in the current term.

This could be a <nop> (no operation, basically a command that does nothing) in the current term. Only then can 4.2 be committed.
A classmate points out that read-only operations could be executed much faster if the Raft leader alone executed them and sent the result back to the client, without sending such operations to the followers. After all, read-only operations don’t modify the state, so executing them on the followers is a no-op. Why might this idea lead to incorrect behavior?
A partitioned node can still think that it is the leader. This means that a node could respond back with stale state, which is a problem.

So, a leader node cannot reply to read-only operation without checking that it is still the leader.

1. A newly elected leader has to commit a `<nop>` to check it is still a leader before responding.
2. Or if a node has been a leader for a while, exchange heartbeat with a majority of nodes before responding.
Imagine that a set of 5 nodes perform a leader election, and node S1 is elected leader. Then imagine that leader S1 becomes partitioned from the remaining nodes (it cannot send or receive messages from them). After a sufficiently long period of time, will S1 still consider itself to be the leader, even though it hasn’t heard from the other nodes? If a client submits a request to this “leader”, what will happen?
Answer

Yes, S1 can still think it is the leader.

To simulate this on raftscope, let a node be elected as leader and then drop all packets coming out of and going into that node.

Even if S1 thinks that it is the leader, it will be unable to commit any operations as it cannot communicate with the cluster. A client will try to do an operation and might time out, before trying to contact some other node in the system.
Resources

When unsure about a certain scenario -

https://raft.github.io/raftscope/index.html


http://thesecretlivesofdata.com/raft/
Questions?