Raft Consensus Protocol

Khai Nguyen
khainguyen@temple.edu
Introduction to Distributed Systems and Networks

1. Problem

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members, hence, they play a key role in building reliable large-scale software systems.

2. Current Solutions

Apart from the current Raft Consensus algorithm, there are other available solutions for consensus:

- Paxos Consensus [1]
- Practical Byzantine Fault Tolerance algorithm (PBFT) [2]
- Proof-of-Stake algorithm (PoS) [3]
- Delegated Proof-of-Stake algorithm (DPoS) [4]

Among these, Paxos is commonly used in creating consensus in distributed systems. However, Paxos is quite difficult to understand, its architecture requires complex changes to support practical systems. Therefore, Raft is introduced as an alternative with novel features:

Strong leader: Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log.

***Leader election*:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.

***Membership changes*:** Raft's mechanism for changing the set of servers in the cluster uses a new joint consensus approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

3. Raft Protocol

A. Components:

i. **Follower** – synchronize their copy of data (*log*) with that of the leader's. A Follower gets the chance to turn into a Candidate when the current leader crashes.

ii. **Candidate** – a Candidate server asks other servers for votes.

iii. **Leader** – whom the client communicates solely with. Raft guarantees at most one leader is elected every consensus round (*term*).

B. Algorithm:

This section will explain the Raft algorithm, including Leader Electrion, Log Replication & Safety protocol.

i. Leader Election:

Nodes (servers) start out as Followers, and use their own randomized *election timeouts* to turn into a Candidate once that time duration expires and it is not receiving heartbeat signals from a leader.

Once turn into a Candidate, the node starts a new election *term.* It first votes for itself and broadcast vote requests out to other nodes. On the Followers side, if a Follower have not voted for "this" *term,* it will vote for the Candidate and reset its *election timeout.* If the Candidate receives vote from a majority of Followers, it turns into a leader.

As a Leader, the node sends out *appendEntries* in heartbeats. Using a *heartbeat timeout,* it will send a response to the Client (external) that the entry has been *"committed"*.

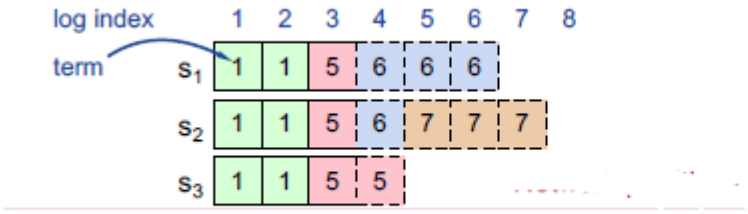
An election *term* continues until a Follower stops receiving heartbeats from the Leader, and eventually turns into a Candidate.

ii. Log replication

The Client initially sends an entry to the Leader, who appends the *entry* directly to its *log*. Leader then start sending *appendEntries* to Followers in its next heartbeat. Two scenarios can happen here:

+ Leader receives "success" responses from majority of Followers, the *entry* got "c*ommitted*" on the Leader's log. The Leader then notifies the Followers that *entry* is now *"committed"*.

+ Leader receives "reject" responses from a few Clients. To resolve this, when sending out appendEntries, the Leader maintains a table, or dictionary, of nextIndex for all Followers, with initial values being (Leader's last index + 1). For every "reject" from a Follower, the Leader will decrement the nextIndex for that corresponding Follower until the latest entry where the 2 logs agree. Then, all entries in the Follower's log will be overwritten with that of the Leader.



Server	nextIndex
S2 (Leader)	8
S3	8

→ decrement →

Server	nextIndex
S2 (Leader)	8
S3	**3**

→overwrite with leader's log

In the end, the Leader send out a response to the Client saying the *entry* has reached consensus.

4. Properties

- ***Election Safety***: at most one leader can be elected in a given term. With this property, we can see Raft is strictly single Leader protocol, hence huge traffic load can overwhelm the system.

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

- ***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.

- ***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.

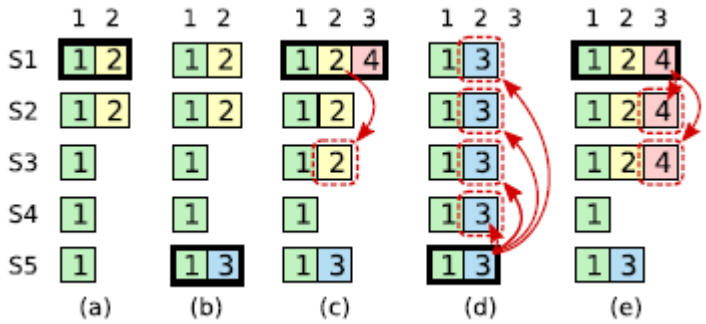
- ***State Machine Safety*:** if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

5. Performance Analysis / Evaluation

i. Leader (Election) Restriction

There are a couple of safety measures that Raft takes into account. An example would be when a Follower is unavailable while the Leader commits several log entries, then the Follower could be elected leader and overwrite these entries with new ones; as a result, different state machines might execute different command sequences. Therefore, restriction on which servers may be elected leader. A Candidate can only win an election if:

- its log contains all committed entries.
- its log is most up-to-date. The *index* and *term* of the last entries are compared to determine which log is more up-to-date. With different terms, then the log with the later term is more up-to-date. With the same term, whichever log is longer is more up-to-date.



The figure illustrates a time sequence showing why a leader cannot determine commitment using log entries from older terms.

(a) S1 is leader and partially replicates the log entry at index 2.

(b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2.

(c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed.

(d) S1 crashes, S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3.

(e) However, if S1 replicates an entry from its current term on a majority of the servers before crashing, then this entry is committed (S5 cannot win an election).

ii. Handling Follower & Candidate crashes

If a Follower or Candidate crashes, then future RequestVote and AppendEntries RPCs sent to it will fail. Raft handles these failures by retrying indefinitely; if the crashed server restarts, then the RPC will complete successfully. Aside from that, if a follower receives an AppendEntries request that includes log entries already present in its log, it ignores those entries in the new request.

Works Cited

[1] Lamport, L. (2001). Paxos made simple. ACM Sigact News, 32(4), 18-25.

[2] Castro, M., & Liskov, B. (2002). Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems, 20(4), 398–461. https://doi.org/10.1145/571637.571640

[3] Vasin, P. (2014). Blackcoin's proof-of-stake protocol v2. URL: Https://Blackcoin. Co/Blackcoin-Pos-Protocol-v2-Whitepaper. Pdf, 71.

[4] Delegated Proof-of-Stake Consensus. (n.d.). Retrieved from https://bitshares.org/technology/delegated-proof-of-stake-consensus/