An Implementation of the Raft Algorithm in Docker Using a Scalable Peer-to-Peer Network

Patrick McDonald

Department of Computer Science, Louisiana Tech University

The Raft algorithm was originally created at Stanford University and outlined in a paper by Diego Ongaro and John Ousterhout. The algorithm was designed to be an easier to understand alternative to Paxos.

This project provides an easy-to-understand implementation of the Raft algorithm in a modern programming language, using Docker and Docker Compose to simplify network configuration and scalability.

OBJECTIVES

- To implement a simple, scalable, peer-to-peer network that can be scaled using the scale functionality provided by Docker Compose.
- To implement the election protocol portion of the Raft algorithm for nodes to elect a leader
- To implement the log replication and commit protocols from the Raft algorithm for nodes to reach consensus
- To implement request proxy for non-leader nodes to make the network singly addressable

METHODS

Languages, Tools, & Frameworks

Node.js and Express were chosen as the runtime and web framework for the services in this project due to their widespread use in the modern software development industry. Additionally, TypeScript was chosen as the implementation language over JavaScript to make the code more readable and easier to follow.

Docker and Docker Compose were chosen to containerize and execute the project in order to simplify the underlying network infrastructure as well as scaling the number of nodes in the network.

METHODS (cont'd)

The Network

Implementation of the peer-to-peer network is based on a simple network registry and announce protocol. A registry service was created to keep a record of the nodes currently participating in the network. It is each individual node's responsibility to register with the registry service and announce its presence to the rest of the network. This is done immediately following node startup, once the node is ready to service requests.

This protocol for joining the network allows new nodes to join a network with an already established leader and immediately begin receiving append requests and quickly have an accurate log and replicated state machine.

Modes & Responsibilities

At any point in time, a node in the network is in one of three modes: *leader*, *follower*, or *candidate*. The leader node is responsible for servicing client requests and telling followers to append entries to their respective logs. There can only be one officially recognized leader node at any given time. The follower nodes are responsible for proxying client requests to the leader and servicing requests to append entries from the leader or for votes from a candidate during an election. A candidate node is responsible for collecting votes in order to become the new leader.

Election Protocol

When a follower exceeds its election timeout without hearing from the leader, that follower becomes a candidate and starts an election. The candidate node votes for itself and requests votes from every other node in the network until it has received votes from at least 50% of nodes, its election timeout is exceeded again, it receives an append request from a new leader, or it receives a higher term from another node.

The election timeout implementation is accomplished using a scheduler object that makes use of the built-in setTimeout function.

METHODS (cont'd)

The scheduler object maintains the current election timeout and handles reset and cancellation of the timeout, as necessary. The scheduler initializes the election timeout when the node starts. Additionally, in order to decrease the likelihood of candidate overlap, the timeout duration is randomized within a range every time the timeout restarts.

Log Replication & Consensus

In addition to the replicated state machine, each node maintains its own log of to-be-committed state machine commands, the index of the highest indexed log entry known to be committed, and the index of the highest indexed log entry applied to the state machine. In the official Raft specification, the log is stored in persistent storage, but for the sake of simplicity, this project stores the log data in memory on each node.

When the leader receives a client request with a new command for the state machine, it appends the new command to its own log. Then it sends requests to the follower nodes to append any entries missing from those nodes' logs. Once the leader has confirmed that more than 50% of the nodes has a given command in their logs, the leader increments the commit index, triggering an update to the state machine and the last applied index.

Proxying Client Requests

In order to make the network appear to clients as a single endpoint, the network exists behind a load balancer, which is automatically created by Docker Compose and handles requests in a round robin to all the nodes. Because it is not guaranteed that a request will go to the current leader with such a simple proxy strategy, and because every node knows the current leader of the network, each node has been implemented to act as middleware and proxy all client requests to the current leader.

CONCLUSIONS

While this project succeeds in providing a simple, easy-to-understand, scalable implementation of the Raft algorithm using a containerized peer-to-peer network, it should by no means be considered production-ready. There are additional steps that need to be taken for this implementation to meet the resiliency requirements of a production system.

Next steps to improve resiliency include error handling around requests to other nodes and protocols to remove nodes from the network when they become unresponsive. The current implementation makes assumptions about overall network health and node responsiveness, which would need to be addressed for a production system.

REFERENCES

1. Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC'14). USENIX Association, USA, 305–320.
2. Ongaro, D. (2015). *An Introduction to Raft. CoreOS Fest.* https://www.youtube.com/watch?v=6bBggO6KN_k.