The University of Texas at Austin

Distributed Systems

Software Engineering - Option III

Fleeing Paxos by Raft

Authors:
Howie Benefiel
Patrick Sigourney

Professor:
Dr. Vijay Garg

November 30, 2017

Abstract

Paxos has been the industry-standard protocol for implementing distributed consensus for the past 25 years. Though the basics of single-decree consensus are easily enough grokked, delving into why it works, forming a replicated log or implementing membership changes is fraught with unproven ambiguity. Raft was designed with a singular design goal, understandability. To achieve this goal of optimizing for understandability, the authors employed two techniques, problem decomposition and minimization of state space. For this paper, we examined and implemented their protocol. We found their protocol performed as well as they claimed and it was much easier to fundamentally grok than Paxos.

1 Introduction

The primary focus of distributed consensus is to take an unreliable machine and replicate it so that it becomes a cluster of individually unreliable but collectively reliable machines. The most studied model for this is the replicated state machine model. In this model, each machine has some internal state which can be changed by external stimulus. Perhaps obviously, the state of these state machines should be replicated by all the machines in the cluster. At a high level, the duty of a distributed consensus protocol is to ensure the state stays consistent across all the machines in the cluster.

The core data structure in a practical consensus algorithm, and indeed Raft, is the replicated log. The replicated log contains all the commands that have been run against the cluster. If a machine were to start at the beginning of the log and execute each command in its log in order that machine should arrive at a state consistent with other machines in the cluster. At a more practical level, a distributed consensus protocol will keep these replicated logs consistent across the cluster.

The FLP Impossibility Proof states that an asynchronous distributed system can not tolerate failure while satifying agreement, validity and termination. [**?**] So consensus algorithms, including Raft, generally satisfy:

1. *Safety* They always return a correct result. If a client randomly queries any server for the state of the system, the result should be the same regardless of which machine was queried.

2. *Availability* If a majority of the servers are available, the system, as a whole should be available.

3. *Time Invariance* The system should not depend on the physical timing of events.

4. *Quorum Limited* The response to the client should be returned once quorum is reached, not when all machines have responded.

Paxos, the leading distributed consensus algorithm thusfar, satisfies these properties, but its original incantation only arrived at consensus on a single value. [**?**] Practical systems need to build a replicated log which is an extension of the original Paxos protocol. This extension is one symptom of a larger problem with Paxos where extensions are needed to build practical systems with these extensions not being formally studied. Furthermore, the single-decree Paxos protocol is challenging enough in itself to understand, so extensions are incredibly hard to fully grok.

Since Paxos is poorly suited for building practical systems due to its complexity, the authors of Raft designed their protocol for understandability. This is obviously a unique goal as compared to most other CS papers, where the goal is to improve some time, memory, or message complexity.

The authors worked toward this goal using two techniques: decomposing the primary problem, distributed consensus, into independent sub-problems and minimizing the state space of the algorithm.

In this work, we examine the Raft algorithm, implement it and examine its performance

2 Algorithm Overview

Whereas Paxos shares a solution detail among multiple sub-problems, Raft targets a few sub-problems independent of one another. Before we discuss how Paxos solves these sub-problems, there are a few structures which we need to explain.

2.1 Structures

First, there is the concept of a *term*. A term is a subset of the system's execution which begins with an election of a leader. A term concludes when

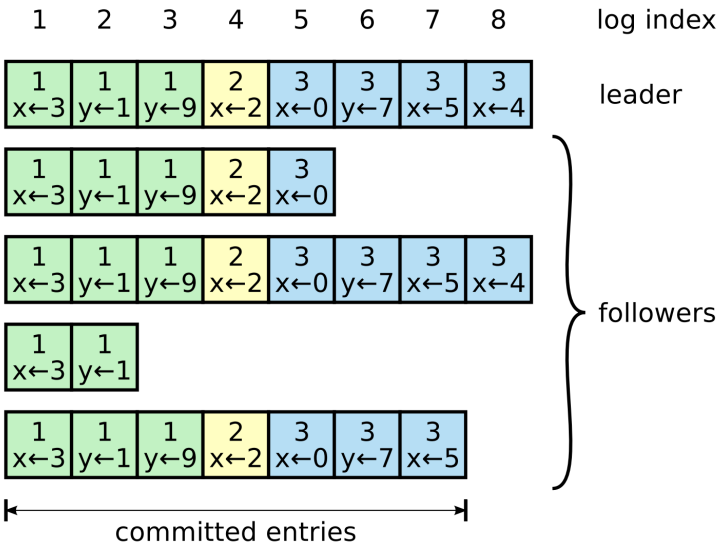

Figure 1: Above is an example of a replicated log. The top number and the color represent the term and the index of each entry is at the top.

the elected leader is not longer able to send messages to the other members of the cluster, usually due to the leader process failing, and a new election is started by the remaining cluster members. In exceptional circumstances, a term can end with no leader being elected due to a draw in voting, which results in a new term and new election being initiated.

The second structure described by Raft is the replicated log. The replicated log, as touched on above, is a list with each entry containing the command received from a client and the term number in which the command was received. Each entry is stored at a unique, incremented index location within the log. An example of the replicated log is shown below in Fig. 1.

Raft also uses two RPC messages, both having well defined schema. The purpose and implications of each will be discussed later in the paper but they will be introduced here. First, the `RequestVote` RPC is used by a candidate who has initiated an election and is sent to all other servers to request their vote for leader in the new term.

Second, the `AppendEntries` RPC is used for two purposes: Primarily it is used by the leader to replicate its logs among the followers. An empty `AppendEntries` RPC is also used as a heartbeat mechanism by the leader,

to notify the followers that the leader is still alive and in control. When a follower's logs are determined to be inconsistent with the leader's logs, the `AppendEntries` RPC is used to rewind the inconsistent follower's state and bring the follower into a state consistent with the leader. We have found that this RPC is somewhat overloaded and it would help understandability to decompose this RPC into a `HandleLogs` RPC and a `Heartbeat` RPC.

2.2 Subproblems

2.2.1 Leader Election

Using these data structures, the first of the sub-problems which Raft solves is leader election. Like Paxos and unlike some other consensus protocols such as Egalitarian Paxos [**?**], Raft elects a leader. In the Raft model, a machine can exist in one of three states: leader, candidate, or follower. A machine initial starts in a follower state. Every follower maintains a countdown timer with a random value within a preset range, the `election_interval`. This randomizing ensures that the timers expire at staggered intervals rather than simultanously. When a follower receives an `AppendEntries` RPC (whether a log update or a heartbeat message), the timer is reset. If the timer expires, the follower will increment its term, convert to a candidate, and send `RequestVote` RPCs to all other machines in the cluster.

That candidate then waits to receive votes from a majority of machines for that term. This rule mandates that a Raft cluster can only withstand the loss of 1/2 n - 1 members and still persist. If greater than half the members fail, none of the remaining machines will be able to achieve an election majority and a new leader cannot be elected.

While holding an election, if a candidate receives an `AppendEntries` RPC with a term greater than or equal to the candidate's term, the candidate knows there is a legitimate leader in the system and reverts itself back to a follower. If an `AppendEntries` RPC is received with a term less than the recipient's, the recipient will reply back to the sender with current term value which will result in the sender (who believes himself a leader) updating his term value and setting himself as a follower. Once a candidate receives a majority of votes from the cluster members, it has been elected leader for the term and will begin sending heartbeat `AppendEntries` RPC messages to the other members which will reset their election clocks and prevent an election from starting.

2.2.2 Log Replication

The second sub-problem Raft solves is log replication. This problem is markedly more complicated.

To more easily handle this problem, Raft defines two properties for the logs: First, two machines with log entries at the same index with the same term will contain the same command. This property allows us to couple ordering to the state of the system. Building on the first property, two machines containing log entries with the same index and term implies all previous log entries are identical between the machines. By defining these axiom, Raft simplifies the safety of the system.

The happy path case of log replication is when a leader receives a request from a client, the leader appends the request to its log, sends the `AppendEntries` RPC and the followers append the log entry to their logs. The last step of the process is the leader marking the log entry as committed once the leader receives replies of `success` to an `AppendEntries` RPC from a majority of the members of the cluster, the leader marks the entry as committed and the entry can then be applied to the state machine with the knowledge that a majority of the other cluster members will also apply the entry.

The unhappy path is when there are a series of leader and follower crashes. In this case, the system needs a way of rectifying the competing logs into a consistent state. The system accomplishes this by overloading the `AppendEntries` RPC. Whenever a leader sends an `AppendEntries` RPC, it includes `prevLogIndex` and `prevLogTerm` values which contains the index and term of the leader's previous committed log entry. If a follower receives this `AppendEntries` RPC and the `prevLogIndex` and `prevLogTerm` do not match the follower's log values, the follower will return false. The leader will then rewind and send step back through the log, sending messages with earlier log entries until the follower finds the last log entry which matches the leader's log. At this point, the leader will play forward through its log to bring the follower's log up to date. Using this mechanism, the current leader can rectify inconsistencies in the follower's logs and ensure consistency across the cluster.

2.2.3 Safety

Raft handles safety simply by ensuring that log entries can only flow from leaders to followers. A follower can never commit an entry until it has received a commit signal from the leader. If a follower receives a command from a client, it will direct the client to the current leader. All new client commands must first go through the leader, to ensure the leader's log is always complete.