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 29, 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 order, in its log, 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.

FLP states that an asynchronous distributed system can not tolerate failure and satisfy agreement, validity and termination. [1] 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. [2] 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. First, they on decomposed the primary problem, distributed consensus, into a couple independent sub-problems. Second, they minimized the state space of the algorithm.

In this work, we examine the Paxos 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. Most of the time, a term then

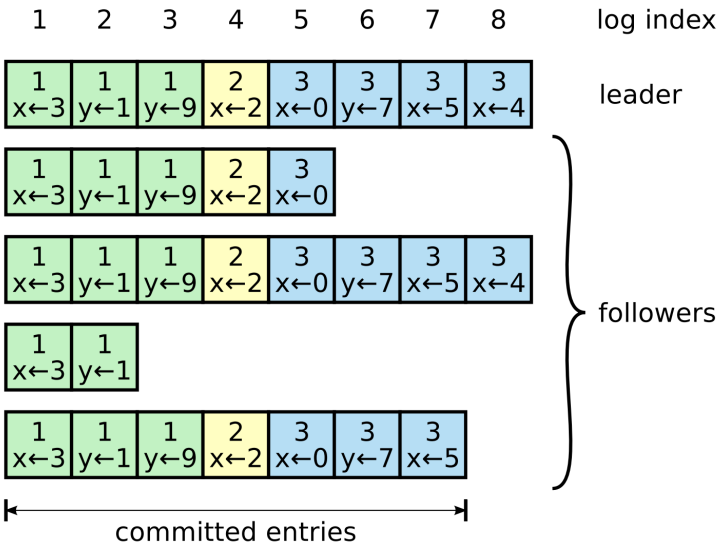

Figure 1: Above is an example of a replicated log. The top number and the color represent the term and the index of each entry is at the top.

ends with normal operation, some time period where the system is receiving commands from clients. In exceptional circumstances, a term can end with no leader being elected which results in a new election.

The second structure described by Paxos is the replicated log. The replicated log, as touched on above, is a list with each entry containing the term number and the command recorded at that index number. An example of the replicated log is shown below in Fig. 1.

Raft also uses two RPC messages which both have a well defined schema. The purpose and implications of each will be discussed later in the paper, but each will be introduced here. First, the `RequestVote` RPC is used by candidates to initiate an election and possibly be elected leader.

Second, the `AppendEntries` RPC is used for two purposes. Primarily, it is used by the leader to replicate its logs among the followers. Secondarily, it is also used as the heartbeat mechanism. Lastly, the `AppendEntries` RPC will rewind an inconsistent follower's state and bring the follower into a state consistent with the current 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 [3], Raft elects a leader. The other two states a machine can be in are follower and candidate. When a machine starts, it is in a follower state. Leader election when a machine has not received a heartbeat message (`AppendEntries` RPC) during some random interval, `election_interval`, the machine initiates an election. The machine, now in a candidate state, sends a `RequestVote` RPC.

That candidate then waits to receive votes from a majority of machines for that term. While holding that election, if the candidate, receives a message with a term greater than or equal to the candidate's term, the candidate knows there is some other legitimate leader in the system and reverts back to being a follower. The else there, the supposed leader's term is less than the candidate's simply means the candidate continues the election. Once a candidate receives a majority of vote's from the cluster, it starts sending heartbeat signals which suppresses other machine's attempts at becoming leader.

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 a log entry at the same index and same term will contain the same command. This property allows us to couple ordering to the state of the system. Inducting on that first property, two machines containing a log entry with the same index and term imply all previous entries are identical. By defining these invariants, 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 (heartbeat) and the followers append that log entry to their logs. The last step of the process is the leader marking that entry as committed. Once the leader receives `success` for an `AppendEntries` RPC back from a majority of the

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. When a leader sends an `AppendEntries` RPC, it also sends `prevLogIndex` and `prevLogTerm` which contains the index and term of the previous entry. If a follower receives this `AppendEntries` RPC and `prevLogIndex` and `prevLogTerm` do not match the follower's log values, the follower will return false. The leader will then rewind until the leader and follower agree on an index and term. Using this mechanism, the current leader can rectify inconsistencies in the follower's logs.

2.2.3 Safety

Raft handles safety simply by ensuring that log entries always flow from leaders to followers. A follower can never commit an entry until it has received a commit signal from the master.

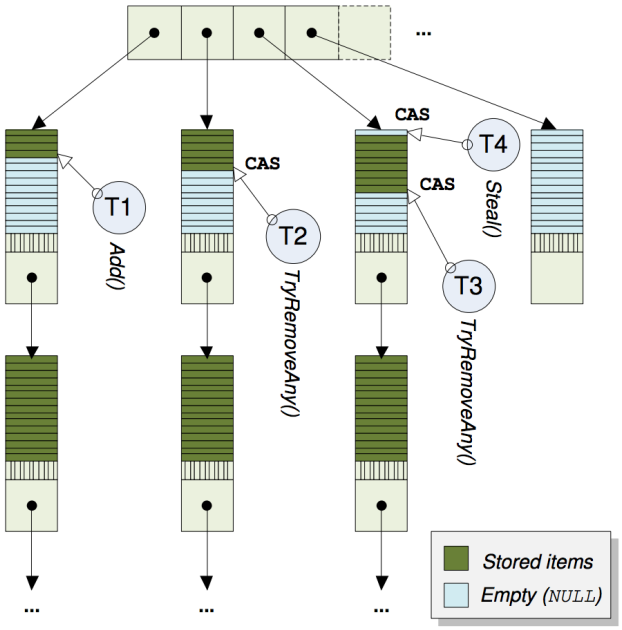

Figure 2: Above is the graphical representation of the basic data structure. The top array is globalHeadBlock. The linked list nodes below the array are the blocks. The green and blue cells are the objects being operated on.

2.3 Everything below this is just an example

2.4 Basic Operations on the Data Structure

There are two basic operations which can be used on a bag, `Add(item)` and `TryRemoveAny()`. `Add(item)` will add an item anywhere in the data structure, as expected. The `Add(item)` algorithm with detailed comments is shown in Algorithm 1. When adding or removing from the data structure, the threads keep track of the position in the block they can access and the first block in their linked list through the thread-local variables, `threadHead` and `threadBlock`, respectively. This is shown in Fig. 2 as the thread-numbered circles pointing to a position. One thing to note about `Add()` is that items will always only be added to the first block in the linked list and at most 1 thread will be adding to a block.

`TryRemoveAny()` is used to remove any random item from the data

structure and return it. This is shown with detailed comments in Algorithm 2. Now the situation may arise where a thread's linked list only contains empty blocks. For this situation, the `TryRemoveAny` operation provides functionality via the `Steal()` method to steal from another thread's block. This is shown in Algorithm 2 at line 4.

Algorithm 1 `Add(item)` Operation

```
1: if threadHead == BLOCK_SIZE then          ▷ the current block is full
2:     Create new block and add it at the head of the linked list
3:     threadHead ← 0
4: end if
5: threadBlock[threadHead] ← item ▷ Insert value at current threadHead
6: threadHead ← threadHead + 1 ▷ Advance threadHead to next position
```

Algorithm 2 `TryRemoveAny()` Operation

```
1: loop
2:     if threadHead < 0 then
3:         if head < 0 and next block == null then      ▷ End of linked list
4:             return Steal()                                  ▷ Go to steal
5:         end if
6:         threadBlock ← threadBlock.next          ▷ advance position in list
7:         threadHead ← BLOCK_SIZE                   ▷ last position in block
8:     end if
9:     item ← threadBlock[threadHead]              ▷ Retrieve item from block
10:    if item ≠ null and CAS(threadBlock[threadHead],item,null) then
11:        return item                    ▷ If CAS successful, return the item
12:    else
13:        threadHead ← threadHead − 1          ▷ If not, try next and loop
14:    end if
15: end loop
```

3 Implementation

4 Conclusion

References

[1] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, 1985.

[2] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–3169, 1998.

[3] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.