hRaft: An Implementation of Raft in Haskell

Shantanu Joshi
joshi4@cs.stanford.edu

June 11, 2014

1 Introduction

For my Project, I decided to write an implementation of Raft in Haskell. Raft[1] is a consensus algorithm for managing a replicated state machine. It is a popular replacement for Paxos with the key differentiating factor being that Raft was designed from the ground up to be easily understandable. Raft has been embraced by the community and has widespread applications in commercial settings. At last count, there were 28 implementations of Raft with only one prior implementation in Haskell [2].

The next section will go over the necessary properties and definitions required to understand Raft with the following section outlining how the different pieces come together. Section 4 highlights some of the important details about my implementation. I wrap up the discussion with a section on future work and lessons learned.

2 Background

2.1 Roles

Each server (node) participating in the consensus can have one of three roles at any point of time : `Leader` , `Follower` or `Candidate`. Having these roles decomposes raft into a series of smaller problems , making it highly modular and thus easier to understand.

2.2 Communication

The nodes communicate with each other through remote procedure calls (RPCs). Raft allows for only two RPCs, The *AppendEntries* RPC and *RequestVote* RPC. The client sends the log instruction(s) to the *Leader* which attempts to replicate it across all the other servers through the AppendEntries RPC. A *heartbeat* is simply an AppendEntries RPC with no payload. Lastly, If any server receives an RPC or the result of an RPC whose payload has a term that is greater than the servers current Term then it is updated to the higher value.

2.3 Log

Raft creates a replicated state machine via replicated Logs across all the servers. An entry in a log has an index and contains the term number along with the instruction to be executed. A log entry is considered to be *committed* once it has been replicated on a majority of the machines. The algorithm guarantees a Log Matching Property which states that:

- If two entries in different logs have the same index and term then they store the same command
- If two entries in different logs have same index and term then the logs are identical in all preceding cases.

The above property along with some rules for election safety ensures that when a leader marks an entry as committed it will eventually be executed by all the servers. In cases where logs across different machines are inconsistent, this property dictates how they are brought into sync again.

3 Raft Consensus Algorithm

Now that we are familiar with the different components of Raft, let us dive into the heart of the algorithm. The rest of this section will focus on providing the key details of implementing Raft.

3.1 Election

The log entries flow in one direction only, from the Leader to the Followers. The algorithm also guarantees that only one server can win the election and become a leader. The default state for each server is to be a Follower. If a Follower does not receive either an AppendEntries RPC or a heartbeat within a set interval of time, then it assumes that there is no viable Leader.

In such a scenario, it changes it's role from Follower to Candidate, increments the value of its current term and starts an Election by issuing RequestVote RPC's to all the other members of the consensus.The following rules are enforced in an election:

- The candidate always votes for himself.
- A server can only vote for one candidate per term. This restriction coupled with the need for a simple majority guarantees that there can only be one winner in an election.
- The candidate can win a vote from a Follower only if the its log is at least as up-to-date as the Followers. Raft determines which of the two logs is more up-to-date by comparing the index and term of the last entries in the logs. The log with the greater term is more up-to-date. If they have the same term then the log which is longer is more up-to-date.
- The election times out if no leader is elected within a specified duration. In order to make these occurrences rare, the election timeout for each server is randomly chosen from within a specified interval.

3.2 Leader

Once a Leader has been elected, it immediately sends a heartbeat to all other servers to establish its authority. Upon receiving this heartbeat, a server updates its term if necessary and converts to being a Follower (from a Leader or a Candidate). On the flip side if a Leader receives an RPC from another server whose term is greater than its own, then it relinquishes its role as a Leader and becomes a Follower.

In addition to the state that is maintained by Raft the leader also maintains some additional information. For each Follower, it maintains the index of the next log entry to send to that particular follower. Further it keeps track of the highest index of the log known to be replicated on another server. This information coupled with some restrictions on when an AppendEntries RPC succeeds (discussed below) ensures log consistency.

Upon receiving a new instruction from a Client the Leader appends the new entry to its Log and attempts to replicate it on all the other machines. A Follower can reject an AppendEntries RPC if the previous Log Entry and Term (relative to the nextIndex value stored in the dictionary for that particular follower) does not match on both the machines. This means that the logs are inconsistent. To resolve this the leader reduces its nextIndex by one and tries again till it succeeds. Once a matching index is found all the entries of the log after that are deleted and replaced by entries from the leader. This procedure ensures that the Log Matching Property is maintained. Additionally, this means that the leader does not need to take any special steps to bring the long of a follower into consistency with its own. The leader also updates its commit index based on the values in the matching index dictionary.

3.3 Follower and Candidate

A Candidate is the only one that can issue a RequestVote RPC. Based on the replies it gets from other Candidates and Followers, it tallies the votes received and determines if it has obtained a simple majority to become a Leader. If not, then it waits for the election timeout to occur and starts again. If another candidate has become a leader then it responds to the heartbeat and becomes a Follower.

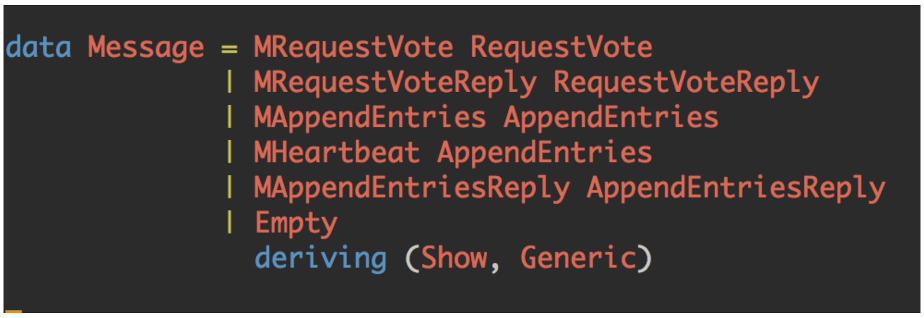
A Follower receives both AppendEntries and RequestVote RPC's. The conditions under which it accepts or rejects both have been discussed in detail in the previous sections.

3.4 Server Crashes

Each server has to maintain some non volatile state: its log, current term and who they have voted for. This information is saved to disk as and when it changes. Upon initialization it looks for that particular file and picks up from where it left off.

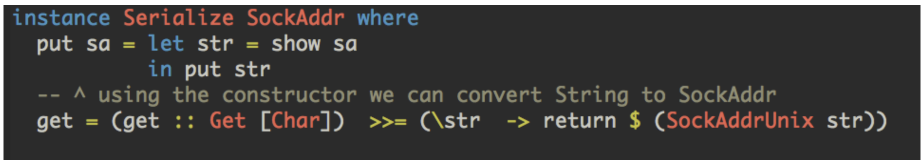
When a Leader Crashes, the election timeout occurs, one or more of the followers becomes a candidate and a new leader is elected.

When a Follower or Candidate crashes, operation continues as normal (as long as a majority of the servers are still running) and it will receive the RPC's it missed when it resumes full functionality.



```haskell
data Message = MRequestVote RequestVote
             | MRequestVoteReply RequestVoteReply
             | MAppendEntries AppendEntries
             | MHeartbeat AppendEntries
             | MAppendEntriesReply AppendEntriesReply
             | Empty
               deriving (Show, Generic)
```

Figure 1: Message is sum type which contains wrappers around both the RPC's and their replies.



```haskell
instance Serialize SockAddr where
  put sa = let str = show sa
           in put str
  -- ^ using the constructor we can convert String to SockAddr
  get = (get :: Get [Char])  >>= (\str  -> return $ (SockAddrUnix str))
```

Figure 2: The definition of a Serializable instance for SockAddr.

4 Implementation and Design Considerations

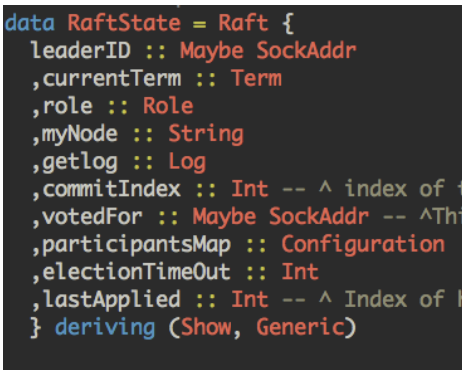
I found the very handy *timeout* function in the **System.Timeout** package. It's type signature is `timeout :: Int` → `IO a` → `IO ( Maybe a)` . This function allows the user to encapsulate a timer around IO actions. The first argument specifies the number of microseconds to wait before timing out. If the action does not finish before the timer reaches zero then it returns `Nothing` otherwise it returns the value of the action wrapped in a `Maybe` constructor.

One of the major design decisions that I had to make was how to model the RPC's. Talking with Bryan, he suggested I look into Cloud Haskell [3]. Cloud Haskell drew inspiration from Erlang and provided the semantics of Typed and Untyped Channels in Haskell. However, after reading their paper I realized that that their current implementation used Template haskell and other bits of unidiomatic haskell syntax. This coupled with the fragmented nature of the tutorials , some of which didn't work at all, convinced me to adopt the message passing method suggested by Prof. Mazieres. .

Figure 1 shows the definition for the **Message** type. Passing around messages instead of RPC's meant I needed to create messages for the results of the RPC as well. In order to differentiate between the reply of a server to an AppendEntries RPC versus a Heartbeat I created the Empty message to be a response for a Heartbeat. This works because a leader would only send a heartbeat if the logs of both the servers are completely in sync and it has nothing new to send.

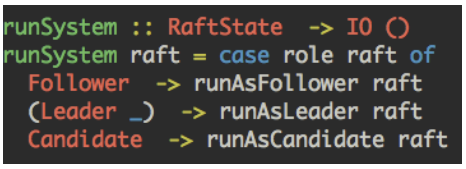
Another issue that I had to deal with because of my message passing approach was to have all the datatypes be *Serializable*.The only problem I encountered was with the **SockAddr** type. I fixed the issue by writing a custom instance of Serializable for it as shown in 2

I chose the simplest possible representation for maintaining the state of the



```haskell
data RaftState = Raft {
  leaderID :: Maybe SockAddr
  ,currentTerm :: Term
  ,role :: Role
  ,myNode :: String
  ,getlog :: Log
  ,commitIndex :: Int -- ^ index of
  ,votedFor :: Maybe SockAddr -- ^Th
  ,participantsMap :: Configuration
  ,electionTimeOut :: Int
  ,lastApplied :: Int -- ^ Index of
  } deriving (Show, Generic)
```

Figure 3: The type definition for RaftState



```haskell
runSystem :: RaftState  -> IO ()
runSystem raft = case role raft of
  Follower  -> runAsFollower raft
  (Leader _)  -> runAsLeader raft
  Candidate  -> runAsCandidate raft
```

Figure 4: An example of top level modularity that arises due to the Roles

system and created a single **RaftState** type (Figure 3) that carried all the information in one place. The LeaderState which is embedded inside the **role** field is the only exception. In many cases only a few fields had to be updated/modified and the record syntax came very helpful in this situation.

The message passing interface and natural decomposition of the system into modules based on Roles made the overall code and especially the top-level functions very modular. Figure 4 provides a glimpse of this. This property was especially useful while **testing**. I could write stub functions that induced particular behaviour in the system without any refactoring. This enabled me to ensure that all possible interactions between two servers worked properly without having to wait till I wrote the entire system.

5 Future Work

Given more time I would have implemented more of the features specified in [1]. Dealing with membership changes, and adding log compaction would be the top items on my list of future improvements.

Improvements that I could make to the existing code include reading from a config file instead of hardcoding the servers participating in the consensus and to utilizing lenses instead of the record syntax everywhere.

I hope to carry on working on this and eventually release the code on Hackage.

6 Lessons Learned

One of my most important takeaways from this project was finally figuring out how the low level *Network.Socket* package worked and using it successfully to implement a substantial system.

An advantage of coding in a purely functional language like Haskell is that most of my functions were very short and performed only one task/transformation. In imperative languages even after testing each function individually we can run into bugs when testing the entire system. A combination of Static typing and purity ensure that such bugs are almost completely absent from Haskell.

Coming into this class I used to think that Functional Programming is only useful in very specific situations. However, after taking the class and working on this Project I've come to realize that its possible to implement a wide gamut of applications in Haskell, from numerical solvers to cryptography to distributed systems to computer vision algorithms etc.

References

[1] D.Ongaro and J.Ousterhout In Search of an Understandable Consensus Algorithm In *USENIX Annual Technical Conference*, 2014

[2] T.Nicolas. Kontiki *https://github.com/NicolasT/kontiki/tree/master/src/Network/Kontiki*

[3] J. Epstein, A. Black and S.P. Jones . Towards Haskell in the Cloud *Haskell Symposium, Tokyo* September 2011