CMSC 23310 Project Paper - Raft Implementation

Roman Amici and Andrew Ding

6/5/2014

1 Introduction

The fault-tolerant protocol we decided to implement is Raft. With a draft published as recently as February 2014, this protocol has been proposed as a safety-first distributed consensus algorithm that can serve as a more understandable alternative to (multi)-Paxos. It boasts the same efficiency as (multi)-Paxos, but with a more modular structure that lends Raft to a better understandability.

Raft solves the problem of replicating a log identically on a set of machines. This log is used to run a distributed finite state machine. This paper references In Search of an Understandable Consensus Algorithm by Ongaro and Ousterhout, Februrary 2014.

The key properties that Raft offers are as follows, in the authors own words:

1. Election Safety - At most one leader can be elected in a given term
2. Leader Append-Only - A leader never overwrites or deletes entries in its log; it only appends new entries.
3. Log Matching- If two logs contain an entry with the same index and term, then the logs are identical in all entries up and through the given index.
4. 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-number terms
5. 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.

The key property is the fifth one, which is analogous to safety in Paxos. Informally, safety means nothing bad happens; in this case amounts to rewriting already correct data. Safety is completely independent of the timing of messages.

The consistency model is strong consistency. If the value of x is requested by a client, and he receives an answer, then he knows the received value is up-to-date. This is due to Property 4, Leader Completeness, since a GET request can be satisfied by asking the leader (who has the most up-to-date info). Of course, this process is not fully available given failures, since a node that is cut-off from the leader cannot guarantee a consistent answer.

Availability is fundamentally tied with timing. Availability is threatened if the election timeout is poorly chosen. It must be significantly longer than the time required to broadcast messages and write to persistent storage, but also significantly shorter than the average.

Other properties: The method by which consensus is achieved guarantees replication as a side-effect. The system makes progress under the condition that a strict majority of the servers are functioning properly. Raft handles failures of the fail-stop form; servers either crash or they crash and resume function. In particular, Raft doesnt address the issue of Byzantine failures; when servers are online, we assume they are functioning correctly. Raft also allows for network issues such as delayed, out-of-order messages and network partitions. Since Raft uses a leader-based system, we assume that all communication among the servers goes through the leader, and that only the leader communicates with the clients.

2 Our implementation (and Get and Set Requests)

2.1 Important Classes and Functions

Our implementation is done in Python. Since we are writing for a single client, we have a class called RAFT_instance.

- Its instance variables contain every variable mentioned in the summary on page 4, including currentTerm, votedFor, etc.
- The other instance variables include name, isLeader (bool), leader (name of leader), self.lastHeard (for timeout purposes). Lastly, we keep track of the number of peers for the sake of calculating the majority.
- We could have used an enumerated type for the three roles: follower, candidate, and leader; however, since the candidate status is usually short-term, we decided simply to have a candidate subroutine that is called if and only if the election timeout elapses and there is no leader.

We have a class for the socket info needed by ZMQ:

- class sock_state

We also have classes for the various RPCs and their respective replies. Each of these has a to_json message that does exactly what it says on the tin.

- class append_message
- class appendReply_message
- class requestVote_message
- class replyVote_message

For each message class as well as get and set, we also have a handler function. They are called by the handle_message function upon receipt of a message.

- handle_get_set
- handle_append
- handle_appendReply
- handle_vote
- handle_voteReply

2.2 Timing and Implementing Get and Set

Handling the timeouts was tricky. Fortunately, there are only two timeouts:

- In the follower state, the timeout length is based on the leader lease time (in our implementation, 5 seconds). If it elapses, it transitions to a candidate and starts an election.
- In the candidate state, the timeout length is a randomized (in our implementation, 1-3 seconds). If it elapses, then we start a new election.

Instead of resetting the timer, we thought it would be simpler to have the timer run out, and act according to the actions that have happened since the timeout was initiated. We have a pair of functions called check_election and check_leader_timeout.

- Follower: Upon timeout, call check_leader_timeout. The usual case is that a heartbeat message has occurred, meaning current time is earlier than raft.lastHeard[leader] + rand_time. We reset the timeout by specifying timeout to be raft.lastHeard[leader] + rand_time. The abnormal cases are when theres no leader (i.e. new term without leader) or when the node fails to receive messages from the leader. In either of these two cases, the follower should become a candidate via the request_vote function.
- Candidate: If the election timeout elapses, then if there is a leader, we simply do nothing (including not creating a new time_out, so the leader never has a time_out). If there is a leader, then we call request_vote, which sets a new timeout.

Note: Ideally, client requests are always sent to the leader in Raft. However, under failure conditions, we cannot know for ascertain the leaders identity, due to factors such as randomization in the timeouts and a lack of leader in a small partition). Therefore, we had to implement a procedure to forward a client request.

We suppose the get message is sent to an arbitrary node A. In the normal scenario, node A will forward the get message to the leader, and the leader will attempt to get a majority of the followers to append the get message in their logs. After all that, the leader forwards the requested data to node A, who returns it to the client. A getResponse sent to a stopped node, or if it is dropped will halt the script since a getResponse will never be sent.

In case a network is permanently partitioned, we must send an error message. With this rationale, we implemented a timeout in the message itself. Upon receiving a get message, we start a timer, where node A must will send the appropriate error message to the client if it has not successfully forwarded it to the leader.

Set messages are very similar. Suppose Node A receives a set message and forwards it to the leader. The leader will use the RequestVote RPC (simulated by the handle_vote and handle_voteReply functions). Again, the leader does not forward the setResponse message to node A until a majority of the followers have appended the log entry. This action gives two desirable properties: accounting for up to n/2 failures, and replication of data.

This leads to the discussion of the fault-tolerant properties of this networks. As mentioned, there is an appreciable amount of data replication, as well as accounting for fail-stop errors (up to n/2). Our implementation of Raft accounts for network partitions in a certain sense; if one partition has a strict majority of nodes, then progress is still made there, and client requests to that block of nodes will succeed as usual. In the event of delayed or dropped messages, safety is never threatened; however, these issues may cause availability issues. In addition, our implementation does guarantee that, in the absence of errors, all client requests will be processed successfully.

3 Scripts and Functionality

3.1 Overview of Test Scripts

- get-set.chi - Tests the get and set functionality with a predefined leader and two follower nodes
- test-leader-election.chi - Artificially triggers a leader election and then ensures that get/set commands work as intended.
- test-replication.chi - Tests that set commands are replicated on multiple nodes by performing a set, and then changing which node is the leader (and thus which serves the request)
- test-leader-failover.chi - Kills the current leader and tests that a new master is spontaneously elected.
- test-partition2election-replication.chi - see below

3.2 Discussion of test-partition2election-replication.chi

This particular script shows off a number of the features that our implementation of raft offers. It shows how one node can be elected as the leader. Then it shows that the datastore continues to function as long as a majority of the total nodes are still present. Next it shows how a new leader may be elected and that this leader will necessarily have the most up to date copy of the log. Finally, it shows that nodes which miss an update to the log can catch up after falling behind. Its text can be seen below:

```
start test --peer-names test2,test3,test4,test5
start test2 --peer-names test,test3,test4,test5
start test3 --peer-names test,test2,test4,test5
start test4 --peer-names test,test2,test3,test5
start test5 --peer-names test,test2,test3,test4
send {"destination" : ["test"], "type" : "debug-startElection"}
after 10 {
set foo bar
after 8 {
split p test4,test5
set test4 foo 1
set test3 foo 1
after 25 {
join p
after 25 {
get foo
after 25 {
send {"destination" : ["test5"], "type" : "debug-startElection"}
after 12{
get foo
}
}
}
}
}
}
```

The script begins by activating five nodes (test through test5). The nodes start without a leader and are thus ill equipped to receive any transaction messages. The script begins by sending a debug-startElection message to test. This message causes test to trigger a leader election. It does this by incrementing its term number and then by sending requestVote RPCs to test2-5. Each of the other nodes will notice that the term sent out by test is higher than the term value for which they were initialized will set their current term to the term value sent by test. They will also notice that their logs are the same length. With each of these conditions met, they send voteReply RPCs with the voteGranted field set to True. When test receives replies from two other nodes, it names itself the leader and then sends heartbeat messages to every node, advertising that it is the leader. It indicates this fact by sending a log message to the broker with the message.

The script waits for 8 messages before sending a set request to ensure that leader election is complete. If the broker had not waited, the set request would be queued at the node until leader election was complete. Once the set message is received, a raft round is completed to replicate the log on the other four nodes. The broker then partitions the network so that test4 and test5 are part of a separate networks. Since this partition does not contain a majority of nodes, it cannot perform get or set commands, nor can it elect a new leader. Thus, when test4 receives the request set foo 1, It attempts, in vain to forward the request to the last node that it knew was the leader. During these attempts, an amount of time given by LEADER_LEASE_TIME elapses without hearing from the leader. This causes both test4 and test5 to attempt, in vain, to elect a new leader. Finally, the timer on the set request expires and test4 sends a response back to the client indicating its failure.

The client then sends the same set request to test3. Since test3 is in the partition with an active leader, and since this partition contains a majority of the active nodes, the operation . After waiting for several sets of heartbeat messages, the broker repairs the network partition. Meanwhile, the partition with test4 and test5 have attempted to elect a new leader and have thus incremented their terms a number of times. When the partition is repaired, test4 or test5 will send a request for votes with a higher term number than the current term. Test, test2 and test3 will increment to the current term once they receive these requests. However, they will not grant their votes to either test4 or test5 because the fulfillment of the previous set request ensures that nodes 1 through 3 have one more log entry committed than nodes 4 and 5. This causes nodes 1-3 to reject all vote requests sent by test4 or test5, thus preventing their election. Since these two nodes cannot be elected, an election timer will eventually expire on one of the eligible nodes (1-3). Once an eligible node tries for election, it will succeed and be elected the leader. Thus, it can successfully handle the get request which is subsequently sent by the broker.

Finally, this script demonstrates the fact that nodes which fall behind can catch up. During the fulfillment of the get request (via a response to a append_message RPC), the leader learned that test4 and test5 have logs which are out of date. The leader sends these logs to the nodes which were behind. Once received, they apply them in order, updating the state of their copy of the data store if necessary. Thus, when the broker sends a debug-startElection message to test5, it is able to win a majority of the votes since its logs are now up to date. Furthermore, it is able to properly handle a get request by returning foo $\implies$ 1. This indicates that the set foo 1 command, for which test5 was not present to handle, is properly replicated on the node. Otherwise, test5 would have incorrectly returned foo $\implies$ bar which it was present for.

3.3 Discussion: test-failstop-hack2.chi (fail-stop)

Like a few other groups, we had trouble with the stop function. Therefore, to simulate a stop, we split the node we wish to stop from the rest of the group. The purpose of the following script is to show that our implementation progresses in the face of fail-stop failures.

```
start test --peer-names test2,test3,test4,test5
start test2 --peer-names test,test3,test4,test5
start test3 --peer-names test,test2,test4,test5
start test4 --peer-names test,test2,test3,test5
start test5 --peer-names test,test2,test3,test4
send {"destination" : ["test"], "type" : "debug-startElection"}
set test foo bar
after 10 {
split p test3
split q test4
get test2 foo
set test foo 3
get test2 foo
}
```

The output statements related to the get and set responses are:

```
2014-06-05 16:01:59,200 [broker] INFO: setResponse (test): foo => bar
2014-06-05 16:01:59,200 [broker] INFO: getResponse (test2): foo => bar
2014-06-05 16:01:59,200 [broker] INFO: setResponse (test): foo => 3
2014-06-05 16:01:59,200 [broker] INFO: getResponse (test2): foo => 3
```

Therefore, given these two failures, Raft is still able to make progress.
Splitting three does not work. We add the line split r test5 and get:

```
setResponse (test): foo $\implies$ bar
2014-06-05 16:51:42,844 [broker] INFO: getResponse (test2): foo
ERROR: Request timedout. Please try again
2014-06-05 17:12:21,089 [broker] INFO: getResponse (test2): foo
ERROR: Request timedout. Please try again
2014-06-05 17:12:25,097 [broker] INFO: setResponse (test): foo
ERROR: Request timedout. Please try again
2014-06-05 17:12:29,102 [broker] INFO: getResponse (test2): foo
ERROR: Request timedout. Please try again
```

Since a get request must be appended to the log, it fails, as does all future requests.

4 Issues, Challenges, and Lessons Learned

There were quite the range of obstacles encountered while trying to implement Raft.

ZMQ and tornado were quite the libraries to work with. I, (Andrew) was a little daunted by all the use of all the new libraries. Perhaps this is something I should have picked up from Networks. I had some difficulty figuring

Also, there was a minor issue with the broker. For some reason, stop was not working correctly for me. When I used a script with stop, two weird things would happen. First of all, when I sent a set/get request to a stopped node, there was no error, and things proceeded smoothly. If I didn't specify the node, the client request had a chance to be sent to a stopped node as well as working nodes, which was weird. The second issue is that, when I stopped a strict majority of nodes, Raft still proceeded. That was definitely not good.

It was also a challenge to implement timeouts. It was confusing how to "reset" a timeout. We eventually gave up and would simply run a function that extended the timer if the condition was met. I felt there should have been an easier way to do this.

One of the lessons I learned was the effectiveness of randomization for timeouts. I think this is quite a simple, effective method for election timeouts. It does a great job of avoiding split votes.

Another lesson we learned is that the separation of the main RPCs made it easier to divide up the work. One of us took AppendEntries, and one took VoteRequest, and we were able to work independently. I suppose that's another bonus in addition to understandability of splitting up the tasks. We could even have written separate modules for the two RPCs.