IMPLEMENTATION OF THE RAFT PROTOCOL

1.Abstract

Raft is a consensus algorithm that aims at producing results that are equivalent to the more widely used Paxos algorithm.
Raft is used for managing replicated log in distributed systems. It uses a structure that is easier than Paxos, thus making it more understandable. The key elements of establishing consensus in Raft are leader election, log replication and safety. It uses a new mechanism to address the issue of cluster membership changes. This mechanism uses overlapping majority to guarantee safety.

2. Introduction

Raft aims at achieving consensus among a network of connected machines, but in an easy and understandable way. Consensus algorithms facilitate all the machines on a network to work as a coherent group in normal circumstances as well as in the events of failure. This makes them a key player while implementing large scale systems that are reliable.
The basic function of RAFT protocol is to elect a distinguished leader who will manage the replicated logs at all nodes in the network. The leader replicated the log entry that it gets from the client and replicated it on all the nodes. The leader also tells the nodes when its safe to apply the log entries to their respective state machines. If in this process, the leader fails or disconnects, a new leader has to be appointed.
One of the very popular protocols used to achieve consensus is the PAXOS protocol.
Paxos was first published in 1989 and has been used to solve consensus in a network of unreliable processors. The problem with Paxos is that it is difficult to understand despite of multiple attempts to make it more approachable. One major reason for that is that it has a complex architecture. It has a large number of trade between the number of processors, number of message delays before learning the agreed value, the activity level of individual participants, number of messages sent, and types of failures. This is why programmers as well as students struggle to understand Paxos.
While designing raft, specific techniques were applied to ensure understandability. The separation of leader election and log replication makes the architecture easy to understand.
It was found that students found it significantly easier to understand Raft as compared to Paxos. When it comes to performance, entries can be added with (f+1) roundtrips whereas for Paxos its 2(f+1).

3. Working of raft protocol

the raft protocol gives the leader complete control of managing replicated logs. The client sends log entries to the leader which in turn replicate the log entries to the other servers. The leader tells the servers when its safe to apply the log entries on to their state machines. This makes the management of servers easy as data flows in one direction. The consensus problem is decomposed into three sub problems, namely:

a. Leader election: one server is voted as the leader and re election is done if the leader fails.
b. Log replication: the server logs are made to replicate the leader's logs.
c. Safely: this property ensures that there is at most one leader per given term.

3.1 LEADER ELECTION

In order to achieve consensus in the cluster, first step is to elect a leader. Every server can have three possible states: Follower, candidate or leader. Every node starts off as being a follower. Leader election is done using two types of timeouts. The first is called the election timeout.

It is the amount of time each follower has to wait before it can become a candidate. It is a randomized time interval chosen between the range of 150- 300ms.

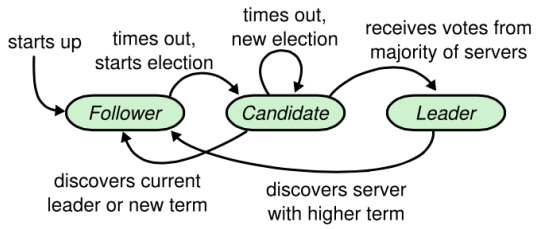

Fig 1: the server can have three possible states; follower, candidate or leader.

Once the followers become candidates, they start what is called their election terms. A leader serves for 1 term. An election term is when a follower becomes candidate and votes for itself. It then sends a Request vote message to other nodes seeking votes to become the leader. The servers store a current term number that increases monotonically over time. If receiving node hasn't yet voted in this term, it votes for the candidate and resets its election timeout.
When the candidate has a majority of the votes, it becomes the leader.
Servers communicate with each other using 2 kinds of RPCs. The first one is the REQUEST VOTE Rpc that are issued by the candidates. The second type of RPCs are the APPEND ENTRY Rpcs that are issued by the leader.
The Leader sends APPEND ENTRIES message to all the followers. This is done at regular intervals that are defined by a Heartbeat timeout.

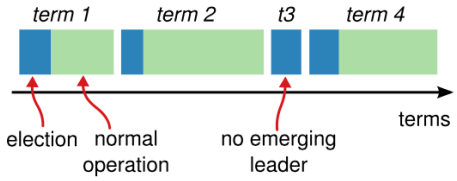

fig 2: each leader serves for one term.
The leaders send periodic heartbeats to all the nodes in the cluster. If the leader fails, the fellow nodes don't get the heartbeat messages, the servers turn into candidates and the leader election process is started.
The Followers respond to each append entry. The append entry messages imply that the leader is still alive. This particular election term will continue till a follower stops receiving heartbeats and becomes a candidate. A split vote can occur when two nodes start an election for the same term. In this case, all the followers wait for a new election and try again after a fixed time interval.
When the leader election process begins, each server first increments its current term. The state of the server changes from follower to candidate and votes for itself. This is when they issue a Request to Vote RPC to the rest of the nodes. The followers then respond to the request with their votes.
When the candidate gets the majority of the votes, it wins the election and becomes the leader.
Another possibility to be addressed is if a candidate doesn't win or lose the election. This can happen if too many followers become candidates at the same instant. In such cases, votes get split and no one gets the majority of votes and each candidate goes through time out and a new election term is started. The term entry is incremented and another set of Request-Vote RPCs. To ensure that the split vote event doesn't occur indefinitely, election timeouts are chosen randomly (150-300 ms). This makes sure that in most of the cases a single server times out and wins the election. Once this is done it immediately sends heartbeats before any of the other candidates sends a request to vote Rpc.

3.2 LOG REPLICATION

Once the leader is elected, it appends the commands issued to the clients to its own log as a new entry. These changes are replicated at the nodes. This is done using the Append-entry Rpc messages. When a client sends an entry to the system, it goes to the leader. The leader adds this change as a new entry in its log. Once this change is added to its log, it then Sends this change to the follower nodes. Each log entry consists of the command and the term number. Initially the log entry is set to be in an uncommitted state. To commit this change, the leader must get a majority of the followers acknowledge the

change and send acknowledgment messages back to the leader. When this is done the leader notifies all the followers and a consensus is thus reached about system state. The leader has to keep a record of the highest index that has been committed so that all the other servers can match up to it and make changes in its state machine. The consistency of the logs are checked by log matching: If there are two entries in different logs that have the same index and term, it implies that they store the same command. Also they must have identical entries in all the previous sessions.

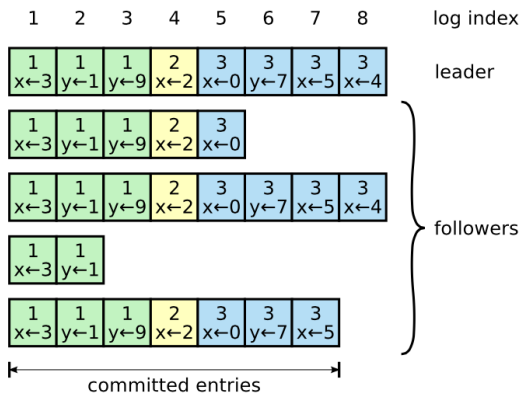

Fig 3: Logs have entries that are numbered sequentially. The term numbers are indicated in each entry.

Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered committed if it is safe for that entry to be applied to state machines. Inconsistencies can arise in case of leader crashes as the old leader may not have finished replicating all of the entries in its log. In such cases, there may be a lot of missing entries in the follower's log, the leader's log or both.

The leader handles inconsistencies by ensuring that the followers' logs are exactly the same as itself. If there are any entries that are a mismatch to its own, the mismatched entry is overwritten to match the leader. The leader also maintains a nextindex for each of the follower in the cluster that tells the leader which log entry it has to send next to the follower. The nextindex entry is initialized to a number that's greater to the leader by 1. The AppendEntries consistency check fails if the followers' logs are inconsistent and the leader will decrement the nextindex till it reaches the point where the entires match exactly. All the conflicting entries are thenn removed and the log is made consistent with the leader.

3.3 SAFETY

It is necessary to ensure that each state machine executes exactly the same commands in the same order. In case the follower becomes unavailable the log entries were being committed, there is a possibility that the leader would overwrite its log entries due to which different state machines would execute different commands.

That's why raft uses certain restrictions on the leader election process. It guarantees that all the committed entries from the previous terms from the previous terms are on the new leader's logs from the instant they are elected. This restriction is applied by the RequestVote RPC- the RPC includes information about the candidate's log, and the voter will deny its vote to the candidate if its own log is more up-to-date than that of the candidate. This is done by comparing the index and term of the last entries in the logs.

In cases where the leader crashes before committing an entry, the future leader will attempt to replicate the entry. No log entry is committed from previous terms just by counting replicas except those from the leader's current term.

If a follower or candidate crashes, then AppendEntries messages sent to it will fail. In such cases, the leader retries indefinitely. If the crashed server restarts then the rpc will be processes successfully. If a server crashes after completing an RPC but before responding, then it will receive the same RPC again after it restarts.

3.4 LOG COMPACTION

the logs have to be compressed in order to save space. As the log size increases, the more time it requires to replay. This might lead to availability issues.

The way raft handles this problem is by snapshotting. The current system state at the given instant is written on a snapshot on stable storage, then the entire log up to the point of snapshot is discarded.

Each node in the cluster has the permission to take a snapshot of its current log with just the committed entries. A small amount of metadata is also saved. This is the last included index ; the index of the last entry in the log that the snapshot replaces.

If a leader has discarded a log entry that was supposed to be sent to the followers later, the leader has to send snapshots to the servers that are lagging behind.

To implement snapshotting, the leader uses a new RPC which is called the InstallSnapshot RPC. When it sends this rpc to a follower, it means that its lagging too far and has to make a decision about what it wants to do with the existing entries.

If the follower finds that the snapshot has log entries that aren't in its own, it might discard the entire log. If there are any cases where a snap-shot that a prefix of its log , then log entries covered by the snap- shot are deleted but entries following the snapshot are still valid and must be retained.

It is important to note that snapshotting deviates from the strong leader principle as the followers may independently take snapshots without the leader's agreement.

4. Follower and candidate crashes

In cases where a follower or candidate crashes, then the RequestVote and AppendEntries RPCs sent to it will fail. The failures are handled by sending the appendEntries RPC indefinitely. When it restarts, the RPC completes successfully. If a follower crashes after it has received the RPC but crashed before it could respond to it, the RPC is received again when it restarts.

5. Cluster Membership Changes

In practical application, the cluster configuration changes in cases of server failure. One way to handle this is by taking the entire cluster off-line, updating configuration files, and then restarting the cluster. But this would lead to unavailabity during this time. This is handled by incorporating the "at most one leader per term" rule.

Raft introduces one more phase before the configuration change where the new servers join as non voting members and only proceed with voting when all the state machines have been updated to match the leader's logs. Another issue is when the cluster leader is left out from the new configuration. The leader has to step down and become a follower after committing the new log entries.

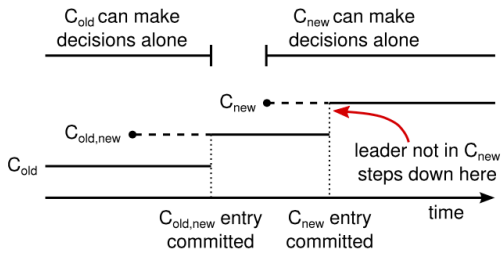

Fig4 : The leader makes changes using the joint consensus algorithm. At no point can the Cold and Cnew make decisions indepenedently.

The third issue that can come up is that once a server is removed it might disrupt the cluster. They don't receive the heartbeat messages and because of this they time out and start new elections. With this they send the requestvote RPC and the current leader goes back to being the follower again. A leader is then elected again

but the removed servers continue causing this disruption. To make sure that there's no unavailability because of this the RequestVote RPCs are disregarded if the current leader exists.

6. Implementation of raft protocol

The implementation for the RAFT protocol was written using C++ running on a Linux system. The system ran locally, but could easily be deployed to a network of machines. The system consists of a Coordinator and a set of Nodes. The Coordinator is responsible for spawning and tracking the Nodes, while each Node is responsible for performing the RAFT protocol.

The system starts out by creating a number of nodes that is specified by the user. Each node is created and set to default values. Then, each node is updated with the address and ports of the other nodes so that all of the nodes can communicate with each other over socket communications. Then, the Coordinator starts each Node as their own processes and threads. Once each node is running, the Coordinator stays on it's original thread to accept commands from the user. The user can input a message to add to the log or issue a command to start and stop nodes. The Coordinator finds the leader node to commit the message to and the Node is responsible for using the RAFT protocol to update the other follower nodes.

Each node is running on it's own thread. The follower nodes are setup to accept incoming RPCs for the other nodes. If the node is a leader, then the incoming messages are discarded because it won't need to respond to them. The nodes keep track of all of the other nodes in the system, but not if they are running or now. The protocol specifies what do to in the event of a leader or follower failing, so the communication protocol doesn't need to keep track. The nodes operate off of two different timeout values. The election timeout is the amount of time each node will wait between receiving RPCs to initiate a new election.

For each node, this value is randomized between 150ms and 300ms. The leader also has an extra timeout that is the time between sending AppendEntries RPCs as a heartbeat to the other nodes to prevent them from starting a new election. This value is set to 50ms to make sure that it always less than the election timeout.

One of the difficulties of implementing this protocol was getting information back from the Nodes as they operated. We included a logging system so that the user can see what all of the Nodes are doing, but there was an issue. Because all of the nodes are running as separate processes on different threads, there is the possibility that they write to STDOUT at the same time and their messages can write basically on top of each other. Because of the we created a thread safe logging class that reads in the messages into a queue before writing them to SDTOUT. This makes sure that not only the message are displayed to the user in the format they were specified in, but that they are displayed in exactly the order they were generated in. This is extremely helpful for seeing the order of events that takes place in the protocol.

The other issue we felt with in the implementation also has to do with showing the progress of the protocol. The RAFT protocol works very quickly and efficiently, so we included a slow down to the protocol so that the output could actually see what the protocol was doing in real time. This makes it easier to debug and demo for our presentation. For future use, all that would need to happen is remove the slow down constant and allow the protocol to run at it's full speed.

8. Conclusion

This protocol aims at achieving understandability. The decomposition of mechanism to segregate leader election for log replication and safety makes it easy to implement the raft protocol. the leaders are elected on an on- term basis and it supports all kinds of changes in the cluster configuration. Raft has very few message types ie 4 RPCs which is lesser than any other consensus algorithm.

A very interesting part of the algorithm is deciding upon the commit point for an entry for which the log entry would persist even if the servers fail. The nodes send their log length and term number in order to elect a leader from the available candidates. The log entry that is on majority of the nodes might sometimes get overwritten when a new leader is elected. To check this, an entry is considered to be committed only if it is stored on majority of the servers.

When it comes to performance, entries can be added with (f+1) roundtrips whereas for Paxos its 2(f+1).

There's also need to consider disk corruptions and log compactions so that there is less dependence on the consensus algorithm. Adding these features would definitely add to the complexity of the protocol thereby making it a little less understandable.

References:

[1] Diego Ongaro and John Ousterhout Stanford University, In Search of an Understandable Consensus Algorithm in Proc. USENIX Annual Technical Conference, 2014

[2] Raft consensus algorithm website: http://raftconsensus.github.io.

[3] Raft user study : http://ramcloud.stanford.edu/ongaro/userstudy/.

[4] distributed systems study group : http://dsrg.pdos.csail.mit.edu/2013/05/23/raft/

[5] engineering health : http://engineering.cerner.com/2014/01/the-raft-protocol-a-better-paxos/

[6] Paxos made simple : http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf