

Summary

For the project, I implemented Chord on top of RAFT. In particular, the nodes in the Chord ring are RAFT clusters. I was inspired by Google's hierarchical approach and decided to create my own. To create RAFT, I followed the *In Search of an Understandable Consensus Algorithm (Extended Version)* and to create Chord, I implemented Chord from the *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, specifically, I implemented the stabilization algorithm that can handle multiple concurrent joins. The design is to be a CP system with low latency compared to just running RAFT. In particular, the system can have 100s RAFT nodes but each RAFT node will only have to communicate with the RAFT nodes in its cluster (a small number, but probably 3).

In this paper, I will go through the process I went through for creating the project. First, I created the base **ChordNode** that can handle routing and joins. Next, I created **RaftNode**, a node in a RAFT Cluster. Finally, I integrated **RaftNode** as a storage engine for Chord, meaning Chord stores state in Raft. The first sections consist primarily of citations of the corresponding papers, noting the methods of **RaftNode** and **ChordNode** respectively that implement the given functionality. The final section describes the creation of **RaftPersistentKeyValueStore** and the refactoring of **ChordNode** and **RaftNode** to allow for the use of **Raft** as a Storage engine.

The Infrastructure

First, I created the **Message** class, which represents a message being sent over the network, and **MessageSender** class, which sends instances of the **Message** class over the network.

Next, I created a way to block a function until a specific message was received (timeout if necessary). This tool is the **AsyncScheduler**. Because it concerns the sending and receiving of messages, the **AsyncScheduler** only deals with async functions. Only async functions can be registered on it, although it does return normal functions.

AsyncScheduler exposes 4 main functions to clients:

- **Wait_for** : (activation_condition: Message -> bool) -> future
 - returns a future that is resolved when a message `msg` is received so that activation_condition(msg) is true. The future's result is this `msg`.
- **Wait_for_with_timeout**: (activation_condition: Message -> bool, timeout: float) -> future
 - Same as wait_for but future's result is None if the timeout expires.

- `schedule` : (monitor_job: function, interval_time: float) -> fn
 - Schedule `monitor_job` to be run every `interval_time`.
 - Returns a function to cancel the job .
- `Register_handler` : (handler: NamedTuple(activation_condition: Message -> Bool, handle_method:function)) -> fn
 - This registers the `activation_condition` to be executed every time a message that satisfies `activation_condition` is received.

The **AsyncScheduler** is the first piece of software to handle a message when it comes in. It dispatches the msg to the relevant functions and handlers waiting for it. Under the hood it maintains two lists, one of the waiting functions and another of the handlers. I call async functions that are executed periodically (every n seconds for some n) `monitors`.

Making Chord

First, I made Chord with node joins and stabilization. For simplicity sake, I decided to name Chord Nodes their address in the ring. To the right is a sample of a `chistributed.conf` from this stage of development.

```
chistributed.conf
1 node-executable: python3 network_node.py
2 nodes: 100, 500, 700
3
```

I divided the Chord node on the ring into 4 Main parts:

1. The Chord Client (the class **ChordNode**)
2. The Chord Router (the classes **SuccRouter** and **FingerTableRouter**)
3. The Key Transferer (the class **KeyTransferer**)
4. The Persistent Storage (the classes **PersistentKeyValueStore** and **RaftPersistentKeyValueStore**)

Persistent Storage

The Persistent Storage is used to set and get keys from storage.

- At this point, it was implemented as **PersistentKeyValueStore**, a dictionary and exposed 2 main functions: get, set.

Chord Router

The Chord Router is used to find the Chord Node responsible for a given key.

- In particular, the Chord Router exposes **find_node_responsible_for_key(key)**, an asynchronous function that searches the Chord ring for the node responsible for the key.
- The Chord Router maintains all information related to routing.

Chord Client

The Chord Client processes client requests for “get” and “set”

- The Chord Client executes the following functionality and restarts if it doesn't complete in a reasonable time for a ‘get’ or ‘set’:
 - The Chord Client uses the router to find the Chord node in the ring responsible for the node.
 - Next, the Chord Client communicates with that node's Chord Client to either get the value of a key or set the value of a key in PersistentStorage

Key Transferer

The Key Transferer processes the transfer of keys between nodes

- The Key Transferer is initiated whenever the predecessor is changed, since this changes the space the current node is responsible for.
- In particular, when the predecessor is changed to p_n from p_o , the Key Transferer transfers all data in $(p_o, p_n]$ maintained by the current node to p_n
 - There are edge cases when there is no predecessor or the predecessor is ourself. These are handled as you would expect by sending the appropriate interval we are no longer responsible for.

Routers

SuccRouter

First, I made a Router that routes only using successor and maintains predecessors. The predecessors are maintained to transfer keys between Chord Nodes. This followed the methodology of the Chord Paper in changing the successor and predecessor. In particular, I implemented the stabilization approach:

Routing (**find_node_responsible_for_addr**) was done by going around the ring asking nodes for their successor until we found the node immediately after or at the given address. I used an iterative rather than recursive approach.

```

n.join(n')
  predecessor = nil;
  successor = n'.find_successor(n);

// periodically verify n's immediate successor,
// and tell the successor about n.
n.stabilize()
  x = successor.predecessor;
  if (x ∈ (n, successor))
    successor = x;
  successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
  if (predecessor is nil or n' ∈ (predecessor, n))
    predecessor = n';

```

I implemented these functions on the **SuccRouter** as:

- n.find_successor is **find_node_responsible_for_addr**.
- n.notify is **notify_i_might_be_your_predecessor**.
- n.join is **startup**.

FingerTableRouter

Next, I made a finger table router.

This was an extension of **SuccRouter** that uses a finger table.

I implemented the following algorithms from the Chord Paper:

```

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
  finger[1].node = n'.find_successor(finger[1].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for i = 1 to m - 1
    if (finger[i + 1].start ∈ [n, finger[i].node))
      finger[i + 1].node = finger[i].node;
    else
      finger[i + 1].node =
        n'.find_successor(finger[i + 1].start)

// ask node n to find id's predecessor
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.successor])
    n' = n'.closest_preceding_finger(id);
  return n';

// return closest finger preceding id
n.closest_preceding_finger(id)
  for i = m downto 1
    if (finger[i].node ∈ (n, id))
      return finger[i].node;
  return n;

// periodically refresh finger table entries.
n.fix_fingers()
  i = random index > 1 into finger[];
  finger[i].node = find_successor(finger[i].start);

```

I implemented these functions on the **FingerTableRouter** as:

- `n.find_successor` is **find_node_responsible_for_addr**.
- `n.notify` is **notify_i_might_be_your_predecessor**.
- `n.join` is **startup**.
- `n.closest_preceding_finger` is **closest_preceding_finger_node**.
- `n.init_finger_table` is **init_finger_table**

To maintain the finger table, I implemented **fix_fingers** and registered it as a monitor job. However, unlike the implementation in Chord which choses a random finger, I replace every finger in the table with every call to `fix_fingers` by using the successor.

Routing(**find_node_responsible_for_addr**) was done using the iterative approach of the algorithm:

SuccList

Before I implemented RAFT, I wanted to see if I could make a successor list and replication work. I created a tool to do this. It maintains a successor list and periodically replaces the successor in the router with a new successor if an old successor has died. This will allow `fix_fingers` to eventually correct the finger table and routing to eventually work.

- The **SuccList** runs a monitor job that sends out heart beat requests to the **SuccList** of every successor in the successor list.
- If a successor dies, the node is removed from the list (shifting the list left) and a new node is added to the end of the list by finding the successor of the last node in the list
- If there are not enough nodes in the network to fill the list, some entries will be None.

The **SuccList** also had the configuration option to `config.USING_SUCCESSION_LIST_REPL` that would replicate the interval the **ChordNode** is responsible to peers. In particular,

- Every time data is set in ChordClient to persistent storage, **SuccList** replicates the data to every successor.
- Every time a new successor is added to the successor list, **SuccList** replicates the keys in `(self.predecessor, self.chord_addr]` to the node using the **KeyTransferer**. There is an edge case when `self.predecessor` is None that is handled by sending all of the data.

RAFT

I implemented RAFT without new node joins. I followed the algorithms laid out on page 4 of the RAFT paper and it worked. These algorithms are in the appendix of the paper.

However, I didn't want the log to be indexed at 1 like they are in the paper, so I made the first element of the log be 0.

To accomplish this I:

- changed the initialized of last_applied to be -1
- changed the initialized of commit_index to be -1
- changed the initialized of match_index for each peer to be -1

When a leader gets a majority of votes, we initialize the match_index and next_index tables. Other than that, I just followed the algorithm on what to do when a new message is received. There were only 2 types of messages (REQUEST_VOTE and APPEND_ENTRIES) and so I didn't really have to do anything special.

When a REQUEST_VOTE is received, I follow the specs in RequestVote RPC.

When an APPEND_ENTRIES is received, I follow the specs in AppendEntries RPC:

Each FOLLOWER/CANDIDATE runs a timer that waits to start a new election nominating themselves. Every time an APPEND_ENTRIES message is received, we restart the timer. When the timer goes off it follows the spec in Candidates (5.2):

Leaders

Leaders send append_entries to clients using **reconcile_logs_with_peers()**. To make sure that only one reconcile is happening at a time, each time a new reconcile is started, the old reconcile is ended. To start a new reconcile, the node calls **start_new_append_entries()**. They follow the following spec:

Leaders run a monitor job that runs periodically and calls **start_new_append_entries()**. This sends an empty append entry if every node is up to date and prevents anyone else from nominating themselves to be leader.

To incorporate the setting of match indexes, I created a new function **set_match_index(peer, match_index)**, which would set the match index for a peer and then try to update the commit index of the leader if it could following the spec:

- If there exists an N such that $N > \text{commitIndex}$, a majority of $\text{matchIndex}[i] \geq N$, and $\text{log}[N].\text{term} == \text{currentTerm}$:
set $\text{commitIndex} = N$ (§5.3, §5.4).

Finally, I implemented a function **set_commit_index(comm_index)**, which increments last_applied to apply all log entries to PersistentStorage that must be applied when we set the new commit index, all non applied entries less than the new commit index.

Gets/Sets

The final piece of the puzzle was making so that set and get requests worked.

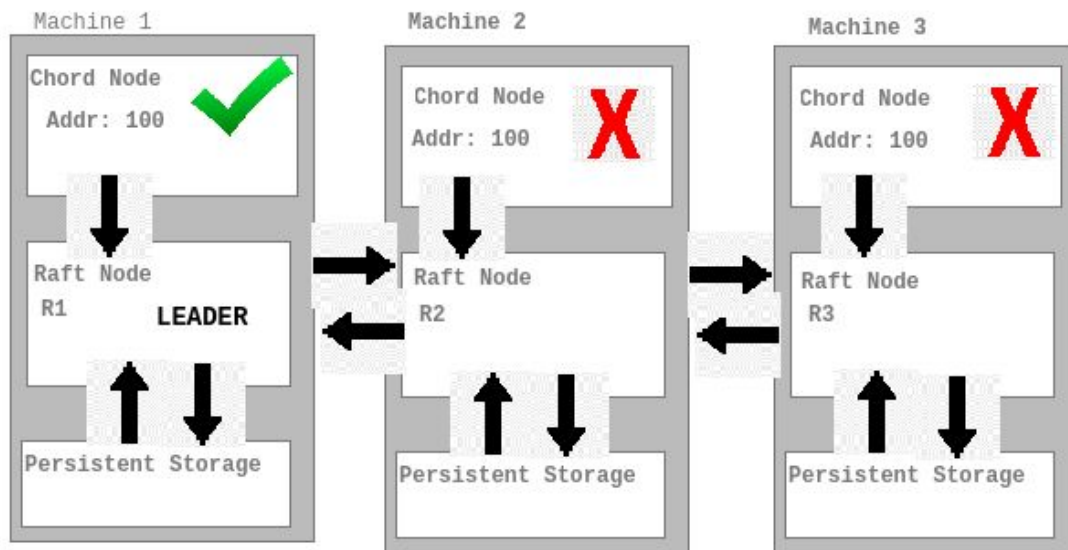
To do this, I implemented the following functionality on the receipt of a GET:

- If we are leader, return the value of the key by fetching from PersistentStorage
- If we are not the leader, return an error alerting the user that we are not the leader and cannot process requests.

I implemented the following functionality on the receipt of a SET:

- I append the entry to the log, and wait until my commit_index is at least the index of that entry
 - To achieve this, I store a list of waiting sets (self.commit_index_waiters) and every time a new commit index is set I go through this list and mark futures (resolve them by giving them a result) that are waiting for an index that has now been committed.

Implementing RAFT With Chord



The Final Design

Each machine starts up with a **RaftNode** and **PersistentStorage** which is the key value store for this raft node. When a **RaftNode** becomes leader it spawns a **ChordNode**. Hence, as in the figure, each machine can be seen as a **ChordNode** on top of **RaftNode** (through **RaftPersistentKeyValueStore**) on top of a **PersistentKeyValueStore** (in-memory dictionary) where the **ChordNode** is active if the **RaftNode** is leader.

The Journey

First, I needed to figure out how to make this work in terms of protocols. After thinking long and hard, I realized that I actually had set up the project perfectly to do this. I could use Raft

as a plug in service. In particular, I could make a new **PersistentStorage** object that would store data in a **RAFT** cluster.

I replaced **PersistentKeyValueStore**, which stores data in an in-memory dictionary, with a new **RaftPersistentKeyValueStore**, which stores data in a RAFT cluster. The **RaftPersistentKeyValueStore** exposes all the same methods as the in-memory **PersistentKeyValueStore** but stores data in a RAFT cluster. The only difference was that now gets and sets are **asynchronous**, as well as all other operations.

With this, I had almost solved my problem. I used **RaftPersistentKeyValueStore** as the **PersistentStore** for **KeyTransferer** and **ChordNode**. Now, I was able to store data and transfer data between **ChordNodes** using **RAFT**.

However, I had a new problem. There should only be one chord node for a given address at a time. In particular, if a cluster of Raft Nodes R1, R2, R3 represent a chord node across different systems, there should only be 1 **ChordNode** structure fielding requests at a time, the **ChordNode** of the leader. To do this, I added 2 new parameters to a **RaftNode**, **on_become_leader()**, **on_step_down()**. This allowed me to run a function on the machine of a **RaftNode** when it becomes leader and when it steps down from being the leader. To accomplish this, I made a new function **set_role(role)** that a **RaftNode** uses to change its role. Now, all state changes are in one place so it's easy to call **on_become_leader()** when the node's role is changed to Leader from Follower or Candidate and **on_step_down()** when the role is changed from Leader to Follower or Candidate. Now, I made it so that when the **RaftNode** of a machine becomes leader, a **ChordNode** is spawned up. Because RAFT ensures only one leader in a given term, this ensures that there is only one **ChordNode** as long as every node is on the same term. If no nodes are allowed to recover, this ensures that there will be only one leader at once since the case where there would be 2 leaders is when one leader recovers after failing.

However, now a Chord Node could use **RaftPersistentKeyValueStore** to store data. I refactored the **ChordNode** to store its state completely using **RaftPersistentKeyValueStore** (the finger table, successor and predecessor) and now when the machine the Leader goes down on dies another node will become the leader and using the **RaftPersistentKeyValueStore** will be able to start another **ChordNode** with exactly the same state.

Chistributed representation

To set up the previously mentioned system with chistributed I encoded the state of a machine in the name of the node (node in this case being a chistributed node). In particular, the node name has the chord_addr and the name of all peer **RaftNode** ids and the Raft Id of the current machine. The first raft id is the raft id of the machine and the next raft ids are those of the peers.

For example, 100:R1,R2,R3 indicates:

- The machine is a part of the RAFT cluster that represents the address 100 in the Chord Ring.
- The Raft Id of the machine is R1
- The peers of this node in its RAFT cluster is R2,R3

Similarly, 100:R2,R1,R3 indicates the machine with raft id R2 in the same cluster.

```
100:R1,R2,R3
100:R2,R1,R3
100:R3,R1,R2
300:R10,R11,R12
300:R11,R10,R12
300:R12,R10,R11
500:R20,R21,R22,R23
500:R21,R20,R22,R23
500:R22,R20,R21,R23
500:R23,R20,R21,R22
```

The distributed node names of a chord ring with 3 RAFT clusters:

1. The chord node with address 100 consists of R1, R2, R3
2. The chord node with address 300 consists of R10, R11, R12
3. The chord node with address 500 consists of R20, R21, R22, R23

Each machine then upon spawn constructs a **RaftNode** (indirectly through creating RaftKeyValueStore) that handles any messages intended for the Raft address of the machine. When the **RaftNode** becomes the leader, it constructs a **ChordNode** that responds to messages intended for the chord address that the **RaftNode** represents. When the **RaftNode** steps down from being leader, it cancels its **ChordNode**. This ensures that there is never more than one **ChordNode** for a given address for an extended period of time (there might be more than one if there are 2 different leaders for different terms still alive and partitioned from each other)

Fault Tolerance

The fault tolerance of this algorithm is basically RAFT. This is a CP system.

In particular, in any given raft cluster, a node can fail-stop. As it is currently, the code does not support fail recoveries. If a node recovers and it is the leader, this might make it so that there are two active leaders at once (in different RAFT terms), and so two active **ChordNodes**. To allow for fail-recoveries, the code would need to be refactored to use quorum reads which would not allow the **ChordNode** to service clients without a quorum (indicating it is the only leader).

If a machine dies, the Raft Node at that machine will stop communicating with other nodes in its cluster. Then, a new leader will be elected. This leader will then spawn up a new **ChordNode** with the same configuration as the previous nodes. If enough nodes die that there is not able to be a quorum, we consider the whole system to be compromised and ‘get’ and ‘set’ requests that must use this node for routing will return an error indicating that the server is down.

We ensure that every get and set will be processed by forcing a request to be made at a leader and repeating them until a response is received. If a request is received at a node not the leader, it returns an error indicating that it is not the leader. If a node is a leader and attempts to serve the request, the node might not get a response due to the network stabilizing, which would be the case if **fix_fingers()** or **stabalize()** are running or another Raft cluster is electing a new leader. If the request times out enough times, this indicates that a Raft cluster is offline and more than the allowable amount of nodes have died, and so there is an error among the service and an error is returned.

If there is a partition between all the nodes in some raft cluster and some other cluster, the system will respond the same as in the previous paragraph and will not route new messages.

Test Cases

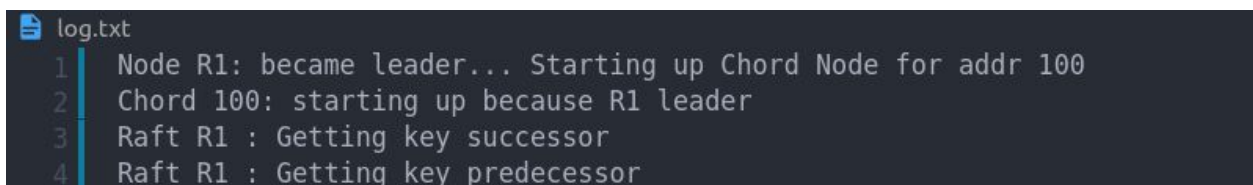
Note: in running the test cases, there will be alot of warnings about Messages with unknown destinations. This is because nodes are listening on sockets that are not their node_names but rather their raft address or chord address. You can safely ignore these.

Test Case 1

In this test case, we will test the setting and getting of a value on a raft cluster and how it reacts to a node that is not the leader fail-stopping.

In this test we will run 1 cluster of 3 raft nodes. This will be to test what happens with a fail-stop of a node that is not the leader in RAFT.

First, we start up the cluster. We can see an leader election happen and R1 becomes the leader. Because it is the leader, it starts up a chord node.



```
log.txt
1 Node R1: became leader... Starting up Chord Node for addr 100
2 Chord 100: starting up because R1 leader
3 Raft R1 : Getting key successor
4 Raft R1 : Getting key predecessor
```

Afterwards, we can see various operations that are occurring because the Chord Node is setting up and persisting its state to the cluster. We will not focus on this here.

After all these operations, we see the log stabilize. At this point, every node is up to date and so what we are seeing is the `append_entries` heartbeat going out to each node again and again to ensure no new election happens.

```
49 Raft R2 : Applying log entry 10 setting finger_table[9] to 100
50 Raft R3 : Applying log entry 10 setting finger_table[9] to 100
51 Raft R1 : Reconciling log with R2 complete
52 Raft R1 : Reconciling log with R3 complete
53 Raft R1 : Reconciling log with R2 complete
54 Raft R1 : Reconciling log with R3 complete
55 Raft R1 : Reconciling log with R2 complete
56 Raft R1 : Reconciling log with R3 complete
57 Raft R1 : Reconciling log with R2 complete
58 Raft R1 : Reconciling log with R3 complete
59 Raft R1 : Reconciling log with R2 complete
60 Raft R1 : Reconciling log with R3 complete
61 Raft R1 : Reconciling log with R2 complete
62 Raft R1 : Reconciling log with R3 complete
```

Next, we try to set the value of a key when we are not the leader. We get an error:

```
> set -n 100:R2,R1,R3 -k dan -v rocks
ERROR: SET id=1 Failed (dan-rocks): Cannot process request. We are not the leader
```

We can see the logs of why this happened. The node R2 did not have a chord node spawned because a chord node is only spawned when a node is the leader. Since R2 is not the leader, it did not have one spawned, so rejected the request.

```
67 NetworkNode 100:R2,R1,R3 :got command from client {'id': 1, 'key': 'dan', 'value
68 100:R2,R1,R3 got set with chord node available False
69 NetworkNode: get request for 100:R2,R1,R3 but we're not leader
```

Next, we can set the value of a key with the leader. We see it completes successfully.

```
> set -n 100:R1,R2,R3 -k dan -v rocks
2020-06-05 19:14:05,588 [chistributed] WARNING: Message with unknown destination R2
2020-06-05 19:14:05,589 [chistributed] WARNING: Message with unknown destination R3
> 2020-06-05 19:14:05,589 [chistributed] WARNING: Message with unknown destination R1
2020-06-05 19:14:05,590 [chistributed] WARNING: Message with unknown destination R1
2020-06-05 19:14:05,590 [chistributed] WARNING: Message with unknown destination R2
2020-06-05 19:14:05,590 [chistributed] WARNING: Message with unknown destination R3
2020-06-05 19:14:05,591 [chistributed] WARNING: Message with unknown destination R1
2020-06-05 19:14:05,591 [chistributed] WARNING: Message with unknown destination R1
SET id=2 OK: dan = rocks
```

If we look at the logs we can see how this happens:

```
78 NetworkNode 100:R1,R2,R3 :got command from client {'id': 2, 'key': 'dan', 'value': 'rocks'}
79 100:R1,R2,R3 got set with chord node available True
80 Chord 100 : set request key dan to value rocks with hash 846
81 Chord 100 : found node responsible for dan 100
82 Chord 100 : setting dan to value rocks
83 Raft R1 : Reconciling log with R2 complete
84 Raft R1 : Applying log entry 11 setting dan to rocks
85 Chord 100 : setting dan to value rocks complete
86 Raft R1 : Reconciling log with R3 complete
87 Raft R2 : Applying log entry 11 setting dan to rocks
88 Raft R3 : Applying log entry 11 setting dan to rocks
89 Raft R1 : Reconciling log with R2 complete
90 Raft R1 : Reconciling log with R3 complete
91 Raft R1 : Reconciling log with R2 complete
92 Raft R1 : Reconciling log with R3 complete
```

First, we get the request. We find that we are the node responsible for this so we call Raft to set the value of the key to dan to be rocks.

Next, R1 creates a log entry for this, log entry 11 here.

Next, R1 sends out an APPEND_ENTRIES to every other node.

After line 83, we see that we caught R2 up to date. This means R2's log has this entry. Hence, a majority of the nodes have this entry. So, R1 is clear to increase the commit_index to 11 and apply the entry.

Concurrently, R3 is caught up to date.

Next, when the heartbeat APPEND_ENTRIES is sent, the commit_index is now 11 so R2 and R3 update their commit index and apply the log entry to their persistent storage.

Next, we get the key.

```
> get -n 100:R1,R2,R3 -k dan
GET id=3 OK: dan = rocks
```

```
93 NetworkNode 100:R1,R2,R3 :got command from client {'id': 3, 'key': 'dan'}
94 100:R1,R2,R3 got get with chord node available True
95 Chord 100 : found node responsible for dan 100
96 Raft R1 : Getting key dan
```

Chord 100, the ChordNode instance running on R1's machine fields the request, finds the node responsible for it is itself and gets the key from storage. Since we do not use a quorum get, a get request does not need a majority quorum.

Next, we fail the node R2.

```
> set -n 100:R2,R1,R3 -k die -v die
```

We see the node die and then R1 continues to send out heartbeat APPEND_ENTRIES but only receives responses from R3.

```

101 NetworkNode 100:R2,R1,R3 :got command from client {'id': 4, 'key': 'die', 'value': 'die'}
102 NetworkNode 100:R2,R1,R3 :got kill command from client {'id': 4, 'key': 'die', 'value': 'die'}
103 Raft Node R2 No longer alive
104 Raft R1 : Reconciling log with R3 complete
105 Raft R1 : Reconciling log with R3 complete

```

Next, we execute another get.

```

> get -n 100:R1,R2,R3 -k dan
GET id=5 OK: dan = rocks

```

```

108 NetworkNode 100:R1,R2,R3 :got command from client {'id': 5, 'key': 'dan'}
109 100:R1,R2,R3 got get with chord node available True
110 Chord 100 : found node responsible for dan 100
111 Raft R1 : Getting key dan

```

Next, we set the value of the key danny to rocks.

```

> set -n 100:R1,R2,R3 -k danny -v rocks
2020-06-05 19:14:37,329 [chistributed] WARNING: Message with unknown destination R2
2020-06-05 19:14:37,331 [chistributed] WARNING: Message with unknown destination R3
2020-06-05 19:14:37,335 [chistributed] WARNING: Message with unknown destination R1
2020-06-05 19:14:40,329 [chistributed] WARNING: Message with unknown destination R2
2020-06-05 19:14:40,331 [chistributed] WARNING: Message with unknown destination R3
2020-06-05 19:14:40,335 [chistributed] WARNING: Message with unknown destination R1
2020-06-05 19:14:43,331 [chistributed] WARNING: Message with unknown destination R2
2020-06-05 19:14:43,332 [chistributed] WARNING: Message with unknown destination R3
2020-06-05 19:14:43,333 [chistributed] WARNING: Message with unknown destination R1
2020-06-05 19:14:43,669 [chistributed] WARNING: Message with unknown destination R2
> 2020-06-05 19:14:43,670 [chistributed] WARNING: Message with unknown destination R3
2020-06-05 19:14:43,671 [chistributed] WARNING: Message with unknown destination R1
2020-06-05 19:14:43,672 [chistributed] WARNING: Message with unknown destination R3
2020-06-05 19:14:43,672 [chistributed] WARNING: Message with unknown destination R1
SET id=6 OK: danny = rocks

```

If we look at the logs:

```

117 NetworkNode 100:R1,R2,R3 :got command from client {'id': 6, 'key': 'danny', 'value': 'rocks'}
118 100:R1,R2,R3 got set with chord node available True
119 Chord 100 : set request key danny to value rocks with hash 949
120 Chord 100 : found node responsible for danny 100
121 Chord 100 : setting danny to value rocks
122 Raft R1 : Reconciling log with R3 complete
123 Raft R1 : Applying log entry 12 setting danny to rocks
124 Chord 100 : setting danny to value rocks complete
125 Raft R3 : Applying log entry 12 setting danny to rocks
126 Raft R1 : Reconciling log with R3 complete
127 Raft R1 : Reconciling log with R3 complete

```

We see that the Chord Node fields the request and then sets the value of danny to rocks in the Raft cluster. R1 appends the entry to its log (entry 12 this time) and sends out a request to reconcile its log with R3. Once it's completed, it knows that R3 has entry 12 into its log and so a majority of nodes have entry 12 in its log and so R1 can commit the entry. R1 sets the commit index to be 12 and applies the entry. Now, the next heartbeat append_entry R3 increases its commit index to be 12 and applies entry 12.

Finally, we try to get the value of danny and find the correct answer.

```

> get -n 100:R1,R2,R3 -k danny
GET id=7 OK: danny = rocks

```



```

128 NetworkNode 100:R1,R2,R3 :got command from client {'id': 7, 'key': 'danny'}
129 100:R1,R2,R3 got get with chord node available True
130 Chord 100 : found node responsible for danny 100
131 Raft R1 : Getting key danny
132 Raft R1 : Reconciling log with R3 complete

```

Test Case 2

In this test case, we run 3 clusters of raft nodes:

1. A cluster of Raft Node R1, R2, R3 representing address 100
2. A cluster of Raft Node R10, R11, R12 representing address 300
3. A cluster of Raft Node R20, R21, R22 representing address 500

In this test case, we first start all of the nodes by running the follow start script

```

start-nodes.chi
1 start -n 100:R1,R2,R3
2 start -n 100:R2,R1,R3
3 start -n 100:R3,R1,R2
4 start -n 300:R10,R11,R12
5 start -n 300:R11,R10,R12
6 start -n 300:R12,R10,R11
7 start -n 500:R20,R21,R22,R23
8 start -n 500:R21,R20,R22,R23
9 start -n 500:R22,R20,R21,R23
10 start -n 500:R23,R20,R21,R22

```

To see the results of the test we look at `log.txt` and see the output.

After waiting some time, we see that each cluster has elected a leader by looking through the log:

```

1 Node R3: became leader... Starting up Chord Node for addr 100

```

For this first leader, we can see that the node eventually got the right successor and predecessor and that these were also saved to the replicas

```

Raft R3 : Applying log entry 14 setting predecessor to 500
Raft R3 : Applying log entry 11 setting successor to 300

```

```

Raft R1 : Applying log entry 11 setting successor to 300
Raft R2 : Applying log entry 11 setting successor to 300

```

```

Raft R1 : Applying log entry 14 setting predecessor to 500
Raft R2 : Applying log entry 14 setting predecessor to 500

```

We can also see that the finger table is saved,

```

Raft R1 : Applying log entry 0 setting successor to 100
Raft R1 : Applying log entry 1 setting finger_table[0] to 100
Raft R1 : Applying log entry 2 setting finger_table[1] to 100
Raft R1 : Applying log entry 3 setting finger_table[2] to 100
Raft R1 : Applying log entry 4 setting finger_table[3] to 100
Raft R1 : Applying log entry 5 setting finger_table[4] to 100
Raft R1 : Applying log entry 6 setting finger_table[5] to 100
Raft R1 : Applying log entry 7 setting finger_table[6] to 100
Raft R1 : Applying log entry 8 setting finger_table[7] to 100
Raft R1 : Applying log entry 9 setting finger_table[8] to 100
Raft R1 : Applying log entry 10 setting finger_table[9] to 100
Raft R1 : Applying log entry 11 setting successor to 300
Raft R1 : Applying log entry 12 setting predecessor to 300
Raft R1 : Applying log entry 13 setting finger_table[0] to 300
Raft R1 : Applying log entry 14 setting predecessor to 500
Raft R1 : Applying log entry 15 setting finger_table[1] to 300
Raft R1 : Applying log entry 16 setting finger_table[2] to 300
Raft R1 : Applying log entry 17 setting finger_table[3] to 300
Raft R1 : Applying log entry 18 setting finger_table[4] to 300
Raft R1 : Applying log entry 19 setting finger_table[5] to 300
Raft R1 : Applying log entry 20 setting finger_table[6] to 300
Raft R1 : Applying log entry 21 setting finger_table[7] to 300
Raft R1 : Applying log entry 22 setting finger_table[8] to 500

```

```
46 | Node R10: became leader... Starting up Chord Node for addr 300
```

```
109 | Node R20: became leader... Starting up Chord Node for addr 500
```

Now, we run the following command to set the value of the key dan (hash value 863)

```
set -n 100:R1,R2,R3 -k dan -v rocks
```

And we see it run in the logs:

```

384 | NetworkNode 100:R3,R1,R2 :got command from client {'id': 1, 'key': 'dan', 'value': 'rocks'}
385 | 100:R3,R1,R2 got set with chord node available True
386 | Chord 100 : set request key dan to value rocks with hash 846
387 | Chord 100 : found node responsible for dan 100
388 | Chord 100 : setting dan to value rocks
389 | Raft R3 : Reconciling log with R1 complete
390 | Raft R3 : Applying log entry 23 setting dan to rocks
391 | Chord 100 : setting dan to value rocks complete

```

First, we see that this raft node indicates that its chord node is available in line 385 so it can process the request.

Next, we see that the hash value of the key dan is 846 and so the node responsible for it is actually the current node..

Because of this, the node does not need to communicate to any other nodes, and so it can set it directly to storage. Because of this, the node sends to directly to RaftStorage to be storage across the cluster.

The node sends an append entries request to R3 and updates R3 to have this new log and so it now a majority of replicas have the given log so it can be committed.

Now, we see the Chord node indicates that the set was successful and sends a setResponse back to the client:

```
> set -n 100:R3,R1,R2 -k dan -v rocks
2020-06-05 18:04:42,109 [chistributed] WARNING: Message with unknown destination 500
> 2020-06-05 18:04:42,111 [chistributed] WARNING: Message with unknown destination 100
2020-06-05 18:04:42,112 [chistributed] WARNING: Message with unknown destination R1
2020-06-05 18:04:42,113 [chistributed] WARNING: Message with unknown destination R2
2020-06-05 18:04:42,113 [chistributed] WARNING: Message with unknown destination R3
2020-06-05 18:04:42,114 [chistributed] WARNING: Message with unknown destination R3
2020-06-05 18:04:42,114 [chistributed] WARNING: Message with unknown destination R1
2020-06-05 18:04:42,115 [chistributed] WARNING: Message with unknown destination R2
2020-06-05 18:04:42,115 [chistributed] WARNING: Message with unknown destination R3
2020-06-05 18:04:42,116 [chistributed] WARNING: Message with unknown destination R3
SET id=1 OK: dan = rocks
```

Next, we fail the node R3 by executing the following command

```
set -n 100:R3,R1,R2 -k die -v die
```

This is the command to kill the Raft Node R3. We see in the log that the node dies (meaning it will no longer send or respond to messages):

```
479 NetworkNode 100:R3,R1,R2 :got command from client {'id': 2, 'key': 'die', 'value': 'die'}
480 NetworkNode 100:R3,R1,R2 :got kill command from client {'id': 2, 'key': 'die', 'value': 'die'}
481 Raft Node R3 No longer alive
```

Next, we wait for a leader election and see that R2 wins.

```
491 Raft R10 : Reconciling log with R12 complete
492 Node R2: became leader... Starting up Chord Node for addr 100
493 Chord 100: starting up because R2 leader
494 Raft R2 : Reconciling log with R1 complete
```

Now, we send a get request from another node to see if the value persisted

```
get -n 300:R11,R10,R12 -k dan
```

```
> get -n 300:R10,R11,R12 -k dan
2020-06-05 18:07:12,711 [chistributed] WARNING: Message with unknown destination 500
2020-06-05 18:07:12,712 [chistributed] WARNING: Message with unknown destination 300
2020-06-05 18:07:12,713 [chistributed] WARNING: Message with unknown destination 100
2020-06-05 18:07:12,714 [chistributed] WARNING: Message with unknown destination 300
2020-06-05 18:07:12,715 [chistributed] WARNING: Message with unknown destination 300
GET id=5 OK: dan = rocks
2020-06-05 18:07:13,380 [chistributed] WARNING: Message with unknown destination R21
```


We see that the request was successful, as the node got the request, routed it and got the answer from the appropriate node

```
1158 NetworkNode 300:R10,R11,R12 :got command from client {'id': 2, 'key': 'dan'}
1159 300:R10,R11,R12 got get with chord node available True
1160 Chord 300 : Found node responsible for key dan 100
1161 Raft R2 : Getting key dan
```

Test Case 3

This test will run the same configuration as Test 2 but will check for results in the presence of network partitions. In particular, we will shut down the first Raft cluster and ensure that the requests time out.

First, we find the leader of the Raft Group for chord address 100.

We find that Raft Node R2 is the leader.

```
1 Node R2: became leader... Starting up Chord Node for addr 100
```

We wait for the system to stabilize.

```
339 Raft R10 : Reconciling log with R11 complete
340 Raft R10 : Reconciling log with R12 complete
341 Raft R20 : Reconciling log with R21 complete
342 Raft R20 : Reconciling log with R22 complete
343 Raft R20 : Reconciling log with R23 complete
344 Raft R2 : Reconciling log with R1 complete
345 Raft R2 : Reconciling log with R3 complete
```

Next, we kill Raft Node R2 by running

```
set -n 100:R2,R1,R3 -k die -v die
```

```
346 NetworkNode 100:R2,R1,R3 :got command from client {'id': 1, 'key': 'die', 'value': 'die'}
347 NetworkNode 100:R2,R1,R3 :got kill command from client {'id': 1, 'key': 'die', 'value': 'die'}
348 Raft Node R2 No longer alive
```

We wait for the next leader to be elected.

```
361 Node R1: became leader... Starting up Chord Node for addr 100
```

So we kill the next leader, Node R1

```
set -n 100:R1,R2,R3 -k die -v die
```

```
400 NetworkNode 100:R1,R2,R3 :got command from client {'id': 2, 'key': 'die', 'value': 'die'}
401 NetworkNode 100:R1,R2,R3 :got kill command from client {'id': 2, 'key': 'die', 'value': 'die'}
402 Raft Node R1 No longer alive
```

Next, we try to get the value of a key that is in the address space of the first raft cluster, which has no leader.

```
get -n 300:R10,R11,R12 -k dan
```

```
> get -n 300:R10,R11,R12 -k dan
2020-06-05 21:50:53,140 [chistributed] WARNING: Message with unknown destination R11
2020-06-05 21:50:53,158 [chistributed] WARNING: Message with unknown destination R12
2020-06-05 21:50:53,164 [chistributed] WARNING: Message with unknown destination R10
2020-06-05 21:50:53,191 [chistributed] WARNING: Message with unknown destination R10
2020-06-05 21:50:53,203 [chistributed] WARNING: Message with unknown destination 500
> 2020-06-05 21:50:53,205 [chistributed] WARNING: Message with unknown destination 300
2020-06-05 21:50:53,206 [chistributed] WARNING: Message with unknown destination 100
```

Approximately a minute later,

```
ERROR: GET (id= failed (b=den): timed out
2020-06-05 21:51:13,216 [chistributed] WARNING: Message with unknown destination 100
2020-06-05 21:51:13,862 [chistributed] WARNING: Message with unknown destination R1
2020-06-05 21:51:13,863 [chistributed] WARNING: Message with unknown destination R2
2020-06-05 21:51:14,152 [chistributed] WARNING: Message with unknown destination R11
2020-06-05 21:51:14,154 [chistributed] WARNING: Message with unknown destination R12
2020-06-05 21:51:14,155 [chistributed] WARNING: Message with unknown destination R10
2020-06-05 21:51:14,158 [chistributed] WARNING: Message with unknown destination R10
```

Issues, Challenges, and Lessons Learned

Lessons Learned

I learned how harder it is to work on a large system in an untyped language. I had countless times where I would misspell a variable (sucessor instead of successor) and would only figure this out at run-time. Additionally, the lack of interfaces really made it hard to create a polymorphic storage engine. I could have used abstract classes but Python really isn't made to do that I think. In the end, I just implemented two classes that have the same functions. It would have been much easier to work in a typed setting on this and I think that python is good for writing scripts but very hard for large projects.

I learned that I should write unit tests for components that are maintained. I made unit tests for the ChordNode and Raft Node but when I combined the two, none of those tests could be run anymore because I had to change core functionality in both classes. I should have thought more about how to merge the two elements (projects in their own right) together.

When I was integrating Chord and Raft, my initial instinct for how to do it was very bad. Unlike most other times, I realized this time that it was very bad and thought there must be a better way. I went and relaxed for an hour and came up with a way that was much simpler and easier. I learned that I should think very hard about how to do things and consider many approaches before deciding on one approach.

Challenges

I had a really hard time working with chistributed at first. I couldn't figure out what was happening and I couldn't get it to work with python3. However, after searching for a while I was able to figure out how to make it work with python3. Additionally, I was able to understand how ZeroMQ works to be able to subscribe for messages from multiple addresses in one NetworkNode.

I had a really hard time figuring out how to make readable code. In particular, I needed code that could block until a message was received. I wanted to find something that looked like promises in javascript and found the asyncio library which has futures and an event loop (promises in javascript are futures and javascript is run on an event loop). Next, I learned how to use asyncio and it worked out really well!

When I implemented Raft, I didn't understand it 100%. In particular, I didn't understand the purpose of all the constants. I implemented the code thinking that the log starts at index 0 but was getting odd bugs. I looked at the paper again and realized that the paper dictates that the log starts at index 1. I realized I didn't completely understand the algorithm and so I stopped working and read the paper again to understand Raft completely before I implemented it. Once I understood it completely, things were much easier and I was able to adapt the algorithm to use my index starting at 0 method.

Appendix:

<p>State</p> <p>Persistent state on all servers: (Updated on stable storage before responding to RPCs)</p> <p>currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)</p> <p>votedFor candidateId that received vote in current term (or null if none)</p> <p>log[] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)</p> <p>Volatile state on all servers:</p> <p>commitIndex index of highest log entry known to be committed (initialized to 0, increases monotonically)</p> <p>lastApplied index of highest log entry applied to state machine (initialized to 0, increases monotonically)</p> <p>Volatile state on leaders: (Reinitialized after election)</p> <p>nextIndex[] for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)</p> <p>matchIndex[] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)</p>	<p>RequestVote RPC</p> <p>Invoked by candidates to gather votes (§5.2).</p> <p>Arguments:</p> <p>term candidate's term</p> <p>candidateId candidate requesting vote</p> <p>lastLogIndex index of candidate's last log entry (§5.4)</p> <p>lastLogTerm term of candidate's last log entry (§5.4)</p> <p>Results:</p> <p>term currentTerm, for candidate to update itself</p> <p>voteGranted true means candidate received vote</p> <p>Receiver implementation:</p> <ol style="list-style-type: none"> 1. Reply false if term < currentTerm (§5.1) 2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)
<p>AppendEntries RPC</p> <p>Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).</p> <p>Arguments:</p> <p>term leader's term</p> <p>leaderId so follower can redirect clients</p> <p>prevLogIndex index of log entry immediately preceding new ones</p> <p>prevLogTerm term of prevLogIndex entry</p> <p>entries[] log entries to store (empty for heartbeat; may send more than one for efficiency)</p> <p>leaderCommit leader's commitIndex</p> <p>Results:</p> <p>term currentTerm, for leader to update itself</p> <p>success true if follower contained entry matching prevLogIndex and prevLogTerm</p> <p>Receiver implementation:</p> <ol style="list-style-type: none"> 1. Reply false if term < currentTerm (§5.1) 2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3) 3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3) 4. Append any new entries not already in the log 5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry) 	<p>Rules for Servers</p> <p>All Servers:</p> <ul style="list-style-type: none"> • If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3) • If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1) <p>Followers (§5.2):</p> <ul style="list-style-type: none"> • Respond to RPCs from candidates and leaders • If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate <p>Candidates (§5.2):</p> <ul style="list-style-type: none"> • On conversion to candidate, start election: <ul style="list-style-type: none"> • Increment currentTerm • Vote for self • Reset election timer • Send RequestVote RPCs to all other servers • If votes received from majority of servers: become leader • If AppendEntries RPC received from new leader: convert to follower • If election timeout elapses: start new election <p>Leaders:</p> <ul style="list-style-type: none"> • Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2) • If command received from client: append entry to local log, respond after entry applied to state machine (§5.3) • If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex <ul style="list-style-type: none"> • If successful: update nextIndex and matchIndex for follower (§5.3) • If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3) • If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction). The server behavior in the upper-left box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §5.2 indicate where particular features are discussed. A formal specification [31] describes the algorithm more precisely.