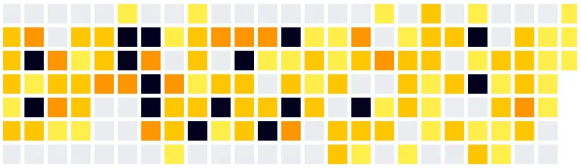
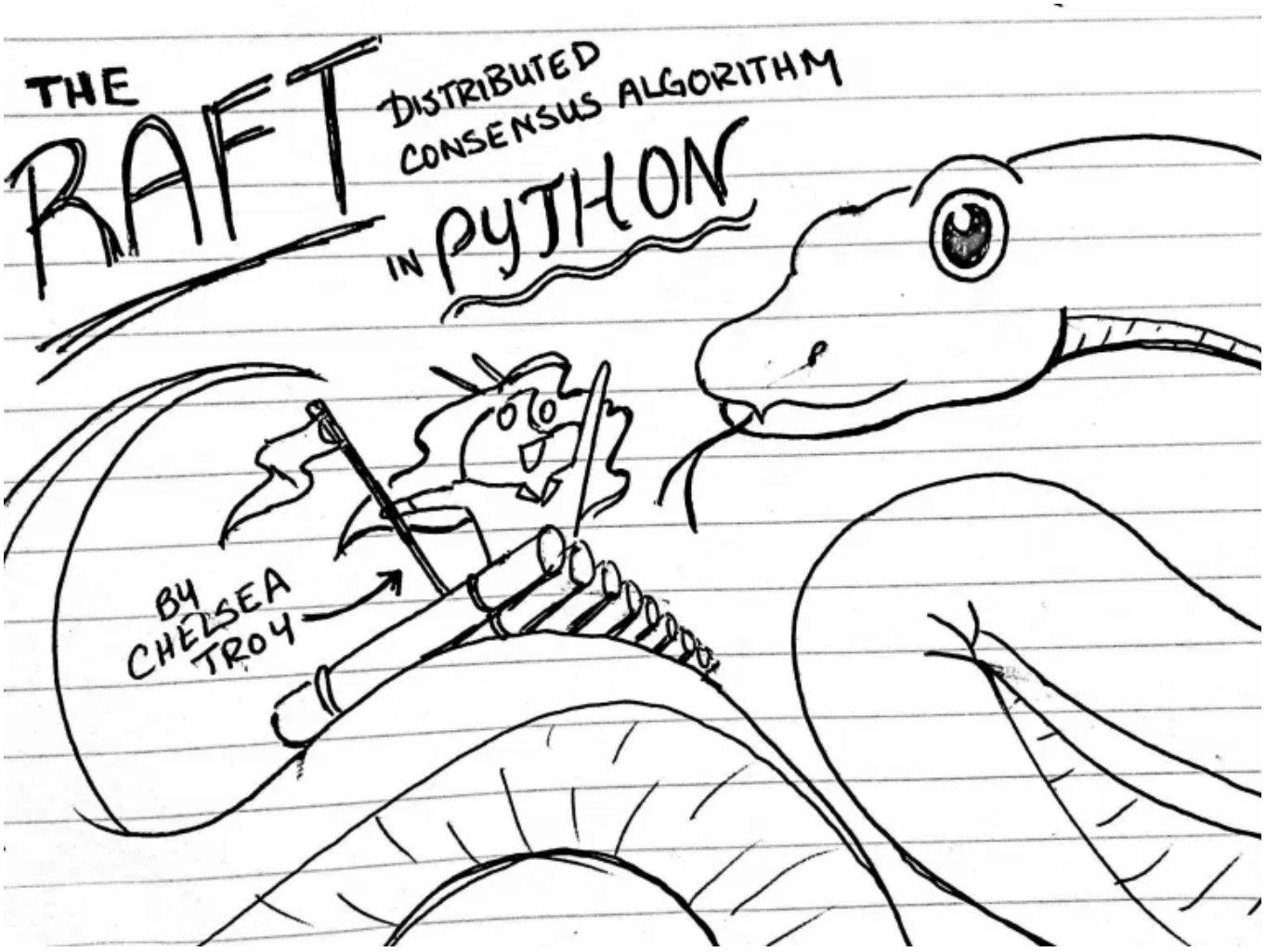


Chelsea Troy

Savvy software comes from savvy people.

Reading Time: 112 minutes



THEY SET OUT TO MAKE SERVICES WORK, EVEN WHEN SERVERS WERE BROKEN.

SHE SET OUT TO IMPLEMENT IT...AND TELL YOU THE TALE.

For Dave Beazley, whose "Rafting Trip" class got me started on this adventure.
To quote the man himself: come to Chicago and take a course.

Introduction

Suppose I'm building a server that sends me information about something that *absolutely has* to be timely and *absolutely has* to be correct: let's say, epidemic contact tracing information.

When a client submits a question about localized cases, we want the server to always respond with the correct answer. This means that the server needs to be:

1. Up. A downed server doesn't answer the clients' questions.
2. Up to date. If case data has changed since this server got the news, it cannot respond with the correct answers to case data requests.

It's a challenge to ensure that both these things are true at once. If one server keeps all the case information, then there's a single source of truth—but that server could go down. So we add redundancy: if there's a cluster of multiple servers, than the others can fill in if one server goes down. But that means we need to update the servers' information in such a way that they all always have the *same* data.

How do we achieve a single source of truth without a single point of failure?

It's not uncommon to use some sort of resilient distributed data store in client applications: think Redis or memcache. Those libraries would be implemented with a **distributed consensus algorithm:** a system designed to keep redundant data sources consistent with one another.

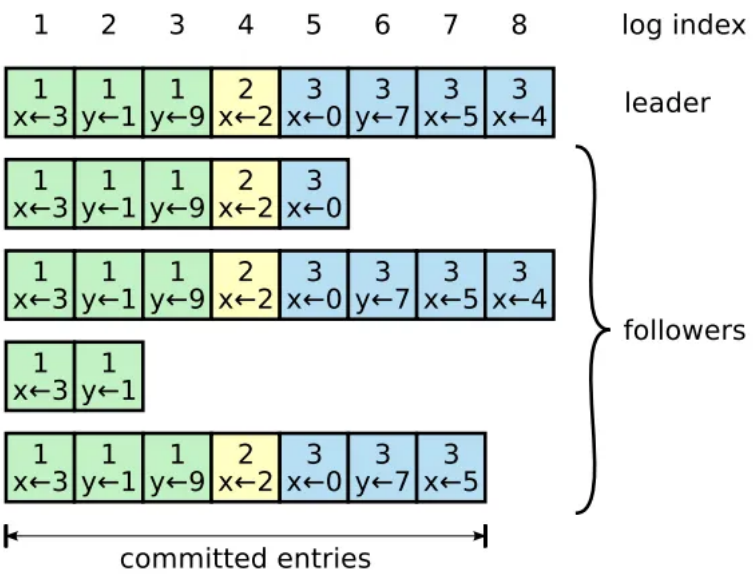
In the '80s, various programmers coalesced around a set of protocols later cemented in a design called Paxos. The original paper about Paxos, though, is notoriously obscure. Clarifying addenda came out later, but the algorithm's reputation for convolution remained.

A 2014 paper presented an alternative approach designed specifically with understandability in mind. It's called Raft. That's what I'll be implementing in Python for you here.

So how does Raft work?

1. Multiple copies of the state (data) are kept on multiple servers.
2. If a *majority* (say, 3 out of 5) of the servers are up, the system is operational.
3. Commands to these servers are kept in an ordered *log*. In order to catch the system up so the data is at the most updated state, a server can replay this log much the way that a client app replays a collection of database migrations in order to reach the most updated database schema.
4. If the majority (3 out of 5, say), of the servers agree on the current, updated state of the system, that is considered "the answer."
5. **So how do we get other servers back up to speed after they go down?** Actions are coordinated by one server, called the **leader**. The leader might change over time (leadership is divided into 'terms'). The leader has two jobs:
    a. Accept all client requests.

b. Get all the other servers caught up to the most updated state of the system.



6. **What if the leader goes down?** The remaining servers will elect a new leader. A subtlety of the Raft algorithm: servers will never elect a server that doesn't have the complete log as the leader.

And then, when one of the servers goes down, the whole system heals itself.

How?

We'll get there over the course of this series. We'll start, though, by setting up:

1. The servers we'll coordinate with raft
2. The data that we want all those servers to access and modify with consistent results

That's our topic for later. In the meantime, I also found this 18 minute explanation of the algorithm that will give you a broad-strokes idea of what we're about to do, before we get into the nitty-gritty.

Part 1: Setting Up Socket Servers

If we're gonna try to build a Raft of servers, we will need:

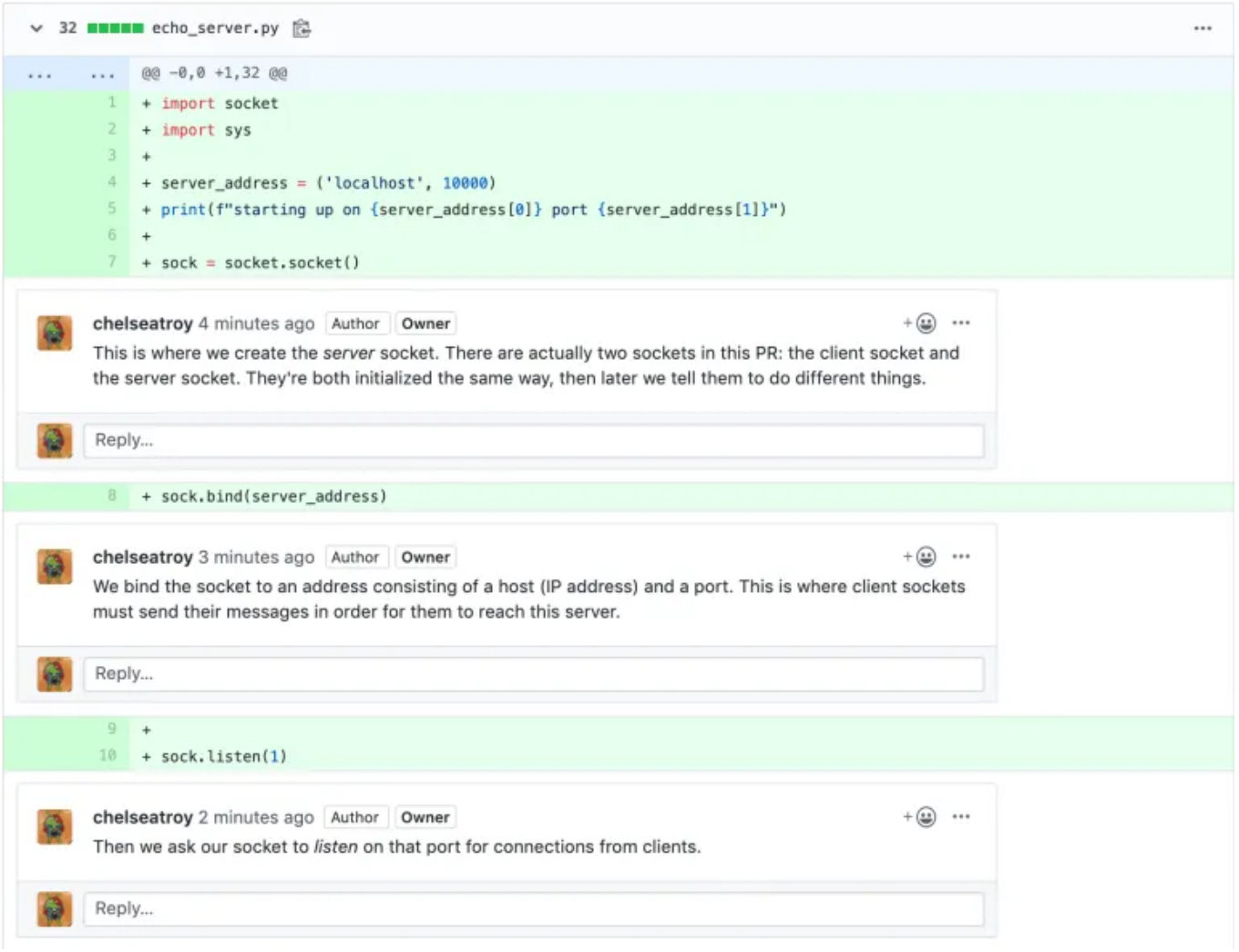
1. Some servers
2. A set of data to keep consistent
3. The ability to recover data if the server goes down
4. Some clients

In this part, I'll go over 1, 2, and 3. Number 4 will be the topic of the next part.

The Servers

I'm doing this with **socket servers.** What's a socket server? I'll show you. I am keeping all the details of this raft project inside one repository, and I'm trying to maintain clearly named, well-circumscribed commits. The first commit implements an echo server and an echo client, both with sockets. I annotated the commit right here if you'd like to copy and paste code or navigate my comments with a screen reader.

First, we have the server:



So there are four main steps: initialize the socket, bind it to an address, tell it to listen, and then open a loop to receive any connections (and, finally, to close the connection when we are done).

By contrast, here is a socket for the client:

30 echo_client.py

```
@@ -0,0 +1,30 @@
+ import socket
+ import sys
+
+ # Create a TCP/IP socket
+ sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

chelseatroy 11 minutes ago Author Owner

This is where we create the *client* socket. There are actually two sockets in this PR: the client socket and the server socket. They're both initialized the same way, then later we tell them to do different things.

Reply...

```
+
+ # Connect the socket to the port where the server is listening
+ server_address = ('localhost', 10000)
+ print(f"connecting to {server_address[0]} port {server_address[1]}")
+ sock.connect(server_address)
```

chelseatroy 11 minutes ago Author Owner

Aha! This is how we know this is a client socket. We give it an *address* (consisting of a host and a port), and we ask it to *connect* to that address. Contrast this to what we tell the server socket to do.

Reply...

```
+
+ try:
+
+     # Send data
+     message = input("Type your message:\n")
```

chelseatroy 10 minutes ago Author Owner

We use `input()` to collect input from the user. `message` will contain whatever the user entered.

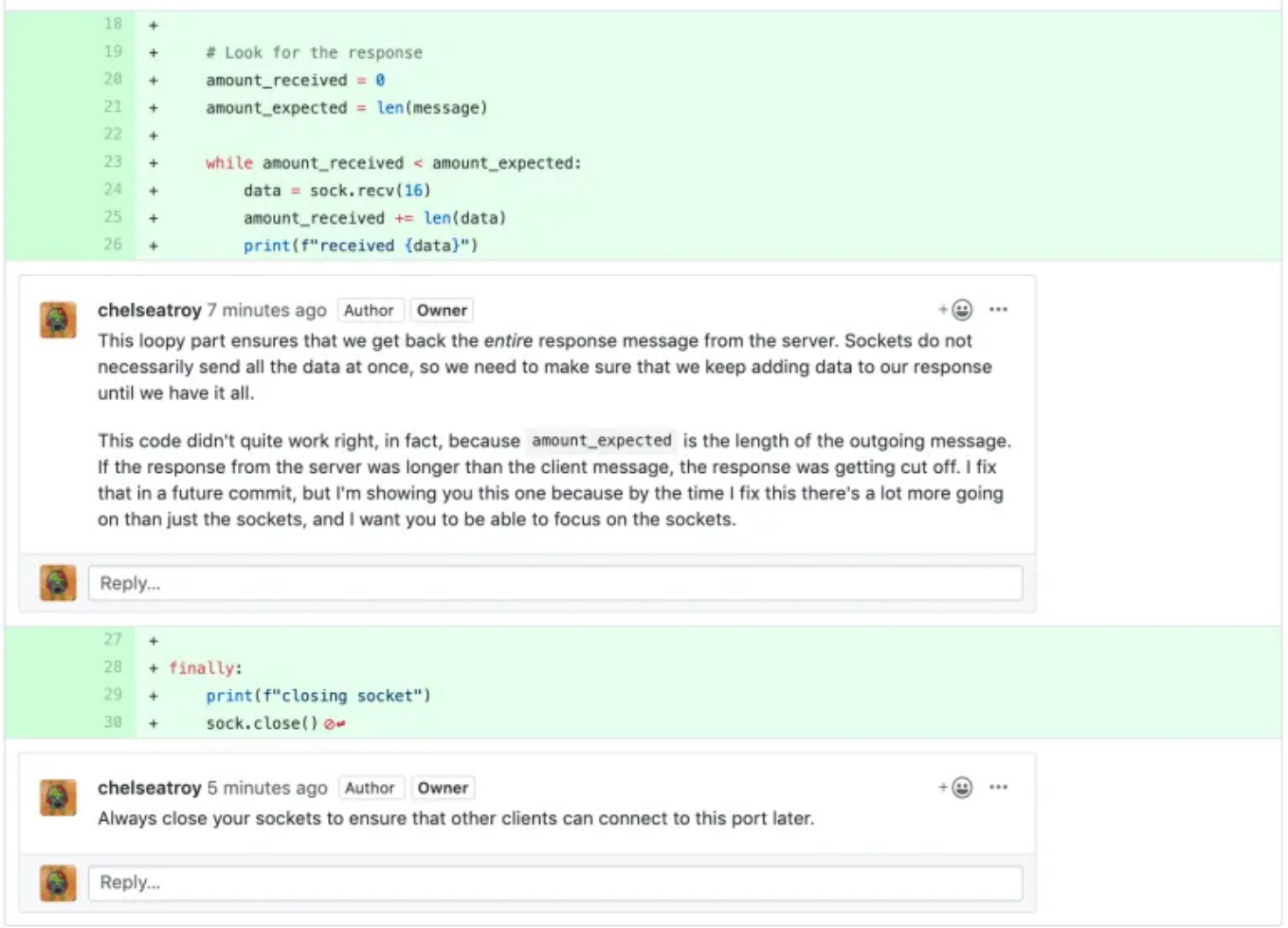
Reply...

```
+     print(f"sending {message}")
+     sock.sendall(message.encode('utf-8'))
```

chelseatroy 9 minutes ago Author Owner

This is where we tell the socket to send what the user entered. We have to encode it because sockets communicate *bytes*, not raw strings.

Reply...



So this time we have three steps: initialize the socket, connect it to an address where another socket is *listening*, and then send over some data (and listen for a response). Finally, we close the connection when we are done.

You can check out this code, fire up a terminal, open two windows, and navigate to the `raft` directory in each window. Then, if you run:

`python echo_server.py` in one window, you should see:

```
starting up on localhost port 1000
```

Now, run `python echo_client.py` in the other window. In the client window, you should see `connecting to localhost port 1000`, followed by `Type your message:`. In the server window you should see `connection from (localhost, [some port])`. This is your client. Hooray!

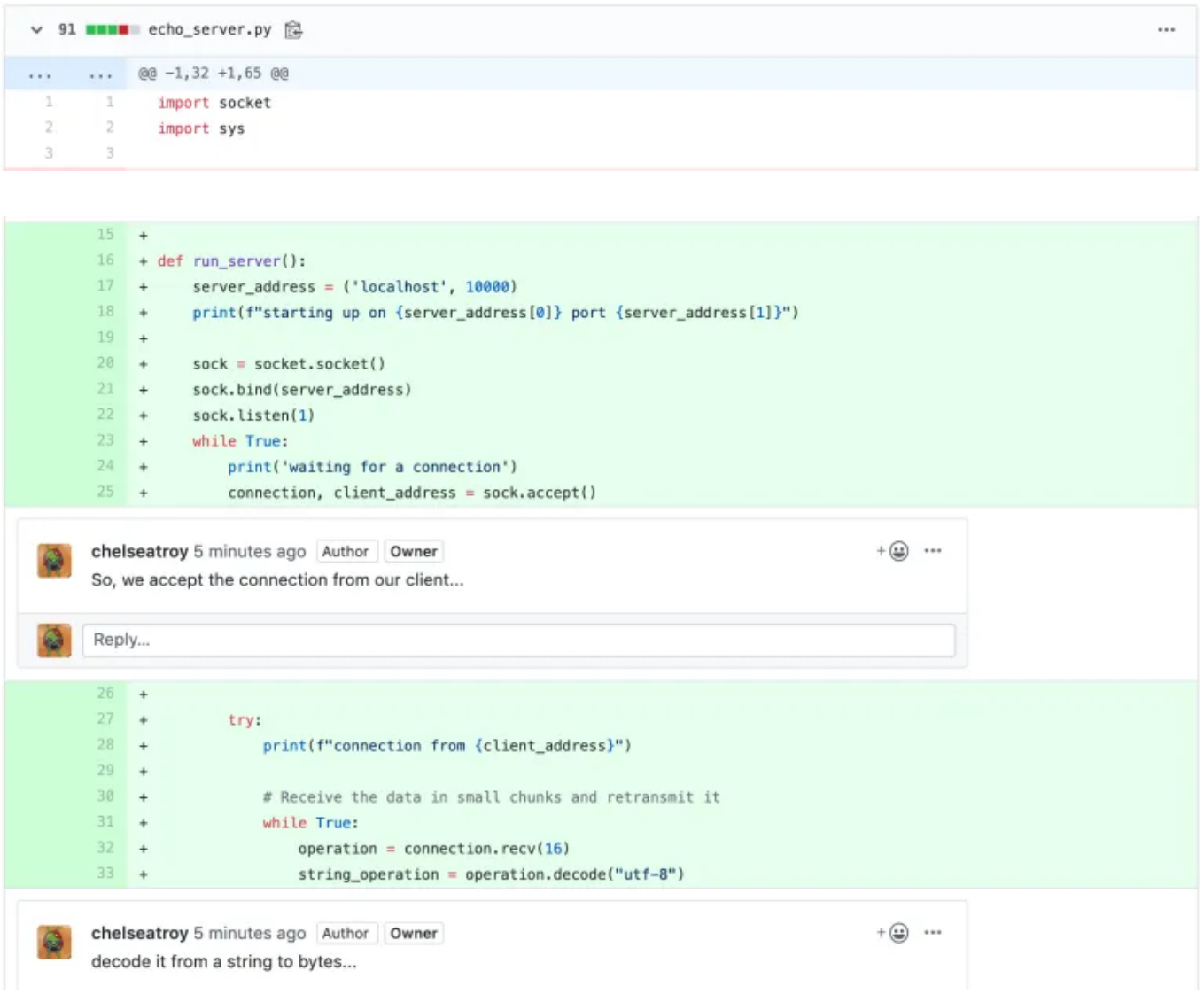
Now, whatever you type into the client window, you should receive *back* as a message from the server.

And if you'd like to learn more about the Python socket API, I recommend this piece that covers the basics.

A Set of Data to Keep Consistent

Now we need these servers to be managing a set of data. This could be any set of data. We're going to use a key value store. In Python this type of data store is called a dictionary. We'll make it possible for clients to tell the server to put a new key and value

in the store (set), fetch a value out of the store by its key (get), and delete a key and its value from the store (delete). [Here's the annotated commit for this](). Again, screenshots:



```
34 +
35 +                print(f"received {string_operation} of type {type(string_operation)}")
36 +                if operation:
```

chelseatroy 5 minutes ago Author Owner

and now, instead of spitting the comment directly back to the client, we are going to do something fancier with it. We will parse it to determine what the client wants!

Reply...

```
37 +                    command, key, value = 0,1,2
38 +                    operands = string_operation.split(" ")
39 +
```

chelseatroy 3 minutes ago Author Owner

Our client will send commands in the following fashion:

```
set key value
get key
delete key
```

For now, we will assume that clients know this, and we won't put in a bunch of code attempting to parse an invalid command. Instead, we'll send back "Sorry, I don't understand that command" if the command does not fit one of these formats.

Reply...

```
40 +                    response = "Sorry, I don't understand that command."
41 +
42 +                    if operands[command] == "get":
43 +                        response = get(operands[key])
44 +                    elif operands[command] == "set":
45 +                        set(operands[key], operands[value])
46 +                        response = f"key {operands[key]} set to {operands[value]}"
47 +                    elif operands[command] == "delete":
48 +                        delete(operands[key])
49 +                        response = f"key {key} deleted"
50 +                    elif operands[command] == "show":
51 +                        response = str(data)
```

chelseatroy 2 minutes ago Author Owner

For debugging purposes, let's add one more command for now: `show`, which will send back a string representation of the whole data store that the server has right now.

Reply...

```
52 +                    else:
53 +                        pass
54 +
```

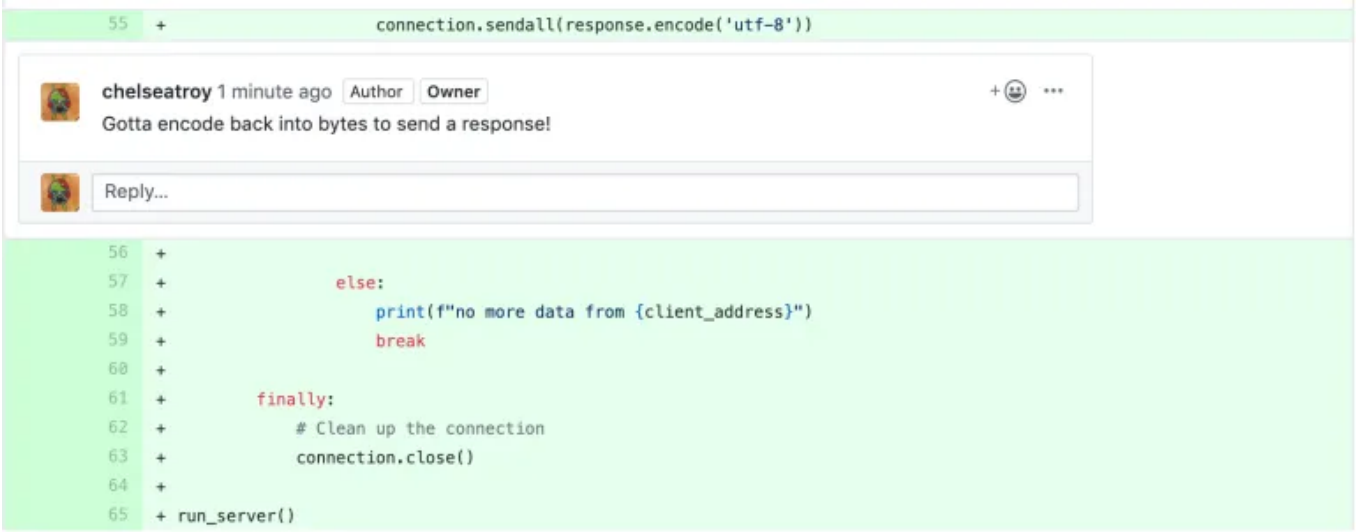
chelseatroy now Author Owner

Chelsea, I hate long chains of `elifs` like that. Why didn't you do that in a more object-oriented way?

Because this is a practice project and all this code is gonna get a lot more complicated later, I promise.

But more broadly, because I don't have enough information about the problem space yet to trust my judgment about where the seams go to separate responsibilities in my app. Moving responsibilities between separate objects, or consolidating them into one object, takes more time and energy than splitting them out. So I will wait until I understand the problem better and feel more confident in my understanding of who should have what responsibilities before I start splitting them apart.

Reply...



You can check out this commit and use the same steps that you used for the echo server to get everything running. Then you can type things into the client like `set chelsea rules`, and you should then be able to type in `get chelsea` and receive the response `rules`. If you type in `data` you should get back `{"chelsea": "rules"}`. Just be sure to avoid extraneous spaces:

```
sending client@ set chelsea rules
received b"every@Sorry, I don't understand tha
t command."
```

Recovering Data if the Server Goes Down

Currently, our key value store lives only in memory on our server. So if the server goes down and comes back up, that data will be lost. We want that data to *persist*—for the server to be able to recover its data. There are a few ways we can do this, but for now, we'll do it by writing any data store commands to a file. Any time the server starts up, it will read this file and execute all the commands in order to return its data to its prior state, much in the way that web frameworks run all the database migrations in order to arrive at the most updated version of a database schema.

As we have continued writing this code, it has become fairly clear that the server code (for coordinating the socket connection) is a separate set of responsibilities from the management of the key value store. So we move the key value store management code into its own methods that will live in an object called `KeyValueStore`.

So we have a method on `KeyValueStore` that accepts a command, executes that command to update (or read from) the data store, and returns a response for the server to pass to the client:

```
def execute(operation):
    string_operation = operation.decode("utf-8")
    print(f"received {string_operation}")

    command, key, value = 0, 1, 2
    operands = string_operation.split(" ")

    response = "Sorry, I don't understand that command."

    if operands[command] == "get":
        response = get(operands[key])
    elif operands[command] == "set":
        set(operands[key], operands[value])
        response = f"key {operands[key]} set to {operands[value]}"
    elif operands[command] == "delete":
        delete(operands[key])
        response = f"key {key} deleted"
    elif operands[command] == "show":
        response = str(data)
    else:
        pass

    return response
```

The server writes all commands to a log before sending them to the `KeyValueStore` to be executed:

```
+                   string_operation = operation.decode("utf-8")
+                   print(f"received {string_operation}")
+
+                   f = open("commands.txt", "a")
+                   f.write(string_operation + '\n')
+                   f.close()
+
+                   response = kvs.execute(string_operation)
```

And each time the server boots up, it initializes a `KeyValueStore` and runs a method on the store called `catch_up()`:

```
+ from key_value_operations import KeyValueStore

  def run_server():
+     kvs = KeyValueStore()
+     catch_up(kvs)
      server_address = ('localhost', 10000)
      print(f"starting up on {server_address[0]} port {server_address[1]}")

      sock = socket.socket()
      sock.bind(server_address)
      sock.listen(1)
+
      while True:
          print('waiting for a connection')
          connection, client_address = sock.accept()
```

That `catch_up()` method reads from the logs and executes the commands to return the `KeyValueStore`'s `data` attribute to its previous state:

```
def catch_up(key_value_store):
    f = open("commands.txt", "r")
    log = f.read()
    f.close()

    for command in log.split('\n'):
        key_value_store.execute(command)
```

If you check out this commit, you should be able to do all the same things that you did at the "A Set of Data to Keep Consistent" stage, except now you should also be able to shut down and boot up the server at will without losing your precious data entries like `chelsea: rules`.

Some Clients

We're doing pretty good, but we still need to take care of a detail. Currently, each of our servers will only accept connections from one client at a time. We would like them each to connect to multiple clients. Why? Because Raft achieves fault tolerance, in part, by electing one server the "leader" of the cluster, and all client requests are redirected to that leader. Any of the servers in the cluster could be elected leader and therefore need to do this.

Why does each server only accept connections from one client at a time? And how would we go about changing that? We'll talk about that next

Part 2: Building Concurrent Solutions

What are our options for allowing a process to facilitate multiple concurrent interactions?

One common solution: **threads.** Before we move on, let's quickly go over the distinction between a thread and a process. A **process** describes an individual running instance of a program. It has its own memory, which it does not share with other programs. An instance of our server running is one process. An instance of our client running is one process. Five instances of our server running represents five processes—they all run from the same code, but they're allocated different address spaces in memory.

A process can run code in multiple separate threads. These threads can run code independently of each other. This is how a server might accept multiple clients: we would spin up a separate thread for each client so the server could have an independent conversation with that client. Unlike processes, they *share* their memory with each other.

The memory sharing thing makes it convenient for threads to communicate with one another. But it also means that, if two conflicting operations execute in separate threads, we have a **race condition.** Suppose one thread spun up in our server sets the key `chelsea` to point to the value `rules`, while another thread sets the key `chelsea` to point to the value `rules`. Does Chelsea rock or rule? Or `chelsea` doesn't get set at all? Or does `chelsea` get set to some amalgamation of the two values like `rucks`? (As it so happens I do that too. but it was the intention of neither client to say so).

We tend to ameliorate this issue with threads by using discrete `Events` (which awaken threads when a specific action needs to be performed) or a `mutex` (short for "mutual exclusion," a mutex prevents all threads except one from executing code at one time). Or we may use a `Lock` object, which locks a specific resource (like a row of a database, or a key value pair in our data store) such that only one thread can do stuff to it at one time.

Sometimes we organize this scheduling with **thread queues**, in which separate threads put stuff that needs to happen onto a queue, and items are popped off the queue in order.

Mutability and State Management

Ultimately our Raft implementation will need to manage two different sets of states:

- The state of the data among all the servers
- The state of the server (who is up, who is down, who is the leader)

We'll talk more about these cases, but first let's talk about the key component here: a **state machine**.

The term **state machine** doesn't originally come from programming. It refers to a mathematical model of computation that can be in exactly one of a finite number of states at any given time. In code, we model this idea with classes that compute, and then report on, the instance being in one of a finite series of states. Usually, we do this via an instance attribute called `state` or `status` that takes one of a finite series of values.

look at a well-circumscribed example: a class representing a traffic light.

```
class TrafficLight:
    def __init__(self, name, state):
        self.name = str(name)
        self.current_state = state

    def progress(self):
        if self.current_state == "green":
            print(self.name + " light yellow")
            self.current_state = "yellow"
        elif self.current_state == "yellow":
            print(self.name + " light red")
            self.current_state = "red"
        elif self.current_state == "red":
            print(self.name + " light green")
            self.current_state = "green"
```

Here we have a traffic light class with a `current_state` attribute that takes three values: `red, yellow, green`.
Our `progress()` method documents the order that these states should happen and moves the light from one state to the next.

Imagine that we wanted to operate two of these lights simultaneously. How might we do that?

```
import time
from threading import Event, Thread
import queue

class TrafficLight:
    def __init__(self, name, state):
        self.name = str(name)
        self.current_state = state

    def progress(self):
        if self.current_state == "green":
            print(self.name + " light yellow")
            self.current_state = "yellow"
        elif self.current_state == "yellow":
            print(self.name + " light red")
            self.current_state = "red"
        elif self.current_state == "red":
            print(self.name + " light green")
            self.current_state = "green"


class TrafficSystem:
    states = {
        "red": 30,
        "yellow": 5,
        "green": 25
    }

    current_time = 0

    def __init__(self, speed=3):
        self.traffic_lights = {}
        self.traffic_lights['1'] = TrafficLight(1, "red")
        self.traffic_lights['2'] = TrafficLight(2, "green")

        self.speed = speed

        self.event_queue = queue.Queue()
        Thread(target=self.timer_thread, args=(self.traffic_lights['1'],), daemon=True).start()
        Thread(target=self.timer_thread, args=(self.traffic_lights['2'],), daemon=True).start()

        self.start_traffic()

    def start_traffic(self):
        while True:
            event = self.event_queue.get().split(" ")

            if event[0] == "progress":
                self.traffic_lights[event[1]].progress()

    def timer_thread(self, light):
        self.event_queue.put('progress ' + light.name)
        time.sleep(self.states[light.current_state] / (self.speed))
        self.timer_thread(light)
```

Here, you see a `TrafficSystem` class. Usually, it makes sense to keep our objects with state separate from the management of that state. Two reasons for this: first, as the state and the management each become more complex, separating these concerns improves the legibility of our code. Second, separating the state from its coordination and timing allows us to write fast, time-independent tests for the state computation itself.

In the example above, we use several of the components we have discussed for concurrent code execution: threads, events, and a queue. We spin up a thread for each traffic light (lines 39 and 40), assign an amount of time that a light should live in each state (line 23), and push events onto our queue to progress each light after waiting the specified amount of time (lines 51-54).

If we new up a `TrafficSystem` and start traffic, our lights start up red and green, then the green one goes yellow for five seconds, then it switches to red as the other switches to green. Repeat.

Now, what if we want to be able to interrupt this state—say, with a button for crossing the street, that should change the light colors as long as the green one has already been green for at least 30 seconds? This now means we must be able to introspect on the state at any given time. It also means that, if events are coordinated, we must have a unified understanding of time. So we cannot accomplish the goal with sleeps (which we can't introspect) or threads (which don't coordinate).

How might we get around something like this?

We might produce a system that divides each state into ticks of the clock, that counts each tick as its own sort of mini-state and transitions only as each state has run its course...or has run at least 30 seconds of its course when the crossing button is pressed.

```
#Initial state of traffic light
Init = {
    'out1': 'G',
    'out2': 'R',
    'clock': 0,
    'button': False,
    'pc': 'G1'
}

# Define functions that determine state membership and state update.
# How to incorporate the button press?
G1 = lambda s: s['pc'] == 'G1' and (
    (s['clock'] < 30 and dict(s, clock=s['clock'] + 1))
    or (s['clock'] == 30 and dict(s, out1="Y", clock=0, pc="Y1"))
)

Y1 = lambda s: s['pc'] == 'Y1' and (
    (s['clock'] < 5 and dict(s, clock=s['clock'] + 1))
    or (s['clock'] == 5 and dict(s, out1='R', out2='G', clock=0, pc='G2'))
)

G2 = lambda s: s['pc'] == 'G2' and (
    ((s['clock'] == 60 or (s['clock'] >= 30 and s['button']))
     and dict(s, out2="Y", clock=0, button=False, pc="Y2"))
    or (s['clock'] < 60 and dict(s, clock=s['clock'] + 1))
)

Y2 = lambda s: s['pc'] == 'Y2' and (
    (s['clock'] < 5 and dict(s, clock=s['clock'] + 1))
    or (s['clock'] == 5 and dict(s, out1='G', out2='R', clock=0, pc='G1'))
)

# Next state relationship
Next = lambda s: G1(s) or Y1(s) or G2(s) or Y2(s)

def run():
    s = Init
    while s:
        print(s)
        s = Next(s)  # State transition


# If here. There was no next state
print("DEADLOCK")

run()
```

This code looks strange, doesn't it? It abuses the implementation of Python in which the predicate of `and` is returned such that state transitions happen when the seconds counter reaches a predefined number. When you run this, you'll see the traffic light cycle through its states in rapid succession. This code manages its transitions independently of a timer (right now), focused chiefly on the order, We could insert a timer when that is needed.

So what might a first pass at concurrency look like for our server?

We might spin up a new thread for each client connection we receive, such that the client and server communications each happen in their own thread:

16 echo_server.py

```
@@ -1,4 +1,5 @@
 import socket
+import threading
 from key_value_operations import KeyValueStore

 def run_server():
@@ -12,18 +13,22 @@ def run_server():
     sock.listen(1)

     while True:
-        print('waiting for a connection')
         connection, client_address = sock.accept()
+        print("connection from " + str(client_address))

-        try:
-            print(f"connection from {client_address}")
+        threading.Thread(target=handle_client, args=(connection, kvs)).start()

+def handle_client(connection, kvs):
+    while True:
+        print('waiting for a connection')
+
+        try:
```

Then we add a lock in our `KeyValueStore` to prevent simultaneous writes on the same key from corrupting our data:

31 key_value_operations.py

```
@@ -1,4 +1,8 @@
+import threading
+
 class KeyValueStore:
+    client_lock = threading.Lock()
+
     def __init__(self):
         self.data = {}
```

```
+        with self.client_lock:
+            if operands[command] == "get":
+                response = self.get(operands[key])
+            elif operands[command] == "set":
+                value = " ".join(operands[2:])
+                self.set(operands[key], value)
+                response = f"key {operands[key]} set to {value}"
+            elif operands[command] == "delete":
+                self.delete(operands[key])
+                response = f"key {key} deleted"
+            elif operands[command] == "show":
+                response = str(self.data)
+            else:
+                pass
```

We now have our building blocks in place. It's time to start getting more specific to the Raft implementation.

As we do that, it's worth reviewing Raft's ultimate purpose: to allow a server cluster to operate as a single source of truth without possessing a single point of failure. For that to work, it must be possible to coordinate the servers' understandings of the up-to-date data. How do the servers catch each other up on the data if some of them go down and come back up?

It is this piece—log replication—that we turn to next.

Part 3: Bare Minimum Log Replication

We need one server to be able to update the logs of another server after that other server goes down.

This requires a few things:

1. Each server needs to have some kind of persistent log such that, if it goes down and comes back up, the state that it knew about *before* it went down is preserved. (This is not strictly necessary; another server could catch up a downed server from zero if it had to. However, this would make the system much more costly to maintain as the shared data store got bigger.
2. Among multiple servers, we need to establish who is doing the catching up and who is being caught up.
3. Once a server is established as the one doing the catching up, it must determine how much catching up the other servers need.
4. Then that server must transmit the logs to be caught up.

In Raft, a *lot* happens to bring this to fruition:

- Servers send "heartbeat" messages to each other periodically to tell each other that they're still up.
- The servers hold elections to determine which of them is the "leader."
- All servers redirect all client requests to write to the data store to the leader.
- If the servers don't get a heartbeat from the leader for a while, they hold a leader election.
- Once a new leader is elected, that leader's job is to get all the other servers' logs up to date.
- In the case of a network partition, it's possible for *multiple* servers to be elected leader. The edge cases here get pretty wild.

Here's a walkthrough, if you're curious. We're not going to worry about all that yet. For now, we want to replicate logs, and the rest of this can wait. Here's my plan for a minimum viable feature set for log replication:

1. Get a server
2. With a key value store
3. that takes the command "you're the leader"
4. and updates the logs of all the other servers

We have our pared down challenge. **Let's go!**

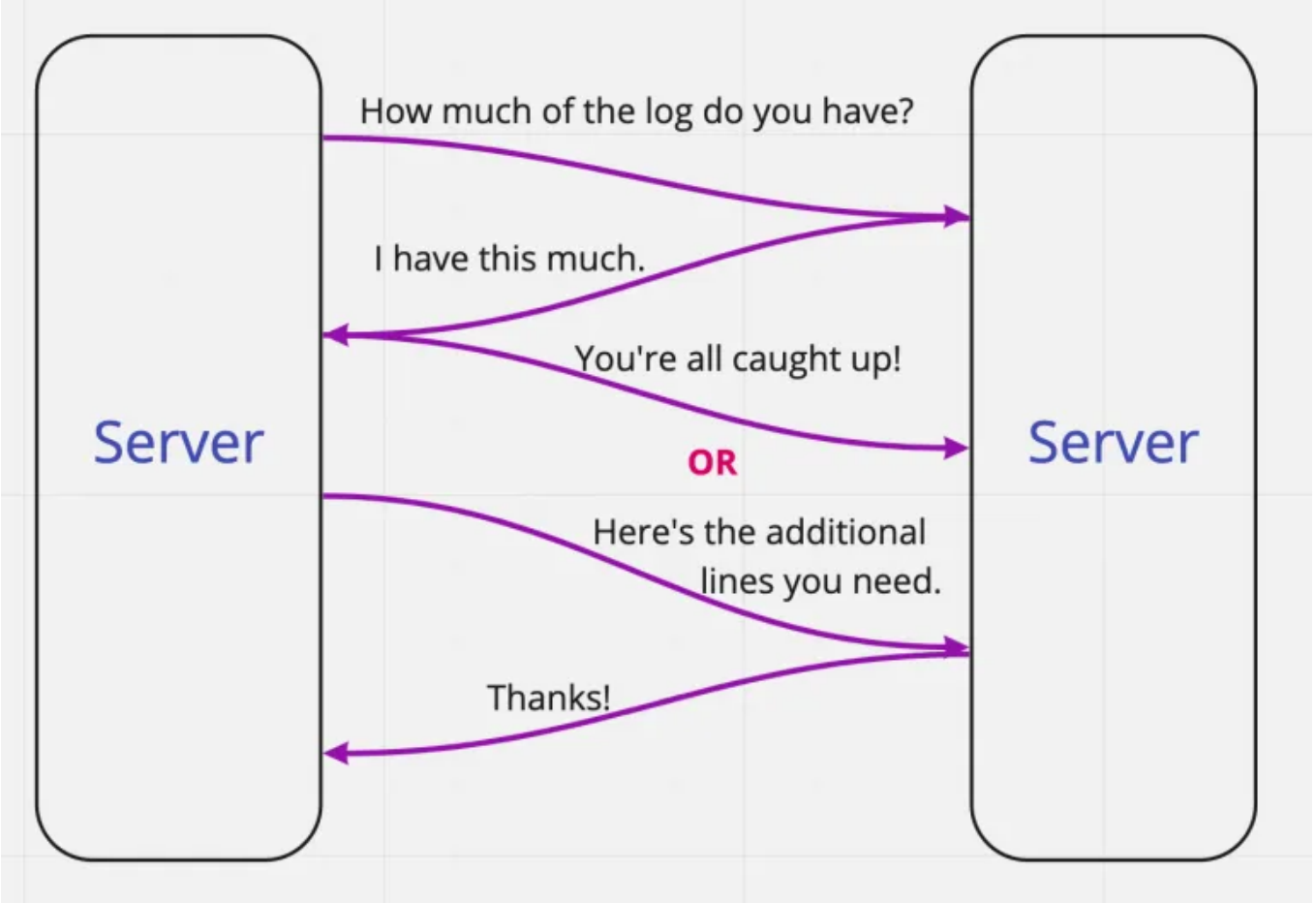
How does Raft replicate logs?

The Raft paper explains that the newly elected leader sends a message to the other servers to append the *last* item in the log to their logs. If that's the only item that the recipient server needs (or if it's already up to date), it responds with `True`. If, however, the recipient server's logs are *more* out of date than that, the server responds with `False`. The leader server then instead sends the last *two* items in their log to append. This continues, back and forth, until the leader is sending the exact set of logs that the follower server is missing.

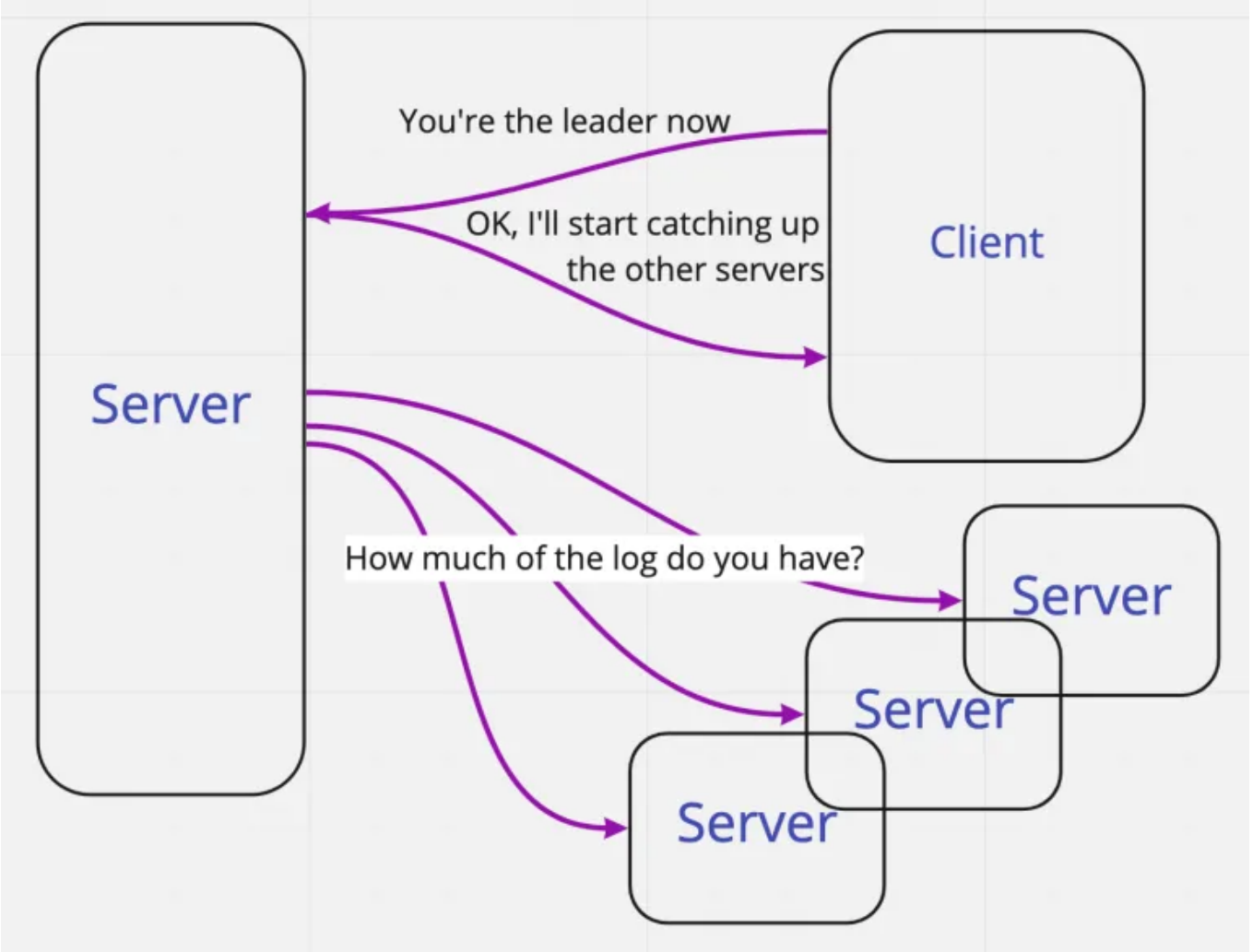
When I read this, it sounded to me like a monumental waste of time. Couldn't the leader ask the follower "how much of the log do you have?" and then send over anything that the follower server doesn't have? (Yes, but it's more complicated than this—as I learned later, when the idea behind the 'waste of time' approach started to make sense).

The Raft paper does discuss such a scheme as an "optimization," and I went ahead and prematurely optimized (perhaps to my detriment, as you'll see). So far, our server responds to messages to get, set, and delete items in the key value store.

Now, I want it to respond to some new messages. I want it to respond to "how much of the log do you have?" with the state of its log. I want it to respond to "I have this much log" with either "Ah, you have just as much log as I do" or "Oh, you're missing things! Here they are!" Finally, I want it to respond to "Oh, you're missing things! Here they are" with an indication of successful catch up (and a thank you, because my software has manners):



In this schematic, the server on the left represents the leader. How does a server become the leader? I don't want to take on leader elections yet, so instead we'll spin up a client and manually nominate a leader from that client. The leader will then begin the conversation with each of the other servers about catching up their logs:



We implement those responses in this commit. Here's the juicy bit from `server.py`:

```
37
38      def handle_client(self, connection, kvs):
39          while True:
40              print('waiting for a connection')
41
42              try:
43                  while True:
44                      operation = receive_message(connection)
45
46                      if operation:
47                          string_operation = operation.decode("utf-8")
48                          print("received " + string_operation)
49
-                         response = kvs.execute(string_operation)
50  +                       if string_operation == "log_length?":
51  +                           response = "log_length " + str(len(self.kvs.log))
52  +                       elif string_operation.split(" ")[0] == "log_length":
53  +                           catch_up_start_index = int(string_operation.split(" ")[1])
54  +
55  +                           if len(self.kvs.log) > catch_up_start_index:
56  +                               response = "catch_up_logs " + str(self.kvs.log[catch_up_start_index:])
57  +                           else:
58  +                               response = "Your info is as good as mine!"
59  +                       elif string_operation.split(" ")[0] == "catch_up_logs":
60  +                           logs_to_append = ast.literal_eval(string_operation.split("catch_up_logs ")[1])
61  +                           [self.kvs.execute(log) for log in logs_to_append]
62  +
63  +                           response = "Caught up. Thanks!"
64  +                       elif string_operation == "show_log":
65  +                           response = str(self.kvs.log)
66  +                       else:
67  +                           response = kvs.execute(string_operation)
68  +
69                          send_message(connection, response.encode('utf-8'))
```

So the server knows how to have the conversation. The server *also* needs to know who to have this conversation with. Somehow, we need the servers to know about all the other servers that should be up, as well as the addresses of those servers so that they can send messages to them.

Up until now, each of our servers has only needed one socket: a socket to *listen* on, to receive messages and respond. Now, our servers need to manage *another* socket. They need to create a temporary socket to *send* messages to servers. They also need to provide some kind of return address so that those servers to whom they send messages can respond to them.

So we're going to have servers register themselves on startup in a server registry file. We implement that in this commit. Again, the juicy bit:

10 server.py

```
@@ -13,9 +13,13 @@ def __init__(self, name, port=10000):
13 13
14 14         def start(self):
15 15             server_address = ('localhost', self.port)
16    -          print("starting up on " + str(server_address[0]) + "port "  +str(server_address[1]))
17    -          server_nodes[self.name] = server_address
18    -          print(str(server_nodes))
   16 +
   17 +          f = open("server_registry.txt", "a")
   18 +          f.write(self.name + " localhost " + str(self.port) + '\n')
   19 +          f.close()
   20 +
   21 +          print("starting up on " + str(server_address[0]) + " port "  + str(server_address[1]))
   22 +          print(str(server_nodes()))
```

I accidentally committed the server registry file, so here's what it looks like:

6 server_registry.txt

```
...  ...  @@ -0,0 +1,6 @@
   1  + myserver localhost 10000
   2  + myserver localhost 10000
   3  + myserver localhost 10000
   4  + myserver localhost 10000
   5  + myserver localhost 10000
   6  + yourserver localhost 10001
```

Entries don't get removed when a server shuts down. You see `myserver localhost 10000` in there five times because I started a server named `myserver` on port 10000, then shut it off, then restarted it, then shut it off, and so on. A leader server should attempt to contact all servers in the cluster, whether or not those servers are up right now. This file is how the leader server will know who to contact. Every server in the cluster has to have been up *at some point* to get registered. I think this is fine. If a server never comes up, I don't mind that none of the leader servers ever know to try to contact it.

Servers can get the addresses the other servers in the cluster (this commit), like so:

6 server.py

```
@@ -11,6 +11,10 @@ def __init__(self, name, port=10000):
11 11             self.kvs = KeyValueStore()
12 12             self.catch_up(self.kvs)
13 13
   14 +      def destination_addresses(self):
   15 +          other_servers = {k: v for (k, v) in server_nodes().items() if k != self.name}
   16 +          return list(other_servers.values())
   17 +
```

Open a temporary socket to talk to a socket at one of those addresses (this commit),

```
+    def tell(self, message, to_server_address):
+        print(f"connecting to {to_server_address[0]} port {to_server_address[1]}")
+
+        self.client_socket = socket(AF_INET, SOCK_STREAM)
+        self.client_socket.connect(to_server_address)
+
+        try:
+            print(f"sending {message}")
+            send_message(self.client_socket, message.encode('utf-8'))
+        except:
+            print(f"closing socket")
+            self.client_socket.close()
```

And even send the same message to all of the other servers at once! (this commit.)

```
+    def broadcast(self, message):
+        for other_server_address in self.destination_addresses():
+            self.tell(message, to_server_address=other_server_address)
```

Then, when a server receives a message that it has become the leader, it can start a conversation about log catchup with each of the other servers (this commit).

```
                    response = "Caught up. Thanks!"
                elif string_operation == "show_log":
                    response = str(self.kvs.log)
+               elif string_operation == "youre_the_leader":
+                   self.broadcast('log_length?')
                else:
                    response = kvs.execute(string_operation)
```

Servers preface all of their sent messages with their name (this commit),

```
+
+    def with_return_address(self, response):
+        return self.name + "@" + response
+
```

whether they're broadcasting or sending a message to a specific server.

```
-                           self.broadcast('log_length?')
87  +                       self.broadcast(self.with_return_address('log_length?'))
88                      else:
89                          response = kvs.execute(string_operation)
90
91  +                   response = self.with_return_address(response)
92                      send_message(connection, response.encode('utf-8'))
```

Servers can then split this name from the message,

```
101 +   def return_address_and_message(self, string_request):
102 +       address_with_message = string_request.split("@")
103 +       return address_with_message[0], "@".join(address_with_message[1:])
104 +
```

such that they can get the address of a server with that name and send their response to the correct recipient.

```
-                       string_operation = operation.decode("utf-8")
65  +                   string_request = operation.decode("utf-8")
66  +                   server_id, string_operation = self.return_address_and_message(string_request)
```

Now we can start up a cluster of servers and watch the conversations happen. In this example, I started up a cluster of servers named `every`, `tom`, `dick`, `and`, `harry` (after a phrase my mother used when I was young to mean "a whole bunch of people." Raft employs an odd number of servers in the cluster—usually five or seven, in examples. So these are our five servers.)

Here, I have started up every Tom Dick and Harry, and I have started a client to send Dick some commands: `set a 1`, `set b 2`, `set c 3`, `set d 4`. Now Dick has four records that none of the other servers have. I then use the client to tell Dick `youre_the_leader`. What follows is Dick's logs about catching up the server named `every`:

```
Default (bash)
starting up on localhost port 10002
connection from ('127.0.0.1', 55402)
from client: received youre_the_leader
connecting to localhost port 10000
sending b'dick@log_length?' to ('localhost', 10000)
connection from ('127.0.0.1', 55406)
from every: received log_length 0
connecting to localhost port 10000
sending b"dick@catch_up_logs ['0 set a 1', '0 set b 2', '0 set c 3',
'0 set d 4']" to ('localhost', 10000)
connection from ('127.0.0.1', 55408)
from every: received Caught up. Thanks!
```

See how much more fun this is when the servers are polite? I try to inject joy into my projects by giving my servers human names and teaching them basic courtesy. It even makes the mistakes more fun:https://platform.twitter.com/embed/index.html?dnt=true&embedId=twitter-widget-0&frame=false&hideCard=false&hideThread=false&id=1207443864593870848&lang=en&origin=https%3A%2F%2Fchelseatroy.com%2F4-get-something-working%2F&theme=light&widgetsVersion=219d021%3A1598982042171&width=550px

Ugh, I hate your `elif`s, Chelsea. Don't you know about objects?

Have you *read* my blog? Of course I know about objects. But software doesn't usually model objects, as we've discussed before:

> *Many authors [have] their examples reference physical objects. Uncle Bob's books famously do a step-by-step software implementation of the Mark IV Coffee Maker. Practical Object-Oriented Design in Ruby, by Sandi Metz, describes the object-oriented approach to building a bicycle.*
>
> *The goal here is noble: to choose a domain that readers already understand, so that they don't have to learn new concepts unrelated to software in order to access the example.*
>
> *The problem: we do not build coffeemakers and bicycles out of software. So when we're discussing separation of concerns, the ease with which you can do that in these examples is fake. Yes, clearly there is a different set of responsibilities for a gear and a handlebar, or a water heater and a coffee pot. The boundaries between the concepts that we represent in most software—like users and accounts—are much, much fuzzier. It's this messiness that programming texts miss, which forces programmers to go forth and bridge the gap between these toy examples and the abstract world on their own.*

So how do we bridge the gap between the infinitely nuanced abstract world and our limited options for representing it in code?

Here, I am keeping all my functionality in one place until the data I have collected on the problem reaches the threshold at which the abstractions I define are satisfactorily likely to be accurate. Pulling out separate responsibilities from a preexisting blob is much less bamboozling than attempting to move responsibilities between a suite of inaccurately separated concerns. So, until I understand the problem better, the code looks like this. And you know what? It's pretty legible.

Now, let's separate some concerns.

Part 4: Disciplining Our Naughty Code

How do we bridge the gap between the infinitely nuanced abstract world and our limited options for representing it in code?

Here, I am keeping all my functionality in one place until the data I have collected on the problem reaches the threshold at which the abstractions I define are satisfactorily likely to be accurate. Pulling out separate responsibilities from a preexisting blob is much less bamboozling than attempting to move responsibilities between a suite of inaccurately separated concerns. So, until I understand the problem better, the code looks like this. And you know what? It's pretty legible.

So far, we've written some naughty code.

- `server.py` does lots of unrelated things
- Nary a test
- Weird behavioral issues to iron out

Now that *something* works, we've gathered enough insights to start separating responsibilities.

First of all, everything we own lives in the root directory. We'll start making a few boxes to put things in (this commit). **I made four to start with:**

- `src`: for our Server, Client, KeyValueStore, and helper methods in message_pass and config.
- `logs`: for the server registry and the log files where our servers write the commands they have received.
- `exercises`: for the state machine exercise that we did with the traffic lights, mostly.
- `tests`: it's time to start getting this system under test.

Why aren't there already tests?

One of the pieces of advice I received for embarking on this project was to manage complexity by splitting up objects and leaning on automated testing. And I think that's fabulous advice for folks who have taken a crack at solving this problem, or a problem like it, before.

I, however, have not solved a lot of problems like this. I don't typically work with socket servers or store structured data for persistence outside some kind of database. And I'm not starting from a framework here like Rails, Django, Android, or Spring, which come equipped with opinions on where to put various types of code in the project. I started with zilch: an empty project.

Tests serve three functions.

1. Documentation. Tests offer us miniature stories to explain how to use our API and what to expect it to do. They're therefore immensely valuable for storing context and transferring it to other developers. But I started out working on this code alone. All the context lived in my head as I wrapped my brain around new concepts. I'm not fast enough to execute enough software in a few days that I can't remember how to use it. I wouldn't put *down* an untested project that I wanted any hope of picking back *up*. I'd need documentation for that. But at the early stages of this project, I didn't need that.

2. Regression Prevention. Tests (particularly integration tests, system tests, and risk-oriented tests) allow us to run a series of small feedback loops simultaneously, so we can check lots of paths through our code. That becomes critical in applications

where we might break something without noticing. Up until now, my little system has had pretty limited functionality. Everything's broken. I notice it all.

3. Design Aids. Avdi Grimm sees unit test-driven development as a tool to write clean, well-circumscribed APIs. Coraline Ehmke describes why this works: in a test, we get to articulate our wishful, greedy, best-case vision of how we'd like the API to read and act. After that, we flip to the implementation to execute on that vision. Without that vision, the lazy slobs in us might take over and implement the code in whatever way is most convenient for our current selves, the writers. But the test holds us accountable to our vision as *readers*. **And code gets read much, much more than it gets written.**

As I started this project, though, I did no design. I threw everything into three files because I didn't yet have enough data about the problem to accurately separate the concerns. Even among the three files I started with, I ran into churn early on. Should the server write to its logs, or should they key value store do it? Are the properties of a cluster *emergent* from the behavior of individual servers, or are they *enforced* by some kind of cluster manager like we had with the traffic lights? Had I invested a lot of time in separating concerns early, I'd often find that I had separated them incorrectly. Re-separating concerns is more frustrating than separating concerns from a single place, and having to move and rewrite unit tests adds to the frustration.

Look, I have to tell you a dirty secret.

Ready? I slung code for the most dogmatically TDD shop in the industry for years. Seriously, programmers love to joke that Labs is a cult, and the fact that we clapped at the end of standup didn't help our case. You can think of us as the ultimate TDD conservatory. And even *we* didn't TDD novel problems from the get-go.

Instead, we did:

1. A spike—experimental blobs of functionality to get something working.
2. Delete it all (well, truth be told, comment it out to reference later).
3. Color-code or otherwise demarcate the comments into hypothetical concerns.
4. Pick a concern. Write a test. Consider how we'd like its API to look and work.
5. Watch the test fail. Make it pass. Look for simplifications that allow it to still pass.
6. Repeat steps 4 and 5 until the concerns are done.

If you start with #4, it's a short road to Circular Dependency Land.

The dirty secret is, TDD isn't for completely novel problems. We need some information first about the risks and responsibilities inherent to our system.

But forging on without tests has its consequences.

As I developed my test-less laboratory of spaghetti code, I also developed alongside it a checklist of things to manually retry each time I changed something. I caught regressions this way. Since this code had only maybe two execution paths, it didn't take long to do the manual thing, and I was poking around manually anyway after each change. As software scales, this doesn't (although keeping a manual regression checklist to run before a *release* is a good idea for an application of any size).

Were I to do this project *again,* I might get to an echo server and then write a single, happy-path system test for the echo server. I might update the system test as I added functionality to the project, leaving unit tests until the separation of concerns stage as I

did here.

Adding Tests

I tried out a new thing on these tests that I haven't done in previous test suites I've written. I'll show you my normal way, and then the new thing, and we'll see what you think.

Typically when I write unit tests, I attempt to use *only* the API method that I am testing in the test itself. I set everything else up by some other means. Here's an example (from this commit):

83 tests/test_keyValueStore.py

```
@@ -1,24 +1,83 @@
- from unittest import TestCase
+ import pytest
+ from src.key_value_store import KeyValueStore

-
- class TestKeyValueStore(TestCase):
+ class TestKeyValueStore():
      def test_get(self):
-         self.fail()
+         kvs = KeyValueStore(server_name="DorianGray")
+         kvs.data = {"Sibyl": "cruelty"}
+         assert kvs.get("Sibyl") == "cruelty"

      def test_set(self):
-         self.fail()
+         kvs = KeyValueStore(server_name="DorianGray")
+         kvs.set("Sibyl", "cruelty")
+         assert kvs.get("Sibyl") == "cruelty"

      def test_delete(self):
-         self.fail()
+         kvs = KeyValueStore(server_name="DorianGray")
+         kvs.data = {"Sibyl": "cruelty"}
+         kvs.delete("Sibyl")
+         assert kvs.get("Sibyl") == ""
+
+     def test_write_to_log(self):
+         test_log_path = "tests/test_kvs_logs.txt"
+         self.cleanup_file(test_log_path)
+
+         kvs = KeyValueStore(server_name="DorianGray")
+         kvs.write_to_log("set Sibyl cruelty", test_log_path)
+         self.assert_on_file(path=test_log_path, length=2, lines="0 set Sibyl cruelty")
+
+         kvs.write_to_log("Set Basil wrath", test_log_path)
+         self.assert_on_file(path=test_log_path, length=3, lines=["0 set Sibyl cruelty", "0 set Basil wrath"])
```

Take a look at the test for `get`. Rather than using `set` to set Sibyl to cruelty, I assign the `data` variable to a hash myself. This isolates each part of the API to its test so that, if something fails I know exactly what part of the API has broken.

Similarly, the only method I call on the subject under test in the `write_to_log` test is `.write_to_log()`.

But how about these tests?

```
17      -     def test_get_latest_term(self):
18      -         self.fail()
        52    +     def test_get_latest_term_after_catchup(self):
        53    +         kvs = KeyValueStore(server_name="DorianGray")
        54    +         test_log_path = "tests/test_kvs_logs.txt"
        55    +
        56    +         kvs.catch_up(path_to_logs=test_log_path)
        57    +         latest_term = kvs.get_latest_term()
        58    +         assert latest_term == 0
        59    +
        60    +         kvs.write_to_log("1 set Sibyl cruelty", test_log_path)
        61    +         kvs.data = {}
        62    +
        63    +         kvs.catch_up(path_to_logs=test_log_path)
        64    +
        65    +         latest_term = kvs.get_latest_term()
        66    +         assert latest_term == 1
```

```
14      33          def test_catch_up(self):
15      -               self.fail()
        34    +         test_log_path = "tests/test_kvs_logs.txt"
        35    +         self.cleanup_file(test_log_path)
        36    +
        37    +         kvs = KeyValueStore(server_name="DorianGray")
        38    +         kvs.write_to_log("0 set Sibyl cruelty", test_log_path)
        39    +         kvs.write_to_log("0 set Basil wrath", test_log_path)
        40    +         kvs.data = {} #Ensure that in-memory data is empty
        41    +
        42    +         kvs.catch_up(path_to_logs=test_log_path)
        43    +         assert kvs.get("Sibyl") == "cruelty"
        44    +         assert kvs.get("Basil") == "wrath"
        45    +
        46    +
        47    +     def test_get_latest_term_before_catchup(self):
        48    +         kvs = KeyValueStore(server_name="DorianGray")
        49    +         null_latest_term = kvs.get_latest_term()
        50    +         assert not null_latest_term
```

Now I'm testing some of the other, more complex methods in the class, and I'm using the `write_to_log()` method even though it isn't being tested. For now I'm calling this approach "**incremental story,**" and I haven't decided if I like it yet.

Here's what it buys me:

- **A simpler, more expressive story.** This flow is perhaps easier to follow, such that a developer could read this test to understand what the class is supposed to do.
- **A check on the legibility of my method names:** If I do this and the methods aren't named well, the story becomes harder to read, not easier.

And here's what I lose:

- These tests could fail because either `.write_to_logs()` or the method they're supposed to test have changed.

For now, I'm okay with this drawback because I can look at the test that tests `write_to_logs` in isolation. If it fails, that's the method that failed. If it passes, a failure in the other tests probably has to do with the method they're supposed to test. For this reason, I think I only want to include methods that have already had their test in the implementation of other tests.

Pulling Things Out of the Server

Finally, out server is a little cumbersome to test. We have to spin it up, regale it with messages, and check the responses. I'd therefore like the server to focus exclusively on managing sockets, which means pulling out everything else it does.

So I know how I want to separate the socket concern. That doesn't mean I know how to separate any of the other concerns in the server. For now, I pull everything else (in this commit) into another file that serves as a grab bag of methods with a temporarily profane name (sorry).

In retrospect I might not name the file a cuss, but I do think that "obviously not the final name" is a fine quality for the file name to have right now. This is not the final separation of concerns. It is, rather, the act of separating a *single* concern—socket management—from everything else, with the acknowledgment that work remains to understand and organize the everything else part.

In this commit, the server goes from 139 lines to 80. It possesses three non-constructor methods:

- `start` (to start its listening socket)
- `handle_client` (to receive incoming connections)
- `tell`(to send responses)

These are the only things I want the server to do.

Oh, the places we can go!

From here, we can dive into:

- Writing system tests that spin up servers and clients
- Writing unit tests for the things in our grab bag file
- Forging ahead with something Raftier

We'll forge ahead with something Raftier and come back to some of these other things: you're here for Raft, and those other things get more interesting once we move a little further into Raft.

Part 5: Terms and Leadership

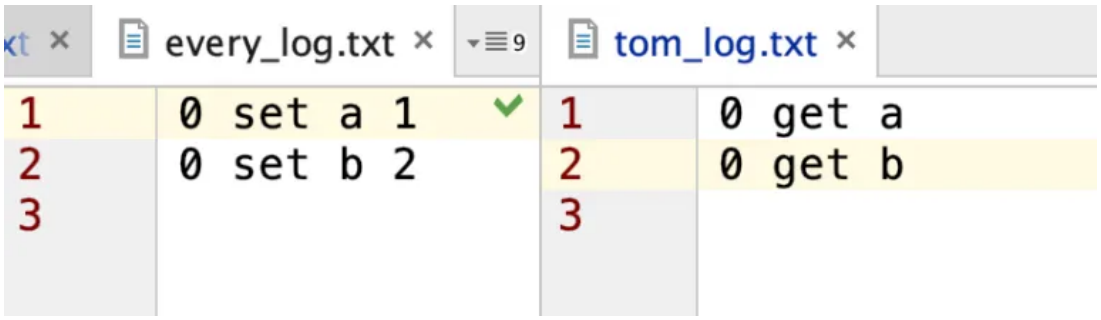
It's time to get log replication to work in a more sophisticated fashion.

Because right now, there's a glaring hole in it.

Suppose I start up two servers and I start a client to tell one of them to `set a 1`, `set b 2`. Then I start a different client to tell the other one `get a`, `get b`...just to make sure that those values *aren't* set on the second server before I attempt to replicate logs from the first to the second server. Here's what I'm going to get:

```
starting up on localhost port 10000
connection from ('127.0.0.1', 62284)
from client: received youre_the_leader
connecting to localhost port 10001
sending b'every@log_length?' to ('localhost', 10001)
connection from ('127.0.0.1', 62287)
from tom: received log_length 2
connecting to localhost port 10001
sending b'every@Your info is at least as good as mine!' to ('localhos
t', 10001)
```

The first server thinks the second server is up to date because these two logs have the same length:



We can fix this *particular* issue by only persisting writes, not reads (this commit):

13 key_value_store.py

```
@@ -31,10 +31,6 @@ def catch_up(self):
31 31         def execute(self, string_operation):
32 32             self.log.append(string_operation)
33 33
34    -         f = open(self.server_name + "_log.txt", "a+")
35    -         f.write(string_operation + '\n')
36    -         f.close()
37    -
38 34             command, key = 0, 1
39 35             operands = string_operation.split(" ")
40 36
@@ -45,14 +41,21 @@ def execute(self, string_operation):
45 41                     response = self.get(operands[key])
46 42                 elif operands[command] == "set":
47 43                     value = " ".join(operands[2:])
   44 +                 self.write_to_log(string_operation)
48 45                     self.set(operands[key], value)
49 46                     response = f"key {operands[key]} set to {value}"
50 47                 elif operands[command] == "delete":
   48 +                 self.write_to_log(string_operation)
51 49                     self.delete(operands[key])
52 50                     response = f"key {key} deleted"
53 51                 elif operands[command] == "show":
54 52                     response = str(self.data)
55 53                 else:
56 54                     pass
57 55
58    -         return response ⊘↵
   56 +         return response
   57 +
   58 +     def write_to_log(self, string_operation):
   59 +         f = open(self.server_name + "_log.txt", "a+")
   60 +         f.write(string_operation + '\n')
   61 +         f.close() ⊘↵
```

That will save us for now, provided that we never have to deal with a network partition.

Suppose, though, that we *did* have to deal with a network partition, such that two servers could each receive different `set` or `delete` commands from clients. When the partition is eliminated, how would these servers adjudicate whose logs to keep?

It's time to start talking about leaders, followers, and terms. Which means, for perhaps the first time in this series, looking at what the Raft paper actually says about this.

The Job of the Lead Server

Think of Raft as a sledding rig and the servers as sled dogs.

One dog acts as the leader. This dog is responsible for taking commands from the driver and guiding the other dogs to navigate the terrain. In raft, one server takes charge of receiving requests from clients and sending updated log information to the other servers. If a client tries to contact a server that is *not* the leader, that server responds by telling the client who is the leader.

Any of the servers could, theoretically, lead the pack. The one that is *currently* leading the pack needs a way to assert that it is still up and leading. It does that by sending periodic messages (referred to in the Raft paper as the "heartbeat").

If another server waits a while and doesn't hear from the leader, it sends out a message to all servers to call a **leader election**. This election will result in another server becoming the leader. It will also result in each online server incrementing its **term**, an integer that keeps track of how many times there has been a change in leadership.

Instead of determining which of two logs is more up-to-date by comparing their length, Raft prescribes comparing these term numbers. It says:

> *Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs. If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.*

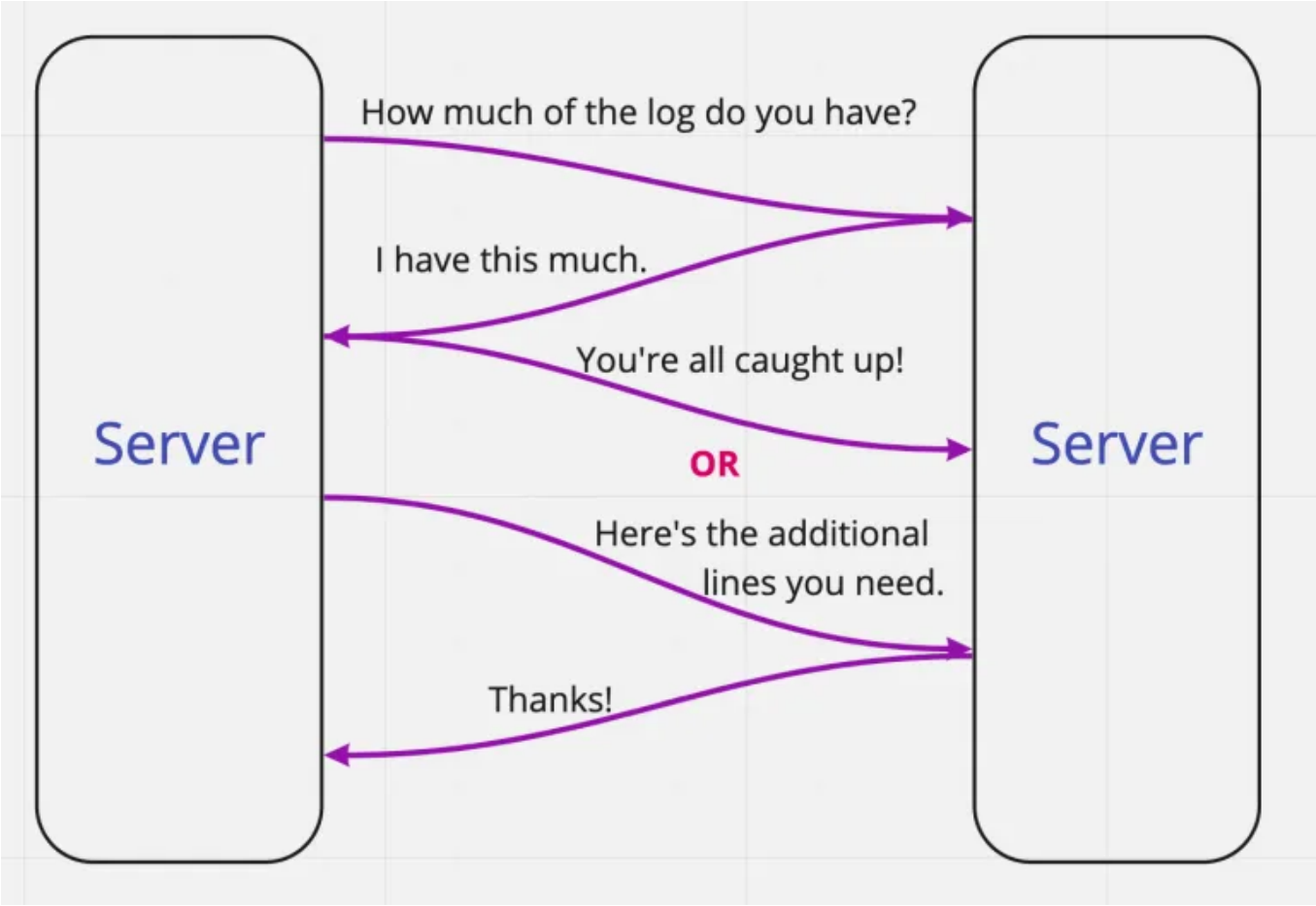
If the last term of one log is `4 set a 1` and the last term of another log is `5 set a 2`, the one that starts with `5` is more up-to-date regardless of the lengths of the two logs.

In fact, we have already *included* a placeholder for the term in our log: the `0` recorded at the beginning of each command. At the moment, the term always remains 0. After all, our servers don't ever undergo a change of leader: we start them all up, tell *one* of them that it's the leader, and let it scramble to have a one-time conversation with each of the other servers to update their logs to match its own log.

Now, we need to get the lead server to send out a **heartbeat** to the other servers. (We'll come back to elections later).

What is the heartbeat?

Until now in this series, I implemented servers to have whole conversations, like this:



Now that I've done all that work, it turns out that Raft prescribes its servers a much smaller set of social skills 😂.

Instead, it says servers should have a message called `AppendEntries`. It doesn't ask the other servers where their logs are; instead, it sends `AppendEntries` with any new entries, and the responsibility falls upon each server to determine whether this is the only entry they're missing or not.

In addition, instead of having a separate heartbeat message, leaders repurpose the `AppendEntries` message, which they can send without any entries to append. It's time for us to remove a lot of extraneous conversation skills from our server, in [this commit](#).

```
-     if string_operation == "log_length?":
-         response = "log_length " + str(len(key_value_store.log))
-     elif string_operation.split(" ")[0] == "log_length":
-         catch_up_start_index = int(string_operation.split(" ")[1])
-
-         if len(key_value_store.log) > catch_up_start_index:
-             response = "catch_up_logs " + str(key_value_store.log[catch_up_start_index:])
-         else:
-             response = "Your info is at least as good as mine!"
-     elif string_operation.split(" ")[0] == "catch_up_logs":
+     if string_operation.split(" ")[0] == "append_entries":
          logs_to_append = ast.literal_eval(string_operation.split("catch_up_logs ")[1])
          [key_value_store.execute(log, term_absent=False) for log in logs_to_append]

          response = "Caught up. Thanks!"
```

I REALLY hope this is the right decision.

Differentiating the Leader

Until now, I have launched five of the same server, then started up a client to tell *one* of the servers that it is the leader. That server would respond, issue a series of commands to update the logs of all the other servers, and then...that's it. The system had no state. No server maintained ongoing leadership beyond doing one round of leader behavior in response to a client command.

Now, we need to introduce a new piece of state to each server: its role as a leader or a follower (in [this commit](#)):

```
+ class Server:
      def __init__(self, name, port=10000):
          self.port = port
          self.name = name
          self.key_value_store = KeyValueStore(server_name=name)
          self.key_value_store.catch_up()
          self.term = self.key_value_store.get_latest_term()
+         self.leader = leader
```

Now, we need that leader to send out the heartbeat to tell the other servers that it is still up and leading.

So we create a method to send out the heartbeat.

```
def prove_aliveness(self):
    if self.leader:
        broadcast(self, with_return_address(self, "append_entries []"))
```

As you can see, the method broadcasts the `append_entries` message to all of the other servers with an empty collection of entries.

We start a five second timer when we listen for new connections, and we run our `prove_aliveness` method every five seconds.

```
+     def send(self, message, to_server_address):
          print(f"connecting to {to_server_address[0]} port {to_server_address[1]}")

          peer_socket = socket(AF_INET, SOCK_STREAM)
@@ -53,17 +54,27 @@ def start(self):
          self.server_socket.bind(server_address)
          self.server_socket.listen(6000)

+         threading.Timer(5.0, self.prove_aliveness).start()
+
          while True:
              connection, client_address = self.server_socket.accept()
              print("connection from " + str(client_address))

-             threading.Thread(target=self.handle_client, args=(connection, self.key_value_store)).start()
+             threading.Thread(target=self.manage_messaging, args=(connection, self.key_value_store)).start()
```

As of now, the server does not send `append_entries` with entries in it anymore. For our next step, we'll reimplement log replication to better assess where two servers' logs diverge.

Part 6: Log Replication: Once More with Feeling!

We now have the **heartbeat**—the means by which the leader server communicates to the other servers that it's still in charge. To do that, though, we removed the part where a server sends new log entries for other servers to replicate.

We need to redo it. So let's get started.

Replication the Raft Way

Let's look at section 5.3 of the Raft paper, which focuses on log replication. I'll highlight the pieces that describe the implementation details we need to address.

Logs are organized as shown in Figure 6. Each log entry stores a state machine command along with the term number when the entry was received by the leader. The term numbers in log entries are used to detect inconsistencies between logs and to ensure some of the properties in Figure 3. **Each log entry also has an integer index identifying its position in the log.**

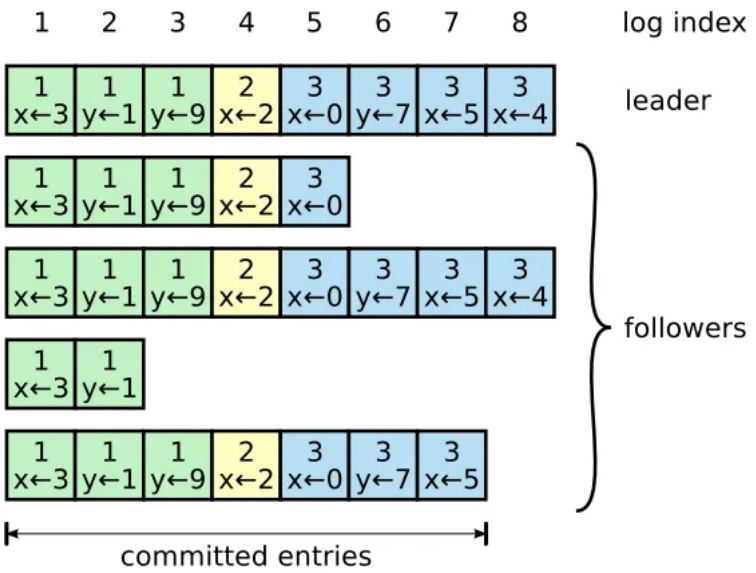


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

Currently, each of our log entries includes a term number and a command for the state machine. This passage indicates that it *also* needs an index, which we will use to determine if logs are current by comparing the index and term number of the latest command in two separate logs. The idea is that we use this to fulfill the **Log Matching Property**:

> *Raft maintains the following properties, which together constitute the Log Matching Property in Figure 3:*
>
> *• 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 the same index and term, then the logs are identical in all preceding entries.*
>
> *The first property follows from the fact that a leader creates at most one entry with a given log index in a given term, and log entries never change their position in the log. The second property is guaranteed by a simple consistency check performed by AppendEntries.*

I think I can get away with *not* adding indices to each row in my current log implementation. I think I should be able to use the indices from the collection that the state machine uses to execute the log commands. We'll find out if I'm right about that.

> *The leader decides when it is safe to apply a log entry to the state machines; such an entry is called committed. Raft guarantees that committed entries are durable and will eventually be executed by all of the available state machines.* ***A log entry is committed once the leader that created the entry has replicated it on a majority of the servers (e.g., entry 7 in Figure 6)****. This also commits all preceding entries in the leader's log, including entries created by previous leaders.*

Once upon a time, our server would *only* update other servers' logs upon being explicitly told to do so. Now, our leader server will endeavor to update followers' logs each time it receives a **write** (a command to set or delete a key-value pair).

My server implementation relies on a conditional statement to parse requests and issue responses. Before my hiatus, I moved this conditional statement into its own file called `parsing.py`. As I got back into the code after six months away from it, I found that

one of the *first* things I wanted to do was move the conditional *back* into the server class. You see that change reflected in this commit.

Now that I've done some more work and reacquainted myself with the code, I find myself wanting to pull it back out again, largely to make it easier to unit test. I feel weird about this because I pride myself on prioritizing 'legibility' in the code I write—but it seems that what makes my code 'legible' to me changes depending on my immediate familiarity with it. Maybe if the conditional statement *were* under unit tests, I'd like it in its own file even after coming back from hiatus. Hmm. Maybe I'll save this for later.

In the meantime, back to Raft.

Sending out logs to be replicated

All right, let's reimplement log replication! First of all, since we have already distinguished a leader server and a follower server, we can give them different behavior.

So when a request related to the data comes through the conditional statement, a leader server should pass on the write commands it receives for follower servers to replicate. A follower server should not process these requests. One might argue that a follower should redirect to the leader, or at least tell the requester where to find the leader. For now, though, my followers slam the door on these requests with a curt `"I am not the leader. Please leave me alone."`

server.py Raw

```
...
        else: #if the request is to get, set, or delete
            if self.leader:
                self.current_operation = string_operation
                key_value_store.write_to_log(string_operation, term_absent=True)

                if self.current_operation.split(" ")[0] in ["set", "delete"]:
                    broadcast(self, with_return_address(self, "append_entries [" + self.current_operation + "]"))

                send_pending = False
            else:
                response = "I am not the leader. Please leave me alone."
```

You'll notice that this implementation focuses on write requests and..completely .drops read requests on the floor.

We don't have to replicate reads. We're focused on replication right now. So, we'll stick with the writes for now, and come back to reimplement the reads later in this commit.

When a leader broadcasts the `append_entries` request to all the other servers on line 8 above, the follower servers accept the request and append the entry to their logs. They send a message *back* to the leader to indicate that they received, and appended, the new command.

```
1  …
2          if string_operation.split(" ")[0] == "append_entries":
3              # followers do this to update their logs.
4              stringified_logs_to_append = string_operation.replace("append_entries ", "")
5              print("Preparing to append: " + stringified_logs_to_append)
6              logs_to_append = ast.literal_eval(stringified_logs_to_append)
7              [key_value_store.write_to_log(log, term_absent=True) for log in logs_to_append]
8              print("State machine after appending: " + str(key_value_store.data))
9
10             response = "Append entries call successful!"
```

view raw

server.py

Now, there are two steps here: write a command to the log, and **commit** the command to the server's state machine. So far, the leader has done the first of these two things. It waits to do the second one until a majority of servers have written the command to *their* logs. So, when the follower servers have sent the `"Append entries call successful!"` message, it's time for the server to make a tally.

We initialize the server with a dictionary that indicates all followers are *not* up to date:

```
1  class Server:
2  …
3          self.followers_with_update_status = {}
4          self.current_operation = ''
5          self.current_operation_committed = False
6
7          for server_name in other_server_names(name):
8              self.followers_with_update_status[server_name] = False
```

view raw

server.py

Then, when one of them responds, that response gets its own branch in the conditional statement, where the leader changes the update status for that server to `True`.

```
...
        elif string_operation == "Append entries call successful!":
            if self.leader:
                self.mark_updated(server_name)
            send_pending = False
```

view raw

Each time the leader server updates the status of a follower server, it checks to determine if we have reached quorum: the point where a majority of servers have replicated the log.

server.py Ra

```
...

    def mark_updated(self, server_name):
        self.followers_with_update_status[server_name] = True

        trues = len(list(filter(lambda x: x is True, self.followers_with_update_status.values())))
        falses = len(list(filter(lambda x: x is False, self.followers_with_update_status.values())))
        if trues >= falses:
            print("Committing entry: " + self.current_operation)
            self.current_operation_committed = True
            self.key_value_store.write_to_state_machine(self.current_operation, term_absent=True, write=False)
            broadcast(self, with_return_address(self, "commit_entries ['" + self.current_operation + "']"))

            self.current_operation_committed = False
            for server_name in other_server_names(self.name):
                self.followers_with_update_status[server_name] = False
...
```

If we have quorum, then we commit the entry: we make the change in the KeyValueStore, updating its `data` dictionary to reflect the collection of keys and values produced by the most up-to-date version of the log. How do we do this? We sit in a `while` loop (see line 10 below) until we get quorum.

server.py Raw

```
...
        else:
            if self.leader:
                self.current_operation = string_operation

                if self.current_operation.split(" ")[0] in ["set", "delete"]:
                    key_value_store.write_to_log(string_operation, term_absent=True)
                    broadcast(self, with_return_address(self, "append_entries ['" + self.current_operation + "']"))

                    while not self.current_operation_committed:
                        pass

                    send_pending = False
                else:
                    response = key_value_store.read(self.current_operation)
            else:
                response = "I am not the leader. Please leave me alone."
...
```

After we commit the entry to the leader server's state machine, we need to let the follower servers know that the entry has been committed, so that they, too, can commit it to their state machines. So, on line 12, we broadcast a *new* message, covered in this commit. Followers respond like so:

server.py Raw

```
...

        elif string_operation.split(" ")[0] == "commit_entries":
            # followers do this to update their logs.
            stringified_logs_to_append = string_operation.replace("commit_entries ", "")
            print("Preparing to commit: " + stringified_logs_to_append)
            logs_to_append = ast.literal_eval(stringified_logs_to_append)
            [key_value_store.write_to_state_machine(command, term_absent=True) for command in logs_to_append]

            response = "Commit entries call successful!"
            print("State machine after committing: " + str(key_value_store.data))
...
```

If you check out this commit of the Raft repo, you are up to date on all of these changes.

This version ignores a lot of sad paths. A few existing fissures in our implementation:

1. **The leader is supposed to keep trying indefinitely if a call to a follower to append an entry fails.** For now, we're going to say that once an entry is committed, a server moves on. Later, we'll implement a follow catching up on multiple log statements from a leader, which will have the same effect in practice in many cases.
2. **For now, if a follower successfully receives the message to commit a command, we assume that it also succeeded in writing that command to its log.** So if a follower server comes back up right in the middle of this transaction, it could have an up-to-date state machine that is not reflected in its out-of-date log.
3. **For now, the cycle of sending out an append_entries call and waiting for quorum on it is blocking on its thread.** For now, I'm fine with this: If the follower servers aren't moving fast enough to get to quorum, I figure they're probably not fast enough to also be sending in loads of other requests.

We may come back to these fissures as we implement additional components of Raft. But I want to explicitly state our assumptions now, as well as where they fall down.

So far, follower servers still don't exercise any scrutiny on the logs: they write and commit what they're told to write and commit. They need to identify when their logs and states are out of date based on the requests they receive from the leader. That way, we can have confidence that our system produces identical logs even if two servers get disconnected for a while. So let's implement that.

I want the implementation to pass, at a minimum, the following set of manual tests:

1. Fire up a leader and a follower server, with the leader having a log with multiple commands and the follower having an empty log.
2. Check that the leader successfully replicates its logs on the follower.
3. Without shutting down either server, issue some write commands to the leader.
4. Check that the new write commands are accurately replicated on the follower log.
5. Shut down both servers.
6. Delete some number of commands from the tail of the follower's log. Replace them with commands that are different from the leader logs, or at least have different index-term pairs at the front.
7. Fire up the leader again.
8. Check that the leader successfully replaces the bogus logs on the follower with the correct logs.
9. Without shutting down either server, issue some write commands to the leader.
10. Check that the new write commands are accurately replicated on the follower log.

So how am I going to do this?

The Raft paper says:

> ***The leader keeps track of the highest index it knows to be committed, and it includes that index in future AppendEntries RPCs (including heartbeats) so that the other servers eventually find out.***

I previously suggested that I thought we might be able to avoid adding this index to the log by just assuming that whichever line a command appeared on in the context of the entire log as an ordered list would count as the index. The thing I wasn't thinking

about at the time was that:

1. I store the state as a dictionary. I do this so clients can ask for values by their keys and the server can look them up lightning-fast. To store them in a list would mean doing something with string parsing and filtering, which would be a) slow and b) no fun to write. BUT...
2. Dictionaries are *unordered* collections. You *can* get a list of the keys and see which one is "first" in the list, but that isn't guaranteed to correlate with which one you put in the dictionary first. FURTHERMORE...
3. Dictionaries don't tell you about change in their contents over time. If one log statement sets "a" to "apple" and a future log statement deletes a, then a third log statement sets "a" to "banana", at the point immediately after *any* of those calls, the dictionary contains *no* evidence that any of the previous calls existed. Indexes in Raft exist to help us coordinate the state of the system *over time.* So they can't be a computed property based on the state of any given server *right now*. They have to live in the logs.

So I went ahead and added the index to the front of each log statement in this commit. I initialize the `key_value_store` with an instance attribute of `highest_index`, set to zero (though this changes in a later commit). Then:

```
if term_absent:
    string_operation = str(self.latest_term) + " " + string_operation
    string_operation = str(self.highest_index) + " " + str(self.latest_term) + " " + string_operation
    self.highest_index = self.highest_index + 1

operands = string_operation.split(" ")
term, command, key, values = 0, 1, 2, 3
index, term, command, key, values = 0, 1, 2, 3, 4
```

I add it to the front of incoming commands that haven't been "marked" yet with their term, and each time the server writes to its logs it also increments that `highest_index` attribute.

Bless Python collection assignment that I can add a new element to the beginning of the string and a new index to the end with the confidence that everything referencing a named index will still work correctly.

Next, I need to include the index and term of the previous log entry in the AppendEntries call itself. A running theme in implementing raft is "The system needs to be able to heal itself if a server goes down and then comes back up," so I *cannot* rely on any of a server's state variables as a source of truth on the state of the system. All of it must be stored in the logs and communicated in the servers' calls to each other.

So here's the commit where I add the index to the `AppendEntries` call. I do another thing in this commit, which is to extract that call into its own object, with a method that gets it into string form and out of string form. This way, if any *more* changes happen to this call, I can change them in one spot. I have repeatedly been changing this call in two spots prior to now.

The new class:

26 src/append_entries_call.py

```
...  @@ -0,0 +1,26 @@
1    + import ast
2    +
3    + class AppendEntriesCall:
4    +     def __init__(self, previous_index, previous_term, entries):
5    +         self.previous_index = previous_index
6    +         self.previous_term = previous_term
7    +         self.entries = entries
8    +
9    +     def to_message(self):
10   +         return "append_entries after " +\
11   +                 str(self.previous_index) + " " +\
12   +                 str(self.previous_term) + " @" +\
13   +                 str(self.entries)
14   +
15   +     @classmethod
16   +     def from_message(cls, message):
17   +         context, entries_as_string = message.split("@")
18   +         previous_info = context.replace("append_entries after ", "").split(" ")
19   +         previous_index, previous_term = int(previous_info[0]), int(previous_info[1])
20   +         entries = ast.literal_eval(entries_as_string)
21   +
22   +         return AppendEntriesCall(
23   +             previous_index=previous_index,
24   +             previous_term = previous_term,
25   +             entries=entries
26   +         )
```

How it's used:

```
if string_operation.split(" ")[0] == "append_entries":
    # followers do this to update their logs.
    stringified_logs_to_append = string_operation.replace("append_entries ", "")
    print("Preparing to append: " + stringified_logs_to_append)
    logs_to_append = ast.literal_eval(stringified_logs_to_append)
    [key_value_store.write_to_log(log, term_absent=True) for log in logs_to_append]
    call = AppendEntriesCall.from_message(string_operation)
    [key_value_store.write_to_log(log, term_absent=True) for log in call.entries]
    print("State machine after appending: " + str(key_value_store.data))

    response = "Append entries call successful!"
```

This looks like pretty standard practice, right? WELL. I'll tell you, in January when I put down the Raft implementation, I extracted the request-parsing and repsonse-generation into their own method in a different file from the server. When I came back to it

this summer, one of my first commits undid that refactor. Why? “I can’t figure out what’s going on with all this indirection.” So we’ll see if this `AppendEntries` object survives first brush with someone who is trying to learn the code base. I’m starting to think many of the lessons we learn about refactoring aren’t as universal as they’re billed to be.

Then there’s the final piece: getting a follower’s state right when it’s catching up from its existing logs. So in this commit, we get the `KeyValueStore` to catch up its `latest_term` and `highest_index` attributes from the logs whenever a server starts up—otherwise it restarts from zero every time. I didn’t anticipate this, but it made itself known during manual testing (lol). Ze fix:

```
last_command = ''
for command in log.split('\n'):
    self.write_to_state_machine(command, term_absent=False, write=False)
    if command != '':
        last_command = command
        self.write_to_state_machine(command, term_absent=False, write=False)

if last_command != '':
    components = last_command.split(' ')

    #increment the index from the last call
    # so that the next log entry continues the count upward
    self.highest_index = int(components[0])
    self.latest_term = int(components[1])
```

Now that we have our index in place, things get interesting.

Let’s turn back to the Raft paper to see how followers are supposed to use this index to figure out if their logs are up to date or not.

When sending an AppendEntries RPC, the leader includes the index and term of the entry in its log that immediately precedes the new entries. **If the follower does not find an entry in its log with the same index and term, then it refuses the new entries.** The consistency check acts as an induction step: the initial empty state of the logs satisfies the Log Matching Property, and the consistency check preserves the Log Matching Property whenever logs are extended. As a result, whenever AppendEntries returns successfully, the leader knows that the follower’s log is identical to its own log up through the new entries.

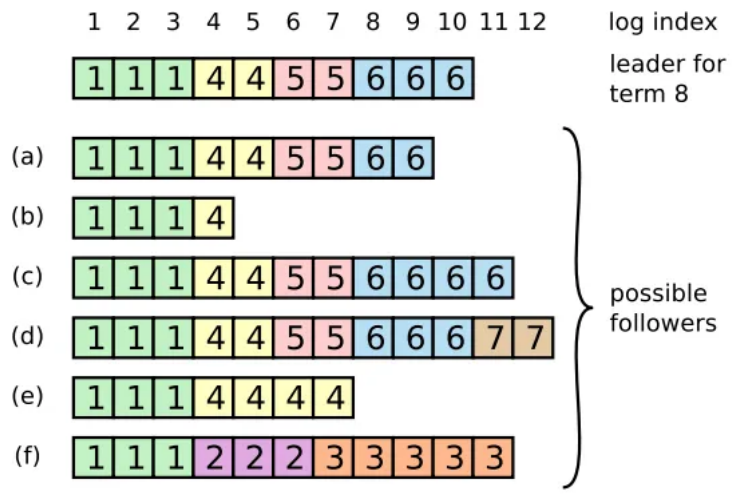


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

During normal operation, the logs of the leader and followers stay consistent, so the AppendEntries consistency check never fails. However, leader crashes can leave the logs inconsistent (the old leader may not have fully replicated all of the entries in its log). These inconsistencies can compound over a series of leader and follower crashes. Figure 7 illustrates the ways in which followers' logs may differ from that of a new leader. A follower may be missing entries that are present on the leader, it may have extra entries that are not present on the leader, or both. Missing and extraneous entries in a log may span multiple terms.

So now our followers need to check, when they receive any `AppendEntries` call, that they possess a line in their logs whose index and term *together* match the one that any lines in the `AppendEntries` call would be appended after.

How the heck am I gonna make *that* a quick lookup?

Whelp, here's what's convenient: any given combination of index and term in the log should be *unique* (only one of them) and *universal* (all servers should have the same set of index-term combinations). So it should be safe to store these combinations as keys, and the subsequent commands as values, in a dictionary.

That happens in this commit. Here's the juicy-juice:

```
def logs_as_dict(self, path_to_logs=''):
    as_dict = {}
    if path_to_logs == '':
        path_to_logs = "logs/" + self.server_name + "_log.txt"

    if os.path.exists(path_to_logs):
        f = open(path_to_logs, "r")
        log = f.read()
        f.close()

        for command in log.split('\n'):
            operands = command.split(" ")

            as_dict[" ".join(operands[:2])] = " ".join(operands[2:])

    print("DICT: " + str(as_dict))
    return as_dict

def previous_command(self, previous_index, previous_term):
    return self.logs_as_dict().get(str(previous_index) + " " + str(previous_term))
```

That code would represent this log:

```
1 0 set h 8
2 0 set i 9
3 0 set j 10
```

like so:

```
{ '1 0' : 'set h 8', '2 0': 'set i 9', '3 0': 'set j 10' }
```

We can then use that to check, when a follower server receives an `AppendEntries` call, whether the follower server has the last line of the log that the server has. If it does, we can go ahead and append whatever the leader has sent. If not, we need to inform the leader that we are not caught up.

src/server.py

```
@@ -137,10 +137,19 @@ def respond(self, key_value_store, operation):
137             if string_operation.split(" ")[0] == "append_entries":
138                 # followers do this to update their logs.
139                 call = AppendEntriesCall.from_message(string_operation)
-                [key_value_store.write_to_log(log, term_absent=True) for log in call.entries]
-                print("State machine after appending: " + str(key_value_store.data))
140
-                response = "Append entries call successful!"
141 +               if self.key_value_store.previous_command(
142 +                   call.previous_index,
143 +                   call.previous_term
144 +               ) != None:
145 +                   [key_value_store.write_to_log(log, term_absent=True) for log in call.entries]
146 +                   print("State machine after appending: " + str(key_value_store.data))
147 +
148 +                   response = "Append entries call successful!"
149 +               else:
150 +                   #TODO: Here is where we have to catch up the logs
151 +                   response = "Append entries unsuccessful. Please send prior log."
152 +
```

So now our follower server can successfully differentiate when it's ready to accept an `AppendEntries` command and when it isn't. Next, of course, comes the part where we catch it back up so that it *becomes* ready to accept those commands.

That means it's time to actually do something about out-of-date logs.

Here's what Raft says is supposed to happen (emphasis mine).

> ***In Raft, the leader handles inconsistencies by forcing the followers' logs to duplicate its own.*** *This means that conflicting entries in follower logs will be overwritten with entries from the leader's log. Section 5.4 will show that this is safe when coupled with one more restriction.*
>
> ***To bring a follower's log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower's log after that point, and send the follower all of the leader's entries after that point.*** *All of these actions happen in response to the consistency check performed by AppendEntries RPCs.*

OK. So this is going to work very similar to a professional disagreement, where the holders of the discordant opinions walk back step by step to figure out where they last agreed. Except with software, it's easier. We have to find the most recent index-term pair in the leader's logs that also appear in the follower's logs.

Let's look back at the paper.

> ***The leader maintains a nextIndex for each follower, which is the index of the next log entry the leader will send to that follower. When a leader first comes to power, it initializes all nextIndex values to the index just after the last one in its log (11 in Figure 7).*** *If a follower's log is inconsistent with the leader's, the AppendEntries consistency check will fail in the next AppendEntries RPC.* ***After a rejection, the leader decrements nextIndex and retries the AppendEntries RPC. Eventually nextIndex will reach a point where the leader and follower logs match. When this happens, AppendEntries will succeed, which removes any conflicting entries in the follower's log and appends entries from the leader's log (if any).*** *Once AppendEntries succeeds, the follower's log is consistent with the leader's, and it will remain that way for the rest of the term.*

So suppose a leader's log looks like this:

```
0 0 set zero 0
1 0 set a 1
2 0 set b 2
3 0 set c 3
4 0 set d 4
5 1 set something old
6 1 set something new
7 1 set something borrowed
8 1 set something blue
```

And a follower's log only has that first line: "o o set zero o." Each call from the leader will indicate what its last index-term pair is:

```
from every: received append_entries after 8 1 @
connecting to localhost port 10000
sending b'tom@append_entries_unsuccessful. Please send log prior to: 8 1' to ('local
host', 10000)
```

And if the follower doesn't have it, it will say so. Then the leader will send it's *next* most recent index-term pair—the one prior to the one mentioned in the follower's response message—while adding the command that the follower doesn't have to the list of commands that it's supposed to append.

```
connection from ('127.0.0.1', 57739)
from every: received append_entries after 7 1 @['8 1 set something blue']
connecting to localhost port 10000
sending b'tom@append_entries_unsuccessful. Please send log prior to: 7 1' to ('localhost', 10000)
connection from ('127.0.0.1', 57741)
from every: received append_entries after 6 1 @['7 1 set something borrowed', '8 1 set something blue']
connecting to localhost port 10000
sending b'tom@append_entries_unsuccessful. Please send log prior to: 6 1' to ('localhost', 10000)
connection from ('127.0.0.1', 57743)
from every: received append_entries after 5 1 @['6 1 set something new', '7 1 set something borrowed',
'8 1 set something blue']
connecting to localhost port 10000
sending b'tom@append_entries_unsuccessful. Please send log prior to: 5 1' to ('localhost', 10000)
connection from ('127.0.0.1', 57745)
from every: received append_entries after 4 0 @['5 1 set something old', '6 1 set something new', '7 1
set something borrowed', '8 1 set something blue']
connecting to localhost port 10000
sending b'tom@append_entries_unsuccessful. Please send log prior to: 4 0' to ('localhost', 10000)
connection from ('127.0.0.1', 57747)
from every: received append_entries after 3 0 @['4 0 set d 4', '5 1 set something old', '6 1 set someth
ing new', '7 1 set something borrowed', '8 1 set something blue']
connecting to localhost port 10000
sending b'tom@append_entries_unsuccessful. Please send log prior to: 3 0' to ('localhost', 10000)
connection from ('127.0.0.1', 57749)
from every: received append_entries after 2 0 @['3 0 set c 3', '4 0 set d 4', '5 1 set something old',
'6 1 set something new', '7 1 set something borrowed', '8 1 set something blue']
connecting to localhost port 10000
sending b'tom@append_entries_unsuccessful. Please send log prior to: 2 0' to ('localhost', 10000)
connection from ('127.0.0.1', 57751)
from every: received append_entries after 1 0 @['2 0 set b 2', '3 0 set c 3', '4 0 set d 4', '5 1 set s
omething old', '6 1 set something new', '7 1 set something borrowed', '8 1 set something blue']
connecting to localhost port 10000
sending b'tom@append_entries_unsuccessful. Please send log prior to: 1 0' to ('localhost', 10000)
connection from ('127.0.0.1', 57753)
```

This process repeats until it reaches an index-term pair that the two servers share.

```
from every: received append_entries after 0 0 @['1 0 set a 1', '2 0 set b 2', '3 0 set c 3', '4 0 set d
 4', '5 1 set something old', '6 1 set something new', '7 1 set something borrowed', '8 1 set something
 blue']
Writing to log: 1 0 set a 1
Writing to log: 2 0 set b 2
Writing to log: 3 0 set c 3
Writing to log: 4 0 set d 4
Writing to log: 5 1 set something old
Writing to log: 6 1 set something new
Writing to log: 7 1 set something borrowed
Writing to log: 8 1 set something blue
State machine after appending: {}
connecting to localhost port 10000
sending b'tom@Append entries call successful!' to ('localhost', 10000)
```

You'll notice something about my implementation here: the state machine on the follower is *empty*. Why? Because we don't need to implement updates to a state machine that isn't used. When a client asks a follower for a value based on a key, it replies that it is not the leader, and that only the leader responds to such requests. Were this server to *become* the leader, *then* we would need to update the state machine.

I'm not thrilled with [my implementation of the above] because I didn't come up with an elegant way to store what I need. Now, *not only* do I need to fetch commands by their index-term combination, but *also* I need to be able to fetch the

command *prior* to the one associated with a given index-term combination.

I suppose I could have done a doubly linked list of key-value pairs, but Python doesn't come with that kind of data structure, and frankly I didn't feel like implementing a data structure in a language that doesn't have it as means to an end in a fun side project. So instead I did this:

```
class CanYouBelieveThis:
    def __init__(self, array, dict):
        self.the_worst_array = array
        self.only_marginally_better = dict
```

Just kidding. I named it better than that:

src/log_data_access_object.py

```
@@ -0,0 +1,7 @@
+
+
+ class LogDataAccessObject:
+
+     def __init__(self, array, dict):
+         self.ordered_logs = array
+         self.term_indexed_logs = dict
```

I now populate this object instead of a lone dictionary with term-index combinations as keys and commands as values. In addition to that dictionary (which goes into `term_indexed_logs` on the `LogDataAccessObject`), I make a list of the index-term combinations and put that into `ordered_logs`. You can see those changes here:

```
class KeyValueStore:
    client_lock = threading.Lock()

@ -47,8 +49,9 @@ def catch_up(self, path_to_logs=''):

        self.catch_up_successful = True

    def logs_as_dict(self, path_to_logs=''):
    def log_access_object(self, path_to_logs=''):
        as_dict = {}
        the_worst_array = []
        if path_to_logs == '':
            path_to_logs = "logs/" + self.server_name + "_log.txt"

@ -58,15 +61,17 @@ def logs_as_dict(self, path_to_logs=''):
            f.close()

            for command in log.split('\n'):
                operands = command.split(" ")
                if command != '':
                    operands = command.split(" ")

                as_dict[" ".join(operands[:2])] = " ".join(operands[2:])
                    as_dict[" ".join(operands[:2])] = " ".join(operands[2:])
                    the_worst_array.append(" ".join(operands[:2]))

        print("DICT: " + str(as_dict))
        return as_dict
        return LogDataAccessObject(array=the_worst_array, dict=as_dict)

    def previous_command(self, previous_index, previous_term):
        return self.logs_as_dict().get(str(previous_index) + " " + str(previous_term))
    def command_at(self, previous_index, previous_term):
        return self.log_access_object().term_indexed_logs.\
            get(str(previous_index) + " " + str(previous_term))
```

So then, when a leader gets a 'Logs not updated' message from a follower and needs to back up a command and send that in an `AppendEntries` response, it can:

1. Get the index in `ordered_logs` of the index-term pair that the follower *didn't* have,
2. Decrement this index to get the term-index pair of the *prior* command,
3. Use this term-index pair to get the prior command from `term_indexed_logs`
4. Add that pair to the list of entries to instruct the follower to append,

5. Use the term-index pair acquired in step 2 as the new pair for the follower to append entries after.

This block of code is kind of long and in teeny font, but feel free to peruse, and if you don't want to peruse it, rest assured that the five steps above explain what it does.

```
        response = "append_entries_unsuccessful. Please send log prior to: " + str(call.previous_index) + " " + str(call.previous_term)
elif string_operation.split(" ")[0] == "append_entries_unsuccessful.":

    response_components = string_operation.split(" ")
    max_index = len(response_components)

    latest_tried_index = int(response_components[max_index - 2])
    latest_tried_term = int(response_components[max_index - 1])

    log_position = self.key_value_store.log_access_object().ordered_logs.index(
        str(latest_tried_index) + " " + str(latest_tried_term)
    )

    #TODO: if log_position < 1:
        #if we are starting from scratch

    ordered_logs = self.key_value_store.log_access_object().ordered_logs
    term_indexed_logs = self.key_value_store.log_access_object().term_indexed_logs
    new_key_to_try = ordered_logs[log_position - 1]

    new_values_to_send = list(
        map(
            lambda x: term_indexed_logs[x],
            ordered_logs[log_position:]
        )
    )

    try_this_index = new_key_to_try.split(" ")[0]
    try_this_term = new_key_to_try.split(" ")[1]

    response = AppendEntriesCall(
        previous_index=try_this_index,
        previous_term=try_this_term,
        entries=new_values_to_send
    ).to_message()
```

Yes, I am aware that this block of code would be very slow if I had millions of log entries. For one thing it calls `log_access_object()`, which we saw has a ton of logic, *three separate times*, when it could easily cache the result after the first call. I'm sure we'll have more to do in this block, which will give us an opportunity to revisit optimizing it later.

But what about deleting any discordant entries in the follower log?

At this point, the follower doesn't delete any entries that appear *after* the one where it agrees with the leader. So anything the leader appends would go on top of those discordant entries. Let's delete any entries that come after the entry where a leader and follower diverge. So let's fix that. This commit is pretty small. We add a method in the `KeyValueStore` to clear out those log entries:

src/key_value_store.py

```
@@ -49,6 +49,22 @@ def catch_up(self, path_to_logs=''):

            self.catch_up_successful = True

+     def remove_logs_after_index(self, index, path_to_logs=''):
+         if path_to_logs == '':
+             path_to_logs = "logs/" + self.server_name + "_log.txt"
+
+         if os.path.exists(path_to_logs):
+             f = open(path_to_logs, "r+")
+             log_list = f.readlines()
+
+             while '' in log_list:
+                 log_list.remove('')
+             f.seek(0)
+             for line_to_keep in log_list[0:index]:
+                 f.write(line_to_keep)
+             f.truncate()
+             f.close()
+
```

And then we call it when the follower has found the term-index pair it's supposed to append after, such that any commands that supersede the one with that term-index pair go away:

src/server.py

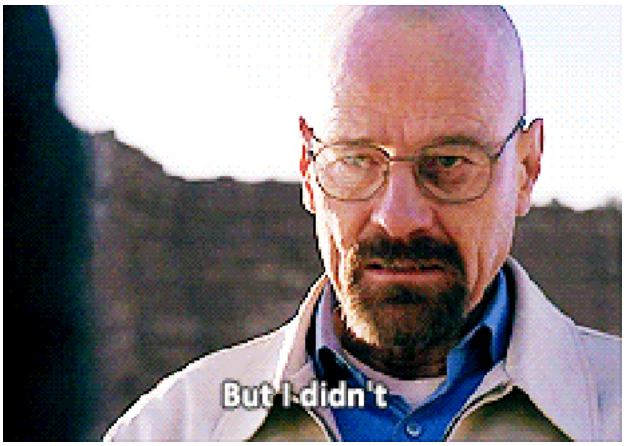
```
@@ -142,7 +142,8 @@ def respond(self, key_value_store, operation):
                call.previous_index,
                call.previous_term
            ) != None:
-               [key_value_store.write_to_log(log, term_absent=True) for log in call.entries]
+               key_value_store.remove_logs_after_index(call.previous_index)
+               [key_value_store.write_to_log(log, term_absent=False) for log in call.entries]
                print("State machine after appending: " + str(key_value_store.data))

                response = "Append entries call successful!"
```

But wait, Chelsea! There's another problem! What if the follower's logs are completely empty?

Good catch. The code we have written will choke if we try to append entries to an empty log, because the leader will *never find* a term-index pair in the follower's log that matches one of its own.

Could I have included conditional logic in the `AppendEntries` response block to deal with this? Of course.



Instead, in this commit, I start instantiating the `KeyValueStore`'s `highest_index` to `1` instead of 0. Then, in the zero spot (when a server starts up and its log is empty), I add a "synthetic" (fake, fabricated, I made it up) command so that every server has the same first log entry.

```
def catch_up(self, path_to_logs=''):
    if path_to_logs == '':
        path_to_logs = "logs/" + self.server_name + "_log.txt"

    if os.path.exists(path_to_logs):
        f = open(path_to_logs, "r")
        log = f.read()
        f = open(path_to_logs, "r+")

        all_lines = f.read().splitlines()
        non_empty_lines = list(filter(lambda x: x != '', all_lines))
        if len(non_empty_lines) == 0:
            f.seek(0)
            f.write("0 0 set  unreachable\n")
        f.close()
```

This command is lying—it's reachable, but you'd only try it if you know it's there. It sets a value on the key of an empty string. In order to get ahold of it, you need to type into a client "get" and then hit the spacebar twice. It works:

```
Type your message:
client@get
sending client@get
received b'every@unreachable'
Type your message:
```

But I don't anticipate an *empty string* becoming an overloaded key that everyone wants to use in their state machine. And you know what, so what if they do? This can be overwritten without a problem. The only reason it's there is to ensure that *every* server's log does have *one* common term-index pair at its root.

Do I feel bad about this decision? Not even a little bit.

As far as I'm concerned, it removes an edge that I no longer have to handle.

Look, the Raft paper *itself* explicitly *tells* us to use an `AppendEntries` call as the heartbeat. As far as I can tell, this has no practical reason besides "Honestly any RPC would have done fine here, so we reused the one we already had."

If they get to do that, I get to do this.

There's one last thing.

It is a lot of back-and-forth for the leader server to send the singular immediate precedent call whenever a follower says it's not caught up, and there are ways to reduce the number of calls here. I'll include the Raft paper's comments on this in an appendix below.

In the meantime, though, we have basic log replication. Get up, make some tea, and celebrate our victory! Then, we take up a new task: leader elections.

Part 6 Appendix: Optimizing Log Catchup, as per the Raft Paper

> *If desired, the protocol can be optimized to reduce the number of rejected AppendEntries RPCs. For example, when rejecting an AppendEntries request, the follower 7 can include the term of the conflicting entry and the first index it stores for that term. With this information, the leader can decrement nextIndex to bypass all of the conflicting entries in that term; one AppendEntries RPC will be required for each term with conflicting entries, rather than one RPC per entry. In practice, we doubt this optimization is necessary, since failures happen infrequently and it is unlikely that there will be many inconsistent entries.*

> *With this mechanism, a leader does not need to take any special actions to restore log consistency when it comes to power. It just begins normal operation, and the logs automatically converge in response to failures of the AppendEntries consistency check. A leader never overwrites or deletes entries in its own log (the Leader Append-Only Property in Figure 3).*
>
> *This log replication mechanism exhibits the desirable consensus properties described in Section 2: Raft can accept, replicate, and apply new log entries as long as a majority of the servers are up; in the normal case a new entry can be replicated with a single round of RPCs to a majority of the cluster; and a single slow follower will not impact performance.*

Part 7: Leader Election

Now, our follower servers can tell when their logs are out of date. They send that information to the leader, who can then backtrack to a common log entry and catch up the follower server from there.

So our system self-heals. But who is the healer?

Illustration from Game Outline

So far the implementation relies on me, its overlord, to arbitrarily designate a server as the leader when I start it up. If I bring the leader down*, there is no more leader. No one will replicate the logs with the other servers. No one will even accept commands from a client. What a sad situation that would be.

The fix for this in raft: leader election. That is, the remaining follower servers will choose a leader amongst themselves.

I want the implementation to pass, at a minimum, the following set of manual tests:

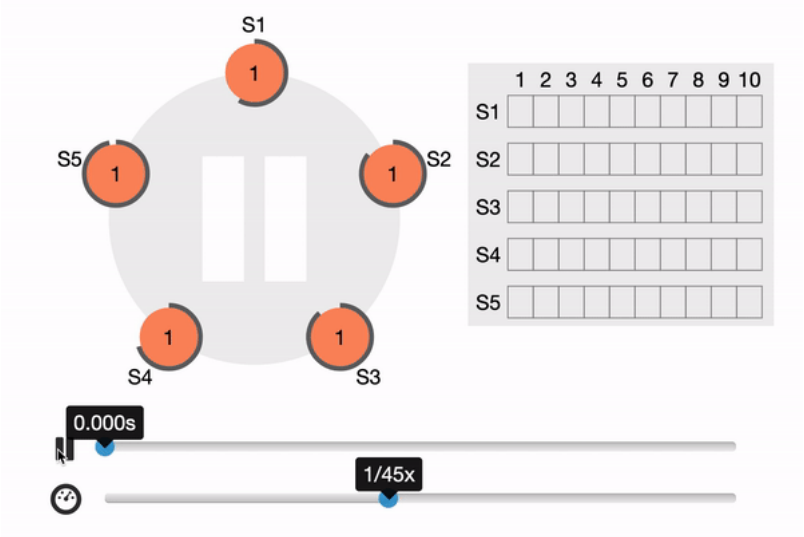
1. Fire up a leader and two follower servers, with the leader having a log with multiple commands and the followers having empty logs.
2. Once the leader has started issuing heartbeats to the followers, shut it down.
3. Ensure that the two remaining followers choose a leader between themselves.
4. Start up a client and issue some read commands to that leader. Ensure that its state matches the totality of its current logs.
5. Issue some write commands to that leader. Ensure that it replicates those commands on the follower.
6. Restart the former leader server, this time as a follower. Ensure that the new leader catches up the restarted server on the commands issued while it was down.
7. Shut down the current leader server.
8. Ensure that the remaining servers again choose a leader.
9. Write some commands to this leader, and ensure they are replicated.
10. Again restart the downed server and ensure the new leader brings it back up to date.

How does the Raft paper prescribe that we implement this?

> *5.2 LEADER ELECTION*
>
> *Raft uses a heartbeat mechanism to trigger leader election. When servers start up, they begin as followers. A server remains in follower state as long as it receives valid RPCs from a leader or candidate. Leaders send periodic heartbeats (AppendEntries RPCs that carry no log entries) to all followers in order to maintain their authority.* ***If a follower receives no communication over a period of time called the election timeout, then it assumes there is no viable leader and begins an election to choose a new leader.***
>
> -The Raft Paper

This can be a tough thing to picture from text alone, so I took a gif from RaftScope. If you want to gain a better understanding of Raft but you don't feel like undertaking to write it (honestly what a ridiculous pastime to *actually write it*, who would do such a thing?), RaftScope shows you a diagram of a server cluster and a handy term table. It also lets you start/stop servers, change election timeouts, and even change the playback speed, so you can see for yourself what would happen in whatever situation you manage to contrive!

Anyway, here's what it looks like when five servers all start as followers with slightly different election timeouts, shown here as the grey arcs surrounding each server:



A step-by-step explanation of the gif above:

1. The first server to time out (S1) starts an election (it turns blue at about 0.120s).
2. S1 votes for itself (the solid dot in the five dots that appear when it turns blue).
3. S1 sends out RequestVote RPCs (the solid green dots). These are all the first RequestVote RPC that any of the servers have received this term, so they all respond with their votes (the green dots with crosses in them).
4. S1 then becomes the leader (the solid black border that appears around it in lieu of a timeout arc, since leaders don't have timeouts for hearing from the leader!)
5. S1 sends out its first heartbeat (the solid orange dots at about 0.230s).
6. When each of the other servers receives the heartbeat, their grey arcs increase to demonstrate that the election timeout resets each time a follower receives a valid AppendEntries RPC. This will prevent them from starting new elections.

Time to make it work!

First, I needed to implement election timeout, which I do [in this commit](). There are three steps here: give each server a timeout, start a timer that counts down from that timeout, and start an election when that timer hits zero. I set the timeout and start the timer when the server gets instantiated, and I call a method `start_election` when that timer gets to zero.

```
class Server:
    def __init__(self, name, port=10000, leader=False):
        self.port = port
        self.name = name
        self.key_value_store = KeyValueStore(server_name=name)
        self.key_value_store.catch_up()
        self.term = self.key_value_store.get_latest_term()
        self.leader = leader
        self.followers_with_update_status = {}
        self.current_operation = ''
        self.current_operation_committed = False

        self.candidate = False
        self.timeout = float(random.randint(5, 30))
        self.election_countdown = threading.Timer(self.timeout, self.start_election)
        self.election_countdown.start()

        for server_name in other_server_names(name):
            self.followers_with_update_status[server_name] = False

    def start_election(self):
        if not self.leader:
            print("THE LEADER IS DOWN LETS START AN ELECTION BOIIIII")
```

As you can see, the `start_election` method doesn't start an election—it just impersonates Jason Mendoza.

Right now, it's going to do that just once, when the original timeout happens, and never again, and there's nothing any other server can do to stop it.

So I reset the timer each time a follower server receives an `append_entries` call, which means there's a server leader acting as the leader somewhere.

```
        if string_operation.split(" ")[0] == "append_entries":
            # followers do this to update their logs.
            self.election_countdown.cancel()
            self.election_countdown = threading.Timer(self.timeout, self.start_election)
            self.election_countdown.start()
```

Great, so we have our election timeouts set up. Now it's time to start an election.

So what does it mean to start an election?

Once again, I'll defer to the paper:

> *To begin an election,* ***a follower increments its current term and transitions to candidate state. It then votes for itself and issues RequestVote RPCs in parallel to each of the other servers in the cluster.*** *A candidate continues in this state until one of three things happens:*
>
> *(a) it wins the election,*
> *(b) another server establishes itself as leader, or*
> *(c) a period of time goes by with no winner. These outcomes are discussed separately in the paragraphs below.*

We'll get to the three things that happen later. First, let's get `start_election` doing something more useful (albeit, perhaps, less entertaining.)

```
43          def start_election(self):
44              if not self.leader:
    -               print("THE LEADER IS DOWN LETS START AN ELECTION BOIIIII")
45  +               self.key_value_store.current_term += 1
46  +               self.candidate = True
47
48  +               self.voted_for_me[self.name] = True
49  +               broadcast(self, with_return_address(
50  +                   self,
51  +                   "can_I_count_on_your_vote_in_term " + str(self.key_value_store.current_term)
52  +               ))
```

First, we increment the term (line 45), put the server in the candidate state (line 46), have the server vote for itself (line 48), and send out a new request to all the other servers asking for their vote (lines 49-51).

In the server initializer, we'll instantiate a dictionary to store votes. We will use this dictionary to determine when a majority of servers have voted for this server as a leader, much in the way we store responses to `append_entries` to determine when a majority have replicated the logs.

```
27            self.candidate = False
      -       self.timeout = float(random.randint(5, 30))
28    +       self.timeout = float(random.randint(10, 18))
29            self.election_countdown = threading.Timer(self.timeout, self.start_election)
30    +       print("Server started with timeout of : " + str(self.timeout))
31            self.election_countdown.start()
32    +       self.voted_for_me = {}
33    +       self.voted_this_term = False
34
35            for server_name in other_server_names(name):
36                self.followers_with_update_status[server_name] = False
37
38    +       for server_name in other_server_names(name):
39    +           self.voted_for_me[server_name] = False
40    +       self.voted_for_me[self.name] = False
```

I also changed the timeout, which previously ranged from 5 to 30 seconds, to range between 10 and 18 seconds. I did this so I knew I'd have more than 5 seconds to start up all of my servers before one timed out, but if I were waiting for timeouts to happen, I wouldn't have to stare at my screen for 30 seconds.

In a "real" implementation, these timeouts would be hundreds of milliseconds, not several seconds. I have things happening very slowly in this implementation so I don't have to scour copious logs to find an error that I just saw fly by. The point here is to implement and learn, not to actually build a production-ready system for multi-master replication across SVN repositories for Google Code (though apparently they used Paxos for that).

So I made up this new call, `can_I_count_on_your_vote_in_term`. Servers need to be able to respond to that. They also need to be able to *interpret* a response to that. So we add two new items to our response conditional:

```
267     elif string_operation.split(" ")[0] == "can_I_count_on_your_vote_in_term":
268         request_vote_call = RequestVoteCall.from_message(string_operation)
269         if request_vote_call.for_term > self.key_value_store.current_term \
270             and request_vote_call.latest_log_term >= self.key_value_store.latest_term_in_logs \
271             and request_vote_call.latest_log_index >= self.key_value_store.highest_index:
272             if self.voted_this_term:
273                 response = "Sorry, already voted."
274             else:
275                 self.voted_this_term = True
276                 response = "You can count on my vote!"
277         else:
278             response = "Your log is out of date. I'm not voting for you!"
```

First, if a server receives a request for a vote (line 267), it looks at the term for which the other server is requesting a vote (line 269). First, it determines if the server is out of date by ensuring that it doesn't think the next term is one that has already passed.

It also makes sure that the server requesting the vote has a log *at least* as up to date as the voting server's log. As per the paper:

> *Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs. If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.*

So, we check that the latest term in the candidate's log is at least as high as the latest term in the voting server's log (line 270), and then that the candidate's log is at least as long as the voting server's log (line 271). (Since the indices increment for every line

in the log, we can use the index as an indicator of the size of the log).

Finally, the voting serve2 determines whether it has voted yet in this term (line 272) . Each server only votes for one server per term—the first one to contact it, or itself, if it's the one starting the election. If not, it votes for the server who requested the vote and sets a boolean to remember that it *has* voted this term (line 275). Anyone else who contacts it for this term will now get a "Sorry, already voted" response (line 273).

Later, I remove `already_voted` from the server altogether by incrementing each server's term *at the time* that it votes for another server. That's what's prescribed by Raft, and it results in any other servers requesting votes for this term getting denied because they're no longer requesting votes for a term greater than the one the voting server is currently in. You can see that change [in this commit](). The juicy addition:

```
274             elif string_operation == "You can count on my vote!":
275                 self.mark_voted(server_name)
276 +               self.key_value_store.current_term += 1
```

And removal:

```
272 -                   if self.voted_this_term:
273 -                       response = "Sorry, already voted."
274 -                   else:
275 -                       self.voted_this_term = True
```

To make it less onerous to include the candidate term, log term, and log index in the request vote call, I [extracted the stringification and hydration of that call]() into its own class similar to `AppendEntries`:

src/request_vote_call.py

```
@@ -0,0 +1,26 @@
+
+ class RequestVoteCall:
+     def __init__(self, for_term, latest_log_index, latest_log_term):
+         self.for_term = for_term
+         self.latest_log_index = latest_log_index
+         self.latest_log_term = latest_log_term
+
+     def to_message(self):
+         return "can_I_count_on_your_vote_in_term " \
+                + str(self.for_term) \
+                + " ? " \
+                + "last_log_entry: " \
+                + str(self.latest_log_index) \
+                + " " \
+                + str(self.latest_log_term)
+
+     @classmethod
+     def from_message(cls, message):
+         numeric_markers = message.replace("can_I_count_on_your_vote_in_term ", "").split(" ")
+         current_term, index, latest_log_term = int(numeric_markers[0]), int(numeric_markers[3]), int(numeric_markers[4])
+
+         return RequestVoteCall(
+             for_term=current_term,
+             latest_log_index=index,
+             latest_log_term=latest_log_term,
+         )
```

Then...

```
279     elif string_operation == "You can count on my vote!":
280         self.mark_voted(server_name)
281         send_pending = False
```

When a server that requested votes *receives* those votes (line 279), it updates its dictionary where it keeps track of who voted for it (line 280). This implementation looks similar to the one we used for determining when to commit entries to the state machine:

```
128 +     def mark_voted(self, server_name):
129 +         self.voted_for_me[server_name] = True
130 +
131 +         trues = len(list(filter(lambda x: x is True, self.voted_for_me.values())))
132 +         falses = len(list(filter(lambda x: x is False, self.voted_for_me.values())))
133 +         if trues >= falses:
134 +             print("I win the election for term " + str(self.key_value_store.current_term) + "!")
135 +             self.key_value_store.catch_up(new_leader=True)
136 +             self.leader = True
137 +
138 +             self.prove_aliveness()
139 +
140 +             for server_name in other_server_names(self.name):
141 +                 self.voted_for_me[server_name] = False
142 +             self.voted_for_me[self.name] = False
```

Every vote triggers a check for whether a majority of servers have voted for this one (line 133), which we get by comparing the number of keys that point to 'True' values with the number of keys that point to 'False' values. Is it super efficient? No. Am I super

concerned about the efficiency of loop-filtering over a dictionary with 5 key-value pairs? Also no.

The server wins! What do we do next (besides, of course, issue an appropriately courteous and triumphant public statement)?

Here's what we do on a newly-elected server:

1. Update the state machine to reflect the logs (line 135). We haven't been doing that continuously because followers don't access their state machines (as discussed over here), but now that this server is a leader, that state machine needs to be ready to respond to client requests.
2. Set this server to be the leader (line 136). Now, anywhere that behavior is conditional on leadership, it will know what it is.
3. Send out an `append_entries` call to ~~assert dominance~~ let the other servers know that they have a leader they can count on (line 138). This will prevent them from starting new elections.
4. Reset the voting dictionary so that everything is false (lines 140-142). That way, if this server somehow loses leadership and then starts a new election, its dictionary will be ready.

At this point, we can get to step 5 in our manual steps above. But when we restart our former leader, things get off the rails. That's because that server doesn't have the updated term, and the other servers aren't checking for an updated term. If that server tried to start an election, it would get "Sorry, already voted" from everyone. Meanwhile, if a server with an outdated term purported to be leader (perhaps upon reconnecting following a partition), it could successfully send `append_entries` calls because the followers don't check the term before naively accepting these commands.

We're about to fix those things.

In Raft, state belongs in the messages.

A system that is resilient to servers going down cannot rely on the state of those servers as a source of truth. So servers need to include all relevant context in their messages to each other.[1] In this case, the current term is relevant context. So the `append_entries` calls need to include it, as per this part of the Raft paper:

> *While waiting for votes, a candidate may receive an AppendEntries RPC from another server claiming to be leader.* ***If the leader's term (included in its RPC) is at least as large as the candidate's current term, then the candidate recognizes the***

> ***leader as legitimate and returns to follower state.*** *If the term in the RPC is smaller than the candidate's current term, then the candidate rejects the RPC and continues in candidate state. '*

We add the term to the `append_entries` commands [in this commit](#).

Conveniently, we extracted the `AppendEntries` call to its own object that helpfully converts all the different components to and from the string format for us, so adding the current term to the call isn't too onerous:

```
3   class AppendEntriesCall:
-       def __init__(self, previous_index, previous_term, entries):
4   +   def __init__(self, in_term, previous_index, previous_term, entries):
5   +       self.in_term = in_term
6           self.previous_index = previous_index
7           self.previous_term = previous_term
8           self.entries = entries
9
10      def to_message(self):
-           return "append_entries after " +\
11  +       return "append_entries in_term " \
12  +               + str(self.in_term) + \
13  +               " after " +\
14                  str(self.previous_index) + " " + \
15                  str(self.previous_term) + " @" + \
16                  str(self.entries)
17
18      @classmethod
19      def from_message(cls, message):
20          context, entries_as_string = message.split("@")
-           previous_info = context.replace("append_entries after ", "").split(" ")
-           index, term = int(previous_info[0]), int(previous_info[1])
21  +       numeric_markers = context.replace("append_entries in_term ", "").split(" ")
22  +       current_term, index, latest_log_term = int(numeric_markers[0]), int(numeric_markers[2]), int(numeric_markers[3])
23          entries = ast.literal_eval(entries_as_string)
24
25          return AppendEntriesCall(
26  +           in_term=current_term,
27              previous_index=index,
-             previous_term= term,
28  +           previous_term= latest_log_term,
29              entries=entries
30          )
```

We then check this latest term when a server receives an `append_entries` call:

```
184 +               if call.in_term < self.key_value_store.current_term:
185 +                   response = "Your term is out of date. You can't be the leader."
```

Everything we previously did in an `append_entries` response—canceling the election timer, checking that the logs are up to date, appending any entries provided by the call—only happens in an *else* block below the conditional on line 184.

I also added a heartbeat timer here. This is because only the leader should send heartbeats. If a purported leader or candidate server receives an `append_entries` call from a server that turns out to have a later term than it does, it reverts to the follower state. So if it was previously sending heartbeats, it now needs to stop.

I start that up when the server becomes the leader:

```
115 +            self.heartbeat_timer = threading.Timer(5.0, self.prove_aliveness)
116 +            self.heartbeat_timer.start()
```

and I cancel it in the event of an up-to-date `append_entries` call:

```
191 +                self.leader = False
192 +                if self.heartbeat_timer:
193 +                    self.heartbeat_timer.cancel()
194 +                self.key_value_store.current_term = call.in_term
195 +
```

At this point, we can get through our ten-step manual test list...most of the time.

There's just one more thing.

What if two servers call an election simultaneously, each garner half of the votes, and no one is elected? They'll ultimately hit their election timeouts and start new elections. But if their timeouts are *the same,* this could happen indefinitely.

Sounds like an edge case, right? Well, it's an easy one to hit when you're running a three server cluster and you bring down the leader. You then have two follower servers, both of whom have timeouts that are some integer between 8 and 18 (inclusive). There's a 1 in 121 chance that they have the same timeout, which translates to a 99% chance that this is going to happen at *some* point within about 500 trials. It happened to me within 10, because I'm just lucky that way .

Since both servers vote for themselves first, it's not even a race to see who can get their vote requests to another follower the fastest: they'll keep voting for themselves, declining to vote for the other server, timing out, and starting over.

So what's the solution? Reset the election timeouts at the start of every election:

> *Each candidate restarts its randomized election timeout at the start of an election, and it waits for that timeout to elapse before starting the next election; this reduces the likelihood of another split vote in the new election.*

We implement that in this commit:

```
45        def start_election(self):
46            if not self.leader:
47                self.key_value_store.current_term += 1
    -             self.candidate = True
48  +             self.timeout = float(random.randint(10, 18))
49  +             self.election_countdown = threading.Timer(self.timeout, self.start_election)
50  +             print("Server reset election timeout to : " + str(self.timeout))
51  +             self.election_countdown.start()
```

I also removed the "candidate" attribute here because, it turns out, I wasn't using it for anything. Instead, servers exist in either the leader or the follower state, and they communicate their candidacy by sending out RequestVote RPCs as followers.

This concludes the implementation of all the *big* pieces. We have a couple of small changes remaining, which I'll show you in the next piece, and finally we'll wrap up this whole thing :).

1. There's a lesson here, by the way, for working in teams. Imagine, for a moment, that the Raft servers are all team members, and the log is their collaborative project. Servers (team members) might join the cluster (get hired), leave the cluster (configuration changes), or temporarily become unavailable (illness, vacation, competing responsibilities) at any time. Under these circumstances, we know that Raft needs to store context, not with any one server, but rather in their shared project itself (the log) or in their messages to one another (the RPCs). So why are we perfectly content on teams to have one person "own" something that they don't communicate, or inadequately communicate? Furthermore, on software teams, not only are teammates usually not evaluated on whether they have communicated—indeed, communication is actively de-prioritized, and developers learn that they receive accolades for cranking features—not by making their work discoverable to anyone else. I have written some pieces about innovative context-sharing for software teams, but ultimately, will anyone use these techniques before they're professionally incentivized and rewarded to do so?

Part 8: Maintenance Repairs

We have one more thing to implement, but before we do, I wanted to go in and clean up some personal annoyances. We've talked about one of these before and I've silently fumed at the other two, so now we'll fix all three.

Fix #1: Cache a Computationally Intensive Operation

In part 7, I showed you this monstrosity and promised you I'd come back to it:

```
        response = "append_entries_unsuccessful. Please send log prior to: " + str(call.previous_index) + " " + str(call.previous_term)
elif string_operation.split(" ")[0] == "append_entries_unsuccessful.":

    response_components = string_operation.split(" ")
    max_index = len(response_components)

    latest_tried_index = int(response_components[max_index - 2])
    latest_tried_term = int(response_components[max_index - 1])

    log_position = self.key_value_store.log_access_object().ordered_logs.index(
        str(latest_tried_index) + " " + str(latest_tried_term)
    )

    #TODO: if log_position < 1:
        #if we are starting from scratch

    ordered_logs = self.key_value_store.log_access_object().ordered_logs
    term_indexed_logs = self.key_value_store.log_access_object().term_indexed_logs
    new_key_to_try = ordered_logs[log_position - 1]

    new_values_to_send = list(
        map(
            lambda x: term_indexed_logs[x],
            ordered_logs[log_position:]
        )
    )

    try_this_index = new_key_to_try.split(" ")[0]
    try_this_term = new_key_to_try.split(" ")[1]

    response = AppendEntriesCall(
        previous_index=try_this_index,
        previous_term=try_this_term,
        entries=new_values_to_send
    ).to_message()
```

Yes, I am aware that this block of code would be very slow if I had millions of log entries. For one thing it calls `log_access_object()`, which we saw has a ton of logic, *three separate times*, when it could easily cache the result after the first call. I'm sure we'll have more to do in this block, which will give us an opportunity to revisit optimizing it later.

I want to make good on that promise. I had a couple of options:

1. Store the result of `log_access_object` at the call site and access that variable thereafter rather than calling the method multiple times
2. Cache the result of the method in the `KeyValueStore` so that multiple method calls would not redo the computation as long as the logs had not changed between the two calls.

I decided on the second of these two approaches: that way, a client operation can use this method multiple times without having to choose between learning `KeyValueStore`'s implementation details and ending up with something super slow.

It'll still have to redo the calculation with every change to the logs, but for now, I decided, this commit was enough to satisfy me.

It instantiates an attribute on `KeyValueStore` called `log_recently_changed`,

```
@@ -15,6 +15,8 @@ def __init__(self, server_name):
            self.latest_term_in_logs = 0
            self.highest_index = 1

+           self.log_recently_changed = False
+
```

Stores the result of the first run of `log_access_object` in an instance variable,

```
-               as_dict[" ".join(operands[:2])] = " ".join(operands)
-               the_worst_array.append(" ".join(operands[:2]))
+           self.cached_log_access_object = LogDataAccessObject(array=the_worst_array, dict=as_dict)
+           self.log_recently_changed = False

-       return LogDataAccessObject(array=the_worst_array, dict=as_dict)
+       return self.cached_log_access_object
```

Keys off whether the logs recently changed to determine whether to send the current cached instance variable or replace that variable with the computation,

```
    def log_access_object(self, path_to_logs=''):
        as_dict = {}
        the_worst_array = []
        if path_to_logs == '':
            path_to_logs = "logs/" + self.server_name + "_log.txt"
        if not self.log_recently_changed:
            pass
        else:
            as_dict = {}
            the_worst_array = []
            if path_to_logs == '':
                path_to_logs = "logs/" + self.server_name + "_log.txt"
```

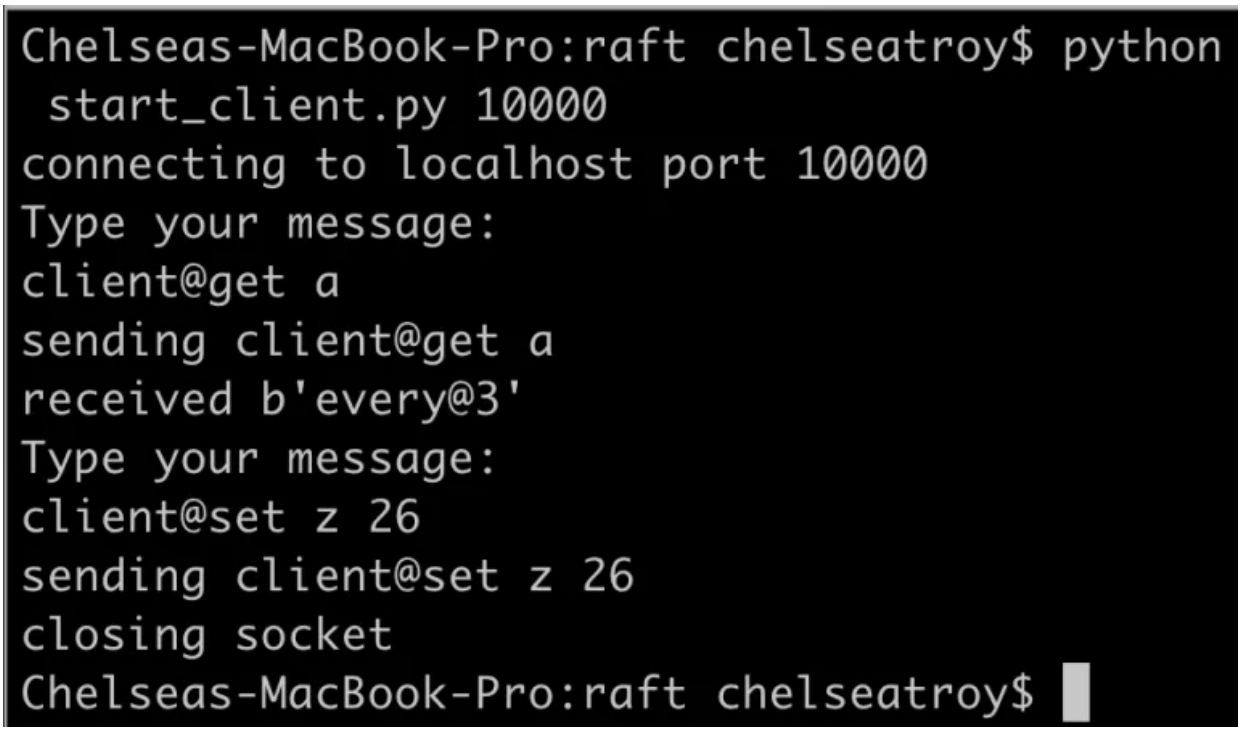
And sets `logs_recently_changed` to True in every method that updates the logs, and False in the `log_recently_changed` method itself.

```
def write_to_log(self, string_operation, term_absent, path_to_logs=''):
    print("Writing to log: " + string_operation)
    self.log_recently_changed = True
```

This would still be slow in the event of a *huge* log and would run every time the log got changed. I'd worry about that if and when this implementation ran into that problem.

Fix #2: Stop Closing the Connection when Client Requests a Write

I like a communicative server. But when I started up a client to talk to a server, I was seeing some inconsistent behavior:



What's happening here:

1. I start up the client and send a request to the leader (the server at port 10000) to send back the value associated with the key 'a.'
2. The leader (which is named "Every" as in "Every Tom, Dick, and Harry") sends back the value '3'. The call ends, and the client prompt "Type your message: " appears, encouraging the user (me) to type in another request.
3. I send the leader a request to associate the key 'z' to the value '26.'
4. The socket closes and my client is shut down.

The 'set z 26' call *works* on the server side, but I wanted it to finish and give me the "Type your message: " prompt like the other calls do. So I fixed that in this commit.

Commit message:

> *Because we break in the event of an empty string response (server.py lines 160 & 161), the server cuts off connection with the client if we do not include a response to the client.*
>
> *This isn't the worst thing in the world—we can easily restart the client. But I wanted continuity in the behavior of the server at the end of responding to any client request. So I added a cute reply from the server to the client when the client sends a write command, and now the connection remains open after the command succeeds.*

The fix:

src/server.py

```
@@ -293,8 +293,7 @@ def respond(self, key_value_store, operation):
                    while not self.current_operation_committed:
                        pass

-                   send_pending = False
-                   #TODO: Something about this block closes the connection
+                   response = "Aye aye, cap'n!"
                else:
                    response = key_value_store.read(self.current_operation)
            else:
```

Fix #3: Tell the Client Who is (Probably) the Leader

One of Raft's keystone characteristics is its strong leadership model: Only the leader directs other servers what to add to their logs, and only the leader accepts client requests. Any follower server, if contacted by a client, should refuse to fulfill the request.

Until now, our servers have responded with an unhelpful and rather curmudgeonly message. This is, first of all, rude.https://giphy.com/embed/57VKVucXSx1tlowvAU

Secondly, it doesn't provide the client with any information about which server to contact instead, which forces the client to keep guessing. Though no follower in Raft gets to be a source of truth, each follower *does* know who is *probably* the leader—it's whichever server last reached out to it in a leaderly fashion. The followers can (and according to the Raft spec, *do*) pass that information to the client.

I implement that in this commit, creating an instance variable to store the name of the last server that sent an `append_entries` call:

```
187             if string_operation.split(" ")[0] == "append_entries":
188                 call = AppendEntriesCall.from_message(string_operation)
189
190                 if call.in_term < self.key_value_store.current_term:
191                     response = "Your term is out of date. You can't be the leader."
192                 else:
193 +                   self.latest_leader = server_name
194 +
```

And then sending that name to any client that contacts this follower server:

```
    -               response = "I am not the leader. Please leave me alone."
307 +               response = "I am not the leader. The last leader I heard from is " + str(self.latest_leader) + "."
```

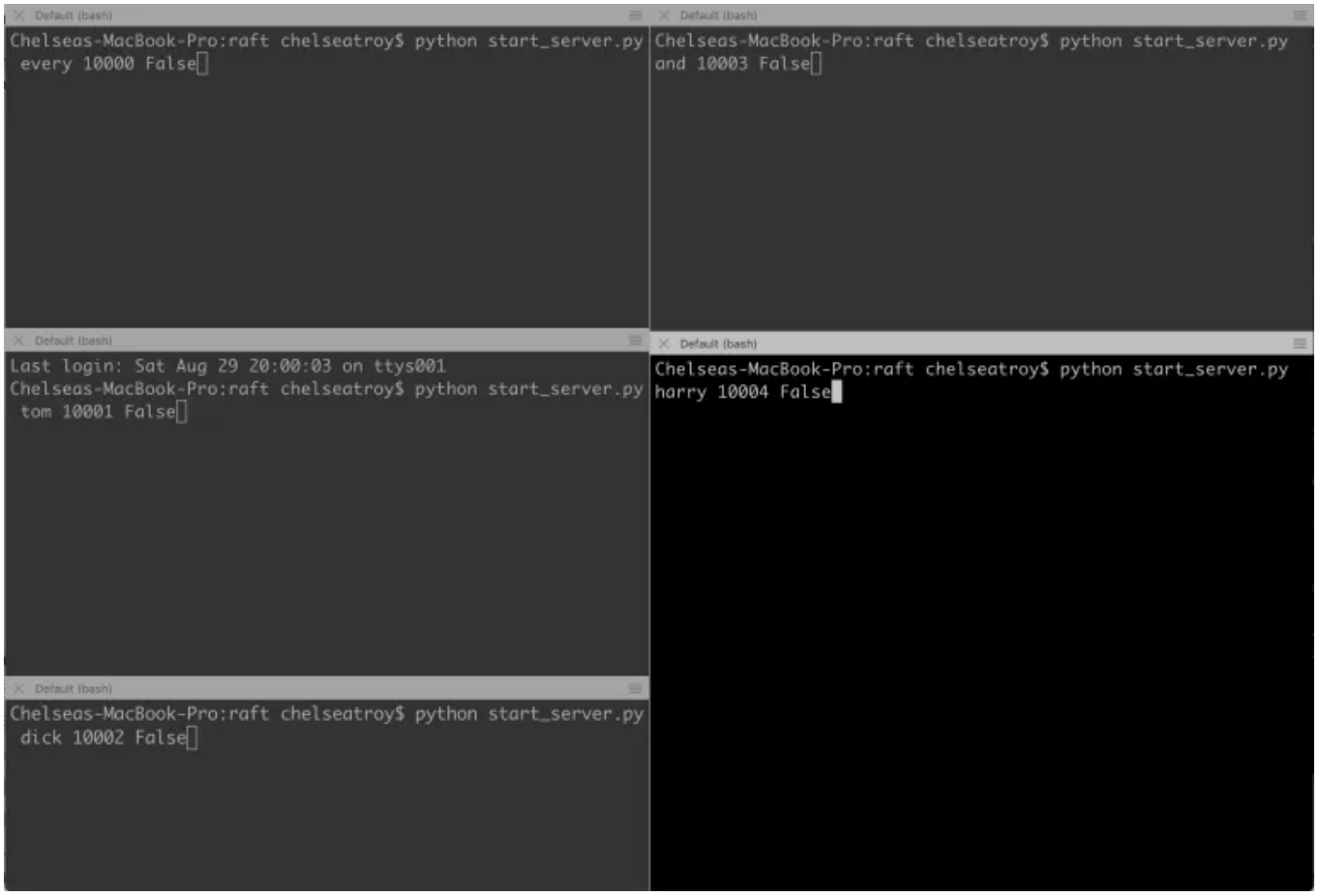
We set a sensible default value that will apply in the most common situation where a follower hasn't heard from a leader yet: on startup of the system.

```
23  +            self.latest_leader = "yet unelected"
```

It's theoretically possible that a client would contact a new follower in the brief period between when it starts up and when the leader sends it an `append_entries` call, in which case this default value isn't totally accurate. I picked this value because it seems like the more likely case, and also because it was more fun to come up with than something like "unknown to me."

And there you have it.

I can open 5 terminal windows like this and start a follower server in each of them (the 'False' argument at the end of each command line determines whether the server starts as a leader):



Once one of these servers' election timeouts elapses, it will start (and probably win) an election and start pinging the other servers. The servers have rafted! I can happily open a client against the leader server and get, set, delete key-value pairs to my little heart's content.

Typically, this is where I might refactor.

I'm going to hold off in this case. I foreshadowed this during log replication:

> *My server implementation relies on a conditional statement to parse requests and issue responses. Before my hiatus, I moved this conditional statement into its own file called `parsing.py`. As I got back into the code after six months away from it, I found that one of the first things I wanted to do was move the conditional back into the server class. You see that change reflected in this commit.*

> *Now that I've done some more work and reacquainted myself with the code, I find myself wanting to pull it back out again, largely to make it easier to unit test. I feel weird about this because I pride myself on prioritizing 'legibility' in the code I write—but it seems that what makes my code 'legible' to me changes depending on my immediate familiarity with it.*

Do I think refactoring is universally bad? Don't be ridiculous. Would I write this much about refactoring if I thought it were universally bad?

No—I think there's something more interesting going on here. And rather than ignore it, I'd like to dig deeper. That, my friends, we will save for dessert. Get excited.

Part 9: Configuration Changes

We're ready to address the final implementation chunk: configuration changes.

What if we have to change out the servers?

Changing the guard at Buckingham Palace

At this point, the Raft paper expects that our current implementation works from a fixed cluster configuration. That's not quite true: each of my servers, upon startup, puts its name and port in a `server_registry` file. Then, when a server needs to broadcast a message, it scrapes this file for all the unique names and broadcasts to those ports. So I can add as many servers as I want, whenever I want, and they'll get added to the Raft configuration.

That doesn't mean that this is a great solution. Why:

1. **The file gets longer indefinitely.** Servers write their name and port to this registry file every time they start, so there are repeat entries as servers go down and come back up. Over time, servers have to scrape a longer file to get the port set, so broadcasts take longer.
2. **n + 1 server replacements will break the cluster.** Nothing gets removed from the registry, and servers treat all the unique names in the registry as voting servers. So if I perform, over time, six replacements on a five-server cluster, I will have 11 servers named in the registry. At that point, it will no longer be possible for the leader to commit entries or for candidates to get elected leader because those things require responses from a majority of the cluster—and six out of the 11 servers will never respond.

Raft prescribes a two-phase approach for configuration changes.

In this approach, there are two sets of servers: the old one and the new one. First, just the old one is up. Then, both the old *and* the new one is up. Then the old one is brought down. In the middle, both the old and the new configurations need to agree on stuff, but only *some* stuff, and they have to get *separate* majorities between them to make things happen.

There's gotta be something I'm missing, because that sounds complicated and finicky, particularly for a system whose *whole design* is meant to be resilient to servers *randomly going down whenever*. If Raft does what it says on the tin, I should be able to make a "configuration change" by mercilessly tanking 40% of my servers. The catch: I need to tell the remaining servers to stop looking for responses from the ones that I tanked.

So I simplified the configuration change process for my purposes. Here's the difference between the canonical approach and mine: I don't configure the cluster by defining the entire old or new configuration in a single decree. Instead, I register and de-register servers one by one. The leader replicates logs and commits when a majority of the *currently registered* servers respond.

To make this work, I need to change the way servers get registered. As of this commit, I no longer write a server's information to the registry on startup:

```
91  -           f = open("logs/server_registry.txt", "a")
92  -           f.write(self.name + " localhost " + str(self.port) + '\n')
93  -           f.close()
```

I also stop using the `server_nodes` method, which reads the `server_registry` and produces the list of servers in the cluster from that registry:

```
16  - def destination_addresses(server_name):
17  -     other_servers = {k: v for (k, v) in server_nodes().items() if k != server_name}
18  -     return list(other_servers.values())
19  -
20  - def other_server_names(server_name):
21  -     other_servers = {k: v for (k, v) in server_nodes().items() if k != server_name}
22  -     return list(other_servers.keys())
```

Instead, servers populate their list of active servers from the log. Commands to register and deregister specific servers get interspersed with write commands to the key value store in the log:

```
153 +               elif operands[command] == "register":
154 +                   self.server_cluster[operands[key]] = operands[values]
155 +                   print("CURRENT OTHER SERVERS: " + str(self.server_cluster))
156 +               elif operands[command] == "deregister":
157 +                   self.server_cluster.pop(operands[key])
158 +                   print("CURRENT OTHER SERVERS: " + str(self.server_cluster))
```

So our log might look something like this:

```
0 0 set  unreachable
1 0 register every 10000
2 0 register tom 10001
3 1 set a 1
4 1 register dick 10002
5 2 set b 2
6 3 set c 3
7 4 set d 4
```

This means that, when we start servers with empty logs, we need to send commands to register them, that get placed in the logs and then executed, before the servers will start communicating with each other. This will work, but I get real nervous when I have to type under time pressure, because if I can't register a server's compatriots before its election timeout, it won't have the information it needs to request votes or get elected leader. So I start the cluster with a prefabricated log where the first few commands go ahead and register my starting servers.

Once servers could change their understanding of the cluster configuration based on log commands, they needed to accept requests that add such commands to the log while the servers are already running. This allows me to change the configuration on a server cluster without having to restart it.

In this commit, the `server` begins accepting commands from a client to register and deregister servers:

```
306 +                if self.current_operation.split(" ")[0] in ["set", "delete", "register", "deregister"]:
307                      string_operation_with_term = key_value_store.write_to_log(string_operation, term_absent=True)
308
    -                    broadcast(self, with_return_address(
309 +                    self.broadcast(self, with_return_address(
```

and the `key_value_store` starts executing those commands to change the state of the dictionary that stores cluster information, much in the same way it executes log commands to change the state of the key value store of data itself.

```
    -    def write_to_state_machine(self, string_operation, term_absent, path_to_logs=''):
126 +    def write_to_state_machine(self, string_operation, term_absent):
127          print("Writing to state machine: " + string_operation)
128
129          if len(string_operation) == 0:
    @@ -197,7 +197,7 @@ def write_to_log(self, string_operation, term_absent, path_to_logs=''):
197          else:
198              index, term, command, key, values = 0, 1, 2, 3, 4
199
    -        if operands[command] in ["set", "delete"]:
200 +        if operands[command] in ["set", "delete", "register", "deregister"]:
```

At this point, there's an issue: ports. If a server comes up with an out-of-date log and does not yet know which server names to associate with which ports, it does not have the port info it needs to respond to a server that contacts it with only a name as the return address.

This is the latest iteration of a recurring theme in Raft: we can never rely on the state of an individual follower server as the source of truth. Instead, all of the necessary context about the system must be communicated in the servers' messages to one another. That includes the port. So in this commit I add an explicit port declaration to the return address that each server already sends with each request.

```
-     def port_of(self, server_name):
-         return self.key_value_store.server_cluster[server_name]
183 +         return server.name + "|" + str(server.port) + "@" + response
184
185       def return_address_and_message(self, string_request):
186           address_with_message = string_request.split("@")
-         return address_with_message[0], "@".join(address_with_message[1:])
187 +         name, port = address_with_message[0].split("|")
188 +         return name, port, "@".join(address_with_message[1:])
189
190       def respond(self, key_value_store, operation):
191           send_pending = True
192           string_request = operation.decode("utf-8")
-         server_name, string_operation = self.return_address_and_message(string_request)
193 +         server_name, server_port, string_operation = self.return_address_and_message(string_request)
194           print("from " + server_name + ": received " + string_operation)
195
196           response = ''

@@ -328,4 +321,4 @@ def respond(self, key_value_store, operation):
321           if send_pending:
322               response = self.with_return_address(self, response)
323
-         return server_name, response
324 +         return server_name, server_port, response
```

There's another issue: at this point, I'm executing the change to the server's cluster dictionary in the `key_value_store`'s `write_to_state_machine` method. That's incorrect. Here's why:

> *A server always uses the latest configuration in its log, regardless of whether the entry is committed.*
>
> -Part 6: Cluster Membership Changes, The Raft Paper

We need to do this to avoid getting stuck in a chicken-egg situation where the new server configuration must commit its own registration changes before it can commit anything.

We fix it in this commit by moving that logic to `write_to_log` instead:

```
@@ -150,12 +150,6 @@ def write_to_state_machine(self, string_operation, term_absent):
                    self.log.append(string_operation)
                    self.delete(operands[key])
                    response = f"key {key} deleted"
-               elif operands[command] == "register":
-                   self.server_cluster[operands[key]] = operands[values]
-                   print("CURRENT SERVERS: " + str(self.server_cluster))
-               elif operands[command] == "deregister":
-                   self.server_cluster.pop(operands[key])
-                   print("CURRENT SERVERS: " + str(self.server_cluster))
                else:
                    pass

@@ -211,6 +205,13 @@ def write_to_log(self, string_operation, term_absent, path_to_logs=''):
                f.write(string_operation + '\n')
                f.close()

+               if operands[command] == "register":
+                   self.server_cluster[operands[key]] = operands[values]
+                   print("CURRENT SERVERS: " + str(self.server_cluster))
+               elif operands[command] == "deregister":
+                   self.server_cluster.pop(operands[key])
+                   print("CURRENT SERVERS: " + str(self.server_cluster))
+
```

So that's our basic cluster reconfiguration implementation. I can issue commands from the client such as "register tom 10001" or "deregester tom 10001" to add and remove a server named tom, available at port 10001, from the list of servers that receives `AppendEntries` calls, or to whom the leader is looking for a majority response to commit an entry.

That leaves three remaining issues, though.

The Raft paper steps through these one by one.

> *The first issue is that new servers may not initially store any log entries. If they are added to the cluster in this state, it could take quite a while for them to catch up, during which time it might not be possible to commit new log entries.*
>
> *In order to avoid availability gaps, Raft introduces an additional phase before the configuration change, in which the new servers join the cluster as non-voting members (the leader replicates log entries to them, but they are not considered for majorities). Once the new servers have caught up with the rest of the cluster, the reconfiguration can proceed.*
>
> -Part 6: Cluster Membership Changes, The Raft Paper

I implemented this with an attribute on the server called "voting." The command that we use to start servers can explicitly set this to False so that we can start servers in a non-voting state when we are adding them to an existing cluster. I implement that change in this commit.

start_server.py

```
@@ -2,8 +2,8 @@

 from src.server import Server

-am_i_the_leader = False
-if len(sys.argv) > 3 and sys.argv[3] == 'True':
-    am_i_the_leader = True
+voting = True
+if len(sys.argv) > 3 and sys.argv[3] == 'False':
+    voting = False

-Server(name=str(sys.argv[1]), port=int(sys.argv[2]), leader=am_i_the_leader).start()
+Server(name=str(sys.argv[1]), port=int(sys.argv[2]), voting=voting).start()
```

The attribute gets set to True when the new server's log has caught up to the others—that is, upon the server's first successful response to an `AppendEntries` call from the leader:

```
+                    self.voting = True
                     response = "Append entries call successful!"
```

As of this commit, servers only respond to a `RequestVotes` call if they are caught up, voting servers...

```
            if request_vote_call.for_term > self.key_value_store.current_term \
                and request_vote_call.latest_log_term >= self.key_value_store.latest_term_in_logs \
-               and request_vote_call.latest_log_index >= self.key_value_store.highest_index:
+               and request_vote_call.latest_log_index >= self.key_value_store.highest_index \
+               and self.voting:
```

...and when a server gets caught up and gains the ability to vote, it broadcasts a message announcing that fact...

```
                    self.voting = True
+                   print("I GET TO VOTE NOW AAAAAHAHAHAHAHA")
+                   self.broadcast(self, self.with_return_address(
+                       self,
+                       "I can vote now!"
+                   ))
                    response = "Append entries call successful!"
```

...so that the leader can add a line to the log to indicate that the server earned voting privileges...

```
+       elif string_operation == "I can vote now!":
+           key_value_store.write_to_log(f"register {server_name} voting", term_absent=False)
```

...and it can add that server to the dictionary of servers that it counts when looking for a majority of voting servers to respond to a `RequestVote` call:

```
+    def update_server_cluster(self, command):
+        operands = command.split(" ")
+        index, term, command, key, values = 0, 1, 2, 3, 4
+
+        if operands[command] == "register":
+            if operands[values] == "voting":
+                self.voted_for_me[operands[key]] = False
+                print("CURRENT VOTING SERVERS: " + str(self.voted_for_me))
+            else:
+                self.server_cluster[operands[key]] = operands[values]
+            print("CURRENT SERVERS: " + str(self.server_cluster))
+        elif operands[command] == "deregister":
+            self.server_cluster.pop(operands[key])
+            self.voted_for_me.pop(operands[key], None) #Skips KeyError in case a non-voting server is being removed
+            print("CURRENT SERVERS: " + str(self.server_cluster))
+            print("CURRENT VOTING SERVERS: " + str(self.voted_for_me))
+
```

Now, we might have a log that looks something like this when the server `dick` at port 10002 gets added to an existing cluster in a non-voting state:

```
0 0 set  unreachable
1 0 register every 10000
2 0 register tom 10001
3 1 set a 1
4 1 register dick 10002
5 2 set b 2
6 3 set c 3
7 4 set d 4
8 4 register dick voting
9 4 set e 5
```

So now we know that a server is caught up before it starts voting. On to the next problem:

> *The second issue is that the cluster leader may not be part of the new configuration. In this case, the leader steps down (returns to the follower state) once it has committed the C_{new} log entry.This means that there will be a period of time (while it is committing C_{new}) when the leader is managing a cluster that does not include itself.*
>
> *It replicates log entries but does not count itself in majorities.*
>
> -Part 6: Cluster Membership Changes, The Raft Paper

In this case, we would have a server where `self.leader` is True but `self.voting` is False. In theory, it shouldn't ever matter what an active leader's `self.voting` status is, because an active leader prevents the start of a new election.

Once again, it's odd to me that we go through this rigamarole with leadership transfer in a system that is literally designed with the express purpose of quickly rectifying the situation if a leader just *disappears*. I can bring down the leader, wait for a new

election to finish, and deregister the leader.

But, I don't even have to do that. I can deregister the leader and then bring it down because the leader only *sends* `AppendEntries` requests; it does not need to receive them. It's true that the cluster leader may not be a part of the new configuration, but with the current implementation, deregistering a leader does not cause any issues.

We have one last situation to address:

> *The third issue is that removed servers (those not in C_{new}) can disrupt the cluster. Those servers will not receive heartbeats, so they will time out and start new elections. They will then send RequestVote RPCs with new term numbers, and this will cause the current leader to revert to the follower state...*
>
> *To prevent this problem, servers disregard RequestVote RPCs when they believe a current leader exists. Specifically, if a server receives a RequestVote RPC within the minimum election timeout of hearing from a current leader, it does not update its term or grant its vote.*
>
> -Part 6: Cluster Membership Changes, The Raft Paper

On my system, the election timeouts range from 10 to 18 seconds. So to resolve this issue, servers should not grant their vote if they receive a `RequestVote` call within 10 seconds of having heard from a leader. That happens in this commit.

First I give the server an attribute called `election_allowed` that gets set to True on a 10 second timer...

src/server.py

```
@@ -33,12 +33,18 @@ def __init__(self, name, port=10000, voting=True):
33            print("Server started with timeout of : " + str(self.timeout))
34            self.election_countdown.start()
35
36  +         self.election_allowed = True
37  +         self.allow_election_countdown = threading.Timer(float(10), self.allow_election)
38  +         self.allow_election_countdown.start()
39  +
40            for server_name in self.key_value_store.other_server_names(name):
41                self.followers_with_update_status[server_name] = False
42
43            if self.voting:
44                self.key_value_store.voted_for_me[self.name] = False
45
46  +     def allow_election(self):
47  +         self.election_allowed = True
```

...and set to False (with the 10 second timer restarted) each time it receives an `AppendEntries` call:

```
201             if string_operation.split(" ")[0] == "append_entries":
202                 call = AppendEntriesCall.from_message(string_operation)
203
204                 if call.in_term < self.key_value_store.current_term:
205                     response = "Your term is out of date. You can't be the leader."
206                 else:
207                     self.latest_leader = server_name
208
209 +                   self.election_allowed = False
210 +                   self.allow_election_countdown.cancel()
211 +                   self.allow_election_countdown = threading.Timer(float(10), self.allow_election)
212 +                   self.allow_election_countdown.start()
213 +
```

It then denies its vote if a candidate reaches out with a `RequestVote` call within that 10 second period.

```
290             elif string_operation.split(" ")[0] == "can_I_count_on_your_vote_in_term":
    -               request_vote_call = RequestVoteCall.from_message(string_operation)
    -               if request_vote_call.for_term > self.key_value_store.current_term \
    -                   and request_vote_call.latest_log_term >= self.key_value_store.latest_term_in_logs \
    -                   and request_vote_call.latest_log_index >= self.key_value_store.highest_index \
    -                   and self.voting:
    -                       response = "You can count on my vote!"
291 +               if self.allow_election:
292 +                   request_vote_call = RequestVoteCall.from_message(string_operation)
293 +                   if request_vote_call.for_term > self.key_value_store.current_term \
294 +                       and request_vote_call.latest_log_term >= self.key_value_store.latest_term_in_logs \
295 +                       and request_vote_call.latest_log_index >= self.key_value_store.highest_index \
296 +                       and self.voting:
297 +                           response = "You can count on my vote!"
298 +                   else:
299 +                       response = "Your log is out of date. I'm not voting for you!"
300             else:
    -               response = "Your log is out of date. I'm not voting for you!"
301 +               response = "I heard from a leader recently, so I'm not voting for you."
```

With this system, I can issue commands to register and deregister servers, then bring them up and down, and keep the cluster going while changing out the individual pieces. In using and testing this system, here's what I do:

- **When I start the cluster, I manually register the initial set of servers by typing them into the log right before startup like so:**

```
0 0 set  unreachable
1 0 register every 10000
2 0 register every voting
3 0 register tom 10001
4 0 register tom voting
```

This way, these servers can start, and participate in, one another's elections to elect a leader. I could also do this by bringing them up and then issuing commands from a client to register them, but I don't trust my typing speed, so I do this instead.

- **When adding a server to the cluster, I issue a command from a client to the leader to register that server. Then, I start the server in the non-voting state.** Once that server has its log up to date, it automatically switches itself to a voting server.
- **When removing a server from a cluster, I issue a command from a client to the leader to deregister the server. Then I bring the server down.** This worked fine whether I was deregistering a follower or the leader itself. Theoretically if I brought down the leader before its deregistration log was replicated, other servers would still look for it. As trite as this is going to sound, I avoided that problem by not doing that (to be honest I was having a hard time engineering a situation that brought the server down fast enough to cause that problem). The deregistration command doesn't even need to be committed yet because a) servers go off the latest cluster configuration dictated in their log regardless of whether the entries are committed and b) only a server with an up-to-date log can become the leader.

A Further Note on Code Design

To handle configuration changes, I needed to pull some functions into the server object that I had previously attempted to put in their own file. In this commit, for example, I move the methods into the server that add a return address to, or strip a return address from, a server request/response.

In other words, I guessed wrong about where to put seams in my implementation. But I find myself wondering whether my mistake lay, not in misplacing seams, but in attempting to introduce seamsin the first place. Raft kind of resists seams. Its implementation revolves around a few orthogonal execution paths that together manipulate a shared set of state variables. Most changes loan themselves, not to adding a new collaborator, but to adding functionality to an existing collaborator.

Let's look at an example from this portion of the implementation.

I have described adding server registration and deregistration that writes to, and gets executed from, each server's log. The object responsible for managing the log is called `KeyValueStore`. That made sense when the log exclusively stored information about the key value pairs that the cluster exists to replicate.

Now the cluster must replicate a different kind of information in the exact same way—in fact, *in the exact same file.* I thought about building a separate concern for this, but that concern would need to replicate much of the functionality of `KeyValueStore`. Perhaps an opportunity for an abstract class? Maybe, though Python doesn't encourage it (I wouldn't call this kludge that leverages `pass` "encouraging").

Even if I did that, then I have two concerns editing the same file. *The* file, in fact, that the whole Raft system exerts itself to make *consistent* across multiple servers. Giving two concerns write access to this file opens up the possibility for race conditions that lead to a non-deterministic log result given the same set of inputs in similar time domains. Instead of adjudicating this, it made sense to me that the log-to-state and state-to-log translations, together, represented a single responsibility, to be covered by a single concern. Instead of making a separate class to `KeyValueStore`, I added the registration and deregistration functionality to that class and renamed it to `LogManager` in this commit.

This decision gets me further away from the takeaways of books like *Refactoring to Patterns* or *Domain Driven Design*, which lay out the imperative paradigm of breaking up our code into smaller objects. *Philosophy of Software Design* proposes another approach: one in which we strive for a deep set of functionality covered by a single API. Who should programmers listen to?

I don't think the answer comes down to Gang of Four vs. Ousterhout (though I'd be remiss not to mention that the Ousterhout of that book is the same Ousterhout who co-authored the Raft paper). Instead, I think the answer comes down to context.

Here's what I mean by that:

I think a lot of software engineering instruction gets presented as universal, when in fact it's context-specific.

Either the instructor doesn't realize that it's context-specific because they've only worked in one context, or they don't communicate that it's context-specific even if they know it is.

A common example piece of programming advice: "legibility is more important than performance." Most programmers who say this don't add the context part: "Legibility is more important than performance...*when building end-user applications, usually*." Because if you're building a *compiler*, this is not true. If your iOS app loops over a collection of 5 things in a relatively inefficient way, it's probably fine. If your programming language parses every line at the speed of cold molasses because your compiler uses tiered conditionals instead of an ugly-but-faster case statement, *it is not fine.*

But back to Raft, and what to do about the objects. I think that there is context here to take into account. Before any refactoring happened—before any of my code got *written*—this system's design impacts the way its implementation gets factored, and that's worth a look. In fact, I think it's worth a *really* close look. So that's what we're discussing next. Take a nap, come back fresh, and bring an espresso: we have *stuff* to discuss.

Part 10: The Barking Spider Circus

I promised you a grand finale with reflection, cross-references, smack talk, and gifs. Then I warned you to look alive and bring an espresso. Go ahead and pull those double-shots, folks: the time has come.

Our story begins with a tweet:

I've thought (and read, and scribbled) about this a lot; I wanted to get to the bottom of what might be happening. I jotted down specific examples as I came across them in my implementation. I started to connect those examples to broader concepts in software development and software instruction.

I refined those notes into "A Further Note On Code Design" at the end of the previous piece in this series. We look at specific choices I made while writing the features discussed in that piece (handling cluster configuration changes), then I mentioned a so-far-largely-unspoken decision lever: context.

What you need to know about Raft's design:

The decisions that impacted the factoring of my Raft implementation start *long* before I open an IDE, with Ongaro and Ousterhout's design goals:

> *Our approach was unusual in that our primary goal was understandability: could we define a consensus algorithm for practical systems and describe it in a way that is significantly easier to learn than Paxos?*
>
> -In Search of an Understandable Consensus Algorithm (Extended Version)
> Diego Ongaro and John Ousterhout
> Stanford University

The authors of the Raft paper make a big deal about Raft's understandability relative to other distributed consensus algorithms (in particular, Paxos). In this piece, we'll explore how they did that and what it can teach us about architecting our own systems.

Before we do that, though, I have to address something.

This piece does not constitute endorsement of the Raft paper's claim about understandability.

When I read the quoted sentence above, I got wary. Why? Because it reads, at first glance, like a claim that Raft is simple. That's not what the authors are claiming, which we'll get to. But what I *don't* want is for other authors to see that quote as precedent license to make such a claim.

I don't appreciate technologists engaging in onanistic monologues how "simple" they find something to be. Most of the time, these descriptors are of most value to the describers, to self-congratulate on their cleverness or, less commonly, their explanatory skills. Calling something "simple" does nothing to make it any easier to understand; it just signals to people who don't understand it the first time that they should feel shame about that. **I *do not endorse* running out and writing a paper that calls your own solution "understandable".**

Folks who have read the Raft paper will point out (correctly) that the authors didn't just *claim* this—they backed it up with a study. I would skip this, but people are going to ask, so I'm answering it now. I saw the study. The study lacks rigor. They showed 43 students two videos: one explaining Raft, the other explaining Paxos. After each video, they issued students a quiz, and the scores proxy the understandability of the algorithms. Some issues:

1. Both the Raft and the Paxos video were made by the Raft team, who admit in the paper that they couldn't figure out exactly how Paxos worked, whereas they literally *wrote* Raft. So the instructors' preparedness to explain the two topics differed between the videos.
2. They say Paxos had an "advantage" because the video was "14% longer." It's one hell of a proxy leap to equate "longer video" to "better instructed." Such a proxy accepts as given the effectiveness of an authoritarian teaching model that much of the past decade of education research has debunked.

3. "Score on a quiz immediately after watching a video" is the proxy here for "understands the algorithm," and in my view as a software engineer and an instructor of computer science graduate students, not a good one. It accepts as given the effectiveness of multiple choice tests as a demonstration of understanding, which is another thing much of the past decade of education research has debunked.
4. Clearly I'm not a fan of using the scores as proxies, but if we're gonna do that, it needs to be explicitly acknowledged that the scores on these quizzes, in absolute terms, were not good for either Raft *or* Paxos (42% and 34%, respectively). This is what I mean about suggesting that something is simple. Would you say, on an absolute scale, that an algorithm is simple if computer science advanced degree candidates get a 42% on a quiz about it? I would not.
5. It's really hard to say that two quizzes on two different subjects were of exactly equal difficulty: harder still if the authors of both quizzes admit to not fully understanding one subject while having authored the other. A more believable (though considerably more work for the subjects) demonstration might be a single integration test suite or feature checklist for a distributed system to which participants' implementations are subjected, using either Raft or Paxos.
6. The *other* metric used besides quiz scores is the students' opinions of how understandable the two algorithms are. I don't think that someone's stated opinion immediately after watching a single video and without ever having tried to implement the thing is particularly instructive.
7. The sample size is 43. I re-ran the numbers for this study; it's true that the confidence intervals of Raft and Paxos quiz scores do not overlap, but aforementioned grievances about the proxies obviate this validity metric. 43 is a suspiciously small number for measuring such a heavily proxied effect.

I just implemented Raft and, while it's not the *hardest* thing I've ever done, it certainly was not the easiest, either. To their credit, Ongaro and Ousterhout are not saying that Raft is a walk in the park. By contrast, they're saying that Raft is simple relative to Paxos, specifically.

And *that*, I believe they are right about. I read the Paxos paper and talked to some folks who implemented (or tried to implement) both Paxos and Raft. It was this effort, not the 43 person video comprehension quiz study, that convinced me that I'd have cussed a lot more trying to get Paxos working than I did trying to get Raft working.

Sure, Chels. But what's with the "Barking Spider Circus" comment?

Many a Twitter reply guy felt that I had forgotten my place by suggesting that orthodox refactoring sensibilities weren't working for me on my Raft implementation. On Twitter itself, I ignored those people. But I'll address the sentiment here: if I thought orthodox refactoring techniques were useless, I wouldn't have made it this far into programming and still have tried them on Raft.https://giphy.com/embed/ihToSRVoYg5lTIqOfD

Here's what happened: when I tried the orthodox refactoring techniques at the points that they are coached in gang-of-four-era literature, they made the Raft implementation harder to follow. Let's talk about some examples.

DRYing code up works when you have two *conceptually* identical things happening: they can't just be *similar*, or else you'e subclassing with no-ops and doing other gymnastics to get around their differences in ways that would be unnecessary if you had just kept them separate. We talk about this in greater detail over here. I ran into this with respect to responding to similar, but not identical, requests in Raft: in particular, a client's `get` or `show` request versus a `set` or `delete` request. I made this mistake once and remembered it when I extracted `AppendEntries` and `RequestVote` logic. Could I employ polymorphism here? I could. But in any case, they're definitely not the same object.

Similarly, extracting lots of little objects works when each of the little objects handles a relatively separate set of concerns—that is, separate enough, that the probability drops below some threshold that a developer needs to hold *both* of those concerns in their head at once to sufficiently understand the system for maintenance or enhancement. When you extract objects while that probability remains above the threshold, the code gets harder to read. And if you're super familiar with the code such that you *already have* both concerns in your head, it can be tough to guess whether understanding just one or the other would be enough to work in that particular section of code. I ran into this when I returned to my Raft implementation after a several month hiatus and found myself constantly switching between the server class and the method I had extracted from the server class for handling requests and generating responses. I needed to move that logic *back* into the `Server` class—which made the `Server` class bigger. Ugh.

Now that we have some specific examples, I want to introduce a terminology shift to clarify our upcoming discussion. The changes I have described thus far have this name, "refactoring," with "re" implying "again"—that is, that this code has already been "factored" at least once when we wrote it to begin with. They're design changes introduced after implementation to improve the code's flexibility and–you guessed it–understandability. But we also use the term "refactoring" to describe other things that *aren't* this specific thing, so moving forward in this piece, I'll call such changes **postfactoring.**

We already use an equivalent term, **prefactoring**, to describe striving for a level of flexibility in the code design that the system doesn't need or doesn't yet need, often at the cost of understandability.

The term "refactoring" tends to cover both prefactoring and postfactoring, and refactoring tends to happen somewhere relative to the system's implementation. Refactoring focuses on finding flexible and understandable ways to handle the details of the system's intended behavior.

Refactoring therefore follows the task of defining the system's intended behavior.

It's at this preliminary step that authors of the Raft paper focus their efforts for making Raft's design understandable. Are implementation details described in the paper? Yes: there's a table detailing which objects to make and which attributes to put on those objects. Are these details the heart of the algorithm's successes in achieving understandability? I don't think so. Why not: at the beginning of this implementation, I set out to read the "what to do" portion of the paper and come up with the "how to do it" portion myself. I wanted to develop my intuition by, hopefully, slamming into the same barriers the original Raft team did while refining their implementation, so I had firsthand experience with *why* we did things the way we did them. I did not look at the paper's implementation table until after my implementation was almost finished. When I *did* look at it, the implementation I had come up with differed a little, but not a ton, from the table. The system's design, rather, strongly implicated a specific approach to implementation. When I tried to editorialize on that approach, I made things worse for myself.

So, more generally: if you think of a system's implementation as a motorcycle, the different kinds of refactoring (prefactoring and postfactoring) are equivalent to improving the shocks on the motorcycle to compensate for uneven territory and provide a smooth ride for the driver. You can think of the system's design, by contrast, as the road that the motorcycle follows. Raft's design focuses on techniques for smoothing out the *road*, such that fancy shocks aren't needed—and, when over-applied or inappropriately installed, make the ride harder than it has to be.

Waterlooville: WWII-era motorcycle painted allied colors. Photograph by Colin Moore

How do you design a system to resist a convoluted solution?

I don't think of length as a proxy for complexity, especially as it could vary widely between implementation languages, but I know folks ask. I wrote my Raft implementation in Python and it clocks in at about 700 lines, not counting logs and tests, but counting all the places where I decided to get cute and give the servers some responses with personality (this project was for fun, after all). I believe the implementation mentioned in the Raft paper itself was in Java and came in at around 2000 lines, being probably somewhat more robust (though indubitably less cute) than mine and in a language that trends more verbose than Python. That's objectively not 60,000 lines. We're not talking about a *massive* body of code.

Here's how the Raft paper describes the team's approach to designing the algorithm:

> *There were numerous points in the design of Raft where we had to choose among alternative approaches. In these situations we evaluated the alternatives based on understandability: how hard is it to explain each alternative (for example, how complex is its state space, and does*
> *it have subtle implications?), and how easy will it be for a reader to completely understand the approach and its implications?*
>
> *We recognize that there is a high degree of subjectivity in such analysis; nonetheless, we used two techniques that are generally applicable. The first technique is the well-known approach of* ***problem decomposition****: wherever possible, we divided problems into separate pieces that could be solved, explained, and understood relatively independently. For example, in Raft we separated leader election, log replication, safety, and membership changes.*
>
> *Our second approach was to* ***simplify the state space by reducing the number of states to consider****, making the system more coherent and eliminating nondeterminism where possible. Specifically, logs are not allowed to have holes, and Raft limits the ways in which logs can become inconsistent with each other. Although in most cases we tried to eliminate nondeterminism, there are some situations where nondeterminism actually improves understandability. In particular, randomized approaches introduce nondeterminism, but they tend to reduce the state space by handling all possible choices in a similar fashion ("choose any; it doesn't matter"). We used randomization to simplify the Raft leader election algorithm.*

> -In Search of an Understandable Consensus Algorithm (Extended Version)
> Diego Ongaro and John Ousterhout
> Stanford University

The techniques that Ongaro and Ousterhout describe here are an altogether different set from the ones we find in *Design Patterns* or *Refactoring to Patterns* or *Effective X*. Instead, they look more like the set of techniques that Michael Feathers describes in *Unconditional Programming*, the book I have not-so-secretly used my blog to badger him to write for three years since I saw him do this talk about it. My badgering has earned me a peek at the table of contents, which we'll get to soon.

In fact, we've discussed a similar set of techniques on my blog, specifically with regard to what software engineers can learn from rocket launchpad design (I try to keep it lively). In that piece, I said:

> *The idea is to identify your edge cases—your what-ifs, and exceptions, and your it-shouldn't-do-thats—and rather than handling them, make them go away. Doing so produces a more robust piece of software.*
>
> *Doing this also requires software engineers (that's us) to identify the moments when we are negotiating with an edge and determine whether that edge deserves negotiation. We have to identify the times when we are answering the question "How should I account for X?" and consider "Can I make X go away?"*
>
> -Lessons from Space: Edge-Free Programming
> Chelsea Troy (that's me!)

The Two Techniques

First, Ongaro and Ousterhout mention **problem decomposition**: they separate leader election, log replication, safety, and membership changes in Raft to operate independently of each other. These four subjects offer us seams along which we can divide our implementation into separate modules: they don't interfere with one another, so we don't get trapped in our own net with the requirement to understand them all at once. This design choice eliminates whole classes of cross-functional interactions: collections of potential pitfalls for which we'd struggle to install a suitable set of shocks.

Feathers might characterize this effect as **removing conditions.** We have three separate states for each server: whether it's the leader, what its log looks like, and whether it's in the cluster or not (we get safety as an emergent property of the system). Mostly, we can handle these three axes of status independently of one another, rather than have to check them all before making any changes. Avdi Grimm talks about separating axes of status specifically in his talk "Build Graceful Processes." If you're interested, I have a series on this with some code examples.

Second, the Raft authors reduce the number of states to consider, and introduce nondeterminism *only* when any of the nondeterministic outcomes would be handled in pretty much the same way. These design choices, in Feathers parlance, **impose edges** on the system where previously some negotiation would happen. A lot of these edges come from Raft's **strong leadership model**: the leader server is in charge. That server *only* appends entries to its log: it never removes or changes entries. If it encounters inconsistencies with a follower's log, it overwrites the follower's log. It issues all `append_entries` commands. It accepts all client requests. It treats its own state machine as the source of truth in responding to those requests. In this respect, it implements another of Feathers' techniques: **tell, don't ask**. There are few conditionals for how a follower responds to a command: the leader isn't accepting consultation on the state of the system as a whole. The sole exception happens in log replication, where a follower notes that its log appears not to match the leader's, in which case the follower receives lines from the leader to catch up its log and, if needed, delete entries that don't agree with the leader's.

Implications

We have sets of patterns for handling complexity in our code, as well as guidelines for when to introduce them. What about a set of practices that we can use before writing code to eliminate complexity before we write it?

That can be tough when we're using an iterative approach, or don't know all the requirements of our system up front. Designing Raft involved trying many different implementations and discovering where each of them broke down, or where edge cases popped up, before alighting on the final design. We might not be able to afford dozens of experimental tries in a production setting.

This is where something like formal verification might serve us well. We can use a language-agnostic formal verification tool like TLA+ or Alloy to describe the states and interactions in our system and identify potential edges and inconsistencies before implementation.

Even for systems where the formal verification approach feels heavy-handed, we can employ a pencil-and-paper risk analysis technique to assess what might go wrong with our current approach and "how bad" the problem might be. I describe such a framework right here, and my students complete a risk profiling workshop with a parrot emergency room app as their example (see this piece for an example diagram). I give the workshops at companies on request, by the way 😊.

But at a minimum, even in an iterative shop, we can evaluate potholes in our pavement before jumping to install shocks. We look at some specific examples of such evaluations in this case study that looks at a mental health management app written for Android.

In any case, we're used to striving for a flexible and understandable implementation of a model. We can perhaps benefit from drawing some of that effort forward into streamlining the model itself so that our implementation boasts fewer concerns to separate and fewer duplicated concepts to DRY up.

Have you enjoyed this series?

You can help me keep writing by tossing a coin at this Patreon 🙂

Categories

- Agile (8)
- Android (16)
- APIs (14)
- Behind the Scenes (16)
- Books (15)
- Careers (60)
- Community (42)
- Concepts (3)
- CSS (2)
- Data (9)
- Data Science (24)
- Debugging (9)
- Design (8)
- Golang (1)
- Hiring (7)
- History (8)
- Inclusion (19)
- iOS (16)
- Java (15)
- Javascript (6)
- Languages and Frameworks (13)
- Leveling Up (24)
- Live Coding (11)
- Machine Learning (27)
- Professional Development (22)
- Python (19)
- Refactoring (14)
- Remote Work (12)
- Ruby (23)
- Scheme (1)
- Spring (4)
- Swift (9)
- Teaching (16)
- Techtivism (3)
- Testing (26)
- Uncategorized (4)