Initial Scope

The initial goal of the project was to have a feature complete implementation of Raft, this included having a leader help lagging servers catch-up, in case of leader failure: holding a leader election to establish a new leader and having the leader correct the logs of the other servers. Along with supporting the basic functionality of log replication. The goal was to build a distributed key-value store and using raft to replicate date across the servers.

What works and what does not work

The initially planned features were not successfully implemented, the biggest one of these is the leader election workflow. Thus, the leader currently needs to be assigned the role manually using command line arguments. If the leader fails, then the system comes to a stop unless we restart the leader or assign a new server the role of the leader.

Features that work are basic log replication (the leader is able to broadcast the client requests and have them replicated), leader helping lagging servers catch-up (the leader fixes missing entries in the follower), the leader correcting wrong log entries (by checking from the tail of the log the leader establishes a point in which both the logs are common; the followers logs thereafter are deleted and replaced by those of the leader).

The Leader changes are also properly handled by the system. If the leader fails and we manually assign the role of the leader to a new server in the group; the system successful follows the commands of the new leader.

There is a heartbeat that is sent out every 10 seconds which reuses the append Entries function call. Apart from this I have not included any of the leader election code as it was not part of the initial submission.

Fixes made from initial submission to improve stability

Because of how the project was setup, I spent a lot of time in the last few hours before the submission undoing the leader election workflow which was still not functional. Doing this in a hurry and haphazardly let to a few bugs still remaining in the system. This causes that system to error out under certain conditions. I have included a list of the changes made here:

- IO exceptions were being thrown by the receive_size call when the socket connection was cloased from the other side. (recv empty message was received)
- Get and delete operations if performed when the key was not present lead to the system throwing invalid index error.

- IP addresses are now part of the command line arguments, instead of everything being localhost.
- All operations sent by the client needed to be prefaced by "@client" manually, this is now done by the client code.
- Key value store (line 174) was modified as it was causing index-out of bound errors during the delete operation.
- The Highest index in key_value_store was wrongly initialized to 1, it was changed to 0 as it was requesting to send logs at "1 0" (index 1, term 0) when there was no entry at that log location leading to an index out of bounds error.

Setting up the system

Part 1: The logs folder

Python 3 is needed to run the system. The "server_registry.txt" file consists of the name, IP address and port number of all the servers that are part of the system. This file is used by config.py to identify a server's peers.

For every server that is add to the server_registry.txt, we need to add a log file with the same server name followed by "_log.txt" (i.e.: server1 will have a log file called server1_log.txt). This file can be left empty (to be later filled by client requests) or could contain dummy entries that we expect a leader to replicate (if the server is a leader) or we expect them to get overwritten by the logs of a leader server (if the server is a follower and the logs are inconsistent with the leader).

Part 2: Running startServer.py

Usage: python3 startServer.py [server name] [server IP] [server Port] [Leader or not]
Example: "python3 startServer.py server1 192.168.55.55 1000 True".

The "leader or not" is indicated by a Boolean (True/False), at any given time there should only be one leader in the system. Because we are setting up leaders manually accidentally having multiple leaders will take the system to an inconsistent state.

By using startServer.py we start-up all the servers that we defined in the server registry and made log files for. Once all these servers are running and there is a leader, the leader will make sure that its current log is replicated across all the other servers.

Part 3: Running startClient.py

The client is used to send requests to the leader, all client requests are prefaced by “client@”. These requests can only be processed by the leader in the system. If you send these requests to any of the other servers, then they reply with “I am not the leader.”

The operations that the system supports are:

- get [key]
    - returns the value associated with the key, if the key is not part of the key value store, then it returns “key does not exist!”
- set [key] [value]
    - Simply sets a given key to a particular value, if the key already exists then it acts as an update operation.
- delete [key]
    - Deletes the key value pair if it is present in the dictionary. If the key that a user wants to delete is not present, then this operation simply does nothing.

Use the above-mentioned commands during the evaluation of the system. These three commands form the basis of our key value store. After executing these commands, the server returns a success status once the replication has successfully occurred.

There is a one second communication delay that was intentionally added by me. This is done to slow down the logs being printed and to making it easier to debug. Because of this delay it might, sometimes appear like nothing is happening as the client did not get a response. If this happens then check the servers to see if any errors have occurred.

Running the code on Khoury VDI clusters

I have tried to run the system on the Khoury VDI servers buy was not able to. The Khoury VDI servers gave me a permission denied error, thus not letting me bind a socket to an address. I looked into solutions for this but have not gotten anything to work. There is probably something really simple that I am missing or something really stupid that I am doing that is causing this error (because we were able to bind sockets as part of the programing assignment without using sudo). Not having much experience with things like this, I have not been able to fix this issue.

Evaluation of the system

To test the correctness of the system, I propose the following manual tests.

1. Set up 3 servers with empty logs, set one of these servers to be the leader. Have the leader replicate logs to the other two followers. Check the log files of the three servers after performing a few set, get and delete operations by using the client.

    a. If the logs of all the server's match, then we can say that we have basic log replication working.

2. Bring one of the follower servers down and add few more entries to the log (like what was done above). Once the leader has more entries than the server that was bought down, bring the server back up.

    a. If the leader can correctly add the missing entries to the follower server, then we can confirm that the leader successfully brings crashed servers back up to spec.

3. One of the key ides of raft is that at a particular "index_number term_number" combination there should exist a particular operation and all the servers should agree on this ordering. This ordering is dictated majorly by the leader; hence the followers need to replicate the log of the leader. To test if this is happening properly or not in a very simple way, I recommend bringing down the leader. Removing some of the logs from the leader and bringing it back up.

    a. If this change is properly replicated across the other servers, then we can say that the system is functioning as desired.

4. Testing if the system can survive a leadership change: To test this, perform evaluation 1, then bring down the leader and a follower server after all logs till that point have been properly replicated. Then bring back the old follower as the new leader of the system. Add a few more log entries by using the client and connecting to the new leader. Once this is done bring back the old leader back up as a normal follower and check to see if the new leader did its job.

    a. If this test passes, then we can conclude that the system is able to survive a leadership change. Here we are hand picking the leaders replacement, but the next step would be to automate this process as described in the raft paper.

System Design and Engineering overview

- The append entries class just defines the marshaling and un-marshaling of the class values into a string and extracting them from a string. This class models the functionality that message.cpp that we dhad in our programing assignment. There was a leader_election_call.py which I later removed as that part did not work properly. The design for both was mentioned in the raft paper and I have followed their recommended class parameters.

- For storing the data “index-number term-number” key associated with a particular operation and making this a fast lookup I have used a python dictionary. But this not let me preserver the order of operations (i.e.: did 2 1 happen first of 2 2). Hence, I am also maintaining an array that preserves the ordering of these “index-number term-number” keys.

- Line 84 in server.py is the function that keeps a tally and check to see if a quorum has been reached or not. Once a quorum has been reached this while loop exits.

```
# waiting till we get quorum!
while not self.current_operation_committed:
    self.current_operation_committed = False
```

- If the log of a server is inconsistent with that of the leader, then the leader keeps decrementing the log entries index to find a point in time where the two logs converge. The logic for this is in the server.py line 144 where a follower sends back append entries unsuccessful and line 148 when a leader receives this message.

- All servers have their first log as “0 0 unreachable” here the key is a space. This is done so that all the logs have 1 entry in common for sure.

- Broadcasting to the other servers is made easier by the helpers in parsing.py and getting all the peer servers using config.py.

- Message_passing.py is like the Recv/Send wrappers over recv/send that we had in our programing assignment. They keep the messages intact, so that they are always received as a single unit and not as chunks.

- The rest of server.py are branches of execution as defined by the Raft algorithm. (not all the potential branches have been properly handled).

- How append entries lead to the term_idx being incremented, Election workflow would lead to the term number being incremented.