Implementing Raft along 3 Human Nodes while Implementing Raft to build a fault-tolerant Key-Value Store

Yes, we had to first implement Raft amongst ourselves before figuring out how to do it in Python

June 2020

Authors: Melanie Chow, Celine Kim, Anant Matai

Introduction

Throughout the Advanced Distributed Systems class, we have focused on the key ideas that drive the relatively-modern concept of Distributed Systems. Issues of keeping time led to issues of understanding synchronicity in decision-making and reaching consensus, where we moved from these abstract ideas into understanding the implementation of consensus and fault-tolerance in databases, clusters, and various other applications of distributed computing. The infinite field of possible failures—byzantine, fail-stop, fail-recover—in machines leads to an obviously less-total system of thinking, but a formal process is required to combat these issues nevertheless. In the formalization of solutions to the consensus problem—how nodes keep track of commands made to a system—we largely have two algorithms, Paxos and Raft.

Paxos, originally brought up in the *Part-Time Parliament*,[6] suggests a method through which nodes pass messages to one another and use a system of proposals and votes to "pass" given proposals. The system, or at least the reduced system of *Paxos Made Simple*,[7] was implemented in our second assignment of the class. The more proper use of Paxos in distributed systems is in the use of Multi-Paxos[9]—moving beyond repeating the Leader election (Prepare/Propose) and allowing the Leader, once elected, to enact and send ACCEPT requests for multiple proposals.

The other contender in consensus algorithms is Raft[3]. Designed to be easier to implement than Paxos, Raft entails the use of a cluster in which all nodes are connected. A node proposes itself as a candidate based on its term, size of log, and when it timeouts, at which point other nodes decide to grant a vote or reject the vote. Once elected, the candidate becomes a leader and signals that it is alive to all other nodes at a regular interval. Alongside this heartbeat, the leader also handles the log replication across other nodes, ensuring a majority of the nodes have the same log. The key difference between Raft and Paxos is the design[8]—Raft is easier to understand and implement, as it solves the problem Paxos creates while solving consensus.

We decided that having experienced enough of Lamport and Paxos, we wanted to try implementing fault-tolerance in our key-value store through building an implementation of Raft and working it into the given construction of node.py[10]. Raft is resilient to fail-stop failures as it is built to tolerate nodes crashing. Raft is also resilient to fail-recover issues as when a node recovers, the leader continuously sends appendEntries to update the recovering machine. Raft is also similarly resilient to network partitioning, as we can treat this as a delay or a crash, and upon rejoining either a new leader forces an election or becomes a follower and is caught up. A good discussion of Raft's failures, efficiency, and correctness can be found here[5].

Our design involves the integration of the Raft Node into the Node Class and the creation of a Role Class. These two classes together form a solution Leader Election and Log Replication, forming the functioning Raft algorithm. We continue to work with JSON objects and deal with the broker when sending messages. The decision to not abstract away message delivery and sending means we don't interface with node.py as much as build on top of it and influenced our decision to integrate Raft into node.py. This was a design decision made in understanding time limitations as well as how closely we were working with chistributed and the node.py file. However, we do feel modularity is cleaner and well-suited to future development in fault tolerance as we add on different algorithms for different uses, and abstracting the Raft and Message modules would be first on our list of improvements.

Implementation and Design

Our *get, set,* and *replicate* functions are taken from node.py, where the interaction of the client-broker-node was initially a *non* fault-tolerant key-value store. In order to make it fault tolerant, we implemented Raft in node.py, following similar naming conventions as the paper. In our code, we also include comments from the paper to make it more clear about why we are doing certain operations. The properties of Raft Nodes are embedded into the original Node object from node.py. To make our design and implementation from paper to project clear, we will refer to the Nodes as Nodes and our Raft Nodes as RNodes, though we see quite clearly that they are essentially equivalent.

We elaborate on some design decisions that stray away from the paper (there aren't many), and some design implementations we made. Our implementation sticks to the specification for Log Replication and Leader Election, and most of our design decisions were made in the creation of the Roles—Leader, Follower, and Candidate.

Leader Election

Initial election

The leader election proceeds as the spec in the Raft paper. After initializing the nodes, we have a candidate appear as one of the nodes times out. The candidate requests the vote, upon which the other nodes will either grant the vote or reject it. Given this is the initial election, there is likely no conflict in terms and nodes will grant requests. The leader is elected and proceeds to handle log replication (and thus our data-store replication). The leader periodically sends an empty appendEntries RPC as the heartbeat, and then proceeds to send more appendEntries as commands come in from clients. We have set our heartbeat interval to 500ms, and use a timer thread to periodically send the heartbeat. The change in timers here and throughout the

implementation are so that the broker has time to send and receive messages between nodes and the client. We have one deviation, in that, any node called ‘leader’ or ‘backup’ have hard-coded times for the timer. This is only for testing purposes, because we wanted to ensure which nodes would become leaders, and in what order during partitions. That way, we can set the appropriate scripts. This is made more clear in the section below detailing the scripts for testing.

Failure Tolerance: Leader Crash

Should a leader crash/disappear, we enter the re-election process. Each node will pick a random time from 1 to 5 seconds. The paper chooses 0.15s - 0.3s, however, we give the nodes more time to receive votes from their fellow peers. The random time is significant, as it lowers the chance of simultaneous candidates. As in, two nodes are unlikely to time out at the same time, so we have less occurrences of a split election. **A split election** is when all the candidates (>1) have the same number of votes, and all the non-candidate nodes have already voted. This election timeout also closely follows the paper, where the first node to timeout after not receiving a heartbeat proposes itself. If the candidate’s term is lower than another node RNode, (in case it failed and just recovered), the RNode will reject the candidate and send an updated term. If the RNode has voted for the candidate before or has not voted, and the candidate’s log is at least as up to date as the RNode’s, then the vote is granted. This is why we always check if the request for a vote is valid before making our decision. A majority results in the candidate becoming leader, and we proceed onwards.

Log Replication

Our log replication sticks to the specification for calling appendEntries. We don’t deviate in any noticeable manner. The leader will send appendEntries, which followers will acknowledge. Upon acknowledgement, the leader updates its commitIndex (applying the logs) and the follower’s matchIndex and nextIndex, and on the next heartbeat/appendEntries the followers receive the updated index, thereby applying their logs as well up to the given information from the leader. Nodes also rollback or commit uncommitted entries as specified.

Failure Tolerance: Follower crash

Should a follower fail and recover its state (an assumption of the Raft module), the leader’s matchIndex for the follower will be up-to-date with the follower’s state, and it will send the appropriate missing logs over appendEntries. Because the leader and the

Should a follower recover and realise it has a different entry at a given index, the follower will realise this upon checking with the leader and will delete that log and all logs that follow it. And, we return to the above paragraph.

Failure Tolerance: Network Partitioning

The network partitioning is, in execution, quite similar to delayed and lost messages or a shutdown. If we partition the cluster of five, then one will either elect a new leader and proceed, while the other partition will continue to update but will be unable to receive a majority. Therefore, the nodes without a leader will become candidates (not all but 1 or more), and they will increase their terms as they restart elections until the partition is healed. During this time, the partition without a leader will not be able to take any writes, as there will be no node to process the set request. Upon removal of the partition, the nodes with the higher term will trigger an election and the situation will proceed as in “Follower Crash”. If the partitioned node has any unapplied logs, these will be rolled back and the broker will return the appropriate error message.

Implementation

One File Design:

We decided to implement Raft all into `node.py` to make it more convenient. While we thought of separating our code into different files (e.g one for every type of role) to make it more modular, we ultimately decided not to due to the structure of the initial given `node.py` file, and because the "architecture requires that [we] implement all those roles in the same program". The old implementation can be found at project/impl/deprecated/alt-node.py.

Navigating Our Code:

Our design and implementation can be categorized into the following:

- Set and Get
- Message Delivery/Sending
- Raft Node
- Log/Entry
- Roles
    a. Leader
    b. Candidate
    c. Follower

Data Structures:

We turned each Role—Leader, Candidate, and Follower—into separate classes. At first all this functionality was integrated into one large Node class (this file can be found in deprecated/node.py), where we used if conditions on the node's role to determine what action to complete. However, after realizing that there are set things we repeat once a node transitions from one role to another (i.e resetting certain fields, starting/cancelling timers), we redesigned our system to make it more modular. The following classes and their methods are described in more details below. The Raft Node is not a separate module and is merely a separate section of the class Node as defined in `node.py`. To assist with log replication, we defined Message object types and Log Entries. We outline each data structure in the subsections of our code design.

Set and Get

The get implementation has not been changed from the code that was already given in node.py. Once a client requests a get, the program will immediately return what is known based on the current state of the data center. For a set, the program will only process the request if the node is a leader. If not, it will send a message directing the client to the proper node to speak to or, it will ask the client to wait a few more seconds for the leader election to conclude. After the client sends a set request to the leader, the leader will then immediately add this to its log, and send a heartbeat to all its clients with the logs needed by each node. If the value is committed by the leader, we return a 'setresponse' showing its success. If the value is later dropped (perhaps due to an error), the program will return a delayed 'setresponse' error noitifying the client that the value had been dropped. This system is definitely more Consistent/Partition resilient than available.

Message Delivery/Sending

Data Structures Used:

- "Messages" : Type = JSON

We continue the use of json objects to send and receive messages between the broker and the cluster (though we now handle messaging through the leader). We built separate functions to handle the five new types of messages: "AppendEntries", "AppendEntriesAck", "RequestVote", "RequestVoteAck", and "Commit", and have a function sendMessage that communicates to the broker.

We also add a few new fields to the JSON objects: src, dest, term, lastLogIndex, lastLogTerm are a few. We note the use of source in "hello", but message direction and provenance matter more in Raft as we must direct to the leader and maintain matchIndex as well as other information about peers.

Our Message Delivery/sending is supposed to imitate the behavior of a Remote Procedure Call. Based on the paper, we are aware that ideally, our system would send messages and expect messages back from the destination node in one function call. However, because of the way chistributed is set up, the “send_to_broker” function (send messages function) is actually a pointer in each node. Therefore, we create a send and handle function for each RPC. This actually does make our code vulnerable to some race conditions, for example, a node might receive two heartbeats before it is able to acknowledge it. This is why, in our acknowledgement message, we also include the index that was just replicated during a success. This will prevent the leader from updating its ‘nextIndex’ and ‘matchIndex’ values for that node more than it should.

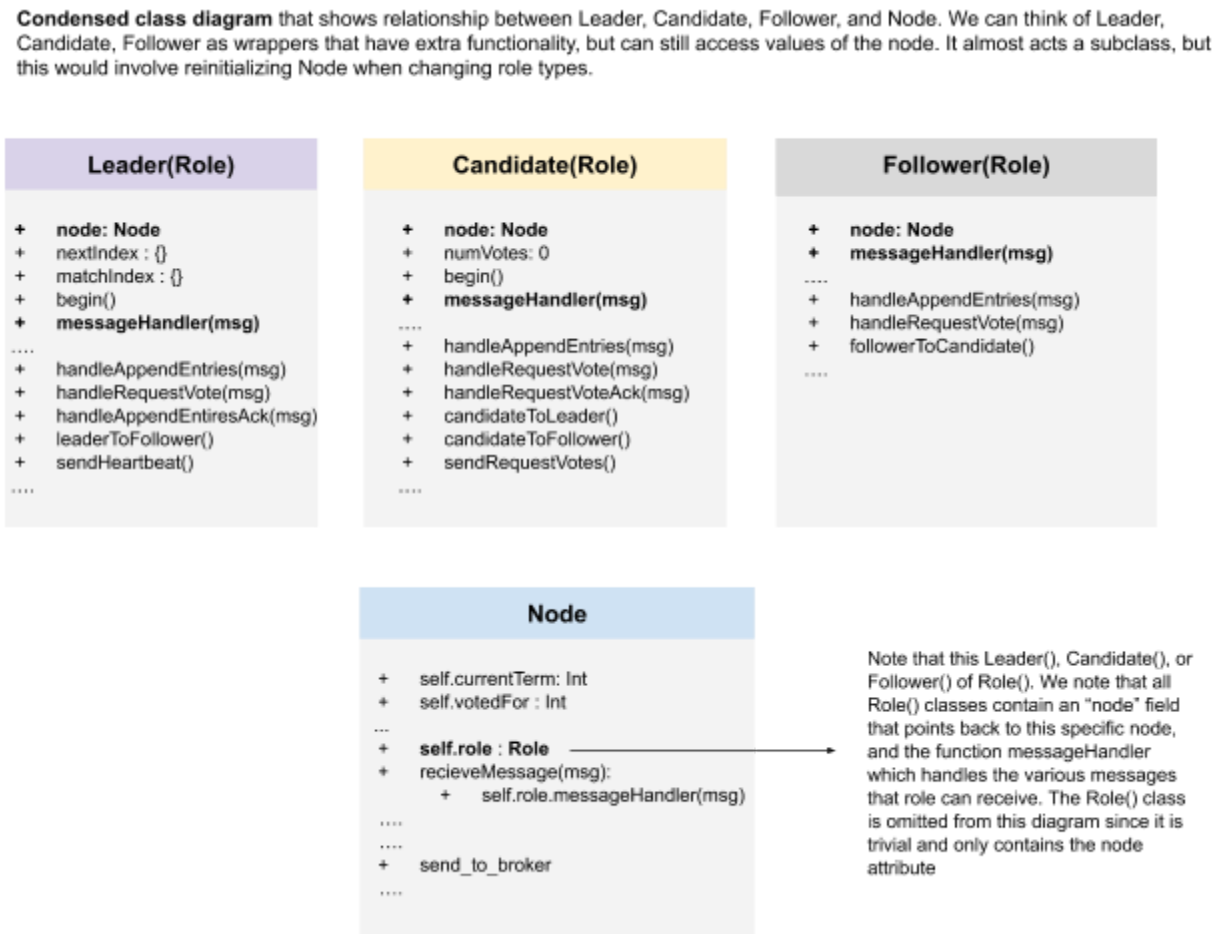

Figure 1. Details the structure of our code, with Leader, Candidate and Follower being a role of the ‘Node’

Raft Node

Data Structures Used:

- Node: Type = Node Class

Our implementation entails building a Raft node that contains the persistent and volatile states and the index into the Node class. The “Persistent” and “Volatile” variables are taken from the specification, while we initialize leader_name and Role. We set the following variables in addition to the node’s name and the list of peers:

- **Persistent**
  - currentTerm
  - votedFor
- **Volatile**
  - lastApplied
  - commitIndex
  - lastLogIndex
  - lastLogTerm
- **Leader_name**
- **Role:** Leader, Follower, or Candidate class

Log/Entry

Data Structures Used:

- Log Entry: Type = JSON
  - key
  - value
  - id
  - term

The Log Entry is a JSON object that we use to store in the RNode log. Rather than think about commands to the state machine (as is abstracted in the paper), we define a log in the node as a list of class Entries—where Entry has the key/value pair, term, and id. The Entry also includes an ‘id’. This is the id of the message from the broker. Therefore, when we respond back to the broker, we know which id value to set the ‘setResponse’ after we commit or drop the “SET” request. This allows us to pack in the key-value store and deal with an abstracted entry (as Raft adds on indexing and terms for log entries) when working with entries in the Raft Node log.

Roles:

Data Structures Used:

- Role: Type = Superclass for Leader, Candidate, and Follower
- Leader : Type = Leader Subclass of Role:
- Candidate : Type = Candidate Subclass of Role:
- Follower: Type = Follower Subclass of Role:

We define three roles that RNodes can take on—Leader, Follower, and Candidate. A Role class has a node field, and each role subclass has its own defined constructor and functions to receive and handle message types (the ones defined in Message Delivery/Sending) that are

specific to their role. This design was implemented when we realized we had several conditionals and repeated code specific to the Node's role.

Aside from messages, each Role keeps track of the following information:

- **Leader:** Sends AppendEntries (heartbeat), manages Log Replication
  - nextIndex
  - matchIndex
- **Candidate:** Begins Election, requests/counts votes, either becomes Leader or Follower
  - electionTimer
  - heartbeatTimer
  - numVotes
- **Follower:** Mostly just handles messages
  - electionTimer
  - heartbeatTimer

Testing on Chistributed

Test Set One: Key-Value Store and Fault-tolerance

five-nodes-test/runtest.chi

Our first script, *runtest.chi*, shows get, set, and replication, and then proceeds to test different levels of failure. Note that running *runtest.chi* loads all the other scripts we refer to as well (they do not work standalone as the start operation is in runtest.chi). If you would like to run the scripts on their own, you may either edit the start component of runtest.chi in or start the nodes in the shell before loading the file.

Why do we have timer commands?
Our scripts have several 'waits' to give our program time to go thought leader elections, commits, and more. Without the waits, the commands will be given much too fast. Our system is highly consistent, not available! In addition, the response may appear a bit after the the print detailing the expected result. For example, we print that a set happened, and the setResponse will be printed perhaps 1-2 logs over due to race conditions. However, we use wait times to reduce such delays as much as we can. So, if our test that our progam received a 'set' command, please also skim the rest of the logs to see the set response print if its no immediately after our log.

This set of tests first shows get, set, and replication, then proceeds to test failure scenarios that we highlight below, where we demonstrate that the Leader Election and Log Replication from the Raft algorithm both work and make our implementation resilient to fail-stop and fail-recover failures. Note that for the purposes of this script we have built in an option into our

code. If a node is named *leader*, it is guaranteed to become leader when the nodes are initially started. If a node is named *backup,* then it is guaranteed to become the leader in the re-election. We use this so that our scripts can be automated as we don't currently have redirection to the leader set up.

1. ***runtest.chi:*** We first want to show that get, set, and replication work. Once a leader is elected, that leader is responsible for AppendEntries and ensuring that the information is set across the cluster. This means that if node-1 is the leader and we set a value for node-2, node-1 is actually responsible for ensuring that node-2 through node-5 receive the value. In this way, get and set working with our implementation of Raft is actually initially a test of Raft working. We test this by setting a value on each node, then checking each node for the given value. This tests *set* on the leader, *log replication,* and *get* on each node. We then test setting a value on a node that is not the leader, only to receive an error and a pointer to the node.
2. ***one-node-failure.chi:*** This script tests the failure of a single node (not a leader), showing that our implementation is fail-stop resilient. Multiple values are set while the node is offline, and once the node comes back online, we see that it now successfully runs the *get* command.
3. ***two-node-failure.chi:*** This script tests the failure of two nodes (neither is a leader), showing the maximum fail-stop resilience Raft has on a five-node cluster. The test is conducted quite similarly to the one-node failure test as both nodes that are offline come back online and successfully update themselves to return the values that were stored while they were offline.
4. ***leader-node-failure:*** Here we test the failure of a leader node. This triggers a re-election once the nodes listen for a heartbeat but don't receive one in the set time interval. We test the setting of a value on the original leader, which will be delayed as the leader cannot reach consensus and has no response, and then test the setting of a different value on the new leader of the remaining four nodes (backup). We then recover the *leader* node and see that it now becomes a follower and successfully updates its log, dropping the old value (which now returns an error). While we know that *get* always returns, *set* does not return until a consensus is reached or a node rolls back an entry. This test shows the implementation of the latter as the leader rolls back its entry.
5. ***leader-transition:*** While this is not included in runtest.chi, we have included it for completion. This tests Leader Election without getting or setting a value in between. We partition off node-2 (backup), which will consequently keep increasing its term as it continues to request votes. Upon healing the partition, an election will be triggered and as the logs are equally up to date, node-2 will become leader. This test is standalone and not run with *runtest.chi*.

The above script shows that with a five-node cluster, our project successfully implements *get, set,* and replication. We also show that our Raft algorithm works by specifically testing Leader Election and Log Replication, and that our implementation is fail-stop and fail-recover tolerant.

Test Script Two: Network Partition

Having shown that our implementation of Raft is fault-tolerant, we next show that it is resilient to network partitioning. Our scripts here focus more on the partitioning than on *get, set,* and replication. We test different partitions, focusing on Leader Elections at the time of partition and on Log Replication upon removal of the partition. We do this by supplying two test scripts: *partition-leader-majority.chi* and *partition-leader-minority.chi*.

network-partition-tests/partition-leader-majority.chi

This test simulates a partition across a five-node cluster, where the leader remains in the partition with a majority (3-2). We demonstrate that *get* and *set* operations on the leader succeed, and that *get* and *set* on the partitioned nodes do not succeed and return an error. We then heal the partition and see that the partitioned nodes are now up-to-date. This test deals with the delayed/lost messages in the network partition, but does not introduce a re-election. Although one of the two partitioned nodes becomes a candidate with a higher term, the more up-to-date log on the leader will result in the leader winning the re-election.

network-partition-tests/partition-leader-minority.chi

This test simulates the leader ending up in the minority on the partition. The three-nodes elect a leader and we operate *get* and *set*. We also test *set* on the original leader before healing the partition and seeing that the *set* is rolled back as the original leader now becomes a follower and updates itself to contain the values set while it was partitioned. This demonstrates the completion of each command once we are failure-resilient. Note that both leader and node-3 drop the entry, so we get an additional “unexpected SET message”. This does not change functionality and is a harmless side-effect that could be dealt with in the future. Basically, all the nodes that drop logs will send a broker this message, but had we more time, we would implement it so that the setresponse with a warning is only sent once.

Challenges, Improvements, and Conclusion

Challenges

We faced several challenges and obstacles over the development of this project. We segment them below into logistical, technical, and spiritual obstacles. The logistical issues sprung out of the remote nature of the quarter and the spread across three different time zones. Technical issues were faced when working with development software, chistributed, and in several cases in the implementation of the algorithm. Spiritually, we had to learn to work well together and deal with burnout from whatever the last few months have been. There are several cases of *“if*

we had the time and energy, we would do X" (try this in Rust or Go, attempt to recreate Scatter, build our own chistributed), but as much as we love programming and have loved this class, we would really just like to get some sleep.

Logistical Challenges

Working together in a group of three on a tough PA isn't easy in-person, and it isn't much easier online. The 24/7 Zoom connection is at times calming, but as we worked in three different timezones across the US, we encountered some difficulties in setting up and pacing initially. Given the complexity of the algorithm, we also found it difficult to communicate about the project. However, we quickly fell into a rhythm, trading off between designing, debugging (pair programming or "triple tripping"), writing scripts/tests, and working on this paper. We observe that while some aspects of remote work are difficult to deal with, the ability to share-screens, stay on call, and work from the comfort of our own homes does come with some benefits.

Technical Challenges

We faced a myriad of these, from understanding the design and implementation of the Raft algorithm, interfacing/factoring our code with the initial setup in node.py, working in Python (we all had different levels of experience with the language), and setting up scripts in chistributed.

In the design and implementation, we faced some troubles in understanding edge cases such as rolling back commits that were sent to a leader that is partitioned away (chistributed doesn't actually shut down the node), comparing logs of identical size when the partition is lifted, and navigating some other unclear parts of the Raft design. We don't really have a conclusion as to whether Raft is easier to implement than Paxos, but we do appreciate the effort and more usable design. We also had the most trouble with certain race conditions and issues that arise with the speed of messages sent. For example, if we set our heartbeat to be too small, then the broker would send messages at a higher ratio than nodes received messages. Therefore, we really had to play with the times for our heartbeats and elections so that we could understand what was happening in debug mode.

In interfacing with Chistributed, we often found that we would run into key errors as nodes would attempt to communicate before the initial "hello" was actually sent, and we had to calibrate the timings away from the suggestions in the original paper to account for chistributed's speed, buffer, and timing. We also had to work in wait times into the scripts in order to work with our Raft election times and chistributed's execution.

Working in Python is always a strange experience, and this project is no exception. In gaining the ability to quickly write and test code, we struggled a little initially with maintaining good, consistent style and building out a more modular structure. The decision to directly implement

Raft into node.py was made in understanding the need to interface directly with the functionality defined in the Node class, but it did create some difficulties when debugging.

Spiritual Challenges

It is also important to note the context of the project. Amidst the pandemic and large protests across the nation, we take time to acknowledge our privilege in being able to still study and the resources that we have. Many of our peers have been severely affected, and our team as well has definitely felt the emotional turmoil of witnessing the heartbreaking events that continue to escalate. We thank you for the extension that has alleviated some of the burdens of all the external factors that have been affecting our studies.

Future Improvements

If we were to continue, we would want to optimize and separate our Raft implementation into a separate module entirely in order to support future development of the key-value store. We would also set up the redirection so that if any non-leader node receives a request, the request is forwarded to the leader instead of returning an error. We also want to work on testing and reviewing edge-cases and better understand how the system resets once it experiences failures beyond the threshold that Raft supports.

We want to also optimize the code as it is sometimes a little slow. In thinking about the product—the key/value store—we think that Raft plays an important role in consensus across servers, but another important aspect of databases is scalability. We looked into this in class and found interesting analyses and descriptions of implementing Raft (or aspects of it) at scale. We talked about MultiRaft,[1] optimizing heartbeats, and thinking about optimizing the leader choice in cases of varying stability in connections.

We discussed Scaling up Raft in class, where one of the authors of this paper wrote a post on the topic (Piazza @113). This paper (2.2)[2] explains an issue with scaling Raft—reliance on a leader. As the state machine/logs/all information must go through the leader (and all nodes must receive the heartbeat), two large issues to scalability are the leader's performance and the volume of messages. Another side issue from "You Must Build a Raft"[4] is distance—higher latency can lead to false positives.

The issue with scaling raft, as described by the post and cited materials, is that the direct connection between nodes and the leader as well as the creation of multiple Raft clusters with leaders that must communicate between themselves does not scale well for replication and sharding. While simple optimizations to Raft itself can include speeding up elections or using quorum reads, the algorithm in its current implementation is not well-suited to scaling up.

Given this, it might be interesting in the future to think about how Raft would work well with distributed hash tables, and particularly working with MultiRaft to have nodes send singular messages that contain information about all overlaps instead of sending one message per group that a node is a member of. Perhaps a good starting point is to first think about DHTs and sharding.

Finally, there is always tidying up the implementation and further modularizing the code. Abstracting message delivery and sending from the JSON objects the broker handles would prove useful if we develop more features on the database.

Observations & Conclusion

Reading and discussing the variety of papers this quarter gave us an appreciation of the formalization and thought that goes into making distributed systems work well. We discussed at a high level the efficiency of such algorithms, their use cases, and their interaction with larger heuristics in the space of consensus, time, and of course, CAP. The two programming assignments gave us a better idea of what it is really like to understand and probe into the consensus problem, but this project allowed us to get a much better look at the practical implementation of such solutions. Raft feels much more “modern” in its usability, and interfacing with the key-value store gave us a different appreciation for the papers we read and mostly thought about. Were we to have more time, thinking about Distributed Hash Tables and the AP-side of things might be an interesting area to delve into.

Fun Facts

A little bit more context of the title. To debug our code, we would each pretend to be a node and read out our lines. Node-1 would say, “I am the leader now! Accept my love, reply to my heartbeats!” Node-2 may say, “Oh crap, I’m out of date”. It made debugging go a lot faster and it was a great way to take apart a problem with 3 people.

References

[1] Darnell, Ben. “Scaling Raft.” *Cockroach Labs*, 27 Jan. 2020, www.cockroachlabs.com/blog/scaling-raft/.

[2] Deyerl, Christian, and Tobias Distler. “In Search of a Scalable Raft-Based Replication Architecture.” *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data - PaPoC 19*, 25 Mar. 2019, doi:10.1145/3301419.3323968.

[3] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm, 2014

[4] HashiCorp. “You. Must. Build. A. Raft! Consul's Consensus Protocol Explained.” *HashiCorp*, 8 May 2019, www.hashicorp.com/resources/raft-consul-consensus-protocol-explained/.

[5] Howard, Heidi. “[PDF] ARC: Analysis of Raft Consensus: Semantic Scholar.” Undefined, 1 Jan. 1970, www.semanticscholar.org/paper/ARC:-Analysis-of-Raft-Consensus-Howard/3665b13932eea50cf9ef5d32b85efc8a06a92b79.

[6] Lamport, Leslie. “The Part-Time Parliament.” *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, 29 Aug. 1998, pp. 133–169., doi:10.1145/279227.279229.

[7] Lamport, Leslie. “Paxos Made Simple.” *ACM SIGACT News*, 32(4):18–25, December 2001

[8] Raft, thesecretlivesofdata.com/raft/.

[9] Renesse, Robbert Van, and Deniz Altinbuken. “Paxos Made Moderately Complex.” *ACM Computing Surveys*, vol. 47, no. 3, 2015, pp. 1–36., doi:10.1145/2673577.

[10] Sotomayor, Borja. “uchicago-cs/chistributed.” *GitHub*, 30 Apr. 2018, github.com/uchicago-cs/chistributed/blob/master/samples/python/node.py.