Still Searching for an Easily Understandable Consensus Algorithm

Anna Kim, Lucy Newman, Ryan Teehan

June 1, 2018

1 Introduction

In this project, we implement a distributed key-value datastore in Python. In order to maintain fault tolerance and replicate values, we use the Raft algorithm. A prominent selling point of Raft is that it is easy to understand and implement, at least compared to Paxos. After having implemented the single-synod version of Paxos, we wanted to implement Raft to see for ourselves how it compares. While the concept of Raft is straightforward, we did come across a few issues in implementation. These include:

1. Helpful error responses are crucial in order for a system to be useful. We came across difficulties in attempting to return error messages in certain failure cases.
   (a) If a leader steps down while they have pending entries that have not yet been committed, then these could fail silently.
   (b) If the leader fails while a follower is attempting to forward it a client request, this could cause the forwarded message to disappear, and would require additional error handling including acknowledgements and a timer on the followers in order to return error messages.
2. Maintaining variables as described in Raft was not perfectly straightforward, and certain cases were difficult to debug. Difficulties encountered include figuring out how to set, modify, and remove timers for each state and in their transitions, and ensuring that all the various variables are set properly, including election votes, commit index, and log entries.
3. We felt at times that there was so much detail spread out throughout the Raft paper that it was difficult to parse and put together. There is a concise summary on page 4 of the paper, but it is missing some key details. The Paxos algorithm was summarized in two paragraphs in Paxos Made Simple, but the implementation of Raft was spread over the majority of the paper. This is not a complaint with the algorithm itself, and the extra detail is definitely helpful. However, we felt that a complete, concise enumeration of the entire algorithm would have made implementation easier.
4. Partially because of how Chistributed is set up, it was difficult to find a reasonable structure to the code. This challenge was not specific to Raft, and we will discuss this in more detail below.

In sum, though we enjoyed implementing Raft, we don't completely agree with the authors that it is vastly more simple to implement than Paxos. Part of the reason for this is that from our perspectives, consensus and distributed computing in general can be difficult to understand. From a conceptual standpoint, you need to understand what it means to reach consensus, why majority is required, and how this is obtained, as well as what the various error cases are and how they are handled, and why the algorithm's solution works for these cases. From an implementation standpoint, you need to understand things like threads and timers, which are not simple to a beginning programmer.

On the other hand, we agree that Raft made significant strides to make consensus concepts accessible. The strong leader and leader append-only properties helped simplify things, as well as their

use of randomness to prevent artificial hierarchies. Many of the design choices in Raft were very smart in their simplicity.

2 Summary of Raft

Raft is a distributed consensus algorithm that allows deployment of a distributed system/datastore while ensuring safety from certain kinds of faults and failures. It functions by electing a leader to propagate log entries to a set of machines in order to achieve a fault-tolerant system to agree on a sequence of commands. Raft prioritizes consistency over availability, and failures can at most cause (indefinite) delay. Furthermore, a minority of slow or faulty nodes will not cause significant damage to the system, as only a majority of requested responses are required for a value to be committed.

Additionally, an important goal of Raft is to keep the implementation as simple and intuitive as possible. There are only two remote procedure calls (RPCs) needed to achieve consensus: RequestVote for electing a leader, and AppendEntries for replicating logs. AppendEntries doubles as a heartbeat mechanism, telling the other nodes who is in charge. Whenever possible, Raft favors symmetry and randomness, rather than imposing an artificial structure on the system such as a rank hierarchy among nodes.

2.1 Leader Election

In order to simplify log management and make the protocol more comprehensible, Raft relies on strong leadership. Leaders accept client requests to write to and read from the log, send log entries to the other nodes in the system, and determine which log messages are committed based on when they have received responses from a majority of nodes[1]. Thus, a crucial part of the protocol is determining who the leader is. Once the leader is determined, that node handles a lot of the important parts of determining when consensus has been achieved.

Raft nodes can be in one of three states: Leader, Follower, or Candidate. Leaders, as described above, perform key roles in the consensus protocol. Followers passively receive and respond to messages from the leader, maintaining replicas of the leader's log. Candidates have detected that there is no leader and are attempting to determine who the new leader will be[2]. A leader regularly sends messages to all other nodes in order to maintain their authority. The messages sent are blank AppendEntries RPCs, so the heartbeat does not require additional machinery for sending and receiving. Each follower maintains a timer to determine whether they have received a heartbeat recently, and if their timer times out, they transition into Candidate state[3].

In Candidate state, a node randomly chooses a new timeout within some interval (the Raft paper suggests 150-300ms)[4]. When this timeout runs out, they send out a `RequestVote` RPC to each other node. The purpose of the randomly chosen time interval is to avoid multiple nodes concurrently requesting votes, which could lead to split votes. The `RequestVote` RPC contains the most recent index of the sender[5]. If the node requesting the vote is not at least as up-to-date as the recipient, the recipient will deny its vote. Since a candidate needs a majority in order to win an election, and a log entry needs a majority in order to become committed, this ensures that any candidate that wins the election has the most recent committed value.

Additionally, `RequestVote` RPCs contain a `term`, which a candidate updates each time it begins an election. If a candidate receives an `AppendEntries` RPC with a term at least as high as its own while waiting for votes, it will move into follower state and accept the sender of that `AppendEntries` message as leader[6]. If a candidate receives a majority of votes for a given term, it transitions to

[1] Diego Ongaro and John Ousterhout, "In Search of an Understandable Consensus Algorithm (Extended Version)". (2014), 1.
[2] Ongaro and Ousterhout, 5.
[3] Ibid., 5-6
[4] Ibid., 6.
[5] Ibid., 7.
[6] Ibid., 6

Leader state. Finally, if a split vote occurs, the candidates who sent out the `RequestVote` RPCs should have another timer in place to restart the election process, incrementing its term again.[7]

2.2 Log Replication

Log replication stems from the leader. The leader maintains a state machine to keep track of what entries have been committed. When the leader receives a `set` request from a client, it appends the entry or entries to its own log and issues an `AppendEntries` RPCs to all of the other nodes. If it receives responses from a majority of nodes that the `AppendEntries` call was successful, it updates the state machine accordingly and informs the client. `AppendEntries` calls contain the index of the entry most recently committed by the leader, so that the followers can update their own state machines.[8]

When a client receives an `AppendEntries` request, they check whether the leader is still in power by examining the term in the message. If so, they check whether they have any conflicting entries in their log, which have the same index but a different term, and if so they remove those entries[9]. They then append the new entries to their log and send an acknowledgement to the leader, to indicate that they were able to successfully replicate the entries.

If a node that is not the leader receives a `get` or `set` request, they forward the message to the node they believe to be the leader.[10] As noted below, we modified this to return an error and as the client to request from the leader.

2.3 Optimizations and Add-Ons

The Raft paper has a number of additional features which would improve performance or expand the cases for which the system can maintain consistency. It discusses snapshotting, which seems to be a fairly standard method for saving space, and describes how they decided to break the strong dominance of the leader for snapshotting[11]. They have an additional RPC which allows the leader to send chunks of a snapshot to a follower, rather than waiting for them to catch up sending the uncompacted log.[12]

Raft also discusses a method for maintaining consistency in the face of network membership changes, by having the consensus be based on a majority of all the nodes in the old and new configurations until the new configuration is decided.[13]

We did not decide to implement either of these possibilities. While they are interesting to consider and certainly add to the usefulness of the system, we decided that they were beyond the scope of this project. This would be an interesting further direction to explore given more time.

3 Description of Implementation

In our implementation, we follow the structure suggested in the Raft paper to a large extent. When we differ from the implementation, it is usually because the implementation did not fully describe how something should be done. For example, we maintain sets of nodes that accepted our `RequestVote` calls, so that we know when a majority has been reached. While not described explicitly in the paper, this is the obvious implementation. Where our code differs significantly from Raft is in the handling of cases where the client might have to receive an error. In order to send helpful error responses, we had to make some small modifications to the algorithm.

[7] Ibid., 6.
[8] Ibid., 6-7.
[9] Ibid., 7.
[10] Ibid., 5.
[11] Ibid., 11-12.
[12] Ibid., 12.
[13] Ibid., 10

3.1 Class Structure

3.1.1 State Class

The `state.py` includes an `enum` class to specify the roles of the nodes. A node can be a leader, a follower, or a candidate. This simplifies transitioning from state to state and is used instead of using strings because it is less prone to errors from misspellings or case issues. This is the one exception to our rule of not importing user defined classes as `message.py` was used as a rough reference rather than anything usable.

3.1.2 Node Class

Each machine in chistributed runs a copy of `node.py`. It specifies the `Node` class, which has the attributes and functions to run a node in chistributed as well as run the full Raft algorithm. We implement the Raft algorithm as described above, with a few small modifications.

In general, we want the attributes in the Node class to keep all information relevant to maintaining and updating the data store, along with general invariants, such as lower and upper bounds for timeouts. The attributes were, generally, split into categories based on the persistent state on leaders, volatile state on all servers, and volatile state on all leaders, as described in the Raft paper's concise summary[14], as well as additional things we found necessary such as sets to keep track of votes received by candidates. Since nodes don't know when they fail, volatile state information is reinitialized in state transitions to simulate that it could be lost. Additionally, each node keeps track of what state it is in, as well as the identity of the current leader, so that it can tell the client who the leader is in case they contact the wrong node. The data store itself has two components: `store_log` is a list of all log items appended to the node's store, and `store` is a dictionary representing the portion of `store_log` known to be committed. The node class has methods for handling each type of request including client request and messages related to leader election and log replication, as well as methods to transition classes.

3.2 Client Requests and Error Handling

In Raft, all `set` and `get` requests are handled through the leader. We decided to send an error message if a client contacts a non-master node. Though this sacrifices a level of availability, it makes it easier to maintain consistency.

3.2.1 Motivation For Sending Error Messages

One design decision that came up is how to deal with the case where a client sends a `get` or `set` request to a node that is not the leader. In this case, the paper recommends forwarding the message to the leader. However, consider the case in which the following things happen sequentially:

1. A client sends a request to a follower node.
2. The follower forwards the message to the node it believes to be the leader.
3. The leader fails.

In this case, the follower will get no response from the leader. The follower could implement a timer system to ensure his forwarded client requests are acknowledged in a reasonable time-frame. However, for simplicity of implementation, we decided to simply return an error message to the client instructing them to re-send their message to the leader. Since leader turnover is ideally infrequent, we can assume that a client who makes a single request will know for a long time who the leader is, so this will not add a lot more overhead in terms of client requests.

[14] Ibid., 4.

3.2.2 Implementation

We implemented this with a client request handler function which takes a callback as an argument. The node receiving the client request checks its own state. If it is a candidate, it simply tells the client to try again later, since elections ideally don't take that long, and it does not know who the current leader is (if any). If it is a follower, it tells the client to re-send the request to the leader, giving the leader's identity. If it is a leader, it calls the callback function that performs the work of the client request.

3.3 Handling `set` Requests

Set requests at the leader closely follow Raft's suggestions. When the leader receives a set request, it adds the entry to its log and sends out an `AppendEntries` call to the rest of the nodes. Since Raft suggests that heartbeat messages can be `AppendEntries` messages, we take advantage of the heartbeats to send log entries to clients which they have not yet already received. Once it has received acknowledgements of the call from a majority of nodes, this entry is considered committed. Acknowledgements are kept track of in a `match_index` dictionary, which maps the name of each node to the highest index received by that node. Since acknowledgements increase monotonically, we can simply replace the old number with a new one upon receiving acknowledgements. In this case, the fact that Raft is leader append-only and all of the entries that a node has not acknowledged are sent every time until they are acknowledged simplifies implementation, since we don't have to deal with gaps in acknowledgments.

We have a property for `leader_committed_index` which is calculated based on the `match_index`. This is because the leader's commit index depends directly on what has been acknowledged by the followers. This is done by calculating a histogram of how many nodes have acknowledged each number, and considering the leader's commit to be the greatest one for which more than a majority have acknowledged. An optimization here is that we maintain a variable `_prev_committed_index` to keep track of the index previously committed, so that when the new commit index is being calculated, the node does not have to start back from one for the histogram.

A complication arose in dealing with the case in which a leader has committed entries, and then fails. When the new leader is elected, they have the most recent committed entries, but they don't know exactly which those are. Raft suggests sending out a no-op in order to check what the committed entries are.

3.4 Handling `get` Requests

Get requests in Raft require the leader to send out a heartbeat to the nodes and receive a majority of responses acknowledging that it is still in power, so that it can avoid servicing stale data. Beyond that, the get request is a simple lookup in the `store` dictionary.

3.4.1 Ensuring Reads are not Stale

In order to ensure that a leader has not been deposed without his knowledge before responding to a `get` request, the leade must send out a heartbeat to all the other nodes and get a response before returning the value to the client. We hanle this by writing the `get` request to the leader's node with the key being the one requested and the value being `None`. The leader then sends out an append entries request. When we receive an acknowledgement for log entries, if the value of the entry is `None`, then we handle the acknowledgement a `get` acknowledgement. We respond to the client with a `getResponse` and don't apply the entry to the store.

3.4.2 Quick Get Requests

On the master node, as soon as a log entry is committed, we add it to a dictionary representing the most recent state of the logs. Thus, whenever the master receives a `get` request, there is a

constant-time lookup to determine whether the value is in the dictionary and retrieve it, since dictionaries are implemented as hash tables in Python. Without this dictionary, lookup would have been linear time in the number of log entries, since we might have to traverse all of the log entries in order to find a key that is at the beginning or not in the log at all. Since a data store may receive many read requests, this optimization will improve performance.

3.5 Fail-Recover Fault Tolerance

Since our implementation is based on the Raft algorithm, it is tolerant of any number of nodes failing and then recovering later. Since the leader handles all `get` and `set` requests, if any number of followers fail, consistency will still be maintained in the store. If followers fail and then recover, they will be notified of the log entries they missed as well as their new commit index upon recovery, since the leader keeps track of what has been acknowledged by each node in `next_index`. If a majority of followers fail, `set` requests will be delayed until they recover, but they will go through when the nodes come back since the leader has stored these requests in it's `store_log` and continues to send all of the entries each node hasn't received until they are acknowledged.

Raft also has infrastructure in place to deal with failure of leader nodes. Each follower maintains a timer to ensure that they have heard from leaders recently, and if they have not, they start a new election. The leader regularly sends out heartbeats in the form of (possibly blank) `AppendEntries` calls to ensure that each follower has heard from it recently in the absence of failures.

We implemented leader elections in a fairly straightforward way following the specifications of the Raft paper. We included a method `start_election` that increments the term and sends out `RequestVote` calls to all nodes that it knows are available. We maintain sets of acceptors and refusers so that the candidate knows how many responses it has received and when it reaches a majority. If it receives a majority of votes, it becomes the new leader. In the case of a split vote, the election is restarted.

3.5.1 Timers

Nodes in each state maintain a timer to ensure that if a majority of the nodes are up, a leader will eventually be elected, and therefore the system will eventually become available. In our implementation, timers are handled by the ZMQ `IOLoop` object. In particular, we use the `call_later` method from `IOLoop`, storing the resulting object as `timeout_oubject` so that it can be removed the node transitions states or needs a different deadline for the timer.

When a node transitions to the candidate state, they start a timer with a randomized amount of time, so that all followers don't concurrently start an election when a leader times out, since this would result in a split vote. If the candidate a

Heartbeats are used in order to make sure that the leader is still alive. The follower timer ensures that if a heartbeat is not received within a certain period of time, an election is triggered to elect a new leader. If they do receive a heartbeat, the current timer object is replaced with a new one with a deadline `heartbeat_timeout` seconds after the current one. The leader's timer object has a callback called `startbeat` which sends out a heart beat and restarts the timer with the same function as a callback.

3.6 Deterministic First Election

In order to facilitate testing, we decided to make the first election deterministic. We did this by ensuring that once a node joins the network (via `start` in Chistributed), they send out a single `RequestVote` call, and get back responses from all nodes in the network. Only when the node has received responses from all of its peers (listed in `peer_names`) can it re-start the election upon failure to receive a majority of acceptances. Thus, the last node to join the network will always re-start the election first, and will win the first election. We found this helpful in testing, since we immediately knew who the first leader is. The subsequent elections are not deterministic since the timers are randomized and all nodes are aware of all other nodes following these initial elections.

Thus, `RequestVote` doubles as a sort of handshake in this case, in which nodes are able to discover who else is on the network.

4 Discussion of Scripts

There are many features to the Raft consensus algorithm that must be tested to ensure correctness of the implementation. Our test scripts (`test.in` or `t1.test`) and their output files `test.out` describes the result of running tests in the Chistributed shell. Any file with the extension `.in` test single features as described in the discussion below and may be run with Chistributed directly. We have added `wait` commands in between each client request to give Chistributed time to process and respond. Test files with the extension `.test` describe the test scenarios that combine the aforementioned single features in different orders, and `[leader]` must be substituted with the correct leader name. Furthermore, the output files that we have included have been processed to remove the `>` character when it appeard in Chistributed. Therefore, the lines that start with a `>` prompt are input, while the others are output. Because of our previous design choices, we first start all nodes as specified in `chistributed.conf`, and then we are able to run a number of tests on these nodes. If more than a minority of the nodes have failed, our system only returns to the consistent state once at least a majority of the nodes are fuctioning.

4.1 Get and Set

In the absense of failures, `get` and `set` requests should return an error message when directed at a non-leader node, and a success message when directed at the leader.In lines 29-44 of our test script we attempt to set and get a value from different followers, and the follower returns a response that the client should re-send the request to the leader. On lines 14, 19, and 24, the leader receives a `set` requests with different key-value pairs, and notifies the client that that was successful. Note that we knew before enquiring followers that `node-5` was the leader, since it was the last one to join the network, and the last node to join the network wins the first election by design. In lines 16, 21, and 31, the leader receives a `get` request and responds with the correct value.

4.2 Concensus and Leader Failure

Now that we know `get` and `set` are working as desired, we can test for consensus. Since the leader is the only one who can process client requests, the only way we can test consensus is if a leader fails and another is elected. If the same value in `set` request sent to leader A is returned with `get` request sent to leader B, we can conclude that a consensus has been achieved.

Furthermore, when a leader fails, the follower timers will run out, and they will start a candidate timer with a randomized timeout, after which they start an election. One node will win the election, or in the case of a split vote, they will re-start the election process after another timeout occurs.

On line 14, we set a value to a key. Then on line 17, we fail `node-5`, the known leader. On lines 22-28, we request the value of the same key set by `node-5` from all other nodes, since we know that one of them will have been elected the new leader. The new leader responds with the same value that `node-5` set.

4.3 Follower Failure

In the case of a minority number of follower failures, consensus is still reached. On line 14, we fail a single follower node. The values returned from lines 19 and 27-31 show that consensus is still reached. In the case another node fails and there are more than a minority of node failures in the network, client requests to the leader will not return a success message. Instead, the leader will return a success message when enough nodes have recovered for a majority functionality. This is seen in response to the recover command in line 42. Furthermore, the system still reaches a consensus after coming back from majority failures as seen in lines 47-51.

4.4 Network Partition

Consensus is still possible with a network partition in Raft. In the case that a leader is partitioned with a minority half of the network, the client requests will not be processed since it will not be committed. On the majority half of the network, a new leader will be elected, and a request to the leader on the majority half will be successful. Furthermore, after the partition heals, the leader with a higher term number will be the new leader, and any requests that was sent to the previous minority-leader will not have taken effect.

Line 14 creates the partition with `node-1, node-3, node-4` on one side and `node-2, node-5` on the other. Since `node-5` was the leader before the partition was created, line 17 should and does not have any responses. Lines 20-24 requests that a key-value pair be set by the new leader in the majority partition. To check for consensus, lines 27-35 requests the same key-value pair be returned, and those in the majority partition succeeds in reaching consensus and returns the correct value while the leader in minority partition doesn't return anything. Furthermore, when the parition is healed on line 38, consensus is reached among all the nodes in the network, as seen in lines 41-49, and the value set to the key is the one sent to the previous majority-leader.

5 Issues and Challenges

5.1 Handling Replies to Clients

One issue that came up here is that we need the ID of a message in order to respond to a client. We decided to add that to the items that are stored in the log. This is not discussed in the paper, but it was necessary for responding to clients.

To achieve fault tolerance yet provide fully functioning `get` and `set`, we had to consider the case in which requests are issued in between the time that the leader fails and another is elected. As described above, we decided to fail client requests sent to non-leader nodes, rather than dealing with saving requests at follower nodes and sending error responses to the client from the follower if the messages time out. This seems like a fairly natural extension of the Raft's strong leader property, and it is the standard implementation in other systems such as Chubby.

5.2 Maintaining State and Variables

There are a few implementation details in Raft that were difficult for us to implement in practice. For example, determining the commit index of the leader requires both the leader to know when it gets a majority of acknowledgements that it has sent, which we implement by histogram counting, and also for a new leader to know which entries in its log are committed.

Additionally, we initially had trouble implementing timers. We first tried a `while` loop that told the node to do an action and then sleep for a set amount of time, but this seemed to block the node from receiving messages while it slept. Upon suggestion of another group, we looked further into the `IOLoop` class, and fount that the `call_later` function did what we needed it to do. However, putting `call_later` in a loop caused the loop to continue to run very quickly and start many timers, since the `call_later` function seems to start a thread and return immediately. Thus, we implemented heartbeats by having the callback call itself rather than using loops. Further, we discovered that we needed to maintain another variable for the timeout object so that it can be removed whenever the timer needs to change. We then had to ensure that the timer was removed and restarted in all the cases that it should be.

5.3 Raft Paper Details

When we were initially implementing this project, we were looking primarily at the concise summary on page 4. While this summary is helpful as a starting point, it is not complete. In several cases, we came across challenges that we weren't sure how to address, but discovered later that

they were addressed elsewhere in the Raft paper. For example, the issue of new leaders discovering which entries are committed was cleverly solved by a no-op in the Raft paper, but we were not sure how to deal with it before we found that part. This detail was described towards the end of the paper under "Client interaction," which is not where we expected it to be. Additionally, the idea of sending a heartbeat before responding to `get` requests was mentioned here as well.

Related to the problem of heartbeats, we felt like there was some ambiguity about how to deal with different leaders in a network partition and whether leaders should care about not receiving an `AppendEntries` response within a certain time period. This was particularly important in the case of a network partition. In the end, we determined that this would run contrary to the rest of the algorithm and confirmed that our implementation behaved correctly. We felt it would have been helpful to have a concise summary of the entire algorithm including client interaction.

5.4 Class Structure

Partially due to our lack of experience with object-oriented programming, and partially because of challenges with Chistributed, we felt that the class structure we decided on was a bit clunky. Raft nodes separate very naturally into leaders, followers, and candidates, and our initially idea was to have a class for each, each of which is a subclass of a `Node` superclass. This would have simplified a number of parts of our code, where we have to check the state of the node and then handle it differently depending on the state. For example, the `handle_set` method could return error messages for clients and followers, and send out `AppendEntries` calls for leaders, rather than dealing with cases and callbacks.

However, when Chistributed is started, there is a single object that is initialized which is able to receive and send messages to the broker. When a node transitions state, we would want the object simulating the node to be of a different class. This could be done by either (a) initializing a new object of the desired class and putting it in the place of the old object or (b) changing the class of the old object to the new class (by changing the `__class__` attribute). We were not sure how to put a new object in the place of the old one in Python, and we were worried that changing the `__class__` might have unintended consequences. Thus, we decided to keep the code in one class, but we feel that a better solution would likely involve separate classes for leaders, candidates, and followers.

6 Conclusion and Lessons Learned

We implemented a distributed datastore using the Raft consensus algorithm to maintain consistency and tolerate failures and network partitions. In doing so, we learned about the difficulties of altering an existing implementation outlined in an academic paper to fit the constraints of a specific testing environment, in this case Chistributed. In part, this was a lesson in good and bad ways to structure code and the drawbacks that come from implementing an algorithm like Raft without the help of additional classes. Furthermore, we learned about how to make compromises between competing desires for consistency, availability, and partition tolerance in real time when trying to deploy a distributed consensus protocol. The design decisions one makes from the outset can often have a large impact on the compromises and decisions one needs to make as the project develops. Beyond simply dealing with our own design decisions, we also had to account for the structure of pre-given sample code and dependent packages, such as those used for asynchronous programming in Python. The need for asynchronicity added another level of complexity to the task and forced us to think about whether particular functions were thread-safe or not. Ultimately, this project gave us exposure to many of the problems inherent in working with distributed systems while giving us the autonomy to determine the best way to handle them.

References

[1] Diego Ongaro and John Ousterhout. *In Search of an Understandable Consensus Algorithm.* 2014.