Anthony Crespo, Brandon Callender, Tasnim Rahman

CMSC 23310 Final Project

Raft Summary

Basic algorithm is RAFT. This is a leader based protocol that allows individual nodes to keep a log and use this log to interact with a key value store. RAFTs correctness and safety properties stem from the fact that only the leader is allowed to commit entries, and the restrictions in place for how to elect a leader[1].

In the lack of failures, RAFT works by having the client send requests to the leader. The leader will then add the request to its own log and forward the request to all followers. Upon hearing from a majority of the nodes the leader will mark the entry as committed, increment the total number of committed entries, and apply the operation to their state machine (in our case a key value map). This number will get forwarded to the followers the next time that the leader sends out an appendEntries RPC to them, and upon receipt they will also mark the entries as committed, and apply them to their local state machine.[2]

AppendEntries RPC	**AppendEntriesResponse:**
This request is sent from a node in the leader state to all other nodes in regular intervals. It includes the fields : ● Term: the term of the leader sending the message, allows other nodes to update their term to the current one or ignore the message if they are further ahead. ● leaderId: The name of the leader sending the request, so others know who to send responses to. ● prevLogIndex: the index of the entry immediately preceding the contained entries. ● prevLogTerm: The term of the entry immediately preceding the	The response a node sends after receiving an appendEntries RPC. Has following fields: ● Term: the term the node is currently in. Used to depose an outdated leader. ● Success: A boolean stating whether the this nodes has an entry that matches the index and term in the original request. ● logIndex (our implementation): The matchIndex the leader can use. Only valid if success is true. More info below.

[1] http://ramcloud.stanford.edu/raft.pdf
[2] https://raft.github.io/slides/raftuserstudy2013.pdf

contained entries. This and the above are used for a recency check on appends. ● Entries[]: A list of entries to append to the log, may be empty. ● leaderCommit;: The furthest index the leader has committed, used to know when the followers can safely commit an entry.	

RequestVote RPC:	**RequestVoteResponse:**
● term: The current term the candidate is requesting a vote for. Used to depose old leaders and update followers. ● candidateId: Name of the node seeking a vote. ● lastLogIndex: The index of the last entry in the candidate's log ● lastLogTerm;: The term of the last entry in the candidate's log. Used with above to check whether the candidate's log is more recent than the receiver's.	● Term: term of the responding node. Used to be able to ignore old votes, and update term if needed. ● voteGranted: Boolean stating whether the request was succesful

The implementation these messages, and other Raft details, such as elections and state transitions are covered in detail during the section on our implementation to avoid repetition.

Implementation Details

ZeroMQ Configuration

The BrokerManager handles all inter-thread and node<-->chistributed communication. As in the sample code, there are two sockets for communicating with chistributed, a subSock that

receives messages from chistributed (eg SET, GET and others)and a reqSock that sends messages to chistributed (and thus, other nodes). Unlike in the sample code however, the reqSock is implemented as type DEALER rather than REQ -- to avoid the rigid restrictions that come with REQ-ROUTER relationships. (The REQ socket will send only one message before it receives a reply; the DEALER is fully asynchronous. -- ZMQ docs). Nodes can spawn threads when they need to send heartbeats or an election times out -- these threads need to send messages, but ZMQ sockets are not threadsafe.

To improve stability, we chose to use the worker PAIR socket pattern described in the ZMQ docs, where the election/heartbeat threads do not send messages directly (as the main thread's socket identity is unique), but instead pass the messages back to the main thread to be forwarded at the next poll. While this is a recommended pattern, it obviously imposes extra latency on heartbeats, which could prove harmful if the heartbeat interval was very close to the election timeout. In practice, this extra latency was negligible, as the main thread rarely does enough work to stop the polling function from always retrieving all of the available messages for relaying. In any case, our electionTimeout is slightly higher than that recommended by the Raft paper to compensate.

After the BrokerManager receives a message destined for a node, it passes it through to the node's handleMessage function. This function parses the JSON message and captures the messageType (GET, SET, etc.). It then dispatches the message to an appropriate message handler for that messageType.

Client Interaction

With our implementation of RAFT, only the leader is allowed to respond to *get* or *set* requests. If any other node receives the request, then it will forward it to who it believes to be leader to respond. When the leader receives a *set* request it first creates a new log entry detailing what the operation is and appends it to its log. In normal operation, this log entry will get sent out with the next batch of heartbeat messages to all follower nodes. It also adds an entry to its commandsInFlight data structure, which keeps track of which requestIds correspond to which commands to reply back to clients. Upon receiving the AppendEntries RPC, the followers will append the entry to their log and respond stating they have done so. Upon receiving confirmation from a majority of the followers, the leader applies the entry to its state machine, responds to the client, and notifies the user that the request was a success. Upon receiving the next heartbeat message, the followers will know the entry is committed and apply it to their respective state machines.

There are several possible points of failure in the above process:

(1) If the node who receives the *set* request doesn't know who the leader is, or forwards it to someone who does not know who the leader is, the request fails with an error stating that the leader cannot be found.

(2) If the leader fails prior to notifying others of the request, then the *set* request will get lost. An error message will be received from the node when it recovers. For this reason, we cannot fully support fail-stop failures -- however, the window of failure is quite narrow, the amount of time between a request and the next heartbeat, around 150-300ms maximum.

(3) If a majority of nodes do not respond successfully to the AppendEntries RPC, then the result will never be committed, however, assuming all machines eventually come back online we should eventually hear back and the request will succeed.

(4) If the leader fails after reporting success, but before notifying the followers of success, then the log entry must still be committed in the following term. Because of RAFT's recency check on elections, and because a majority of nodes have successfully entered the request into their log only one of the nodes that has entered the log entry into their log can become the next leader. However, the next leader is only allowed to commit requests that match its term number, so it cannot commit this entry until it has committed something else. We get around this restriction by having each leader commit a no-op upon winning an election, which will implicitly commit any lingering log entries.

The response message includes a field not present in the original RAFT paper, which is the size of the node's log. On failure this field is unused, but on success this is the field that is up to date with the current leader, and is used to update leader state for further RPCs to that node. The field returned is used in the leader's matchIndex for that node, and the leader's nextIndex is just one more. This is safe because the follower node returning success means that it's log matches the log that the leader sent to it. In our implementation this also means that if the leader sends out an empty list of list of following entries, then any following entries on the follower will just get deleted so that a successful appendEntriesResponse means that the follower's log exactly matches that of the leader when the original appendEntries was sent.

For *get* requests, the request is similarly forwarded to the current leader. After receiving the request, the leader waits until it has heard back from a majority of the followers and then will respond using the current state of their store. Failing a *get* request is much simpler because there are no lasting effects. We only fail a get request when we are unable to determine the leader (the node that received the request or was forwarded it does not know who the current leader is, or the node is currently in/beginning an election) or the leader does not have the key in its persistent storage. These two cases have different error messages.

A successful *get* request occurs if a leader receives the request, verifies it is still the leader, then replies to the request using the most recent value in its persistent store. This verification step ensures that reads and writes to Raft are linearizable, and is described in section 8 of the Raft paper -- once a set operation has returned to the client as complete, any concurrent/subsequent *get* operation that returns after that *set* must reflect the new value. After each successful appendEntriesResponse is received, the responding peer is added to the set of affirmative responses for each get command currently in flight (stored in

commandsInFlight) for the leader. When a quorum of nodes is added to this response set, we know that we are still the leader and can respond safely and linearizable to the get request with the current value in our persistent store.

Leader Elections

Leader elections proceed quite closely to their reference implementation in the Raft paper with a few key optimizations. Like Raft, all nodes start up in the FOLLOWER state (unless the command to force a node into a leader state is used). If they do not receive a heartbeat message from a node claiming to be the leader within the window of their randomized election timeout (500ms-1000ms in our implementation), the node will begin an election to attempt to select a new leader. In our implementation, this is accomplished using a ScheduledFuture to set a timer after which a new Thread will be spawned to handle election logic. This new thread transitions to the CANDIDATE role, updates its term, then sends out RequestVote messages with the information discussed in Section 1 through chistributed to the rest of the nodes in the cluster. A candidate continues doing this until (1) it wins the election, (2) another node wins the election, (3) the candidate does not receive information about a new leader or enough information to know if it has won the election within the election timeout.

(1) A node knows it has won an election when it receives RequestVoteResponses (with the current term and success:true [equivalent to voteGranted:true in the raft paper]) from a quorum of nodes. Alternatively, a node can lose an election by receiving a quorum of no votes without any candidate winning the election. This means that this node's log is behind a quorum of nodes in the cluster -- in our implementation, such a node will not attempt to compete for re-election again, as it is impossible for it to *become* up to date during the same election term, an optimization.
(2) A node knows when another node has become the leader when it receives an AppendEntries heartbeat from a new leader on a non-stale (currentTerm or greater) term. It transitions to FOLLOWER, and in doing so updates its term and restarts its election timeout thread
(3) In this case, we don't receive responses from a quorum of nodes or an appendEntries within the election timeout. In this case we begin the election again, incrementing the term. Randomized election timeouts guarantee that elections won't continue forever if a quorum of nodes is available to vote.

A node that receives a RequestVote checks that either the vote's term is higher than the current term or that, if the vote term is the current term, that it hasn't voted in this election or previously voted for the node sending the RequestVote. If these tests pass, the node checks to see if the RequestVote it received came from a node that is "as up-to-date" as itself, making sure that either the lastLogTerm of the request is greater than the term of the latest item in its log or that if the lastLogTerms are equal, the lastLogIndex is higher. If the node

that sent the RequestVote is as least as up-to-date as the receiving node, it receives a successful RequestVoteResponse and the receiving node's election timeout is reset (not explicitly mentioned in the Raft paper but avoids cascading elections -- the first to send out a RequestVote, if it can become the leader, will become the leader faster). Otherwise, it responds with a failed response and does not reset its timeout.

As discussed in Section 8 of the Raft paper and above, when a leader wins an election it commits a no-op to its log to commit potentially uncommitted entries from the previous term.

Fault Tolerance Details

As previously mentioned, it is technically impossible for our implementation to be fail-stop within the constraints of our assignment, as a node can fail before it can respond to a message, meaning it will never get a response. Moreover, the leader is the only node during a *set/get* that is keeping state on requests in flight, so a leader failure, while safe[3], will result in all in flight requests to be dropped without response.

Our implementation is resilient to fail-recover events with re-delivery (but not without re-delivery, as explained in our script section and below in the network partition section).

Our system is, moreover, resilient to network partitions (if the messages are redelivered) and will maintain linearizable reads and writes. In the ideal case, a partition will have a quorum of nodes on one side (a major partition). That partition will be able to elect a leader (or retain its old leader) and continue to make progress processing gets and sets. The smaller partition (a minor partition) may have a node in it that thinks it is the leader (perhaps the leader got partitioned off with a couple of other nodes from the cluster), but will be unable to respond to *set* commands because it cannot replicate log entries across a quorum of nodes and will be unable to respond to *get* commands with potentially stale data because it will not be able to verify with a quorum of nodes that it is still the leader[4].

The worst case scenario (as explored later) is one in which the two partitions are of even size and neither has quorum -- if sets are sent to both sides of the partition, none will respond until the partition is healed. Once the partition is healed, behavior is *nondeterministic* as to which side of the partition will 'win', but either way, the behavior is *consistent* -- if the client receives an affirmative or negative *setResponse*, that will be reflected in future gets to any node in the healed partition.

[3] Nothing that was responded to as committed by the leader has not been propagated across a majority of nodes in a cluster, nothing that was denied was replicated or committed on any nodes.

[4] The latter condition can be relaxed if the delay of waiting for a heartbeat before responding to *get* requests adds too much latency for your needs. If one allows any leader to respond to get requests immediately, a client could receive stale data if it asked a leader in a minor partition.

Our system is not resilient to network partitions without re-delivery, however, as there is a 500-1000ms window after a partition in which the nodes on the side without a leader still believe the old leader is valid (before timing out) and will forward *get* and *set* requests to it -- never to be heard from again. This window can be narrowed by lowering the election timeout, but is a necessity unless we only responded to reads and writes at the leader (or allow for the possibility of multiple nodes sending back a response to a query[5]) , which isn't practical for this assignment. These *set* requests will not be committed, so the safety of our implementation is correct, but it is not lively during network partitions without re-delivery.

Auxiliary Implementation Details

The log starts with an initial no-op at index and term 0. All nodes commit this at start up, and so are assumed to have this as the initial matching term. This allows us to start indices at 1 like they do in the RAFT paper, but also not have to check if the log is empty because the no-op is never deleted.

SET and GET messages are forwarded by constructing a setForward or getForward message, then sending that to the node currently believed to be the leader. To prevent duplicate responses (discussed in-depth above, in Fault Tolerance and in [3]), no state about messages forwarded is kept in the node's commandsInFlight data structure.

Both the appendEntriesResponse and requestVoteResponse use the same message structure. RPCMessageResponse With a general "succes" field taking on the role of the "voteGranted" and "success" field in the RAFT paper.

AppendEntriesMessage, RequestVoteMessage, RPCMessageResponse, and ErrorMessages are subclasses of the Message class. They are serializable/de-serializable to/from JSON using GSON. Their specific variables, when they differ from the the raft reference, are documented.

To persist committed operations, we use a file-backed key-value database library for Java called MapDB. It supports far more complex operations, but we essentially use it as a serializable HashMap that writes to stable storage after an operation is committed to the store. It is configured to create its databases in the /tmp directory for ease of use/testing, but could easily be configured to read to a permanent file db named after the node, meaning that nodes could easily recover from fail-stop type failures and pick up their store where they left off.

Chistributed Scripts

[5] A node -- even one that is not the leader, could keep state in its commandsInFlight of set requests that it receives and forwards, and could fail the request if it has its log overwritten by a heartbeat. However, if the leader did receive the forwarded request, it might begin committing the 'new' set request after the original receiving node had sent out a set failure.

partitionSet.chi:

```
start -n node-1 -- --force-leader node-5
start -n node-2 -- --force-leader node-5
start -n node-3 -- --force-leader node-5
start -n node-4 -- --force-leader node-5
start -n node-5 -- --force-role LEADER
start -n node-6 -- --force-leader node-5
start -n node-7    -- --force-leader node-5
start -n node-8    -- --force-leader node-5
wait -t 2.0
fail -n node-1
create_partition -n small -p node-5,node-6,node-7
set -n node-5 -k key -v value
wait -t 1.0
remove_partition -n small --deliver
recover_node -n node-1
wait -t 3.0
get -n node-1 -k key
```

The basic idea in this script is that node-5 is the first leader, but is separated into a partition with 2 other nodes. Before that node-1 fails which prevents the larger side of the partition from being able to elect a leader while the partition exists. Then in the partition with an old leader a set request is issued. The leader propagates this request to the other two nodes in its partition, but there are not enough for it to decide that the entry is committed. Next we remove the partition and recover node-1 and roughly the same time. This allows two different possibilities depending on who wins the election. Nodes 5, 6, and 7 are unable to vote for anybody except each other because their nodes are considered more up to date than the others because of the lingering set request, however the other nodes are able to vote for anybody. If a node in 5, 6, or 7 wins, then the log changes will get propagated out immediately when the leader sends out the no-op at the beginning of their term (this can be forced by adding a small wait between removing the partition and recovering node-1). The new leader will send a success response, and the *get* will succeed. However, if any other node becomes leader, then the no-op it commits will overwrite the log entry in nodes 5, 6, and 7, causing node-5 to respond with a failure message because the command it received was overwritten. The Following *get* request will also fail. In either case the response to the *get* request matches that from the *set* request. Another rare possibility, especially considering the 3 second wait, is that there are constant split elections and the *get* is actually processed before the *set* in which case the *get* respond negatively, and the *set* will function depending

on who becomes leader. This possibility is a good example of why you should have an odd number of nodes.

failRecover.chi

```
start -n node-1 -- --force-role LEADER
start -n node-2 -- --force-leader node-1
start -n node-3 -- --force-leader node-1
start -n node-4 -- --force-leader node-1
start -n node-5 -- --force-leader node-1
start -n node-6 -- --force-leader node-1
start -n node-7 -- --force-leader node-1
start -n node-8 -- --force-leader node-1
set -n node-1 -k key -v value1
set -n node-5 -k key1 -v value2
set -n node-7 -k key2 -v value3
set -n node-5 -k key3 -v value4
set -n node-2 -k key4 -v value5
fail -n node-1
wait -t 2.0
recover -n node-1 --deliver
get -n node-1 -k key
get -n node-5 -k key1
get -n node-7 -k key2
get -n node-5 -k key3
get -n node-2 -k key4
```

failRecoverNoDelivery.chi

```
start -n node-1 -- --force-role LEADER
start -n node-2 -- --force-leader node-1
start -n node-3 -- --force-leader node-1
start -n node-4 -- --force-leader node-1
start -n node-5 -- --force-leader node-1
start -n node-6 -- --force-leader node-1
start -n node-7 -- --force-leader node-1
start -n node-8 -- --force-leader node-1
set -n node-1 -k key -v value1
set -n node-5 -k key1 -v value2
set -n node-7 -k key2 -v value3
set -n node-5 -k key3 -v value4
set -n node-2 -k key4 -v value5
fail -n node-1
wait -t 2.0
recover -n node-1
get -n node-1 -k key
get -n node-5 -k key1
get -n node-7 -k key2
get -n node-5 -k key3
get -n node-2 -k key4
```

In the example to the left (with redelivery), behavior is *nondeterministic but consistent/within the parameters of the assignment*, all the set responses will always receive a setResponse. The first couple of sets may fail or succeed depending on timing, but regardless, all *sets* will be responded to and the corresponding *gets* will return the value of the successful *sets.* The key is that since all messages are eventually delivered to node-1, it becomes aware of the sets that were forwarded to it during the window in which all the other nodes still assumed it was the leader. It starts processing those (adding them to its commandsInFlight) before realizing it is no longer the leader -- realizes it is no longer the leader and purges them from its log, sending back failed setResponses.

In the right example, these messages that were forwarded are lost to the abyss. This means that node-1 never heard about the setForwards from nodes 5, 7, and 2 (or perhaps only heard

from a subset of them), so there is no state about them stored in any node, and thus no way of sending back a setResponse to the client.

Issues, Challenges and Lessons Learned

Basic implementation took much less time than bug testing. The basic implementation of RAFT was functional about 2 days after we were done with the groundwork and actually started working on the algorithm. Getting the implementation to work through different corner cases took a lot of thought, re-examination of the source material, and effort. Considerable time went into refactoring key parts of our implementation that corresponded to assumptions that later turned out to be wrong. We should have put more work into sketching out the architecture/design with the paper in front of all three of us to try and catch these mistakes before they happened.

A major challenge we did not anticipate was determining what sort of consistency model to present with regards to errors on gets and sets. The original implementation unknowingly had the condition where a set failure didn't always mean that the set had failed because of how the fault tolerance functioned. Determining when exactly we could be sure that a set would never occur proved to be an interesting challenge, and we're still not sure that it's something we completely covered in our implementation. Having to forward sets and gets made our implementation flexible, but opened up a world of concurrency/timing problems. We wonder if we would have been better off sticking more closely to the specification and building in more deterministic leader elections for testing for this assignment.

We should have spent more time on the basics of ZMQ before we jumped in and implemented it on the project. A lot of headaches could have been saved by realizing earlier that our entire model was too simplistic and relied on many threads sharing the same socket -- a total no-no in the ZMQ world. This situation resulted in many random crashes and non-deterministic behavior of both our application and chistributed. Having to refactor it so quickly before the deadline was definitely a scary challenge and I wish we had more time to test the new changes.

Debugging was quite difficult, especially trying to establish good logging levels that didn't overwhelm us but still provided enough information to find out where problems existed. A lot of the time, one has to turn on --debug mode and stare at the output like you're in the Matrix and make deductions based on brief flashes of different log colors.