* 20 points: Summary of fault-tolerance algorithm/protocol used in your project, citing all sources you consulted.

Basic algorithm is RAFT. The main modification we have made is that there

* 35 points: Description of your implementation. This must include, at least, a description of how *set* requests are processed, how *get* requests are processed, and the specific fault tolerance features supported by your project. Note that this is distinct from the README file included in your implementation, which should be a to-the-point (and not necessarily prosaic) summary of how your implementation is structured, to help us navigate the files and main data structures of your implementation.

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. 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:

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.

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.

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.

* 30 points: Discussion of at least two example chistributed scripts (with failures) and an explanation of how your implementation reacts in the presence of those failures.
* 15 points: Discussion of the issues, challenges, and lessons learned while implementing your data store.

Basic implementation took much less time than bug testing