Marc Beitchman
CSE P 552
Assignment 2
Two Phase Commit Data Store Implementation Report

My code is structured into 3 main components: client, replica and master. The client connects to the master and allows a user to issue commands against the 2-phase commit cluster. The client has automated tests, which spins up the master, and the replicas, performs and verifies operations, and shuts down the cluster. The client also has an interactive mode, which takes input commands from the user, parses them and issues the appropriate commands to the master.

The master connects to the replicas and spins up its RPC server. The master's RPC server is implemented in a class called MasterRPCInterface implemented in MasterRPCInterface.rb. The master's RPC interface exposes the get, put and delete methods to the client. The master recovery code is implemented in a class called MasterRecover. This class exists in MasterHelper.rb.

The replica is structured similar to the master. The replicas's RPC server is implemented in a class called ReplicaRPCInteface in ReplicaRPCInterface.rb. The replica recovery code is implemented in a class called ReplicaRecover. This class is implemented in ReplicaHelper.rb.

The data store is implemented in class called KeyValueStore in the DataStore module in DataStore,rb. This class handles data store initialization as well as primitive data store operations. Each replica contains an instance of the KeyValueStore class.

The replicas RPC interface exposed to the master includes the following methods: vote_request_put, vote_request_delete, commit and get. Get requests are handled by asking a random replica for the value and are not handled as a two-phase commit transaction. I defined the replica interface based on the definition of the rounds of two-phase commit. Therefore, the master calls vote-request on a replica to introduce the transaction to the replica and get a result of whether the replica would accept the value. The master can then call commit on the replicas if all responses are successful from all replicas. The master stores replica id's internally and provides these id's to the replicas so they can keep track of transactions they've seen and responded to.

There are a few ways I detect failures. The first is I defined that keys in the data store must be unique. Therefore my implementation will reject put requests for an existing key. I also detect basic failures for delete requests to make sure a key exists for the request key to delete. I also detect failures for multiple concurrent requests for a key by storing the key of the active transaction in a data structure so I can detect key conflicts. The main way I detect communication failures is through timeouts.

The master RPC interface returns true/false values to the client to indicate success or failure. Special error message are returned to the client in the case of an unrecoverable failure such as a replica timeout during the commit phase.

I explored many test cases trying to cover as many corner cases as possible. The following scenarios were tested:

- basic functionality
- timeouts in replicas in the vote phase and commit phase by adding sleeps in the replica in order to force the timeout to occur
- recovery of the replicas during vote and commit phases
- master going down in vote and commit phase to ensure recovery was implemented correctly
- concurrency by spinning up multiple clients and launching conflicting requests simultaneously

I was able to automate testing for correct functionality and replica timeout during the vote-phase. I was planning to automate all of the scenarios mentioned previously but was not able to complete this due to time constraints.