CS 162 Project 3: Key-Value Store

April 22, 2015

Contents

1 Overview **2**

2 Code Style and Commit Messages **2**

3 Getting Started **3**

4 Checkpoint 1 **4**
- 4.1 Overview . . . 4
- 4.2 KVClient . . . 4
- 4.3 KVMessage . . . 5
- 4.4 KVConstants . . . 5
- 4.5 KVCache . . . 6
- 4.6 KVCacheSet . . . 6
- 4.7 KVStore . . . 6
- 4.8 Work Queue . . . 7
- 4.9 Socket Server . . . 7
- 4.10 UTHash, UTList . . . 7
- 4.11 KVServer . . . 8
- 4.12 Tests . . . 8
- 4.13 Deliverables . . . 9
  - 4.13.1 Design Document . . . 9
  - 4.13.2 Design Review . . . 9
  - 4.13.3 Functionality . . . 9
  - 4.13.4 Tests . . . 9
  - 4.13.5 Scoring . . . 9

5 Final Code and Report Handin **10**
- 5.1 Overview . . . 10
- 5.2 Slave Server Registration . . . 12
- 5.3 Consistent Hashing . . . 12
- 5.4 Two-Phase Commit . . . 13
- 5.5 Tests . . . 13
- 5.6 Final Report . . . 13
- 5.7 Scoring . . . 14
- 5.8 Bonus . . . 14

6 Closing Comments **14**

1 Overview

In this project, you'll be implementing a distributed key-value store; essentially a database with three simple operations: get, put, and delete. Data will be replicated across multiple slave servers to ensure data integrity, and actions will be coordinated across these servers by a master server. Two-phase commit logic will be used to ensure consistency across slave servers, and a persistent log will be maintained at each server to be able to recreate its state upon a crash, which you must be able to handle gracefully (i.e. still maintain consistency when a server recovers from a crash).

We divide this project into two checkpoints, each described separately with necessary background and checkpoint requirements. Every checkpoint is based on the understanding and implementation of former checkpoints.

2 Code Style and Commit Messages

10 points of your final checkpoint 2 grade is "Efficient Design + Clean Code". Your GitHub commit messages and git hygiene are part of this criterion. **You should write your commit messages professionally, and you should never commit binaries/temporary files to GitHub, or you may lose a point here.**

3 Getting Started

The skeleton we've added for project 3 is available in the group0 github repo. If you're using Vagrant, you can pull the skeleton by running the following command:

```
cd ~/code/group
git checkout master
git pull staff master
cd kvstore
make
```

If this doesn't work for you, double-check that the group0 repo is setup as a tracked repository:

```
git remote -v
```

You should see staff linked to the group0 repo. If not, add the group0 repo and try again:

```
git remote add staff git@github.com:Berkeley-CS162/group0.git
```

If these commands don't work for you or you aren't sure how to setup git, please try the following (most preferred solution first):

1. Read the spec
2. Read the spec again
3. Look on Google
4. Ask a classmate
5. Ask a GSI
6. Post a question on Piazza

4 Checkpoint 1

This checkpoint is due **Wednesday 4/29 11:59:59 PM**. Submit your code to the **release/proj3/checkpoint1** branch.

4.1 Overview

Initially, you will build a key-value storage system complete with code for a central key-value server (with caching), client-side libraries, and all necessary networking components.

Multiple clients will be able to communicate with a single key-value server in a given messaging format (JSON, encoded by KVMessages). Communication between the clients and the server will take place over the network through sockets. The KVServer uses a work queue (wq) to support concurrent operations across multiple sets in a set-associative KVCache, which is backed by a KVStore. This checkpoint should be relatively straightforward, but it is IMPERATIVE you get it correct in order to be able to succeed in checkpoints 2. In later parts of the project, you will extend your single node key-value server to a distributed key-value store that runs across multiple nodes (KVServers) coordinated by a single master (TPCMaster).

For all of the files below, their header files have similar information to what is here, as well as comments on the structs which are defined there. You will want to look through these as well as reading through the spec. Functions and their usages are heavily commented within the .c files.

As you work on implementing the functionality described, you **may not** change the function signatures of any of the functions which have been provided in the skeleton code, remove or change any fields from the provided structs, change the names of any existing `typedef`, etc. You may, however, add functions, structs, and fields to existing structs as you wish. We have added comments to help guide you to where you should be filling in code.

Please note that Checkpoint 2 is **much** longer and more difficult than Checkpoint 1, so we suggest getting a head start on it if you manage to finish Checkpoint 1 early.

4.2 KVClient

For this project, we have provided an interactive client written in Python that can be used to submit requests to the server. The types of valid requests are GET, PUT, DEL, and INFO. The client will connect to your server, submit the serialized request in JSON format, and parse the response. Note that there is nothing special about this implementation: you could build a compatible client in any language, as long as you conform to the server message format.

To use the client, run `make` and `cd` into the bin directory. You will need to start up a server, then run the client from a separate shell window. First, start the server using:

```
./kvslave
```

Then run the client using:

```
./interactive_client
```

For both of these commands, you can configure what port they run on by running, for example:

```
./kvslave 8162
./interactive_client -p 8162
```

Once in the interactive client shell, you can submit requests to the server:

```
get("key")
put("key", "value")
delete("key")
info() // returns information about the server
quit() // exits the interactive client shell
```

For now, all of these commands will return an error message that they have not been implemented.

4.3 KVMessage

KVMessage is used to send messages (requests, responses) across sockets. This is how clients will communicate with a server, and later, how a TPC Master will communicate with its slave servers.

`kvmessage_send` first sends the size of the entire message in the first four bytes. The following bytes contain the contents of the message in JSON format. Only fields which are non-null in the passed in message are sent.

`kvmessage_parse` reads the first four bytes of the message, uses this to determine the size of the remainder of the message, then parses the remainder of the message as JSON and populates whichever fields of the message are present in the incoming JSON.

The types and formats for KVMessages are as follows. It is up to you to correctly populate and read from messages according to their type.

GET Request	Type: `GETREQ` Key: *key*
PUT Request	Type: `PUTREQ` Key: *key* Value: *value*
DEL Request	Type: `DELREQ` Key: *key*
Successful GET	Type: `GETRESP` Key: *key* Value: *value*
Successful PUT/DEL	Type: `RESP` Message: `MSG_SUCCESS` (“SUCCESS”)
Unsuccessful Request	Type: `RESP` Message: *error message*
Server Information	Type: `INFO`

This functionality has been fully implemented for you.

4.4 KVConstants

This file defines some of the constants you will use throughout the project. You should familiarize yourself with all of them, as you will be using them extensively throughout the project. Whenever you are returning an error, if there is a constant which corresponds to that error, be sure to use that constant, otherwise you may use -1 (when returning an integer status) or `ERRMSG_GENERIC_ERROR` (when returning a string error, i.e. in a KVMessage).

4.5 KVCache

KVCache defines the in-memory cache which is used by KVServers to quickly access data without going to disk. It is set-associative.

A KVCache maintains a list of KVCacheSets, each of which represents a subset of entries in the cache. Each KVCacheSet maintains separate data structures; thus, entries in different cache sets can be accessed/modified concurrently. However, entries in the same cache set must be modified sequentially. This can be achieved using a read-write lock maintained by each cache set. The lock should be acquired/released from whoever will be calling the cache methods (i.e. KVServer and, later, TPCMaster).

The cache uses a second-chance replacement policy implemented within each cache set. You can think of this as a FIFO queue, where the entry that has been in the cache the longest is evicted, except that it will receive a second chance if it has been accessed while it has been in the cache. Each entry should maintain a reference bit, initially set to false. When an entry is accessed (GET or PUT), its reference bit is set to true. When an entry needs to be evicted, iterate through the list of entries starting at the front of the queue. Once an entry with a reference bit of false is reached, evict that entry. If an entry with a reference bit of true is seen, set its reference bit to false, and move it to the back of the queue.

This file has been mostly implemented for you, since most of its logic is simply calling out to the correct KVCacheSet. You will only need to implement `get_cache_set`, which determines which cache set to use for a given key.

4.6 KVCacheSet

KVCacheSet represents a single distinct set of elements within a KVCache.

Elements within a KVCacheSet may not be accessed/modified concurrently. The read-write lock within the KVCacheSet struct should be used to enforce this. The lock should be acquired/released from whoever will be calling the cache methods (i.e. KVServer and, later, TPCMaster).

This has been left almost completely unimplemented.

4.7 KVStore

KVStore defines the persistent storage used by a server to store (key, value) entries.

Each entry is stored as an individual file, all collected within the directory name which is passed in upon initialization. If you are running multiple KVStores, they MUST have unique directory names, else behavior is undefined.

The files which store entries are simple binary dumps of a `kventry_t` struct. Note that this means entry files are NOT portable, and results will vary if an entry created on one machine is accessed on another machine, or even by a program compiled by a different compiler. The LENGTH field of `kventry_t` is used to determine how large an entry and its associated file are.

The name of the file that stores an entry is determined by the djb2 string hash of the entry's key, which can be found using the hash() function. To resolve collisions, hash chaining is used, thus the file names of entries within the store directory should have the format: `hash(key)-chainpos.entry` or `sprintf(filename, "%lu-%u.entry", hash(key), chainpos);`. `chainpos` represents the entry's position within its hash chain, which should start from 0. If a collision is found when storing an entry, the new entry will have a chainpos of 1, and so on. Chains should always be complete; that is, you may never have

a chain which has entries with a chainpos of 0 and 2 but not 1.

All state is stored in persistent file storage, so it is valid to initialize a KVStore using a directory name which was previously used for a KVStore, and the new store will be an exact clone of the old store.

This functionality has been fully implemented for you.

4.8 Work Queue

WQ defines a work queue which will be used to store jobs which are waiting to be processed.

WQ will contain whatever synchronization primitives are necessary to allow any number of threads to be waiting for items to fill the work queue. For each item added to the queue, exactly one thread should receive the item. When the queue is empty, there should be no busy waiting.

As given, WQ is just a linked-list of items with no synchronization. It is your job to modify `wq_push` and `wq_pop` to create a synchronized queue.

4.9 Socket Server

Socket Server defines helper functions for communicating over sockets.

`connect_to` can be used to make a request to a listening host. You will not need to modify this, but you will likely want to utilize it.

`server_run` can be used to start a server (containing a KVServer, or, later, a TPC Master) listening on a given port. It will run the given server such that it indefinitely (until `server_stop` is called) listens for incoming requests at a given host and port. A callback function can also be passed in, which is used only for testing purposes. It is called with a `NULL` parameter immediately after the server is actively listening for requests, to notify a test when it is safe to begin submitting requests to the server. You will want to use this in your end-to-end tests.

As given, `server_run` will synchronously handle only a single request at a time. It is your task to modify it such that it can handle up to the specified max number of requests at a time asynchronously. You will likely want to use a work queue.

The server struct stores extra information on top of the stored TPCMaster or KVServer for use by `server_run`.

4.10 UTHash, UTList

UTHash and UTList are two header-only libraries which have been supplied for your use. They help to make creating linked-lists and hash tables in C easy.

For UTHash, a simple pointer of the type of the struct you will be storing in the table will represent the table itself. The structs you insert into the hash table must have a member of type `UT_hash_handle` with name `hh`, and of course you'll want a field to use as a key as well. The macros you will probably find most useful are `HASH_ADD_STR`, `HASH_FIND_STR`, and `HASH_DEL`. You can read more about their usage at `https://troydhanson.github.io/uthash/`.

For UTList, again a simple pointer of the type of the struct you will be storing in the list will represent the list itself. The structs you insert into the list, to keep a doubly-linked list (which we recommend), must

have `prev` and `next` pointers of the same type as the structs you are storing. The macros you will probably find most useful are `DL_APPEND`, `DL_DELETE`, and `DL_FOREACH_SAFE`. You can read more about their usage at `https://troydhanson.github.io/uthash/utlist.html`.

4.11 KVServer

KVServer defines a server which will be used to store (key, value) pairs.

Ideally, each KVServer would be running on its own machine with its own file storage.

A KVServer accepts incoming messages on a socket using the message format described previously, and responds accordingly on the same socket. There is one generic entrypoint, `kvserver_handle`, which takes in a socket that has already been connected to a master or client and handles all further communication. We have provided the topmost level of this logic; you will need to fill in `kvserver_handle_no_tpc`, which takes in a request message and fills a provided response message appropriately.

A KVServer has an associated KVStore and KVCache. The server should attempt to get an entry from cache before accessing its store to eliminate the need to access disk when possible. The cache should write-through; that is, when a new entry is stored, it should be written to both the cache and the store immediately.

A KVServer can operate in two modes; TPC or non-TPC. In non-TPC mode, all PUT and DEL requests go immediately to the cache/store. This is the only mode you will implement for this checkpoint.

Because the KVStore stores all data in persistent file storage, a non-TPC KVServer can be reinitialized using a DIRNAME which contains a previous KVServer and all old entries will be available, enabling easy crash recovery.

A KVServer also has an associated TPCLog, but this is only used in TPC mode, and you will not need to worry about it for this checkpoint. For this checkpoint, you will implement all functions in KVServer except for `kvserver_handle_tpc`, `kvserver_register_master`, and `kvserver_rebuild_state`.

Finally, a KVServer should be able to service INFO requests, responding with a message formatted as such:

```
TIMESTAMP (format of your choosing)
{HOSTNAME, PORT}
```

where each newline is a simple `\n` character.

4.12 Tests

For this project, we will be using the same testing framework as Homework 4, which you are hopefully familiar with by now. To build the testing executable, use `make test` in the `kvstore` directory. This will create the file `kvtests` in the `bin` directory. You can run all of the tests for checkpoint 1 by running `./kvtests checkpoint1` from within the `bin` directory, or you can run a single test suite (i.e., only KV-Cache tests) by running `./kvtests suite-name`. Running `make check1` from the `kvstore` directory will automatically build the test executable and run `kvtests checkpoint1` for you.

We have provided some tests, but they are far from an exhaustive test suite, and we also have numerous tests which we have not released to you, unlike previous projects. You will be graded on your performance both on the tests provided and the tests not provided, so it is strongly suggested that you write many additional tests to ensure that you have implemented all functionality correctly. You are required to write

at least one end-to-end test and a minimum of three unit tests. You can view an example of how to write an end-to-end test in `tests/endtoend_test.c`.

4.13 Deliverables

4.13.1 Design Document

The design template for this project located on Piazza will contain specific questions designed to get you thinking and reading about the code in this project. It will also include space for you to detail how your group is planning to implement the functionality for the next checkpoint. Please attach your design document in an email to your GSI in similar fashion to the previous projects.

4.13.2 Design Review

Setup a 30-minute meeting with your GSI. You should be able to answer questions about the project.

4.13.3 Functionality

After this checkpoint, your group should have a working key-value storage server. It should be able to receive connections from one or more clients, accept requests and deliver success or failure responses, and handle errors gracefully.

4.13.4 Tests

The following test suites should pass for Checkpoint 1, as well as the tests you write. Don't forget to add, at a minimum, one end-to-end test and three unit tests. You can run them all using `make check1` or `./kvtests checkpoint1`:

```
kvcacheset
kvcache
kvserver
endtoend
wq
socket_server
```

4.13.5 Scoring

Design Document: 10 points
Design Review: 10 points
Simple Node Key Value Server: 10 points

5 Final Code and Report Handin

This checkpoint is due **Friday 5/6 11:59:59 PM**. Submit your code to the **release/proj3/checkpoint2** branch. The final report is due **Friday 5/8 11:59:59 PM**. If you decide to use slip-days, please include the number of slip-days you are using in the final report submission (we will verify this number against your group's total as well as your code submission).

5.1 Overview

In this part of the project, you will extend your single node key-value server to a distributed key-value store that runs across multiple nodes (KVServers) coordinated by a single master (TPCMaster).
Multiple clients will be communicating with a single master server in a given messaging format (JSON, encoded by KVMessages). The master contains a set-associative cache (KVCache), and it uses the cache to serve GET requests without going to the key-value (slave) servers it coordinates. The slave servers are contacted for a GET only upon a cache miss on the master. The master will use the TPCMaster library to forward client requests for PUT and DEL to multiple slave servers and follow the TPC protocol for atomic PUT and DEL operations across multiple slave servers. Operations on the store should be atomic (succeeds completely or fail altogether) without any side effects, guaranteed by the TPC protocol.
Data storage is durable, i.e. the system does not lose data if a single node fails. You will use replication for fault tolerance.
The key-value store should operate in SERIAL on elements of the same cache set (determined by the KVCache of the master) but should allow for CONCURRENT gets across different sets. Note that TPC operations (PUT and DEL) are always in serial.

You can run a TPC Master using the following command from within the bin directory:

```
./kvmaster port  (default = 8888)
```

You can start up a slave server in TPC mode using the following command from within the bin directory:

```
./kvslave -t slave-port master-port
```

`slave-port` is the port at which the slave will listen for requests from the master (must be unique for each slave), and `master-port` is the port at which the master is listening.

You can start up a full set of slaves and a master using the following commands:

```
./kvmaster &
./kvslave -t 9001 &
./kvslave -t 9002 &
```

Remember that using an `&` allows a program to run in the background so that you can run multiple at once. To simplify this, we have created a script which runs the above command, as well as adding a bit of cleanup that will kill all three processes when you hit ctrl-c. You can run this from the bin directory:

```
./tpc_server
```

Again, you can use the interactive client (described in checkpoint 1) to hit the master with requests. Make sure you configure the interactive client to use the correct port (this assumes the default master port):

```
./interactive_client -p 8888
```

Slave Registration. Implement registration logic in `tpcmaster.c` and `kvserver.c`. In the former, you will need `tpcmaster_register`, `tpcmaster_get_primary` and `tpcmaster_get_successor`. In the latter, `kvserver_register_master` should be sufficient.

Logs and Slave Durability Implement proper logic for logging the state of slave servers and for rebuilding from the log after unexpected slave server termination. A KVServer should log every action it receives that is relevant to TPC (PUT, DEL, ABORT, COMMIT) immediately upon receiving the request. You can add entries into the log using `tpclog_log`, which will make the entry persistent on disk (and thus durable through crashes). When a server starts back up after crashing, it should be able to recover the state of any TPC transaction that was currently open when it crashed. Because the store is also durable, fully committed TPC transactions do not need to be recovered from the log. You may find the functions `tpclog_iterate_begin`, `tpclog_iterate_has_next`, and `tpclog_iterate_next` useful, which help you to iterate over all entries of the log and determine what state the server should be in. You can also use `tpclog_clear_log` to clear the log of entries when you are certain that they will not be needed at a later time (e.g. all previous transactions have been fully committed). The logic to rebuild a server from its log will be implemented in `kvserver_rebuild_state`.

Master GET handling. Implement `tpcmaster_handle_get` in `tpcmaster.c` to correctly handle GET requests. GET requests from the client are serial within the same set but parallel across different sets. Sets are defined by the KVCache of the Master. You do not need to implement two-phase commit logic (Note that GET is not a TPC command). If the key requested by the client is resident in the cache, return it and do NOT proceed to forward the request to the slave.

Master TPC handling. Implement `tpcmaster_handle_tpc` in `tpcmaster.c` to correctly handle PUT and DEL requests. Refer to the section on Two-Phase Commit for details on the implementation of two-phase commit logic. If the slaves vote for a commit, make sure to update the master cache accordingly.

Slave TPC handling. Implement `kvserver_handle_tpc` in `kvserver.c` to correctly handle requests coming from a master server, as opposed to those coming directly from a client. This is where you will implement the slave-side logic for two-phase commit.

Info Messages. Implement a handler for INFO requests in `tpcmaster.c`. The master should relay the request to each of its slaves, waiting for each response for a maximum of 2 seconds. If a slave responds, append its info to the response; else, ignore it and move on to the next one. The point of this functionality is to give the client an insight on what slave server are currently alive. The format of the response should be different from that in `kvserver.c`, looking instead like this:

```
TIMESTAMP (format of your choosing)
Slaves:
{HOSTNAME1, PORT3}
{HOSTNAME2, PORT3}
{HOSTNAME3, PORT3}
```

where each newline is a simple `\n` character. There should be one bracketed line per slave. In the example, three slaves responded to the ping.

5.2 Slave Server Registration

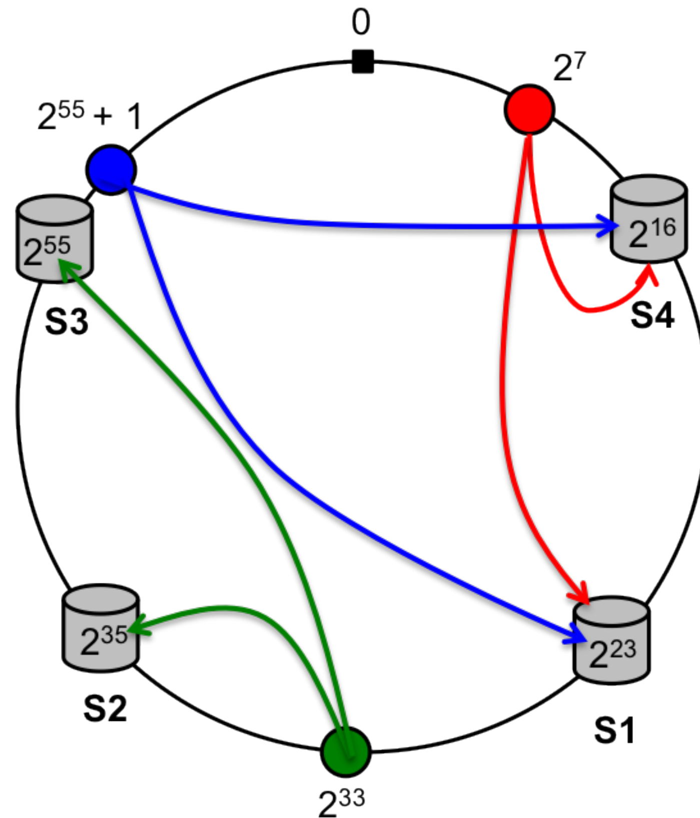
The master will have to keep the information regarding its slaves in a list of `tpc_slave_ts`. Each slave server will have 64-bit globally unique IDs (`int64_t`) assigned to them by the master as they register using their hostname and port number. For simplicity, you can assume that the total number of slave servers is fixed. There will not be any slaves that magically appear after registration completes, and any slave that dies will revive itself. A slave has successfully registered if it receives a successful response from the master.

The master will listen for registration requests on the same port that it listens for client requests on. When a slave starts, it should start listening on its given port for TPC requests and register that port number with the master so that the master can send requests to it. This is already handled with main/kvslave.c.

5.3 Consistent Hashing

Each key will be stored using TPC in $n \leq N$ slave servers (where N is the total number of slaves, and n is defined upon initialization of the master as `master->redundancy`); the first of them will be selected using consistent hashing, the second will be the successor of the first one, and so on. Note that the hash function is provided for you in `tpcmaster.c`.

There will be at least two slave servers in the system. Each key-value (slave) server will have a unique 64-bit ID. The master will hash the keys to 64-bit address space. Each slave server will store the first copies of keys with hash values greater than the ID of its immediate predecessor up to its own ID. Note that each slave server will also store the keys whose first copies are stored in its predecessor (and possibly more, based on the chosen redundancy index). In the following image, the redundancy index is 2.



5.4 Two-Phase Commit

TPCMaster must select replica locations using consistent hashing. Only a single two-phase commit operation (PUT, DEL) can be executed at a time. You do not have to support concurrent update operations across different keys (i.e. TPC PUT and DEL operations are performed one after another), but retrieval operations (i.e. GET) of different keys must be concurrent unless restricted by an ongoing update operation on the same set.

The master will include a set-associative cache, which will have the same semantics as the cache you used before. If the master finds the key in its cache, it must not contact any slaves.

You must wait until all slaves register before servicing any requests. This means you must drop any requests you receive until the master's `slave_count` reaches its `slave_capacity`. When sending phase-1 requests, the master must contact all (relevant) slaves, even if one of the slaves sends an abort.

A slave will send a `VOTE_ABORT` to the master if the key doesn't exist for DEL or an oversized key/value is specified for PUT, and a `VOTE_COMMIT` otherwise. If the master doesn't receive a response before timing out, the slave's should be counted as a `VOTE_ABORT`. If there is no COMMIT consensus (that is, if there at least one `VOTE_ABORT`), the master must send an ABORT command in phase-2. Otherwise, the command will be to COMMIT.

If the master receives any response from a slave in phase-2, it should be an ACK. If no response is received (timeout), the master must keep sending its phase-2 message to the slave, which upon recovery will be able to send an ACK back.

The function used to handle this logic, `tpcmaster_handle_tpc`, takes in a `callback` parameter which is used to synchronize tests. It is imperative that you place calls to this function within your logic exactly as described in the comment above `tpcmaster_handle_tpc`.

The types and formats of KVMessages unique to TPC are as follows. It is up to you to correctly populate and read from messages according to their type.

Registration request	Type: `REGISTER` Key: *hostname* Value: *port*
COMMIT vote	Type: `VOTE_COMMIT`
ABORT vote	Type: `VOTE_ABORT`
COMMIT command	Type: `COMMIT`
ABORT command	Type: `ABORT`
ACK from slave	Type: `ACK`

5.5 Tests

All of the provided key-value store tests should pass for this checkpoint. We will weight each of these tests equally in determining your final score. As with the previous checkpoint, you will need to write a significant amount of your own tests, including both unit tests and end-to-end tests (see `endtoend_tpc_test` for an example). You can run all of the tests for this checkpoint by running `make check` in the kvstore directory.

5.6 Final Report

For your final report, please email your GSI with a polished report describing your implementation of the project, following the template in the initial design. **Please create a new section in this document**

called "Final Design Changes" detailing the major changes to your design and why you changed them.

5.7 Scoring

- Checkpoint2 (hidden + public test cases) - 50 points
- Efficient Design + Clean Code - 10 points
- Final Design Document - 10 points

5.8 Bonus

Homeworks 1,2,3,4 and Project 3 are all user space programs. It is possible to create an application that combines all the assignments. How you do so is up to you. If you can present an application that sucessfully and correctly combines the functionality of the above assignments, you may receive up to 5 points of extra credit. Please push your bonus application to the branch release/proj3/bonus, and email the staff a short description of your application and a readme (if applicable)

6 Closing Comments

Congratulations! You have completed your last assignment in CS162, hopefully you had a fun and educational time here. Good luck on the final!