HW 4: Two Phase Commit

Due December 4 2015

Contents

1 Overview

In this homework, you will be building a distributed key-value store with three operations: get, put, and delete. Data will be replicated across multiple follower servers to ensure data integrity, and actions will be coordinated across these servers via a single leader server.

Your job is to implement (a) two-phase commit, which will be used to ensure consistency across follower servers, and (b) logging to save a follower server's data to a persistent log, as well as the ability to restore from the log after a crash. An implementation of a persistent log and a non-distributed key-value store has been provided for you.

To get started, `ssh` into to your Vagrant VM and run:

```
cd ~/code/personal/
git pull staff master
cd hw4/
```

2 API

Your key-value system will provide an HTTP API to the client (i.e. the user). Internally, these API requests and responses are unmarshalled into the C types `kvrequest_t` and `kvresponse_t` respectively. Below are tables that summarize the API. Refer to Section 4.4 for advice on how to use it.

Table 1: **API Requests**

Request	**HTTP Request**	**KVRequest**
Get	GET `/?key=<key>`	type: `GETREQ` key: *key*
Put	PUT `/?key=<key>&val=<value>`	type: `PUTREQ` key: *key* val: *value*
Delete	DELETE `/?key=<key>`	type: `DELREQ` key: *key*
Commit Transaction	POST `/commit`	type: `COMMIT`
Abort Transaction	POST `/abort`	type: `ABORT`
Register Follower	POST `/register?key=<addr>&val=<port>`	type: `REGISTER` key: *follower address* val: *follower port*

Table 2: **API Responses**

Response	**HTTP Code**	**KVResponse**
Successful Get	200	type: `GETRESP` body: *val*
Successful PUT/DEL	201	type: `SUCCESS`
Vote Commit/Abort	202	type: `VOTE` body: "commit"/"error: *message*"
ACK	204	type: `ACK`
Error	500	type: `ERROR` body: "error: *message*"

3 Existing Code

This homework starts with a rather large codebase. The following subsections provide a detailed run-through of what you'll likely be working with.

In addition to reading this spec, we expect you to read through and understand the header (`.h`) files for each section mentioned below. Struct definitions and functions are thoroughly commented within the `.h` and `.c` files, respectively. The Appendix contains detailed information about the other files not mentioned here.

3.1 KVMessage

`kvmessage.h/c` detail two message types: `kvrequest_t` and `kvresponse_t`. These types encapsulate API calls to our key-value store. They provide a convenient abstraction for the actual HTTP messages we recieve across our sockets, but are only useful internally.

- `kvrequest_send/kvresponse_send`
  Marshalls the kvrequest/kvresponse into an HTTP message according to our API, then sends it over the specified socket file descriptor.
- `kvrequest_recieve/kvresponse_recieve`
  Recieves the HTTP message from the specified socket, unmarshalls and returns the message as a kvrequest/kvresponse.

The formats for KVRequests and KVResponses are summarized in the API section above. It is up to you to correctly populate and handle messages according to their type. (Fortunately, we've abstracted most of the nitty-gritty HTTP and URL related things with the libraries mentioned in the appendix.)

3.2 KVConstants

This file defines some of the constants you will use throughout the homework. You should familiarize yourself with all of them, as you will be using them extensively throughout the homework. In particular, we've written a couple of convenient macros that you should definitely use.

- `alloc_msg(buf, msg)` will stuff `msg` into a `malloc`'d buffer and assign the specified `buf` pointer to that `malloc`'d buffer. This should come in very handy when dealing with allocing/deallocing KVMessages throughout your program—specifically, if you assign a KVMessage field to a not-`malloc`'d, static string and later attempt to free that field, you will trip a `SIGABRT`; this macro can be used to prevent said issue.
- `fatal(msg, code)` and `fatal_malloc()` macros will fatally exit your program with a stack trace upon execution. The latter is a convenient shorthand for exiting your program if `malloc` fails.

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 reporting a string error, i.e. in a KVResponse).

3.3 KVServer

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

In a real-world scenario, each KVServer would be running on its own machine with its own file storage. Your final implementation should be able to support this.

A KVServer accepts incoming HTTP messages on a socket using the API described before, 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 leader or client and handles all further communication. We have provided the topmost level of this logic; you will need to fill in `kvserver_handle_tpc`, which takes in a KVRequest, fills a provided KVResponse appropriately, and logs any necessary TPC transaction steps.

You will also have to implement `kvserver_rebuild_state`, which rebuilds a follower server's state from a persistent log after a crash.

3.4 TPCLeader

TPCLeader defines a leader server which will coordinate two-phase commit logic among registered followers.

The TPCLeader will become the main point of contact for the client. In the final product, all API requests from the client are sent to the TPCLeader. Similarly to KVServer, there is one generic entrypoint at `tpcleader_handle`, which takes in a connected socket and handles all further communication.

You will need to implement `tpcleader_handle_get` and `tpcleader_handle_tpc`. These two functions relay API requests to the appropriate followers. The TPCLeader maps keys to followers according to a consistent hashing scheme, described in a Section 4.2 below.

4 Your Assignment

Your job is to implement two-phase commit in this distributed key-value store, so that multiple nodes (KVServers) are coordinated by a single leader (TPCLeader). You will also implement logging on the KVServers so that they can recover from crashes.

As you work on implementing the functionality described, you **should not** change the function signatures of any of the functions which have been provided in the skeleton code, and you **should not** remove or change any fields from the provided structs. You may, however, add as many additional functions, structs, and additional fields to existing structs as you wish. We have added comments to suggest places where you should be writing code.

Multiple clients will be communicating with a single leader server according to the given HTTP API. The leader will forward client GET, PUT, and DELETE requests to multiple follower servers and follow the TPC protocol to perform atomic operations across multiple follower servers. A basic non-distributed thread-safe key-value data store has been implemented for you. Operations on this key-value store are guaranteed to be atomic.

Data storage must be redundant, i.e. the system should not lose data if only a single KVServer fails. You will use simple replication for fault tolerance.

The provided Makefile compiles binaries to the `bin` directory. You can run a TPCLeader using the following command:

```
./bin/tpcleader [port (default=16200)] [followers (default=1)] [redundancy (default=1)]
```

You can start up a follower server using the following command:

```
./bin/kvfollower [follower_port (default=16201)] [leader_port (default=16200)]
```

`follower_port` is the port on which the follower will listen for requests from the leader (must be unique for each follower), and `leader_port` is the port on which the leader is listening.

You can start up a full set of followers and a leader using the following commands:

```
./bin/tpcleader 16200 2 2 &
./bin/kvfollower 16201 16200 &
./bin/kvfollower 16202 16200 &
```

Using an `&` allows a program to run in the background so that you can run multiple programs simultaneously. To simplify this, we have created a script which runs all 3 above commands and also kills all 3 processes when you type `Ctrl-C`:

```
./bin/kvsystem
```

4.1 Tasks

Follower TPC handling. Implement `kvserver_handle_tpc` in `kvserver.c` to correctly handle two-phase PUT and DELETE requests.

Leader GET handling. Implement `tpcleader_handle_get` in `tpcleader.c` to correctly handle GET requests. Refer to Section 4.2 on Consistent Hashing to determine which followers to communicate with.

Leader TPC handling. Implement `tpcleader_handle_tpc` in `tpcleader.c` to correctly handle PUT and DELETE requests. Refer to Section 4.3 on Two-Phase Commit for details on the implementation of two-phase commit logic. Refer to Section 4.2 on Consistent Hashing to determine which followers to communicate with. You do not need to support the concurrent execution of multiple TPC transactions (they can occur serially).

Logs and Follower Durability Implement proper logic for logging the state of follower servers and for rebuilding from the log after a follower server crashes and recovers. The logic to rebuild a server from its log will be implemented in `kvserver_rebuild_state`.

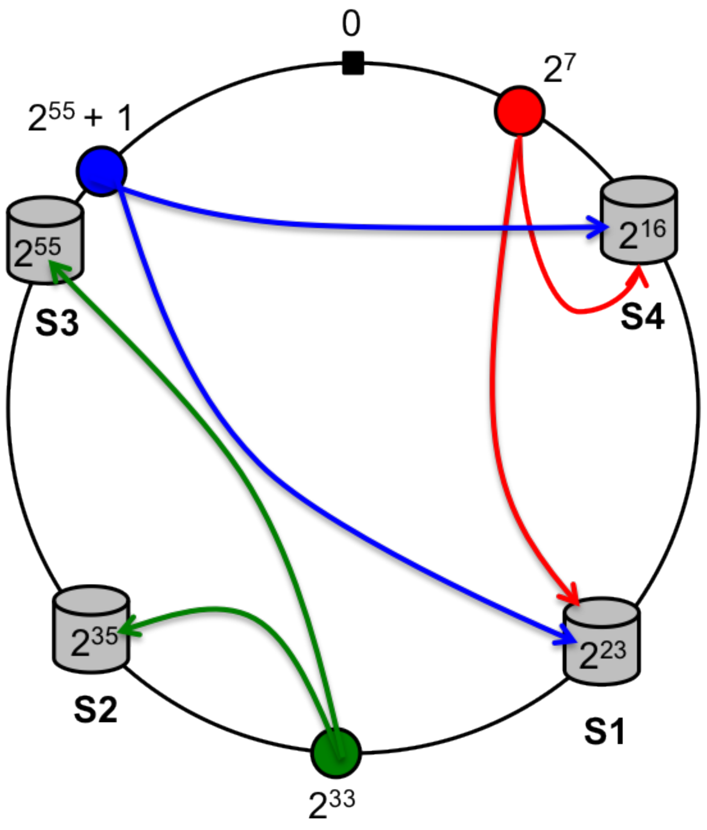
A KVServer should log every action it receives that is relevant to TPC *immediately upon receiving the request*. You can add entries into the log using `tpclog_log`, which will make the entry persistent on disk (and thus durable through crashes). When a server starts back up after crashing, it should be able to recover the state of any TPC transaction that was currently occurring when it crashed. 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.

Because the store is also durable, fully committed TPC transactions do not need to be recovered from the log. You should 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).

4.2 Consistent Hashing

Each key will be stored in r follower servers (where r is less than the total number of followers N). The value of r is available as a member of a TPCLeader (`leader->redundancy`). The first follower, a.k.a. the "primary", will be selected using consistent hashing. The second will be the successor of the primary, and so on. Note that the hash function is provided for you in `kvconstants.h`. *You may not change the hash function.*

Each key-value (follower) server will have a unique 64-bit ID. The leader will hash keys to the same 64-bit address space, and choose the follower with the nearest, higher ID than the key hash as the primary, wrapping around in the case of overflow. The successors are chosen contiguously after the primary. Please refer to the following image for a better explanation. In the following image, the redundancy index is 2. Each circle corresponds to the hash value of a particular key. Each cylinder corresponds to the hash value of a KVServer.



4.3 Two-Phase Commit

Only a single two-phase commit operation (PUT, DELETE) can be executed at a time. You do not have to support concurrent update operations across different keys (i.e. TPC PUT and DELETE operations are performed one after another), but retrieval operations (i.e. GET) of different keys must be concurrent.

When sending Phase-1 requests, the leader must contact all (relevant) followers for PUT and DELETE, even if one of the followers sends an abort. The leader can access the primary and successive followers with the following functions, respectively: `tpcleader_get_primary`, `tpcleader_get_successor`.

A follower will send an "error: ..." `VOTE` to the leader if the key doesn't exist for DELETE or an oversized key/value is specified for PUT, and a "commit" `VOTE` otherwise. If the leader doesn't receive a response from its follower before some timeout, the follower should be counted as casting an abort vote. If there is no commit consensus (that is, if there is at least one abort), the leader must send an `ABORT` request in Phase-2. Otherwise, the leader must send a `COMMIT`. Furthermore, `COMMIT` and `ABORT` requests are always valid: the follower should always respond with an `ACK`.

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

The types and formats of KVRequests and KVResponses unique to TPC are detailed in the API section above. Again, it is up to you to correctly handle messages according to their type.

4.4 Using the API

We've provided you with a fancy web client to interact with your KVServers using our HTTP API. Once you spin up a server (TPCLeader or KVFollower), open your web browser and navigate to the root path of your server's address (e.g. `http://192.168.162.162:16200` for TPCMaster and `http://192.168.162.162:16201` (or 16202, 16203, etc.) for KVFollowers). Voila! Hopefully the interface is self-explanatory.

If you prefer, you can also interact with your server via command line. The `curl` command line tool can easily make HTTP requests to your server:

```
$ curl -i "192.168.162.162:16200/?key=cs162&val=bar" -X PUT
HTTP/1.1 201 Created
Content-Length: 0
```

Note: `curl` uses `GET` by default. Check out `man curl` or `curl --help` for more info.

5 Appendix

5.1 Follower Server Registration

The leader will keep information about its followers in a list of `tpc_follower_t`. Each follower server will have 64-bit globally unique ID (a `uint64_t`) assigned to them by the leader as they register using their hostname and port number. *You should not serve API requests until all followers are registered.* Furthermore, there will not be any followers that magically appear after registration completes, and any follower that dies will revive itself. A follower has successfully registered if it receives a successful response from the leader.

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

5.2 `libhttp` and `liburl`

`libhttp.h/c` together are a modified version of the library we provided for you in HW2. `libhttp` provides some helper functions and structs to deal with the details of the HTTP protocol.

Similarly, `liburl.h/c` provide helper functions and structs for dealing with marshalling and unmarshalling parameters present in HTTP request URLs. In other words, `liburl` helps unpack "`PUT /?key=foo&val=bar`", into a convenient `kvrequest_t`, and vice versa.

5.3 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.

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 hash of the entry's key, which can be found using the strhash64() 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, "%llu-%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.

5.4 Work Queue

`wq.c/h` define a synchronized work queue which will be used to store jobs which are waiting to be processed.

For each item added to the queue, exactly one thread can receive the item. When the queue is empty, there is no busy waiting.

5.5 Socket Server

Socket Server defines helper functions and structs that abstract communication over sockets.

- `connect_to`
  Used to make a request to a listening host.
- `server_run`
  Can be used to start a server (containing a KVServer or a TPCLeader) 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.
- `server_t`
  This struct stores extra information on top of the stored TPCLeader or KVServer for use by `server_run`.

5.6 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 we 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 we 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.