HW 4: Two Phase Commit

Due April 25, 2016

Contents

1 Your Tasks **2**

1.1 Task 1: Two-Phase Commit . . . 2

1.1.1 Follower 2PC handling . . . 2

1.1.2 Leader GET handling . . . 2

1.1.3 Leader 2PC coordination . . . 2

1.2 Task 2: Crash Recovery . . . 2

2 Getting Started **3**

3 Reference **4**

3.1 API . . . 4

3.1.1 Using the API . . . 4

3.2 Important Existing Code . . . 5

3.2.1 KVMessage . . . 5

3.2.2 KVConstants . . . 5

3.2.3 Socket Server . . . 5

3.2.4 TPCFollower . . . 6

3.2.5 TPCLeader . . . 6

3.3 Consistent Hashing . . . 7

3.4 Two-Phase Commit . . . 7

3.5 Appendix . . . 8

3.5.1 Follower Server Registration . . . 8

3.5.2 Socket Server Continued . . . 8

3.5.3 `libhttp` and `liburl` . . . 8

3.5.4 KVStore . . . 8

3.5.5 Work Queue . . . 9

3.5.6 UTHash, UTList . . . 9

1 Your Tasks

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.

1.1 Task 1: Two-Phase Commit

Multiple clients (users) will be communicating with a single leader server according to the given HTTP API. In your final implementation, the leader should forward client GET, PUT, and DELETE requests to the appropriate follower servers according to a Consistent Hashing scheme. The system should follow the two-phase commit (2PC) protocol to ensure that (1) operations are performed atomically across multiple follower servers, and (2) backup data is consistent across multiple follower servers. As a side-effect of consistency, we should also obtain integrity: if one of our follower servers crashes, we should still able to retrieve the keys stored at that node because we should have placed backups on neighboring follower servers.

We highly recommend that you complete this task in the following order:

1.1.1 Follower 2PC handling

Implement `tpcfollower_handle_tpc` in `tpcfollower.c` to correctly handle two-phase PUT and DELETE requests in a follower. Refer to Section 3.4 on Two-Phase Commit for details on the implementation of two-phase commit logic. Refer to Section 3.1.1 on Using the API for how to bypass the leader to communicate with followers directly. *Make sure to complete this subtask entirely before moving on.*

A basic non-distributed, thread-safe key-value data store has been implemented for you in `kvstore.h`. Operations on this key-value store are guaranteed to be atomic.

1.1.2 Leader GET handling

Implement `tpcleader_handle_get` in `tpcleader.c` to correctly handle GET requests coming from the client. Refer to Section 3.3 on Consistent Hashing to determine which followers to communicate with.

1.1.3 Leader 2PC coordination

Implement `tpcleader_handle_tpc` in `tpcleader.c` to correctly handle PUT and DELETE requests coming from the client. Refer to Section 3.4 on Two-Phase Commit for details on the implementation of two-phase commit logic. Data storage must be redundant, i.e. the system should not lose data if only a single TPCFollower fails. You will use simple replication according to Consistent Hashing for fault tolerance; refer to Section 3.3 on Consistent Hashing.

You do not need to support the concurrent execution of multiple 2PC transactions (you should serialize transactions).

1.2 Task 2: Crash Recovery

We have provided you with an implementation of a follower server that already saves its fully committed transations to persistent storage on the local filesystem. However, our implementation will lose data if a follower server crashes *in the middle* of a 2PC transaction. You must implement 2PC state logging and post-crash recovery to mitigate this issue. In general, you cannot prevent a follower server from crashing; however, when a fallen follower server reboots, it should be able to recover the state of any 2PC transaction that was currectly occurring when it crashed. Implement the logic for logging the 2PC state of a follower server in the various 2PC methods of `tpcfollower.c`. Implement the logic to restore a follower server's 2PC state from its log in `tpcfollower_rebuild_state` in `tpcfollower.c`.

`tpclog.h` exports a reasonable set of helper functions to aid you in this task. Make sure you to read through the provided functions before diving too deep.

A follower should log every relevant 2PC action it receives *immediately upon receiving the request.* You can add entries to the log using `tpclog_log`, which will make the entry persistent on disk (and thus durable through crashes). You may find the functions `tpclog_iterate_begin`, `tpclog_iterate_has_next`, and `tpclog_iterate_next` useful in iterating over the log during recovery.

Finally, because the store is 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).

2 Getting Started

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

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

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 an individual follower server using the following command:

```
./bin/tpcfollower [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/tpcfollower 16201 16200 &
./bin/tpcfollower 16202 16200 &
```

Using an `&` allows a program to run in the background so that you can run multiple programs simultaneously. You may also find that for some test cases, killing and restarting a `tpcfollower` may be necessary. To kill a `tpcfollower` on port `16201` use the following command:

```
kill -KILL $(lsof -ti :16201)
```

For context, this shell command sends a `SIGKILL` to the PID attached to the TCP port 16201. To restart this server just use the same command you used earlier.

In order to make it easier for you to test your kvstore, we have created a script which starts the `tpcleader` and a configurable number of `tpcfollowers`. In addition this script has some utilities that will allow you to kill and restart certain `tpcfollowers`. To use simply do

```
./bin/tpcsystem
```

Finally, you **should not** change the command line interface exported from the TPCLeader or TPC-Follower. Feel free to change/add any function, structs, or additional fields to existing structs (although, the staff believes that you should not need to do this). We have added `TODO: Implement me!` comments for your convenience.

3 Reference

3.1 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 struct types `kvrequest_t` and `kvresponse_t` respectively. Below are tables that summarize the API. Refer to the next section 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*”

3.1.1 Using the API

We've provided you with a fancy web client to interact with your servers using our HTTP API. Once you spin up a server (TPCLeader or TPCFollower), open your web browser and navigate to the root path of your server's address (e.g. `http://192.168.162.162:16200` for TPCLeader and `http://192.168.162.162:16201` (or 16202, 16203, etc.) for TPCFollowers). Voila! We hope that 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.

3.2 Important Existing Code

This homework provides a rather large skeleton. The following subsections provide a runthrough of what you'll likely be working with.

In addition to reading this spec, if you read through and understand the header (`.h`) files for each section mentioned below, you'll be in much better shape for chugging through your tasks. 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.

3.2.1 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 receive 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_receive/kvresponse_receive`
  Receives the HTTP message from the specified socket, unmarshalls and returns the message as a kvrequest/kvresponse.
- `kvrequest_clear/kvresponse_clear`
  Convenience function to clear the information stored in a kvrequest/kvresponse struct for reuse.

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.)

3.2.2 KVConstants

This file defines some of the constants you will use throughout the homework. You should familiarize yourself with them as you will be using them extensively throughout the homework.

Whenever you are returning an error to the user and there is a constant which corresponds to that error, be sure to return that constant. Otherwise you may `ERRMSG_GENERIC_ERROR`.

3.2.3 Socket Server

`socket_server.h` defines helper functions and structs that abstract communication over sockets.

- `connect_to`
  Used to connect a client socket to a listening server socket. Returns the socket file descriptor of the connected socket, or -1 on error. *Do not reuse a socket for more than one HTTP request/response pair.*

3.2.4 TPCFollower

TPCFollower defines a follower server which will be used to store (key, value) pairs. In a real-world scenario, each TPCFollower would be running on its own machine with its own file storage. Your final implementation should be able to support this.

A TPCFollower accepts incoming HTTP messages on a socket using the API described above, and responds accordingly on the same socket. There is one generic entrypoint, `tpcfollower_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 `tpcfollower_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 `tpcfollower_rebuild_state`, which rebuilds a follower server's state from a persistent log after a crash.

3.2.5 TPCLeader

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

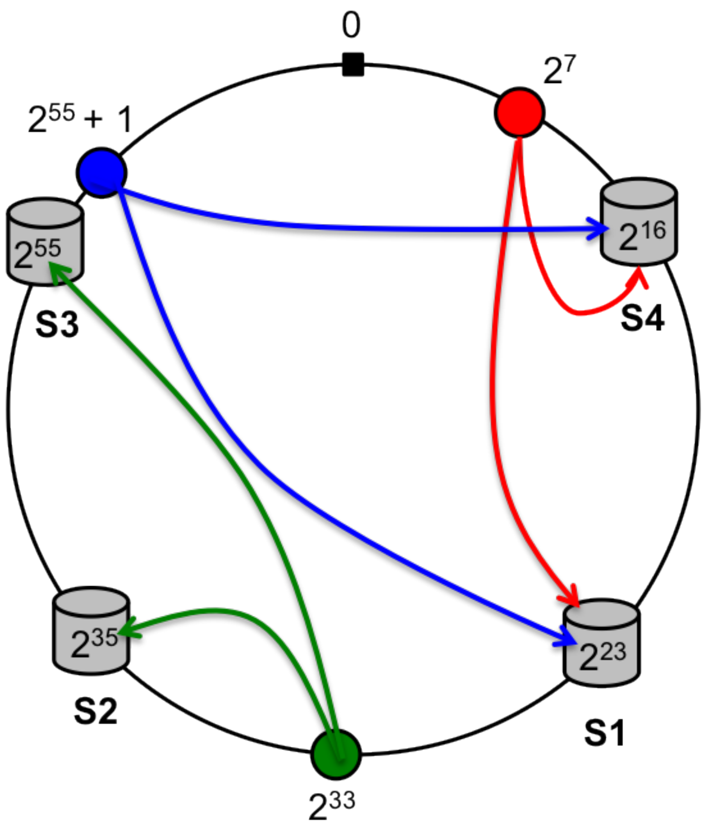
The TPCLeader will become the main point of contact for the client. In the final product, all API requests from the client are sent to the TPCLeader. Similarly to TPCFollower, there is one generic entrypoint at `tpcleader_handle`, which takes in a connected socket and handles all further communication.

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 Section 3.3 below.

3.3 Consistent Hashing

Each key will be stored on 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`. For reference, we are using the MD5 cryptographic hash function. *You may not change the hash function.*

Each 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. The following image provides 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 follower server.



3.4 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 `TIMEOUT` (specified in `socket_server.c`), 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 since followers should be able to recover their 2PC state.

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.

3.5 Appendix

3.5.1 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 appear after registration completes, and any follower that dies will revive itself. A follower has successfully registered if it receives a `SUCCESS` 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/tpcfollower.c`.

3.5.2 Socket Server Continued

- `server_run`
  Used to start a server (containing a TPCFollower or a TPCLeader) listening on a given port. This function 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`
  Struct type that stores extra information on top of the stored TPCLeader or TPCFollower for use by `server_run`.

3.5.3 `libhttp` and `liburl`

`libhttp.h/c` together are a modified version of the library we provided for you the HTTP Server homework. `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.

3.5.4 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. To resolve collisions, hash chaining is used. File names of entries within the store directory have the format: `hash(key)-chainpos.entry` via `sprintf(filename, "%llu-%u.entry", strhash64(key), chainpos)`. `chainpos` represents the entry's position within its hash chain, which starts from 0. If a collision is found when storing an entry, the new entry's `chainpos` will be incremented to 1, and so on. Chains are always contiguous.

All state is stored in persistent file storage. Thus, it is valid to initialize a KVStore using a directory name which was previously used. In this case, the newly initialized store will reflect the state saved in that directory.

3.5.5 Work Queue

`wq.c/h` define a synchronized work queue which is used to store client requests which are waiting to be processed.

For each item added to the queue, exactly one thread pops and serves that item. When the queue is empty, there is no busy waiting.

3.5.6 UTHash, UTList

UTHash and UTList are two header-only libraries which make creating linked-lists and hash tables in C easy. You can read more about their usage at `https://troydhanson.github.io/uthash/` and `https://troydhanson.github.io/uthash/utlist.html`.