Lab 4: Shard Transactional KVS Service

Due: 12-27-2021 23:59 (UTC+8)

Introduction

In this lab, you'll build a shard key-value store (KVS) service that supports transactions based on lab3's KVS. We call it **Chdb**. Recall that KVS is a method for storing and managing data in a memory device, and in lab3 you are supposed to finish one single node key-value store, keeping the consistency of replicas by consensus algorithm, Raft. Now it's quite easy to maintain consistency in a raft_group.

A shard in the KVS is a subset of the key-value pairs stored in the overall key-value store. For example, if the key is range from integer [1,100], and we distribute all of the keys equally into 5 shards, those in range [1,20] could be on one shard, and [21,40] on another, etc. Sharding is a horizontal scaling technique, by dispatching operations to multiple key-value servers.

Transactions are a set of operations used to perform a logical set of work, and they ideally have four properties, commonly known as ACID. You could see the TX-lock for more information. In this lab, you are required to implement a KVS which is transactional. More specifically, two-phase commit (**2PC**) and two-phase locking (**2PL**) are required to be implemented in this lab. To maintain the fault-tolerant property, simple **replications** should be employed on each shard, and the KVS system uses **a raft group** to satisfy consistency.

Your system has three main components. First, **the view server** is responsible for handling requests from client applications, dispatching different key-value pairs out to the managed shards. Raft group is adopted in view server for fault-tolerant. Second, the view server is managing multiple **shard clients**. Each serves as one shard. To interact with Chdb transactionally, applications use **transaction regions**, in which all of the operations are supposed to be transactional.

To sum up, lab4 has the following three parts. Your system is required to dispatch the key-value pairs into multiple shard clients, and implement a 2PC transaction protocol (part1). Then let each shard client support replication (primary backup) and introduce raft group to view server (part2). To deal with the concurrency problem, you are required to finish 2PL and handle the deadlock problem (part3).

There are 5 parts in this lab.

- In part 1(30 points), you will implement the 2PC protocol.
- In part 2(40 points), you will implement replication (primary backup) on shard client side and adopt raft group on view server side.
- In part 3(30 points), you will implement 2PL protocol.

Getting stared

First back up all of your prior lab code before starting lab4.

```
% cd cse-lab
% git commit -a -m "finish lab3"
```

Then, pull this lab from the repo:

```
% git pull
% git checkout lab4
```

Merge with lab3, and solve the conflicts.

```
% git merge lab3
```

After merging the conflicts, you should be able to compile the new project successfully.

Overview of the code

All of the core code are under `./chdb`, and your implementation should be under `./chdb/src/`. You can find four classes as below:

- `chdb`: Defined in `./chdb/src/ch_db.h`. It's the abstraction of your shard KVS system
- `shard_client`: Defined in `./chdb/src/shard_client.h`, and denotes as **shard client** abstraction
- `view_server`: Defined in `./chdb/src/ch_db.h`, and denotes as **view_server** abstraction
- `tx_region`: Defined in `./chdb/src/tx_region.h`, and denotes as **transaction region** abstraction

Now we are going to look through the example in `./chdb_dummy_demo.cc`, explaining the whole control path in detail. Hope it can help you better understand the entire code framework :)

The code of this example is quite short:

```
#include "tx_region.h"

int main() {
    chdb store(30, CHDB_PORT);

    for (int i = 0; i < 4; ++i) {
        tx_region db_client(&store);
        // dummy request
        int r = db_client.dummy();
    }
}
```

First, the included file `tx_region.h` contains all of the related code in Chdb. Upon the main entry, the program creates one Chdb at first (`store`), and then creates 4 transaction regions. Each region borrows the reference of `store`, and calls `tx_region::dummy()` . After running this program, the console would output as below:

> To run this file, just `make chdb_dummy_demo` and run `./chdb_dummy_demo`.

```
tx[0] begin
Receive dummy Request! tx id:0
tx[0] commit
tx[1] begin
Receive dummy Request! tx id:1
tx[1] commit
tx[2] begin
Receive dummy Request! tx id:2
tx[2] commit
tx[3] begin
Receive dummy Request! tx id:3
tx[3] commit
```

1. Chdb Establishment

This subsection guides you to dig deep into the control path of `chdb store(30, CHDB_PORT)`

The constructor of `chdb` takes three parameters `chdb(const int shard_num, const int cluster_port, shard_dispatch dispatch = default_dispatch)` . The `shard_num` represents the static number of shard client in the system, the `cluster_port` is used to assign the listening port of RPC protocol. `shard_dispatch` is the dispatching algorithm of a specific key, and its default parameter is `chdb::default_dispatch`.

The class `chdb` maintains one view server and multiple (`shard_num`) shard clients. Those are the main components in the KVS system, and they interact with the help of class `rpc_node`. The class `rpc_node` in `./chdb/src/common.h` has implemented a simple abstraction of rpc protocol (based on the rpc code in `chraft/rpc.h`).

At this stage, you could glance through the call stack of the `chdb` constructor, to understand how those components establish the rpc connections.

2. Transaction Region

Now you have got a rough understanding towards the chdb establishment, and this subsection will move one more step, introducing the control path of `tx_region::dummy()`. Till then you'll find it easy to understand the console output and set out to start the lab.

First, please focus on the constructor and destructor of class `tx_region` in file `./chdb/tx_region.h`

```
class tx_region {
    ...

    tx_region(chdb *db) : db(db),
                          tx_id(db->next_tx_id()) {
        this->tx_begin();
    }

    ~tx_region() {
        if (this->tx_can_commit() == chdb_protocol::prepare_ok) this->tx_commit();
        else this->tx_abort();
    }

    ....
}
```

The usage of `tx_region` is to put it into one closure.

```
    // transaction region
    {
        // tx-begin
        tx_region db_client(&store);
        db_client.put(1, 1024);
        r = db_client.get(1);
        // tx-commit OR tx-abort
    }
```

This closure is one transaction region. That is, calling `tx_region::tx_begin()` once entering the region, and calling `tx_region::commit()` or `tx_region::abort()` (depending on the result of `tx_region::tx_can_commit()`) before leaving the closure. In the lab, you have to implement the 2PC and 2PL based on this framework.

Next let's focus on the control path of `tx_region::dummy()`. It's quite straightforward at this stage. Just use the `rpc_node` in view server, post one RPC request of type `Dummy` (defined in `./chdb/protocol.h`), and the view server will dispatch the request to the target shard client. Each shard client has registered the `Dummy` handler (`shard_client::dummy()`) in the establishment stage. In the lab, you are required to replace the straightforward synchronize call-reply mode with `raft_command` in `chraft`, and make the view server to be one `raft_group`.

Part-1

In this part, you have to dispatch the key-value pairs into multiple shard clients, and then implement the 2PC transaction protocol.

Recommended steps:

1. Mimic the control path demonstrated in code overview. Add more RPC handlers (**PUT**, **GET**) in class `shard_client`.
2. The 2PC protocol contains two stages: **prepare** stage and **commit** stage. Implement the correct semantics of these two stages in `tx_region::tx_begin`, `tx_region::tx_can_commit`, `tx_region::commit`, `tx_region::abort`. Feel free to add more RPC handlers if necessary. Notice that we simply let one shard client reply `not_ok` in **prepare** stage, if already setting its state to be `prepare_not_ok`, thus this state can be used in `shard_client::check_prepare_state`.

You should pass the 6 test cases of this part. (5 points + 5 points + 5 points + 5 points + 5 points + 5 points)

```
% ./chdb_test part1

Pass 6/6 tests. wall-time: 18.08s, user-time: 7.21s, sys-time: 1.08s
```

Part-2

In this part, you are required to support replication on shard client side, and adopt raft group you've implemented in lab3 on the view server side.

Recommended steps:

1. Support replication migration in a single shard client. (At which point to do the data replication ?) In the testcase, we shuffle the primary replication by calling `shard_client::shuffle_primary_replica`. You are suppose to read the correct `shard_client->store` even after shuffle the `shard_client->primary_replica`.
2. Enable macro `RAFT_GROUP` in `chdb/src/common.h`. You may use this macro in later implementations. If the raft protocol is not fully implemented in lab3, just disable this macro.
3. Adopt raft group in class `view_server`, and implement the `chdb_state_machine` defined in `chdb_state_machine.cc`. Modify `view_server::execute` to ensure that, all of the operations in a `tx_region` have reached consensus among the raft group, before posting the requests to shard clients.

You should pass the 3 test cases of this part. (15 points + 15 points + 10 points)

```
% ./chdb_test part2

Pass 3/3 tests. wall-time: 20.75s, user-time: 4.74s, sys-time: 0.41s
```

Part-3

In this part, you'll handle the concurrency problem. First, you should implement the big lock. It is supposed to pass all of the testcases. Furthermore, 2PL is needed to be implemented instead of the big lock. Meanwhile, you have to handle the dead lock problem.

Recommended steps:

1. Implement big lock in `tx_region`. Acquiring big lock at the beginning of one transaction and release this lock before the end of the transaction. You are supposed to pass all of the testcases at this point (except the bonus part).
2. (Bonus) Change big lock into more fine grained lock, which statisfies 2PL protocol. Handle the dead lock problem as well. You may refer WaitWoundDie to help you solve it.
3. Note that you should disable (`#define BIG_LOCK 0`) the macro `BIG_LOCK` in `chdb/src/common.h` to get the bonus score.

Run test by:

```
% ./chdb_test part3

Pass 6/6 tests. wall-time: 18.99s, user-time: 9.64s, sys-time: 1.77s
```

You should pass the 6 test cases of this part. (6 points + 6 points + 6 points while using dead lock; 4 points + 4 points + 4 points for 2PL protocol with deadlock problem solved). Thus the score without bonus is supposed to be 88/100.

Grading

After you have implmented all the parts above, run the grading script:

```
./grade.sh
```

IMPORTANT: The grade script will run each test case many times. Once a test case failes, you will not get the score of that case. So, please make sure there are no concurrent bugs.

Handin Procedure

After all the above done:

```
% make handin
```

That should produce a file called lab4.tgz in the directory. Change the file name to your student id:

```
% mv lab4.tgz lab4_[your student id].tgz
```

Then upload lab4_[your student id].tgz file to Canvas before the deadline.

You'll receive full credits if your code passes the same tests that we gave you, when we run your code on our machines.