Operating Systems – Spring 2017 Project 3

Database and Commit

In the second half of this semester, we start to work on some problems related to Distributed Systems. Although this project still runs on single machine, we start to face the first issue in distributed environment: crash failure. We ask you to provide a key-value database that commits data to the disk, and can recover after a crash failure by reading from the disk. Specifically, the key-value storage we will build here will not only store the final states of each account, but all the transactions in the database, in a log format. Nowadays many people use a fancy name ``block chain'' for such logs. In this assignment, we only use a single server implementation, assuming the disk files are persistent (i.e. disks never fail). In Project 4, we will extend the solution to multiple machines to tolerate disk / server failures.

As usual, we provide a sample implementation of a simplest form of key-value database: all data are stored in-memory, and of course, they will be lost when the process is killed. In this assignment, we require you to enhance the original implementation, or write a brand-new implementation using the same interface. You are free to choose your favorite language when implementing the database, as long as you stick with the original communication protocol (RPC); please familiarize yourself with **gRPC** and **protobuf**. Database clients will send requests over gRPC, with data structure as defined in `db.proto` file. You can refer to the documentation (and installation guideline) of gRPC at http://www.grpc.io/docs.

Definition of Database Operation

The key-value database need to support five transactions:

1. GET, which shows the balance (an integer) of a given account ID;
2. PUT, which sets the balance of an account;
3. DEPOSIT, which adds a number to the balance of the given account.
4. WITHDRAW, which subtracts a number to the balance of the given account.
5. TRANSFER, which moves some amount of money from one account to another.

As an integrity constraint, the database guarantees that all account balances must be non-negative, which means WITHDRAW and TRANSFER operation may fail if the balance is not sufficient for the operation. Amount of money in all kinds of operations is non-negative. For simplicity, we only allow account IDs as strings of length 8 with alphabet letters and numbers. The initial balance of any new account is zero.

When processing a transaction, in addition to the in-memory database state (i.e. the database table holding the balance of each account). In order to make sure the data can be recovered on a crash, we need to write a transaction log to the disk before returning the result to the client, such that we can recover the database from the transaction history from the disk. However, the log cannot grow

infinitely long; here we require "log retention" to delete old logs and transfer the transaction history into formatted "blocks", each with N transactions (and other metadata). We store the blocks in a chain forever. We choose N=50 in this assignment.

You can design your own format of the transient transaction log, with arbitrary data structure, as long as you can read it back from the disk after restarting from crash failure. The "blocks" should be written in JSON files in a directory, with the following information:

```
{
    "BlockID":1, // Starting from 1, and save as "1.json", "2.json", ...
    "PrevHash":"00000000", // The hash of the previous block; not used in this assignment, always set to "00000000".
    "Transactions":[
        // A list of N transactions recorded:
        {
            "Type": "PUT", // Type of transaction
            "UserID": "Test0000", // UserID of the transaction
            "Value": 10 // The amount of money in this transaction, an integer
        },
        {"Type":"TRANSFER","Value":10,"FromID":"Test0000","ToID":"Test1234"}
        // The TRANSFER transaction requires two IDs: FromID and ToID.
    ],
    "Nonce":"00000000" // A random 8-digit nonce. Not used in this assignment, always set to "00000000".
    // (Please remove all comments)
}
```

This format is defined in the protobuf schema file db.proto as Block message; you can save the protobuf data structure directly into JSON and parse JSON data into protobuf data structure using APIs from protobuf library.

There are several other pre-defined formats for this assignment:

- The file `config.json` contains basic configurations for the database server; please read listen address, RPC port and data directory from this file. The test client also reads from this file to connect to the server. All your block files and log files should be saved in the given data directory.
- The file `compile.sh` is used to compile the code of your database server.

- The file `start.sh` is used to start your database server.

Reliability Requirements

The database may crash at any moment. Meanwhile, we require that:

1. If an update operation (non-GET) returned successfully, then any subsequent successful GET operation should be able to see its effect. Note that the server may crash between these two requests.

2. If a GET operation arrives later than an update operation, and the update operation returned success, the GET operation should see its effect.

3. GET operation does not need to appear in JSON log blocks. The size of your transient log should be O(N).

Please think carefully about how and when to commit operations to disk. (You can assume that anything successfully written to disk is persistent against crash failure.)

Notes

We have provided an example client and a test script, as well as a very simple server RPC implementation. If you run the test script against a full-function database server, the first output block should be equal to `example_1.json` file, an example JSON output block we provided.

JSON files are not checked verbatimly (the order of fields doesn't matter), instead they are compared for their meanings. Think about the Objects.deepEquals() method.

You should submit and document your test cases with the project. Keep in mind that a database should suffice ACID, and you should cover all four aspects!

Tasks

1. (20%, 50 lines) You need to program a database server and implement the RPC calls to support the five basic database transactions we mentioned.

a. The in-memory database can be modelled as a hashmap: map{string (UserID) -> int (Balance)}. We require O(1) time access for items, so a hashmap need to be kept in memory. We do not want a disk-based table data structure, and instead, we want to keep only logs on disk for recovery. For every new transaction, the log is written before the action is performed.

b. When the transient log starts to grow longer than N, you need to flush the first N non-GET transactions into a new block, and remove the corresponding transient log. Write the block into the JSON format we previously defined.

c. Please be familiar with gRPC and implement RPC handlers for the five database transactions, as defined in db.proto.

d. Additionally, your implementation of the database needs to support the following RPC calls to interact with our grader: LogLength(): return the length of transient (non-block) log on disk.

2. (30%, 20 lines) The data in the database need to be persistent through server crash. When the server crashes (got killed) and started again, it needs to read back all previous transactions from the disk (first the blocks, then the transient logs) and recover the data, unless it is the first run ("clean start"). During the recovery process, it does not have to serve any client transactions (but when it starts to serve transactions, the results must be correct). If block files or log files are deleted, missing, or inconsistent, the database should fail to start and report an error.

3. (20%, 20 lines) The transactions need to satisfy the integrity constrains; in our case, the database need to guarantee illegitimate transactions cannot go through. The system shall reject transactions that creates negative balance.

a. The illegitimate transaction should receive a failure result.

b. Remember, integrity constraints require all transactions lead the database from a correct state to another correct state. Thus no two contradictory transaction may coexist in the log. For example, if Account A has balance $15 and the server receives two transfers A->B ($10) and A->C ($10) simultaneously, exactly one of them should succeed and one of them should fail (aborted transactions should NOT be put into the log).

4. (20%, 30 lines) The non-GET transactions need to be saved as JSON blocks.

a. Each block contains exactly N=50 transactions.

b. The blocks should all be put in the given data directory, with file name as `ID.json`. The block ID starts from 1 (i.e. the first file is `1.json`).

5. (10%, 2 lines) You need to modify and provide the startup script of your program (start.sh), and a compile script to compile your program (compile.sh).