Operating Systems – Spring 2017 Project 2

Distributed Database and High Availability

In the second half of this semester, we start to work on some problems related to **Distributed Systems**. The goal of this project is to achieve high-availability / crash-tolerance by using multiple servers. In the lecture, we discussed multiple protocols to achieving consensus.

A recent trend in distributed systems is to use a "blockchain", or a chain of blocks, to achieve distributed consensus and tamper-proof validity. In this assignment, we implement a simplified blockchain system and see how the concepts like Proof-of-Work, help reach consensus.

You need to reuse some of your key-value database codebase from Project 3. The client interface will be similar to the gRPC interface in Project 3. The system works this way: the client can contact any server to submit a transaction. The server broadcasts the (tentative) transactions to the entire network, hoping that most of the servers learns about it. Then a leader will assemble the transactions into a ``block''. Our goal is reach consensus on the content of each block. As re need to support server-server replications, we need to introduce an additional set fo gRPC interfaces for communication among these servers.

Definition of Database Operation

Each block shall contain **at most** N=50 transactions. When there's less than 50 transactions, your server should still produce and broadcast new blocks, such that transactions can quickly go into blockchain. Empty blocks (with no transactions) are invalid.

For the project, let's assume that all transactions are of the same type: *Money Transfer.* The transfer transaction has the following attributes:

- Type: "TRANSER"
- FromID: Money to transfer from this account.
- ToID: Receiver.
- Value: Integer, the amount money to send out (including the fee).
- MiningFee: Integer, money received by the block producer (Miner).
- UUID: 128-bit hexadecimal string, unique for each transaction.

The amount of money that the receiver (specified by ToID) gets equals to (Amount - MiningFee). The MiningFee is payed to the leader (aka the winning miner) as an incentive to do the work. Note that you need to verify MiningFee is a positive integer, and the Amount is larger than MiningFee; otherwise, the miner rejects this transaction and does not include it in the block.

The new RPC interface is defined as follow:

1. GET operation now shows the balance of the account ID, based on the latest block already produced.
    a. A new account will have balance 1000 instead of 0 in this assignment. Meanwhile, there's no Deposit or Withdraw anymore.

    b. If there are multiple "newest" blocks, use the longest branch (see definition later).
2. TRANSFER operation is initiated by client, with a new UUID each time. Besides the business logic requirements from project 3, the server returns a successful response only after the transaction has been pushed to at least one other server.
3. Verify: check if a transaction has been committed. Return SUCCEEDED if the transaction has been written into a block and another 6 valid consecutive blocks have been generated after this block. Or return PENDING if the server sees the transaction but it's not included into a block or not enough blocks have been appended after the block containing it. Or return FAILED if the transaction is not received by the server or is invalid on the longest branch.
4. PushTransaction operation is used to share transactions between servers. If you receive a new transaction, you are required to forward this transaction to other servers. **The server is not allowed to create fake transactions (cheating) in any way!** (In the testing phase, you can cheat/misbehave in other ways, but not this one.)
5. PushBlock: when the server has successfully produced a new block, it must broadcast the block to all other servers using the PushBlock function (as a RPC client).
6. GetHeight: the client or other servers can query about the current length of blockchain. If multiple branch exist, use the longest branch.
7. GetBlock: the client or other servers can query about a past block by its hash. If the server doesn't know about this block, please return Null. Otherwise, you should return the JSON string of this block (such that the hash of this string is equivalent to the hash in query).

- To ensure the consistency of Hashing between different servers, we have provided a Hash class for you (use like a black box, do not modify); please make sure you always invoke this class when calculating hash. Our hash function is based on SHA256, the same as Bitcoin system.
- To make this project more excited, we will add a cross-testing phase as usual. The servers of each group will connect with each other.

Definition of Block Generation and BlockChain (Tree)

In this project, the servers use a "proof-of-work" model to perform leader-election and ensure data integrity.

- First of all, each block is represented by its JSON encoded string, and a legitimate block will satisfies hash(JSONstring)="000...000xxxx..xx", where the number of 0s represents the difficulty of mining.
    - In this assignment, we have set the difficulty to 20 bits (5 hex digits); you must use the provided CheckHash function to check if the hash string is legitimate.
- Each block will contain the hash of its predecessor block; the first block has hash "000...0" (all 256 bits are zero). This mechanism allows everyone to know which blocks are on the same chain. Also, it helps verify that there is no modification to any existing blocks.
- Obviously, the blocks have formed a tree. If everything happens as we want (i.e. there is only a single leader per round), the tree is very simple and only contains a single branch. However, if there are multiple leaders per round, there will be multiple branches, you should always "follow" the longest branch.

    - This means your new blocks should always be “mined” from the tail of the longest branch, making it even longer. If you are aware of another branch becoming the longest, you need to stop mining.
    - If there is a tie, working on the block with smallest hash string.
- When you receive a block from other miner, you need to check:
    1. If the block’s string hash is legitimate;
    2. If the block’s hash to its previous block is indeed a block on (one of) the longest branch;
    3. If the block’s transactions are new transactions; and
    4. If the block’s transactions are all legitimate (Amount>=Balance), w.r.t. the branch it’s extending. (It’s possible for a client to double-spend money on different branch, but not on the same branch.)
- If this block is legitimate, you extend the previous branch and continue mining after this new block. If you accept this block, all the MiningFees in that block goes to the miner’s account (written in the block).

The block format is defined as follow:

```
{
    "BlockID":1, // Starting from 1, and you should save them as “1.json”, “2.json”, ...
    "PrevHash":"0000000000000000000000000000000000000000000000000000000000000000", // The hex hash of the string of previous block; set to “000..” for the first block. 64 digits.
    "Transactions":[
        // A list of N<=50 transactions recorded:
        {
            "Type": "TRANSFER", // Type of transaction
            "FromID": "Test0000", // Client ID of the transaction, 8-digit
            “ToID”: "TestRecv", // Receiver ID of the transaction, 8-digit
            "Value": 10 // The amount of money in this transaction, an integer
            “MiningFee”: 1 // The amount of mining fee for block producer, an integer
            “UUID”: "52fdfc072182454f963f5f0f9a621d72"// A unique UUID, 32 digits
        },
        {"Type":"TRANSFER","Value":100,”MiningFee”:2,
    "FromID":"Test0000","ToID":"Test1234",”UUID”:”b954431a7c0a426e8bc6a1970782d7e4”}
    ],
    “MinerID”:”Server01”, // The server’s account ID, formatted as “Server%02d”.
    "Nonce":"00000000" // A random 8-digit string nonce. You need to try various nonce before finding the “Lucky” one, producing a “good-looking” hash for this block.
    // (Please remove all comments)
}
```

Again, the ordering of fields are arbitrary and you should accept the block if its out-of-order, as long as the hash of JSON is correct. You may also reorder transactions within a block. However, please note that the ordering of transactions is very important and reversing the order may cause some transactions to become illegitimate, and you must ensure the sequence of transactions in every block is legitimate.

If there's any new transactions pending, every server should try its best to find the next "good" block, until being notified that someone else has achieved that. When one transaction is already included in a previous block in the chain, you can stop adding it to new blocks (every transaction is added exactly once).

Definition of config and command-line flag

- The "--id" flag specifies your ID, with 1<=ID<=nservers. Please read your configuration from the correct part of config.json.

The format of config.json, compile.sh and start.sh remained the same as project 3. However, please note that your server should take command flags and read the correct part of config.json. You also need to refer to config.json to talk to other servers. An example of config.json looks like the follow:

```
{
        "1":{
                "ip":"127.0.0.1",
                "port":"50051",
                "dataDir":"/tmp/server01/"
        },
        "2":{
                "ip":"127.0.0.1",
                "port":"50052",
                "dataDir":"/tmp/server02/"
        },
        "3":{
                "ip":"127.0.0.1",
                "port":"50053",
                "dataDir":"/tmp/server03/"
        },
        "nservers":3
}
```

In the example, if you received "--id=2" as command line flag, you should listen to 127.0.0.1:50052 and communicate with other servers via 127.0.0.1:50051 / 127.0.0.1:50053.

Reliability Requirements

Each of the running database servers may crash **at any moment**.
You may assume that, at any moment, the number of crashed server will be less than NServers/3. Also, you don't need to worry about Hash Collision (two block must have different hash) or UUID uniqueness (two transactions must have different UUID).
Meanwhile, we require that:

1. If only a minority of servers crash, clients can still submit operations to other servers and proceed. The system should guarantee its liveliness.

2. Keep in mind that after a server crashes and recovers, it may lost its data on disk! In this case, the server should recover its blocks by replicating from another server.
3. If one server is not responding correctly to GetBlock() calls, the recovery process shall continue by asking other servers. Similarly, if the blockchain recovered from one server looks weird/suspicious/problematic (e.g. forms a loop, or does not look like a chain?), you should find another server. (The chain is correct if it can trace to the first block and all transactions are legitimate.)
4. If the different server responds with an inconsistent result, you should try your best to recover some knowledge about the blockchain (tree), and follow the longest chain.
5. Note that other server's action (e.g. discovery of new blocks) may change the response of queries to your server. For example, a GET operation will return different value after a new block is pushed to your server. Please make sure your server updates accordingly when receiving such update from other servers.

Please think carefully about how servers communicate, recover, and compete.

Special Notes about Cross-Testing

As a tradition of IIIS OS course, the final project includes a cross-testing phase in which students test each other's implementation and find weird bugs. For this project, it contains two parts:

1. The client-role test, where a client can send any transactions, at any time, to a group of servers (all the same servers from another group). The client may also send GET, GetBlock and GetHeight query. Then, the client should report how many tests are passed / failed.
    a. You will get bonus points if your test case is reasonable and other groups failed your test case.
    b. Please do not perform pressure test (sending significantly large amount of transaction). Please do not attack the bug / implementation detail of underlying gRPC/ProtoBuf library.
2. The mining test, where one server of each group runs against each other. In this mode, some clients will generate some "random" transactions, and your server is busy mining and compete with others.
    a. If your server crashed or cannot respond to GET correctly, you will receive penalty. We may or may not restart the crashed servers.
    b. You will get bonus points if you win the mining contest, e.g. at the end of the day your miner's account balance is top 2 out of all groups. Think carefully! (Note that each block may contain ~50 mining fee.)

In addition to this, we still have conventional testing cases; however, your creative test cases is very important. During the test, other servers you met may go byzantine and behave strangely (return anything on GetBlock/GetHeight calls), but your server should always ensure correctness.

Notes

Again, we provide a very simple server RPC implementation and a sequence of example operations. If you run the test script, you will probably get a chain with reasonable length (depending on your luck) and

your miner's account will collect some fees. In the client example, we also provided reference implementation for generating UUID.

When checking JSON files' structure, keep in mind that the order of fields doesn't matter, as long as the structure is same as defined (no extra field can present). 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. (30%, 50 lines) Basic Database Operations:
    - Receive transactions, and broadcast transactions;
    - Produce blocks, and broadcast blocks;
    - Respond to GET request using the blocks on the longest chain.
    - Upon startup, try recovering / catching up the longest chain from other servers; mine from the head 0000 only if you believe the longest chain has length 0.
2. (20%, 20 lines) Transactions need to satisfy business logic; i.e. the fee, the value, and the remaining balance of FromID should be legitimate.
3. (20%, 20 lines) Reject illegitimate blocks:
    - If hash cannot pass the check or JSON failed to decode into correct format, reject;
    - If some transactions ordering lead to negative account balance, reject;
    - If the receiver ID of mining fee is not a legitimate server ID, reject;
    - etc.
4. (20%, 30 lines) Maintain the structure of blockchain tree:
    a. If someone pushes you a legitimate block on the tree, you must save it (even if it's not on the longest chain -- they may form a longer chain). When others query about any old block, you should return such blocks if you know them.
    b. You need to maintain which chain is the longest chain, and change to another chain when another chain becomes the longest. Make sure your block is always extending from the longest chain.
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).
6. (Bonus 10%, unknown) You need to improve the performance of your miner, given the same computation power (hash per second). When 8 identical miners are running, we can expect they collect approximately the same mining fee; however, when 8 different servers from each group are running together, we expect one or two of them will earn much more fee than others. Are you one of those groups?
7. (Bonus 10%, unknown) There are many transactions in the bitcoin world and it's impossible to save all blocks / transactions in memory. When there are many many blocks, can you still respond to GET requests with reasonable efficiency?