CS 171 : Introduction to Distributed Systems
Final Project
Demos: Tuesday March 16, 2021

A key–value database, or key–value store, is a data storage paradigm designed for storing, retrieving, and managing associative arrays, and a data structure more commonly known today as a *dictionary* or hash table. The key-value store maintains data as a collection of key-value pairs, in which each key uniquely identifies each object.

To ensure fault-tolerance, the key-value store is replicated on several servers (5 in this project). You will build a distributed key value application on top of a private blockchain to create a trusted but fault-tolerant decentralized system such that $\forall k \in keys$, the value of key k is the same among all copies of the key-value store.

1 Overview

In this project, you will be building a simplified key-value store backed by a blockchain. For this project, you will be using Multi-Paxos as the method for reaching agreement on the next block to be appended to the blockchain. None of the nodes are assumed to be malicious, but some might crash, *fail stop*. The network may also exhibit failures, in particular, the network might partition.

1.1 Key-Value store application

You will build a key-value store which is used to serve a simple student portal. The key value store will store keys which are your netIDs. The value would contain your phone numbers(for the sake of simplicity). For example,

```
Key: alice_netid
Value: {"phone_number": "111-222-3333"}
```

2 Functional Specification

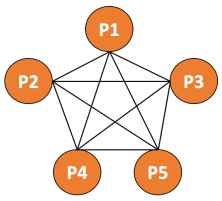

Figure 1: Processes connected to each other

The key-value store system you will build will have five nodes, $P1$ through $P5$ (as shown in Figure 1). Each node provides an interface for clients to perform two operations on the key-value store. (1) **put(k,v)** which inserts a new key value pair $< k, v >$, (2) **get(k)** which returns the value v of key k if it exists.

Every node will maintain three data structures:

- Blockchain: In this simplified Blockchain implementation, you will implement the blockchain as an *append-only* data structure, which stores *blocks*. A *block* contains a single operation. An *operation* is the action sent to the key-value store to put or get data. An *operation* has the structure $< op, key, value >$. Value is optional based on the *op* type. Here, *op* can be **put** or **get**.

  1. If *op* is **put**, *operation* would look like $< put, alice\_netid, sample\_value >$, where *sample_value* could be:

     ```
     {"phone_number": "111-222-3333"}
     ```

  2. If the **op** is **get**, *operation* would look like $< get, alice\_netid >$.

  The blockchain is consistently replicated across the five nodes.

- Queue: This is to store all the temporary operations until the set of operations are added to a block and appended to the blockchain.

- Dictionary: Maintaining the key-value store.

2.1 Contents of each block

Each block consists of the following components:

- **Operations**: A single $< op, key, value >$ operation.

- **Hash Pointer**: Since the blockchain is implemented as an append only data structure, the previous block is always immediately before the current block. All you need to include is the *hash* of the contents of the previous block in the blockchain. To generate the hash of the previous block, you will use the cryptograhic hash function (SHA256). You are **not** expected to write your own hash function and can use any appropriate pre-implemented library function. SHA256 returns an output in hexadecimal format consisting of digits 0 through 9 and letters a through f.

$$O_{n+1}.Hash = SHA256(O_n.Operation||O_n.Nonce||O_n.Hash) \quad (1)$$

- **A Nonce**: A *nonce* is a random string. The nonce is calculated using the content of the *current* block (unlike a hash pointer, which is based on the previous block). Basically, you

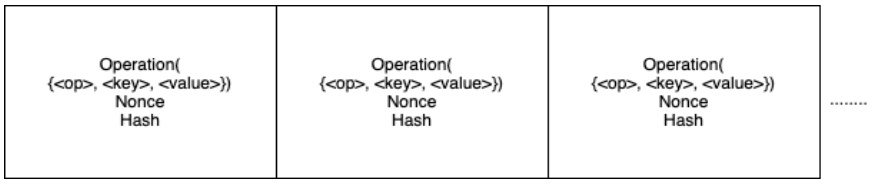

Figure 2: Structure of blockchain

need to find a nonce so that when the nonce is concatenated to the operation in the current block, the last character of the hash is a *digit* between 0 and 2.

$$h = SHA256(Operation||Nonce) \tag{2}$$

Hence, in Equation 2, a successful nonce will produce a *digit* between 0 and 2 as the last character of h. In order to do that you will create the nonce randomly. *The length of the nonce is up to you.* If h does not end with a digit between 0 and 2 as its last character, you will have to try another nonce. In other words, you need to create a sequence of strings randomly until the right-most character of the resulting hash value is a digit between (0-2). This is a simplification of the concept of *Proof of Work (PoW)* used in Bitcoin. Although you will use Multi-Paxos for consensus, the idea of nonce increases the tamper resistance of the system. To tamper with the blockchain, an adversary will need to recalculate all hash pointers, and for each hash of the previous block, it will have to calculate its **correct nonce**. If we add more restrictions on the calculation of the nonce, the amount of computational resources the adversary needs to have will increase, hopefully prohibitively.

2.2 Persisting blockchain to disk

To ensure persistence in the presence of failures, you will persist both the key-value store as well as the blockchain in a file on disk. If the node is the leader, when agreement is reached on a block, the leader writes this block to a file. If the node is a participant, when it receives a block from the leader it needs to write the block to the file and tag is as *tentative*. When it receives the decide message on a block from the leader, it tags the block in the file as *decided*. Make sure that you write all the contents of the block to the file and you should be able to read that file and reconstruct the blockchain in case of node failures. Once a block is decided on disk, a put operation in the block is executed on the key-value store (on disk too).

2.3 Protocol implementation details

2.3.1 Clients

You will assume 3 different clients. Each client issues a single put or get operation and waits for a response before issuing the next operation. Each client has a "hint" on who the current Paxos leader is, and sends its operation to that leader. If the server that receives the operation is not

the leader, it forwards the operation to the server it thinks is the current leader. If a client does not receive a response to operation, it times out and sends a special message *Leader* to any of the servers requesting it to become a leader.

2.3.2 Multi-Paxos

Any node that receives a *Leader* message tries to be the leader by executing the first (leader election) phase of Paxos. Once the leader has a **non-empty** queue it proposes the next block in the blockchain to the other *acceptor* nodes. In this project, you are to incorporate the leader election phase of Paxos to support blockchains as follows:

1. A node intending to become the leader sends prepare messages and must obtain a majority of promise messages for its ballot number.

2. Before it can propose the next block with one of the pending operations that were stored in its queue, the node should compute the acceptable hash value (the last character of the hash must be a digit between 0 and 2) by finding an appropriate nonce. Note that if a process takes too long to compute the acceptable hash, another process might time out and start leader election.

3. The leader ensures that a majority of servers have identical blockchains and hence key-value stores.

Once the node becomes a leader, it proposes a sequence of blocks using accept messages for all operations in its *queue*. After a majority of nodes accepted the block, the leader appends the block to its chain, *updates its key-value store(dictionary)*, and removes the operation from the queue. The leader then (1) send a response to the client, ie, the value for a *get* operation and an "ack" for a *put* operation, and (2) sends out decision messages to all the nodes, upon which they append the block to their blockchain and *update their key-value stores* depending on the operation present in the block.

The Paxos algorithm described in class obtains agreement on a value for a *single* entry in a replicated log. Since your application needs agreement on multiple blocks of the replicated blockchain, the ballot number should additionally capture the depth (or index) of the block being proposed. Hence, the ballot number will be a tuple of $< seq\_num, proc\_id, depth >$. An acceptor does not accept prepare or accept messages from a contending leader if the depth of the block being proposed is lower than the acceptor's depth of its copy of the blockchain.

When a node is elected to be a leader, it collects information about the depth of all the blockchains among the participants. This information is compared and the leader attempts to update all nodes based on the most up to date blockchain (longest in terms of number of blocks). This part is tricky. Keep it till the end and think about all the edge cases.

This recover algorithm is an extra credit of 10%. Please address all other tasks first and then comeback to tackle the recovery. You are free to choose any ways to recover

the log of the crashed or partitioned nodes and update their corresponding key-value stores. One suggestion is to maintain in information of the fist uncommitted index at both the acceptors and the proposer (leader). The first uncommitted index is attached to the accept, accepted and decision messages. For instance, when the leader receives the accepted message from an acceptor, if the attached index is older than what the leader has then the leader will send all necessary information to repair the log of that particular acceptor. On the other hand, if the index of the leader is older than the acceptor, then the leader will request all necessary information to repair its log from that particular acceptor. The full details of this recovery is left for you to discover.

Client-Server interaction: The client issues a **put** or a **get** and expects a reply. When the leader receives a majority response during phase 2, it should send a response to the client. In case of **put**, this would be an *ack*. In case of **get**, this would be the value of the key in the key-value store if present. If the value is absent, return *NO_KEY* back to the client.

3 User Interface

The user must be able to input the following commands:

1. *Operation(op, key, value)*: Depend on the type of op, the client either wants to put or get the key.The client expects a response for each operation it issues: 'ack" for a put and a value corresponding to a get operation.

2. *failLink(src, dest):* This command must emulate a communication link failure between the *src* process and *dest* process. For example: if we want the communication link between P1 and P2 to fail, we will use this interface as a user input on process P1. And your program should then treat the link between the two nodes P1 and P2 as failed and hence should not send any message between the two nodes. For simplicity, links are considered bidirectional and breaking a link allows neither the *src* nor the *dest* to communicate with each other.

3. *fixLink(src, dest):* This is a counterpart of failLink input. This input fixes a broken communication link between any two processes, upon which the two nodes will be able to communicate with each other again. This input will be provided on the *src* process.

   A suggestion to simulate such a behaviour could be to use a dictionary data structure that has the network links in the system and has an active or non-active boolean. You can use such a map to check if the network link is active before sending out any message. This is one suggestion; you can use any other technique as long as the behaviour of network failure and fixing of it is emulated.

4. *failProcess*: This input kills the process to which the input is provided. You will be asked to restart the process after it has failed. The process should resume from where the failure had happened and the way to do this would be to store the state of a process on disk and reading from it when the process starts back.

5. *printBlockchain*: This command should print the copy of blockchain on that node.

6. *printKVStore*: This command should print the key value store on that node.

7. *printQueue*: This command should print the pending operations present on the queue.

4 System Configuration

NOTE: We do not want any front end UI for this project. All the processes will be run on the terminal and the input/output for these processes will use stdio.

1. In this assignment all nodes are directly connected to each other. You can use a configuration file with the IP and port information, so that the nodes can know about each other.

2. All message exchanges should have a **constant** delay (e.g., 5 seconds). This simulates the network delays and makes it easier for demoing concurrent events. This delay can be added when sending a message to another process.

3. You should print all necessary information on the console for the sake of debugging and demonstration. You are expected to print the nonce and the hash values once the correct hash is computed, so we can verify that your nonce results in a hash with the correct specifications. Also, print the hash pointers in the blockchain. You can also print details such as: Committing the block, Sending proposal with ballot number N, Received acknowledgment for ballot number N, etc.

5 Failure scenarios

Your project should handle crash failures and network failures. The system should make progress as long as a majority of the nodes are alive. A leader failure should be handled as described in Paxos in the lectures.

6 Test scenarios

While developing your project, you can test it for scenarios such as: sequential operations of different processes, concurrent operations on different processes (multiple concurrent leaders), failing one or more processes and eventually bringing them back, partitioning the network and eventually fixing the network.

7 Deliverables

As part of this project, you will have two deliverables.

7.1 Deliverable 1: Mid-project evaluation

This evaluation will allow us to track your progress. We encourage you to follow the below instructions which will help you break your project down into smaller achievable tasks. For this evaluation, we expect the following:

- A simple *skeleton* of your system. This means, you should be able to demo the connections between all the clients and servers. Similar to what you did in PA1 and PA2, you will need to establish sockets between all the communicating entities. For **demonstration**, we will ask you to run all your clients and servers and ask you to pass simple messages between them. Please use print statements to allow us to see the messages being sent and received.

- There are various data structures that you will be using in this project - one each for the queue, key-value store and blockchain. We would like you to have these data structures defined in your servers. This just means that you must decide how you are going to implement these data structures. For **demonstration**, we will take a cursory glance at your source code, and also ask you some basic questions about your data structure implementation.

- You will be writing your blockchain to the file for persistency. At this stage, we would like you to figure out the format in which you plan to write the blockchain to the file. For this, you can *populate* the blockchain with some dummy data(say, 2 blocks) and then write it to a file. To ensure that you are doing this right, read from the same file, and see if you are able to *reconstruct* the blockchain and key-value store data-structure. We just need to see that you have figured out this stage. This can be done outside of the client-server interaction FOR NOW. For **demonstration**, we would like you to write some data into the blockchain(please do this before the demo begins) and write this blockchain to a file. After that, your code will read that file and *reconstruct* the blockchain.

At the end of this stage of the project, you will have all the building blocks of the project ready.

7.2 Deliverable 2: Final demo

This includes implementation of the algorithm following all the instructions mentioned in this document.

8 Evaluation

We plan to conduct the mid-project evaluation on March 3rd. This evaluation will compromise 30% of the project grade. The final demo will be on **March 16, 2021** via Zoom. Zoom details will be posted on Piazza.

9 Teams

Projects can be done in team of 2.

10 Afterword

In this project, we use Paxos as a consensus protocol to agree on the next block of operations to be added to the blockchain. Alternatively, we could have used a blockchain protocol as a consensus mechanism.

Both Paxos and Blockchain provide a way to replicate the log (in Paxos) or a chain of blocks (in blockchain) across all the participating nodes. Paxos provides a way to obtain consensus on a value for *one* entry in a replicated log using communication and quorums. Blockchain, on the other hand, obtains consensus on a block using computational power as in Proof of Work. (You will learn more about Blockchain later in the class). Blockchain is usually built on top of a network where the nodes do not trust each other. But we will simplify the problem by assuming trust amongst the nodes in this project. Finally, in blockchain, nobody trusts anybody else, while in Paxos, nodes are trusted, and failures are all assumed to be crash.