CSCE678/ECEN757 (Summer 2024)

Lab 1: Key-Value Store

This lab is adapted from MIT's 6.5840: Distributed Systems course. The original labs are written in Go. We will use Python to implement this lab instead.

Introduction

In this lab, you will build a key/value server that ensures that each operation is executed exactly once despite network failures and that the operations are linearizable.

Clients can send three different RPCs to the key/value server: `put(key, value)`, `append(key, arg)`, and `get(key)`. The server(s) maintains an in-memory map of key/value pairs. Keys and values are strings. `put(key, value)` installs or replaces the value for a particular key in the map, `append(key, arg)` appends arg to the key's value *and* returns the old value, and `get(key)` fetches the current value for the key. A `get` for a non-existent key should return an empty string. An `append` to a non-existent key should act as if the existing value were a zero-length string. Each client talks to the server through a `Clerk` with put/append/get methods. A `Clerk` manages RPC interactions with the server.

Your server must arrange that application calls to `Clerk` Get/Put/Append methods be linearizable. If client requests aren't concurrent, each client Get/Put/Append call should observe the modifications to the state implied by the preceding sequence of calls. For concurrent calls, the return values and final state must be the same as if the operations had executed one at a time in some order. Calls are concurrent if they overlap in time: for example, if client X calls `Clerk.put()`, and client Y calls `Clerk.append()`, and then client X's call returns. A call must observe the effects of all calls that have completed before the call starts.

Getting Started

Go to this Github Classroom and click "Accept this assignment". You will get a private repository which you can use to push and submit your code. You can use any platform to work on the project, but you will need at least Python 3.8 to run the code.

If you are not familiar with running Python in the commend line (either on Linux, MacOS, or the Windows' Subsystem for Linux), we recommend using PyCharm (the community version) as the IDE to develop the code.

We supply you with skeleton code and tests. You need to modify `client.py` and `server.py`.

To get up and running, execute the following commands.

```
$ git clone <your private Github URL>
$ cd <your repo name>
$ python3 -m unittest test_test.*
...
$
```

Key/value server with no network failures (easy)

TASK

Your first task is to implement a solution that works when there are no dropped messages.

You'll need to add RPC-sending code to the Clerk put/append/get methods in `client.py`, and implement `put`, `append()` and `get()` RPC handlers in `server.py`.

You have completed this task when you pass the first two tests in the test suite: "one client (`TestBasic`)" and "many clients (`TestConcurrent`)".

Key/value server with dropped messages (easy)

Now you should modify your solution to continue in the face of dropped messages (e.g., RPC requests and RPC replies). If a message was lost, then the client's `ck.server.call()` will raise a `TimeoutError` (more precisely, `call()` waits for a reply message for a timeout interval, and then raises a `TimeoutError` if no reply arrives within that time). One problem you'll face is that a `Clerk` may have to send an RPC multiple times until it succeeds. Each call to `Clerk.put()` or `Clerk.append()`, however, should result in just a *single* execution, so you will have to ensure that the re-send doesn't result in the server executing the request twice.

TASK

Add code to `Clerk` to retry if doesn't receive a reply, and to `server.py` to filter duplicates if the operation requires it.

You have completed this task when you pass the next two tests in the test suite: “unreliable net, one client (`TestUnreliable`)" and "unreliable net, many clients, one key (`TestUnreliableOneKey`)".

- **Hint:** You will need to uniquely identify client operations to ensure that the key/value server executes each one just once.
- **Hint:** You will have to think carefully about what state the server must maintain for handling duplicate `get()`, `put()`, and `append()` requests, if any at all.
- **Hint:** Your scheme for duplicate detection should free server memory quickly, for example by having each RPC imply that the client has seen the reply for its previous RPC. It's OK to assume that a client will make only one call into a Clerk at a time.

Key/value cluster with static sharding (moderate)

In this lab, you'll build a key/value storage system that "shards," or partitions, the keys over a set of replica groups. A shard is a subset of the key/value pairs; for example, all the keys starting with "a" might be one shard, all the keys starting with "b" another, etc. The reason for sharding is performance. Each replica group handles puts and gets for one shard, and the groups operate in parallel; thus total system throughput (puts and gets per unit time) increases in proportion to the number of groups.

We use a static sharding scheme similar to **Dynamo**:

- In total of N servers will manage N shards (i.e., partitions) of the entire key range.
- The 1st server will store the first 1/N of the key/value pairs, and the 2nd server will store the second 1/N of the key/value pairs, etc.
- The server will replicate its key/value pairs to the next (R - 1) servers.

TASK

Add code to `Clerk` and `KVServer` to handle the sharding and replication of data.

You have completed this task when you pass the last three tests in the test suite: “static 3-way sharding (`TestStaticShards`)", "rejection by one shard (`TestRejection`)", and “unreliable net, sharded, many clients (`TestUnreliableShards`)”. Note that the last test (`TestRejection`) will pass if you didn’t implement the sharding at all, but you need to make it pass with the two other tests.

- **Hint:** To ensure the even distribution of the k/v pairs across the shards, don’t use Python’s built-in hash() or hashlib functions to determine which shards to send to. Just use int(key) % nshards to determine the shards.

- **Hint:** The Clerk must alternate through the replica servers for the shard if it cannot receive response or timeout. For example, if there are 5 shards and 3 replicas, and the key that is being called put() is within the range of the 1st shard, then the Clerk will try to 1st server, and then the 2nd server, and then finally the 3rd server, if it does not receive response from the previous one.
- **Hint:** To pass the last test (TestUnreliableShards), you will need to implement linearizability among all the replicas. The easiest way is to forward all the operations through the first server responsible for the shard to order the operations.

Submission

There's no need to submit this project through Canvas or Microsoft Teams. We will grade the latest commits in your repository.

If you want to use any free late day (each person has up to 3 free late days for the whole semester), you must create a file named `late.txt` in the root of the repository and put the number of free late days you want to use in the file.

If you have used any online resources or Generative AI tools (like ChatGPT), you must provide the references using the comments in the source code.