P1: Client/Server Key-Value Store

Published Assign To Edit ⋮

Overview

This project is the first in a series of projects where we practice distributed system coding and build **MadKV (https://github.com/josehu07/madkv)**, a distributed, replicated, fault-tolerant, and performant key-value store in steps.

In the first step, we will build a straightforward key-value store service and create proper clients for it. We will get familiar with the template codebase, implement a client/server architecture, and exercise remote procedure calls.

The GitHub repo of the starting codebase can be found at: **https://github.com/josehu07/madkv (https://github.com/josehu07/madkv)**. Please follow the README for details about the repo structure, requirements, and testing utilities. All actions relevant to grading should be made invocable through `Justfile` (https://github.com/casey/just) recipes (some of which you will need to fill out in the project-level `Justfile` to surface your own code). For example:

```
just [module]  # list available recipes
just tree
just p1::build
just p1::server [listen_addr]
just p1::test1 [server_addr]
just p1::kill
...
```

Project Groups Policy

You can work on this project with exactly one other person. You should find a project partner by **Wednesday, February 5th.**

Once you've finalized your project partner, place yourselves in the same "Project 1" Group (which we have defined) in Canvas; if you are unable to find a project partner, just leave yourself unassigned and let us know -- we'll match you with another unassigned student.

You are welcome to talk with others in the course about the project. You may use piazza to ask any level of question.

Key-Value Store

Key-value stores (https://en.wikipedia.org/wiki/Key%E2%80%93value_database) (KV stores) are a ubiquitous form of storage in the cloud computing era. Large-scale **object store (https://aws.amazon.com/s3/)** s and **NoSQL database (https://www.scylladb.com/)** s have become the norm of cloud storage. KV stores are also building blocks of more complex forms of storage services, such as **distributed file system (https://ceph.io/en/)** s and **relational database (https://www.foundationdb.org/)** s.

Think of a KV store as a hashmap: clients `Put` key-value pairs in, and `Get` values out by querying with keys. We may call the key-value entries as *objects*, where *keys* are their unique names and *values* are their content. To additionally support key-range-based `Scan`s, the KV store must also efficiently maintain some sorted order of the stored keys.

Durability

In Project 1, assume an in-memory KV store with no durability and memory pressure requirements. We will take care of durability and persistent storage in the next steps.

Consistency

We expect a *linearizable* KV store service out of this project. We will cover ordering and consistency models in later lectures; for now, just assume the most straightforward consistency level: linearizability. That is, the server *appears* to all clients as a single-threaded entity that serves requests one by one. There exists a global sequential ordering of all the completed requests, where each request must be ordered after all the requests acknowledged before it was issued in real-time.

Don't worry if the paragraph above makes little sense to you now. For this project, it is totally fine to make the server simply single-threaded, serving client requests one after another synchronously. Doing so trivially implements linearizability. Our fuzz tester for Project 1 also actually checks for a consistency level that's less strict than linearizability (~= RT causal).

Client/Server Architecture

You will implement both the KV store server and its clients. The *server* in Project 1 is a single long-running process that maintains in-memory state and exposes an `ip:port` to clients. Each *client* is a separate process that connects to the server and makes **remote procedure calls (https://en.wikipedia.org/wiki/Remote_procedure_call)** (RPCs) to the server to complete key-value workloads.

Client/Server APIs

We expect the KV store server to provide the following basic APIs to its clients:

- `Put`(key, new_value) -> key found or not
- `Swap`(key, new_value) -> old_value if found, else null
- `Get`(key) -> value if found, else null
- `Scan`(start_key, end_key) -> ordered list of key-value pairs in range, inclusive
- `Delete`(key) -> key found or not

It is safe to assume that all keys and values are valid ASCII **alphanumeric (https://en.wikipedia.org/wiki/Alphanumericals)** strings, case sensitive. For Scan, keys should be sorted in **dictionary order (https://en.wikipedia.org/wiki/Lexicographic_order)**.

The implementation details of these APIs are up to you; you may design more or fewer RPC types with various arguments to support these operations. You will showcase your RPC specification in the report as part of the grading.

Language and Libraries of Your Choice

We impose no hard requirements on the programming language you choose to implement your server and clients in; feel free to pick your favorite one. It just has to be a compiled system programming language: Rust, Go, C/C++, Java, Swift, C#, or Zig, otherwise performance would be unacceptable. The instructors are more familiar with Rust, Go, and C/C++ than other languages; we recommend them if you want maximum help from us, but feel free to choose your favorite one.

Communication between the server and clients should be done through RPCs. Feel free to define your own internal RPC specifications and choose your favorite RPC library: **gRPC** (https://grpc.io/), **Cap'n Proto** (https://capnproto.org/), Apache **Thrift** (https://thrift.apache.org/), **tarpc** (https://docs.rs/tarpc/latest/tarpc/), or another. You don't need to consider secure connections (https, etc.). Async programming paradigms are welcome.

You may bring in other auxiliary libraries as you see fit, as long as they are readily packaged upon submission. For example, a B-tree-like data structure could be useful. You should NOT use any high-level dependencies that implement an entire key-value store or some critical distributed communication functionalities for you; those defeat the purpose of the project. When in doubt, please don't hesitate to ask.

Functionality Testing

Testing is a crucial part of distributed systems engineering. We won't impose restrictions on implementation details of the server and clients. Instead, we just require a few lines of shell scripts (to be filled in a `Justfile` (https://github.com/casey/just)) to launch your server process and client executables. We will run tests and benchmarks on top of these invocation recipes. Check out the codebase template for details.

Your Own Testcases

You will create at least 5 of your own testcases and showcase their outputs:

- 2 cases of a single client each, covering all operation types
- 1 case of multiple clients issuing requests to non-conflicting keys
- 2 cases of multiple clients issuing interfering requests to overlapping keys, to show that clients observe acknowledged operations by other clients

The testcase clients could be, for example, separate binaries based on a shared KV client library. Or, they could use the same binary that takes workload inputs in some configuration format of your choice. Feel free to create other client modes if helpful; for example, you may create your own fuzz tester or an interactive REPL-style client.

Stdin/out Interface for Automated Workloads

To make the project more interesting and to ease bulk testing, we need a way to drive the clients with automated workloads. To achieve this, we require a way to invoke your KV client such that it listens on standard input (*stdin*) and interprets each line of input as a KV API call. The inputs have the following format:

```
PUT <keyabc> <valuexyz>
SWAP <keyabc> <valuexyz>
```

```
GET <keyabc>
DELETE <keyabc>
SCAN <key123> <key456>
STOP  # stop reading stdin, exit
```

In this mode, the client should promptly print to standard output (*stdout*) the result of each API call ending with a newline, in the input order, in the following format:

```
PUT <keyabc> found
PUT <keyabc> not_found
SWAP <keyabc> <old_valuexyz>
SWAP <keyabc> null
GET <keyabc> <valuexyz>
GET <keyabc> null
DELETE <keyabc> found
DELETE <keyabc> not_found
SCAN <key123> <key456> BEGIN
  <key127> <valueaaa>
  <key299> <valuebbb>
  <key456> <valueccc>
SCAN END
STOP  # confirm STOP before exit
```

All keywords are case-sensitive (`PUT`, `SCAN`, `BEGIN`, ..., and `found`, `null`, ...); all keys and values are valid ASCII alphanumeric, case-sensitive strings.

We provide testing utilities in the starting codebase that will drive your client with automated workloads in this way. We also provide a (functionally incorrect) client implementation to demonstrate input/output parsing in this format in Rust, just for reference. Our fuzz tester and benchmarking driver rely on this stdin/out interface to test and measure your KV store system in Linux (Ubuntu 22.04) environments.

Performance Benchmarking

A good KV store is not only functionally correct but also performant. To measure the performance and scalability of your KV store system, we use **YCSB** (https://github.com/brianfrankcooper/YCSB), the de-facto standard of key-value benchmark workloads. We provide a YCSB adapter that translates YCSB workloads into the aforementioned stdin/out format, so you don't have to connect YCSB to your system by yourself. However, we highly recommend you take a look at the **default workload profiles** (https://github.com/brianfrankcooper/YCSB/tree/master/workloads) (A to F) and understand their meanings (but ignore the transactional workload). The benchmark gives us a bunch of tunable knobs other than workload splits, but the default values are good enough for us.

As part of the deliverables, you will report single-client performance results under workloads A-F, as well as multi-client scaling results under workloads A, C, and E. Our report generation script will tell you which configurations to run before plotting. Make sure you understand the benchmarking results and comment with your observations.

CloudLab Instance

We will grade this project on two **CloudLab** (https://www.cloudlab.us/) instances (one running the server and the other running all client processes) with >= 20 CPU cores and >= 128GB memory each, running Ubuntu 22.04. We recommend that you develop in a similar environment. It is probably a good idea to first develop on a single machine using local addresses, and then validate and benchmark on distributed machines.

Deliverables & Grading

You will deliver both your code and a short report at the end of this project. We will grade the project by having each group do a short (10 min) demo with us, where you walk through your code and report, and check the grading bullet points with us one by one.

Archived Codebase

Bundle your codebase into a single \`tar.gz\` archive to submit it through Canvas. When archiving, run the following command at the root level of your repo (i.e., where you find `README.md` and the top-level `Justfile`).

```
tar --exclude-vcs -zcvf <somename>.tar.gz ./*
```

Verify your archive by untaring it with:

```
mkdir -p ../testrepo
mv <somename>.tar.gz ../testrepo/
cd ../testrepo
tar -zxvf <somename>.tar.gz
ls  # should see README.md
```

Project Report

The code will be graded together with a short report PDF, which should contain the following information. Please be concise when explaining your design or commenting on your testcases and benchmark results. Restrict each point to a few sentences max.

- Design walkthrough [**15%**]
  - Code structure [5%]
  - Server design [5%]
  - Explain your RPC protocol between server and clients [5%]
- Your own testcases & explanation [**25%**]
  - Single client, 2 cases [10%]
  - Concurrent clients, no conflicts, 1 case [5%]
  - Concurrent interfering clients, 2 cases [10%]
  - Show your testcases, explain their outputs, and discuss their coverage
- Fuzz testing [**30%**]
  - Single client [10%]
  - Concurrent clients, no conflicts [10%]
  - Concurrent interfering clients [10%]
- Performance benchmarking & explanation [**30%**]
  - Report YCSB A-F workloads performance [15%]

    - Report YCSB A, C, and E workloads scalability results [15%]
    - Comment on the results: why are trends different or similar? do you see a scalability bottleneck?
    - The absolute performance numbers are irrelevant and we won't grade based on that. Focus on patterns and observations
- Any bonus content [**+10%** max]
    - For example, additional clients, RPC streaming optimizations, testing, benchmarking, and/or interesting observations

We provide utilities that help run everything and generate a template Markdown report from your code. Please fill the report with text and comments at proper markers. You may submit the Markdown source file (in this case also submit the plot images), or render it into a PDF (recommended) and submit the PDF instead. Your comments should reflect your code *faithfully* and *comprehensively*.

We will grade the project by having you do a short (10 min) demo with us, where you show us your code running live, walk through your report, present your test & benchmark results, and explain your observations.

Points	100
Submitting	a file upload
File Types	gz, md, pdf, and png

Due	For	Available from	Until
Feb 20	Everyone	-	-

+ Rubric