P2: Durability & Keyspace Partitioning

Published | Assign To | **Edit**

Overview

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

In this step, we will build upon our Project 1 solution and add two critical features: durability and partitioning.

- A server node should store its necessary states *durably* in persistent storage instead of keeping everything in volatile memory. After a software failure and a re-launch, the server should be able to read the durably stored state from the machine and recover from it.
- The key-value store service should be composed of multiple server nodes, each responsible for serving requests on a *partition* of the keyspace (i.e., a disjoint subset of the keys). This distributes client load across multiple machines, enhancing throughput/latency performance and improving scalability.

You will use your Project 1 solution as the starting point. In case we have made updates to the project template, pull our updates to the `main` branch from the GitHub repo: **https://github.com/josehu07/madkv (https://github.com/josehu07/madkv)**, and merge into your development branch. See the README for instructions.

As usual, all actions relevant to grading should be made invocable through **Just (https://github.com/casey/just)** recipes. You will need to fill out the blanks in the project-level `Justfile` to surface your own code.

Project Groups Policy

You can work on this project with exactly one other person. You can work with the same person as you did for Project 1 or someone new. Just be sure you communicate these decisions clearly with your previous project partner!

Once you've finalized your project partner, place yourselves in the same "Project 2" Group (which we have defined) in Canvas.

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

Durability

In Project 1, we allowed the server to store all its states in volatile memory. This is dangerous, because if the server crashes at any point (due to software bugs, hardware failures, or operator errors), we lose everything in the KV store.

In this project, our first task is to add persistent storage to each server process. When receiving a command that *mutates* state (i.e., is not *read-only*), the server should store necessary information durably to disk before acknowledging the client.

What information to persist is up to you and will be a key part of your design. Here are some example design choices:

- You could make the key-value map itself persistent and maintain nothing else in memory. (This is easy to implement, but hard to extend to incorporate replication for the next project.)
- You could maintain a key-value map on disk that is always kept in sync with an in-memory copy (analogous to write-through-style caching).
- You could model the server as a **state machine** and durably log the incoming commands, instead of persisting the state itself. (This is the recommended approach, because it will make your life easier when we ask you to practice *consensus* and *state machine replication* in the next project.)

Storage Backend

The choice of persistent storage backend is totally up to you. You are very welcome (and recommended) to incorporate your favorite local storage library, such as **LevelDB (https://github.com/google/leveldb)**, **RocksDB (https://github.com/facebook/rocksdb)**, **Sqlite (https://www.sqlite.org/)**, or **DuckDB (https://duckdb.org/)**. Most of these libraries have bindings for all system programming languages.

You may instead write your own durable logging I/O logic on plain files. Getting this 100% right will be tricky: the durable state of the files must be kept consistent across crashes. In other words, an on-the-fly operation at the time of a crash either must leave no partial/dirty states on durable storage, or must let the restart procedure detect and discard them. This requires knowledge out of the scope of this course.

What is ultimately required is that each server process accept an argument which is a path to a directory. All durable states (no matter which library you use) must be stored under that directory.

Don't worry about an unbounded growth of durable state size. You can assume that we will never run out of memory and disk space. Enabling garbage collection (e.g., through snapshots) is a hard problem and we don't require solving it in mandatory projects.

(De-)Serialization

One problem you will practice solving here is the serialization/deserialization between your in-memory language-specific data structures and their compact bytes representation on durable storage. Feel free to find and use a good format & library to do this for you. Examples include lower-level (but more verbose) **Protobuf (https://protobuf.dev/)**, **FlatBuffers (https://flatbuffers.dev/)**, or **Cap'n Proto (https://capnproto.org/)**, and higher-level (format-flexible but language-specific) **Cereal (https://uscilab.github.io/cereal/)**, **Serde (https://serde.rs/)**, or **Mus-go (https://github.com/mus-format/mus-go)**.

Serialization/Deserialization is a common step of storage and network I/O in distributed systems. For network messages, your RPC library likely already handles this for you. The durability task lets you practice this a bit more.

Server Recovery

When a server process failed but the physical machine survived, we should allow re-launching a new server process on that machine with the same parameters to resume the service. In particular, if a server process is started with a path to a non-empty directory, it should recover its state from whatever is in that directory before resuming service to clients.

All KV requests that have been acknowledged (i.e., replied to clients) must be recoverable from the durable files. Any on-the-fly requests (i.e., those not yet acknowledged to clients) at the time of a crash may or may not have been durable stored. This is fine because we require clients to retry failed operations and broken connections indefinitely to the same address until success (details below), so it does not break **idempotency** (https://en.wikipedia.org/wiki/Idempotence) .

In this project, don't worry about the unavailability window before the restart, and don't worry about cases where the machine becomes inaccessible after the crash. We will address these issues in Project 3.

Keyspace Partitioning

The second part of this project involves scaling the KV service *out* to multiple servers. The way we do so is to partition the keyspace into disjoint subsets, distribute them to the servers, and let each server take care of its assigned subset(s).

We assume a fixed, known number of servers when launching the KV store service. Also, assume the number of servers is no larger than 50 (so decisions could be based on even a single ASCII character); in our tests and benchmarks, we will use a much smaller number than that.

Given the fixed number of servers, the strategy of partitioning is up to you. It could be a static, even partitioning of the keyspace. It could be a simple hashing-based solution. You may also opt for fancier algorithms such as **consistent hashing** (https://en.wikipedia.org/wiki/Consistent_hashing) or dynamic load balancing.

Cluster Manager

How do servers know which partition(s) they are in charge of? How do clients know which server(s) to contact for a request? These are hard questions in general distributed storage systems. Although advanced configuration management designs exist, let's take an easy approach: introducing a cluster *manager* node.

The manager is the first entity launched in the KV store service, before any server. It takes an argument that is the expected number of servers in the cluster. Partitioning roughly works according to the following procedure:

- First, the manager is launched with an argument which is a *comma-separated list* of the servers' public addresses.
  - For example: `./yourmanager --man_listen 0.0.0.0:3666 --servers 1.2.3.4:3777,5.6.7.8:3778,9.10.11.12:3779`
  - Argument names are not important; fill in the just recipes to connect our recipe parameters to your executable invocation commands.
  - The manager decides how to partition the keyspace, and listens for servers' registrations and clients' inquiries.

- Then, all servers are launched with an argument that specifies the address of the manager and another argument that specifies a unique server ID starting from zero (0, 1, 2, ...).
  - For example: `./yourserver --manager_addr 1.2.3.4:3666 --api_listen 0.0.0.0:3778 --server_id 1 --backer_path ./backer.1`
  - The first thing a server does is query the manager for its partition assignment information (e.g., a partition ID and the start/end range of keys, or a hash function seed, etc.). Let's say the server *registers* itself with the manager.
- Finally, clients are run with an argument that specifies the address of the manager (instead of the address of a server).
  - For example: `./yourclient --manager_addr 1.2.3.4:3666`
  - The first thing a client does is query the manager for the list of partitions and the server IDs they are assigned to. The client remembers this information, connects to all the servers, and uses the partitioning information to make KV requests to proper servers.

Since clients contact the manager only once upon running, the manager does not become a performance bottleneck if we assume a modest number of long-running clients. However, you can easily see how the manager could become a single point of failure in the system. For this project, assume that the manager is always up and never fails. We will add fault tolerance in a later project.

Error Handling

Your client/server/manager implementation should have decent error handling capability.

Whenever a KV store component `A` is establishing connection to another component `B` but fails due to `B`'s unresponsiveness, `A` should implement an *indefinite retry* logic spaced by some *wait time*. Cases include:

- server -> manager when manager is not up yet
- client -> manager when manager is not up yet or when not all servers have been registered yet
- client -> a server when that server is unresponsive

When a client KV operation fails due to a server crash causing client-server connection to break (or when the operation is taking too long to complete and throws a timeout error), the client should handle the error gracefully. Retry indefinitely to repeat the operation or to re-establish the connection.

You should NOT use any high-level libraries that implement a partitioned KV store for you; those defeat the purpose of the project. When in doubt, please don't hesitate to ask.

Consistency of Scans

Given that each server individually implements *linearizability*, partitioning the keyspace into static, fixed, disjoint subsets will not affect the consistency of single-key operations: `Put`, `Swap`, `Get`, and `Delete` (why? think about this for a bit). However, `Scan`s are different in that they touch a range of keys, possibly scattered across multiple servers. Making `Scan`s linearizable w.r.t. each other and w.r.t. all other operations requires knowledge about **transaction processing** (https://en.wikipedia.org/wiki/Transaction_processing) and is out of the scope of this project.

We relax the consistency requirement on `Scan`s. Each key-value pair in the scan result just has to come from some real-time-preserving point in the history of the server responsible for that key. In other words, each key in the scan result is checked individually, and the `Scan` operation as a whole does not have to appear "atomic".

Correctness Testing

You will craft one deterministic testcase (see below for requirements) and also run our fuzz tester with some required configurations.

Your Own Testcase

The implementation work for this project is rather heavy, so we will only ask you to handcraft one deterministic testcase. This testcase should include the following list of actions:

1. Launch a partitioned cluster with at least 3 servers
2. Do a bunch of `Put`s and `Swap`s
3. Do a bunch of `Get`s and `Scan`s, all of which should succeed
4. Kill one of the servers
5. Do a `Get` and a `Scan` on unaffected partitions; both should succeed
6. Do a `Get` or a `Scan` that involves the failed partition; should timeout
7. Restart a server process for the failed partition
8. Do a `Get` to an existing key in the just-recovered partition; should succeed and return its latest put value

A fully automated recipe for this testcase is appreciated but not mandatory. You just need to be able to demonstrate this testcase during the in-person demo.

Fuzz Testing

We will reuse the fuzz tester utility for fuzz testing. This time, we always run 5 clients with conflicting keys. We ask you to manually set up the following three scenarios on the service side:

- Launch 3 partitions (i.e., do `just p2::service m ...`, `just p2::service s0 ...`, `just p2::service s1 ...`, and `just p2::service s2 ...`), then run the fuzzer till completion.
- Similarly, launch 3 partitions then run the fuzzer, but this time kill server 1 half way through the test. The fuzzer should soon hang if your client implements retries correctly as required. Then, re-launch server 1 and observe that the test continues till completion.
- Repeat the same test as above, this time with 5 partitions, and crashing server 1 and 2 half way, then re-launching both.

Make sure to empty the durable storage directory before starting the service, so that it starts from a fresh, empty state.

CloudLab has limited resources, so you may find it hard to start experiments with a lot of machines. It is totally fine to run testing & benchmarking on a single machine (or have some servers share the same machine). In the report, write clearly the physical topology you used in testing and benchmarking.

Performance Benchmarking

Similar to project 1, we will run **YCSB** (https://github.com/brianfrankcooper/YCSB) benchmarks through our benchmarker utility and report some performance trends. Focus on relative trends across your own results; the absolute numbers may vary due to various factors (physical instance type, implementation language, etc.) and are not important. We recommend CloudLab instance types with >= 20 CPU cores, >= 128GB memory, and a decent disk, running Ubuntu 22.04.

As part of the deliverables, you will report the following results:

- 10 clients on all workloads A to F, varying the number of partition servers across 1, 3, and 5
- workload A with 1 or 5 partitions, scaling the number of clients from 1 to 30

Our report generation script will list all configurations needed for plotting. Make sure you understand the benchmarking results and comment with your observations.

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 testing and benchmarking results. Restrict each point to a few sentences max.

- Design walkthrough [**30%**]
  - Code components structure [6%]
  - Durability/recovery design & implementation [8%]
  - Partitioning design & manager implementation [8%]
  - Error and timeout handling [8%]
- Your own testcase [**10%**]

    - One testcase according to the description [10%]
- Fuzz testing [**30%**]
    - 3 partitions, no crashing [10%]
    - 3 partitions, with crash/recovery of server 1 [10%]
    - 5 partitions, with crash/recovery of server 1,2 [10%]
- YCSB benchmarking & explanation [**30%**]
    - 10 clients across workloads A to F with 1, 3, 5 partitions [15%]
    - Workload A with 1 or 5 partitions, scaling #clients [15%]
- Any bonus content [**+20%** max]
    - For example, fine-grained partitioning, dynamic load balancing, unit tests of your code, additional testcases or benchmarking results, or other interesting observations
    - Additional scripts on top of `just p2::service` and other recipes to help you launch service and do benchmarking

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
Mar 18 at 3pm	Everyone	-	-

+ **Rubric**