**Project 3: Distributed Key-Value Store**

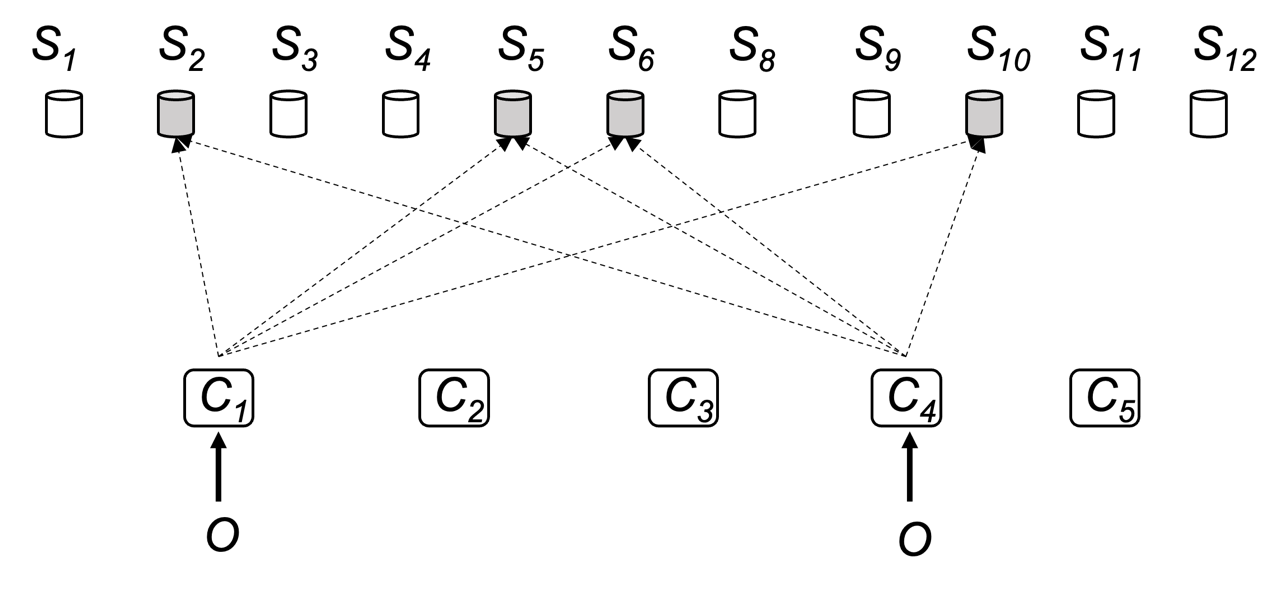
CSC 6712 Distributed Storage Systems

**Overview**

You're going to expand your clients to support connections to multiple instances of your server. They will both distribute and replicate the data across the servers. Clients will use a common set of rules (based on hashing the keys) for assigning keys to servers so that two clients can access the same key on the same server without needing to communicate with each other. Note that the client will only support the case where servers are removed, not added.

**Part I: Distributing and Replicating Data**

You will modify the client to maintain open connections to multiple servers and use [rendezvous hashing](https://en.wikipedia.org/wiki/Rendezvous_hashing) to distribute and replicate keys across multiple servers. The distributed component allows the key-value database to store more data than can fit on a single server by distributing the data across multiple servers. The main problem with only distributing keys is that it offers no improvement in reliability over a single server. In fact, the reliability is worse because the failure rate goes up as the number of servers increases. To resolve this, we're going to introduce replication as well. The key set will still be partitioned across servers while replicating individual keys to multiple servers.



Modify the client so that it takes a replication factor and list of server IP addresses and their ports (e.g., 192.168.1.2:25125). For example, the connect the client to three servers running on the same computer (but different ports) and a replication factor of 1:

$ ./client 1 127.0.0.1:25120 127.0.0.1:25121 127.0.0.1:25122

Keys can be assigned to servers like so:

1. Concatenate the key with each server address (e.g., "xxxxxxx127.0.0.1:25120", "xxxxxxx127.0.0.1:25121", "xxxxxxx127.0.0.1:25122")

2. Hash each key-server string using [MurmurHash32](https://en.wikipedia.org/wiki/MurmurHash) (use a library, do not implement yourself).

3. Sort the server addresses by their hash values

4. For a replication factor R, take the R first server addresses

For a write, issue PUT commands to each of the R servers. For a read, randomly pick one of the R servers and perform the GET operation.

Your client should handle the graceful shutdown of one of the server connections. If the connection is closed, the client should remove the server from its list but continue operations as normal.

Hints:

* When testing your implementation, you should ensure that each of the M servers has approximate faction of 1/M of the keys when using a replication factor R = 1.
* If you have a replication factor of R, you should be able to kill R – 1 servers without losing data.

**Part II: Benchmarks**

You should design and perform benchmarks to evaluate the following:

1. Impact of replication read and write performance of a single client with varying numbers of servers (e.g., 1 – 10) and replication number (e.g., 1 – 5).

2. Scaling of read performance as the number of clients and servers are changed.

**Part III: Written Report**

You should write up a several page report that evaluates your overall distributed system. The report should address the following items:

1. Design and implementation of your benchmarks.

2. Benchmark results and their interpretation to how distribution and replication impact scaling of data sizes, the number of clients that can be supported, and read and write performance of individual clients.

3. Ways in which your solution can fail and cause data lose or inconsistency. As part of this, determine if your solution is AP or CP according to the CAP theorem.

4. Describe ways you might mitigate some of the risks identified in (3).

**Submission Instructions**

Create a new branch to capture the state of your GitHub repository at submission time. Submit a link to the GitHub repository and your report (in PDF format) on Canvas.

**Rubric**

|  |  |
| --- | --- |
| **GitHub Repository:** Has a README.md file with instructions sufficient for building and running the code. | 5% |
| **Distributed / Replicated Client:** Implementation is functionally correct. Rendezvous hashing is implemented correctly. PUT writes to all replicas. GET grabs from a random replica. | 40% |
| **Written Report (Overall):** Report is written in a professional manner using proper grammar and spelling. Report is a useful standalone document that can be shared with another student. Report contains a detailed description of the implementation, experimental designs for benchmarks, benchmark results, and interpretations of the results. | 10% |
| **Plots:** Appropriate types of plots were chosen for each analysis. Axes are properly labeled. Used legends if appropriate. Chose appropriate axis limits to make plots readable and avoid misleading interpretations. Font sizes are legible. Figures are saved at high resolutions. | 5% |
| **Benchmarks:** Experimental designs are correct. Benchmark results are interpreted correctly. | 40% |