CS 442: Distributed Systems and Algorithms

Project Report

20.12.2024

Abdurrahman Bilal Kar

Cahit Ediz Civan

Kanan Zeynalov

Efe Kaan Fidancı (Withdrew)

Introduction

In this final report, we present the design, implementation, and evaluation of a distributed in-memory key-value store with disk snapshots for data persistence. The system is built to provide fast data access through in-memory storage while ensuring durability through periodic snapshots saved to disk. The system is scalable, fault-tolerant, and supports data replication across multiple key-value store nodes to handle increased load and prevent data loss during node failures. A broker server manages the routing of client requests to the appropriate key-value store nodes, ensuring dynamic load balancing and efficient use of resources.
This report outlines the system architecture, including HTTP-based communication between the broker and key-value store servers, and provides a step-by-step guide on how to set up and test the system. It also details the contributions of each team member to the project's success, including the design of the broker and key-value store endpoints, the implementation of peer recovery mechanisms, and the development of the data replication strategy to ensure minimal downtime in case of node failures.

What we promised in the proposal

In-Memory Storage with Disk Snapshots:

- ☑ Fast Access: The system will provide quick read and write operations by storing data in-memory.

  Our system does precisely that, it is an in memory key value store.
- ☑ Data Durability: It will periodically create disk snapshots to ensure data persistence in the event of system failures (e.g., power outages).

  Our system does this by periodically taking snapshots of its contents and saves it to disk. This way it ensures data persistence.

Scalability and Distributed Architecture:

- ☑ Horizontal Scalability: The system will support scaling by adding more key-value store nodes, allowing it to handle increased data and requests.

  Our system is able to register new key value stores on the run. This way it can scale horizontally.
- ☑ Distributed Architecture: The system will include a broker to manage and route client requests.

  Our system has one broker that is doing that. Its endpoints are stated below as "Broker HTTP Endpoints".
- ☑ Brokers will distribute incoming requests to appropriate key-value store nodes, ensuring efficient use of resources.

Our broker keeps track of the loads of the key-value store nodes by keeping track of its contents and routes new key additions to the least loaded key-value store. This way it is as scalable as the number of key-value store nodes allow.

Fault-Tolerance and Replication:

- ☑ Data Replication: The system will replicate data across multiple nodes to ensure reliability.

  Each key-value store node is replicated at two places, the place where the key-value store node is and the place where its peer node is (Except for the last added node).

  When it fails and the broker realizes that, it uses its peer node to restore from its snapshot at its peer (which is also periodically taken). This way fault tolerance is ensured.

- ☑ Master-Slave Replication: A single master node will manage data replication and maintain consistency.

  In our case, the node starting replication is the broker node.

- ☑ Minimal Downtime: The replication strategy ensures that data remains accessible even if a node fails.

  Data replication at two sites ensures that there is 0 minimal time in case of failure until one key value store remains alive.

Load Balancing:

- ☑ Dynamic Load Monitoring: Brokers will track the load on each key-value store node.

  Our broker keeps track of the loads of the key-value store nodes by keeping track of its contents dynamically. Then distributed the loads based on that.

- ☑ Workload Redistribution: The broker will dynamically route new records to nodes with lower loads, optimizing performance and avoiding bottlenecks.

  See above matter on dynamic load balancing.

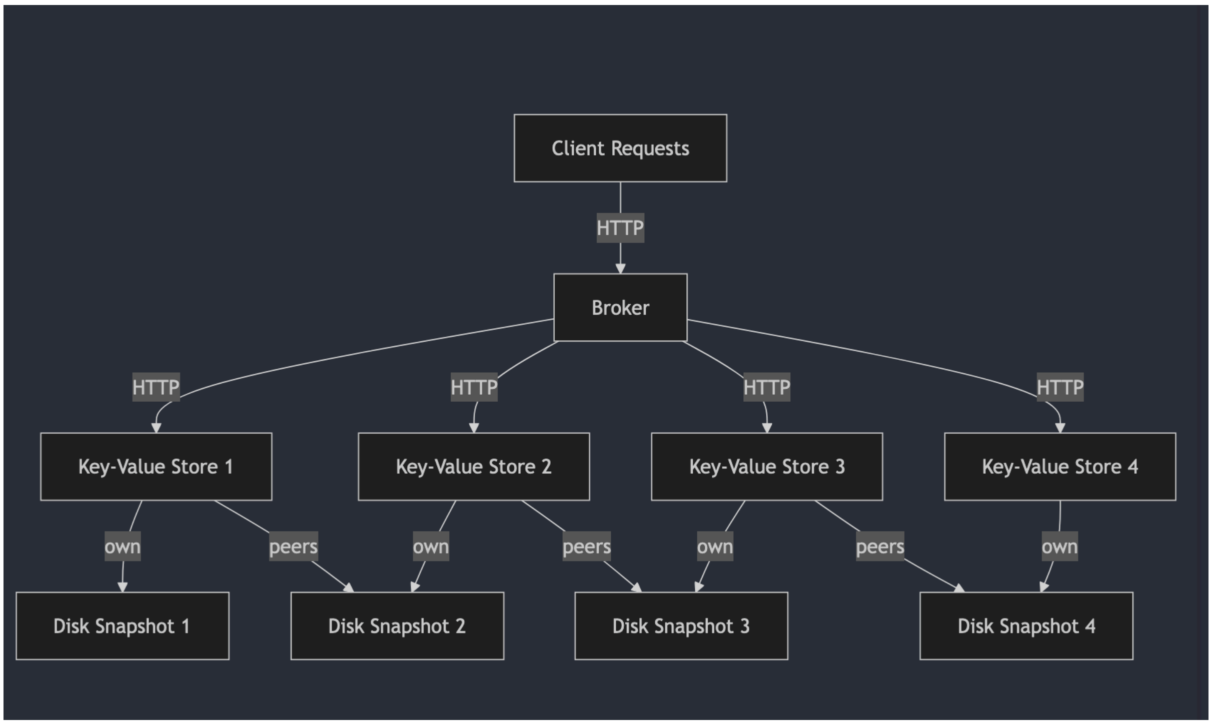
Broker HTTP Endpoints and Their Corresponding Functionalities

```
http.HandleFunc("/set", h.SetHandler)
http.HandleFunc("/get", h.GetHandler)
http.HandleFunc("/getall", h.GetAllHandler)
http.HandleFunc("/kvstore/snapshot/manual", h.ManualSnapshotHandler)
http.HandleFunc("/stores/list", h.ListStoresHandler)
http.HandleFunc("/delete", h.DeleteHandler)
http.HandleFunc("/register", h.RegisterHandler)
```

1. /set: Handles client requests to add or update a key-value pair in the distributed system.
2. /get: Retrieves the value associated with a given key. If it is not found, it returns not found. Note that it also triggers peer-backup, since while iterating kv stores to find the key, it may realize that a kv store is dead, therefore automatically notifying its peer to restore the peer into itself.
3. /getall: Returns all key-value pairs stored across all key-value store nodes.
4. /kvstore/snapshot/manual: Triggers a manual snapshot for data persistence. It can be used to save the data to persist in the disk without waiting for the automatic snapshot to do it.
5. /stores/list: Lists all active key-value store nodes registered with the broker.
6. /delete: Removes a key-value pair from the distributed system.
7. /register: Registers a new key-value store node with the broker for dynamic scaling. Not for direct HTTP call, but rather automatically called while creating a kv store.

You can read the Readme file and look for curl commands to use these endpoints. You can also find a copy of the code as well as the Readme file at https://github.com/KananZeynalov/Distributed-Key-Value-Storage/

System Architecture



How to run?

1. Run the broker: *go run servermain/main.go*
2. Run *export BROKER_URL="http://localhost:8080/register"* command. This is used so that newly created kv stores are registered at broker automatically on the run.
   For Windows: $env:BROKER_URL="http://localhost:8080/register"
   For Mac/Linux specified at the first line.
3. Create a kv store: *go run kvstoremain/kvstore_server.go store1 8081*
4. You can see that after some time, thanks to periodic snapshotting, the store creates its own snapshot automatically. Note that peer snapshotting has not started yet since there are no other kv stores to peer.
5. Set a key by sending a http POST request to http://localhost:8080/set. In the body, you need to have a json as such:

```
{
  "key": "k1",
  "value": "v1"
}
```

   After some time, you will be able to see these values in the snapshot.

6. Set another key by doing the same.
7. Retrieve the key by sending a http GET request to http://localhost:8080/get?key=k2. Check the response for the value.
8. See all keys by sending a http GET request to http://localhost:8080/getall. Check the response.
9. Now add a new kv store with a different name and a port. This will achieve three things:
    a. The system can add new nodes on the run even after operations, showing horizontal scalability.
    b. New set key requests will be forwarded to the new kv store. showing dynamic load balancing.
    c. Since now there is more than one kv store, multiple replication will begin. If the newly created kv store fails, for example, we will see that the older store will recover from failed kv stores' snapshots.
10. Set a key and check */getall* endpoint to see where it landed.
11. Set some other keys.
12. Now let's see the snapshots and observe it over the period of time where we set and delete keys.
13. Now we will test manual snapshot by sending a http POST request to http://localhost:8080/broker/snapshot/manual. This is so that we do not wait for periodic snapshots.
14. Now prepare the show time for multiple replication. We will create 4 nodes, each will hold some data. Then we will fail them one by one until one kv store remains alive and check what happens. You may need to send multiple get requests to retrieve value from a failed node since recovering may take time.

Contributions of each member

Abdurrahman Bilal Kar: In this project, I contributed to the design and implementation of the HTTP-based communication system for both the broker and key-value store servers. On the broker side, I developed endpoints for initiating manual snapshots (/kvstore/snapshot/manual), retrieving all data across stores (/getall), and listing all key-value store nodes (/stores/list). On the key-value store server, I contributed to the implementation of core functionalities such as setting and retrieving key-value pairs (/set and /get), fetching the store's name (/name), and retrieving all data within a store (/getall). Additionally, I handled peer-related operations, including receiving peer assignments (/notify), handling peer failures

by loading their data from disk (/peer-dead), and enabling inter-peer data backups (/peer-backup).
I also contributed to the implementation of the peer recovery mechanism, ensuring seamless restoration of data from disk to memory and over HTTP, thereby enhancing fault tolerance. Furthermore, I developed snapshot-related endpoints, including saving and loading data to and from disk (/save and /load) and managing periodic snapshots (/start-snapshots). My contributions extended to the overall system design, particularly in architecting HTTP interactions and designing general client functionality to ensure a robust and efficient distributed system.

Cahit Ediz Civan: Contributed to the overall system design. Contributed to the broker server and the kvstore server. Contributed to the implementation of the GetHandler endpoint in the broker server, and the DeleteKey endpoint in the kvstore server. Designed the peering logic which essentially is a doubly linked list representation of the key value stores in the broker. In this representation, each node is responsible for keeping the next node’s snapshot. Contributed to automatic key-value store and peer snapshot features.

Kanan Zeynalov: I contributed to the system architecture and coordination between the broker and key-value store servers, implementing the /register endpoint for dynamic scaling and managing load balancing for efficient data distribution. I worked on data replication to ensure fault tolerance, replicating data across two sites and handling node failure recovery through peer snapshots. Additionally, I contributed to the /delete endpoint and worked on both periodic and manual snapshot handling, ensuring data persistence. I also tested the system’s scalability and fault tolerance, verifying correct data replication and recovery during node failures. Implemented all HTTP requests between kvstore and broker by