**Cloud Storage System**

Dworschak Richard 1200121, Potuzak Bernhard 01406826

**Architecture:**

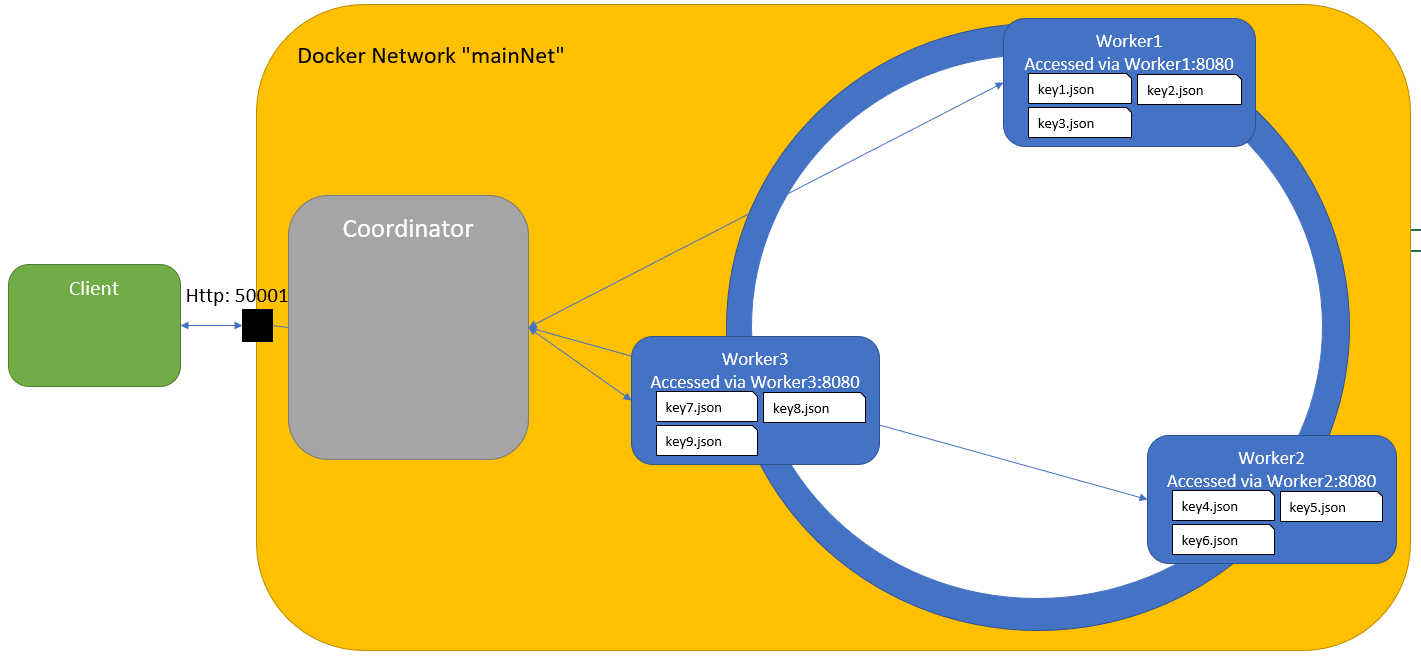
We decided on the Consistent Hashing for our System. We push Servers on a Ring like Structure and want to map the key-value pairs to the corresponding server, so it is distributed evenly. Check the Section “Hashing Method” on which Hashing Function we used and why

For our System we created three Parts:

1. Worker
2. Coordinator
3. Client

The Big Picture

The Coordinator functions as Master for our Consistent Hashing and is placed as Interceptor between the Client and the Workers. The Coordinator knows how many Workers are currently on the ring and where to redirect the Clients Request. The Workers don’t know each other and act on their own. The Workers can’t be reached from outside since all traffic needs to go through the Coordinator. For testing, the Workers are open to 10101 to 1010n where n is the number of workers on the Ring. We can access the Workers from the Client directly but only for testing.



Worker

The Worker represents the Bucket/Server of a Consistent Hashing Algorithm. It stores many files and can accept insert, delete and search Requests from outside. We decided to make the Worker a Docker image so we can create multiple containers to run separately. With that we can assure that each Bucket/Server runs on a different process and is isolated from all other Buckets/Servers. The downside of this decision is, that we can’t start the Cloud Storage System solely from the Client. Instead we start the Worker separately with a docker-compose or multiple docker runs.

Each Worker is assigned a Volume to work with which is mounted to the /shared directory. This volume represents the Bucket filled with files with a single key-value pair. The filename consists of the key of the corresponding value which is stored as JSON-Format inside the file. With that we can just check if a certain file (the key) exists within the Worker without having an in-memory array to load/persist on Worker start-up and shut-down.

The Worker itself hosts a small distribution of a tomcat server which serves on Port 8080. Each Method inside the Worker

Coordinator

The coordinator is responsible for the hashing and the right handling and forwarding of the user input sent by the client. The coordinator knows the addresses of the workers and how many are currently on the ring as well.

All traffic goes through the coordinator and are sent depending on the request on different endpoints. There are 5 endpoints available: insert, delete, search range and getStatus. Each endpoint consumes multipart form data, hashes the key from the received data and forwards to the right address, which it computes with the hashing.

For the hashing the coordinator uses FNV Hashing, for a more detailed look check the Hashing section.

Client

The Client is a Standalone Java Jar with dependencies build with Maven. Check the “How to Run” Section to see how you start it. The Client is a testing ground to check the functionality of the System. It is a console application with accepts simple user Inputs to send to the Coordinator. For a more advanced look at the client check the Section “How to Use the Client”

The Client uses multiple csv files to store a timestamp, the action used, the time it needed to compute and what the Coordinator returned from the Request. If you test single Methods the Client puts that into the file CSV\_Timestamp\_<currentTimeStamp> where the current Timestamp is when the main method is firstly called. With this we can have separate logs for multiple client restarts.

**Hashing Function**

**How to Setup:**

Worker:

The Worker run as Docker Containers on seperate processes to simulate a distributed System. To start up the System you need to follow these Steps

1. Create a Network named "mainNet" with docker network create mainNet
2. You can startup individual docker images by:
   1. Choosing an unique identifier for the Docker Image
   2. Create a volume with whatever you wanna call it (You can use the id in the name)
   3. Build the Docker Image with docker build -t RepoName directory (This can take awhile since its download the Maven Dependencys aswell
   4. Run the Docker file with docker run --rm -v Volume\_name:/shared --name ContainerName --network mainNet -p external\_port:8080 repoName

You can also use the Docker-compose file and start the containers. Before you use the docker compose you need to build the Worker and name the image worker. The docker-compose has 4 predefined Containers with volumes attached. The Volumes are created automatically if they not exist

Coordinator:

The coordinator also runs as a Docker Container.

1. Build the coordinator
2. Start the workers first since there you also create the network on which the coordinator also runs
3. Then run the coordinator with: docker run --publish 50001:50001 --network mainNet koordinator

Client:

The Client is a Maven build executable Jar in Command Line Move to the Client Folder and package the jar with mvn clean package To execute the Jar you must at least have java 1.8 installed It creates a fat Jar that can be executed with: java -jar Client-jar-with-dependencies.jar

**Implementation**

Worker:

The Worker is implemented similar to the first task. It runs on the jersey 2.0 Implementation of JAX-RS. On Context Created it creates a Singleton Instance of Runner which is started in a separate Thread to save some of the Context for the Worker (Id, where the Volume is mounted).

The Worker listens on 5 Methods. (Four of them are POST Methods since after creating the Client I saw that I only trigger POST Methods with a single Wrapper Function and had no time to change it.)

The Methods *insert, search, delete, range* are very similar to the corresponding coordinators methods since the coordinator only redirects the request to the right worker on the ring.

Each Method returns a JSON Object on which Worker was called, if a file was created/deleted and how big the file is. On the search/range method, it responses with the found data if the key matches the criteria.

Coordinator:

The coordinator was implemented with Java Spring Boot. It is defined as Spring Boot application and uses the RestController and RequestMapping for the RESTful communication.

Insert:

The insert function is mapped to /insert and consumes multipart/form-data. Its parameters are the two RequestParam strings k, for the key and jsonRaw for the receiving json. It computes the address with hashing and returns the response with the getResponse function, which builds again a multifile and sends it to the right worker.

Delete:

The delete function works very similar to the insert function. It is mapped to /delete and also consumes multipart/form-data. The only difference is that it receives just one RequestParam, string k, because for deletion we only need the key. The address and forwarding works the same as in insert.

Search:

It works pretty much the same as delete. It is mapped to /search and also consumes multipart/form-data.

Range:

**How to Use the Client**

Start-up the Client described in the “How to Run” Section of this Report.

After Startup you are presented with a Menu an what you can do with the Client to Test the System. Some of the Methods require additional user input which is separated with ENTER.

Import CSV (c):

Imports the Netflix.csv which is located in src/main/resources into the Client. Without that you cannot insert new Data to the Buckets. The Client saves all Records in an accessible Array.

Secondary Input is the absolute Path to the csv file

Exit (x):

Exits the application. Be sure to use it to close the current CSV File.

Set Url (u):

Sets a new Url for the single tests. This is used if you want to go around the coordinator and access the Workers directly. The default URL is http://localhost:10005

Input -> Press Enter -> Enter new URL

Ping Url (p):

Pings the set URL and returns the status of the pinged url

Insert Data to worker (i)

Only Works if you have imported the CSV of the test Data. Calls the insert Endpoint of the Coordinator and returns the Time used to compute and the respond it got. As secondary input you enter the index of the imported records you want to use

Input p -> Press Enter -> <0 – (RecordCount-1)> as Number

Search for a key (s)

Searches for a key in the DataStore and returns a JSON if its found or empty if its not found. As secondary key input the key you want to search.

Input s -> Press Enter -> key

Make a Range Search (r)

Same as Search but searches within a Range and Returns a JSON Array

Input r -> Press Enter -> key1 -> Press Enter -> Key2

Delete a key (d)

Deletes a key from the buckets if it exists.

Input d -> Press Enter -> key

Show Menu (m)

Shows the Menu again

Insert all Data (a)

Inserts the complete Dataset that is imported to the buckets

Check CSV Data (l)

Prints the content of the imported CSV. Secundary Input is the index of the Record you want to check.

Invalid Input

Prints the Url and a hint that this is an invalid Input

**Git Commits Logs**

You can find our Project at: <https://github.com/Bernhard-Potuzak/cc-task4>

We divided the work

1. Implementation of Worker: Richard Dworschak (Tayi22)
2. Implementation of Coordinator: Bernhard Potuzak/Arthur
3. Implementation of Client: Richard (Tayi22)
4. Simulation: Bernhard/Arthur

**Sample Tests**

A Sample Run with inserts searches and deletes would give the output:

You can see the Timestamp when the action responded and how long it took in ms. In the Response you can see the Result of a Search and which worker responded to the action.