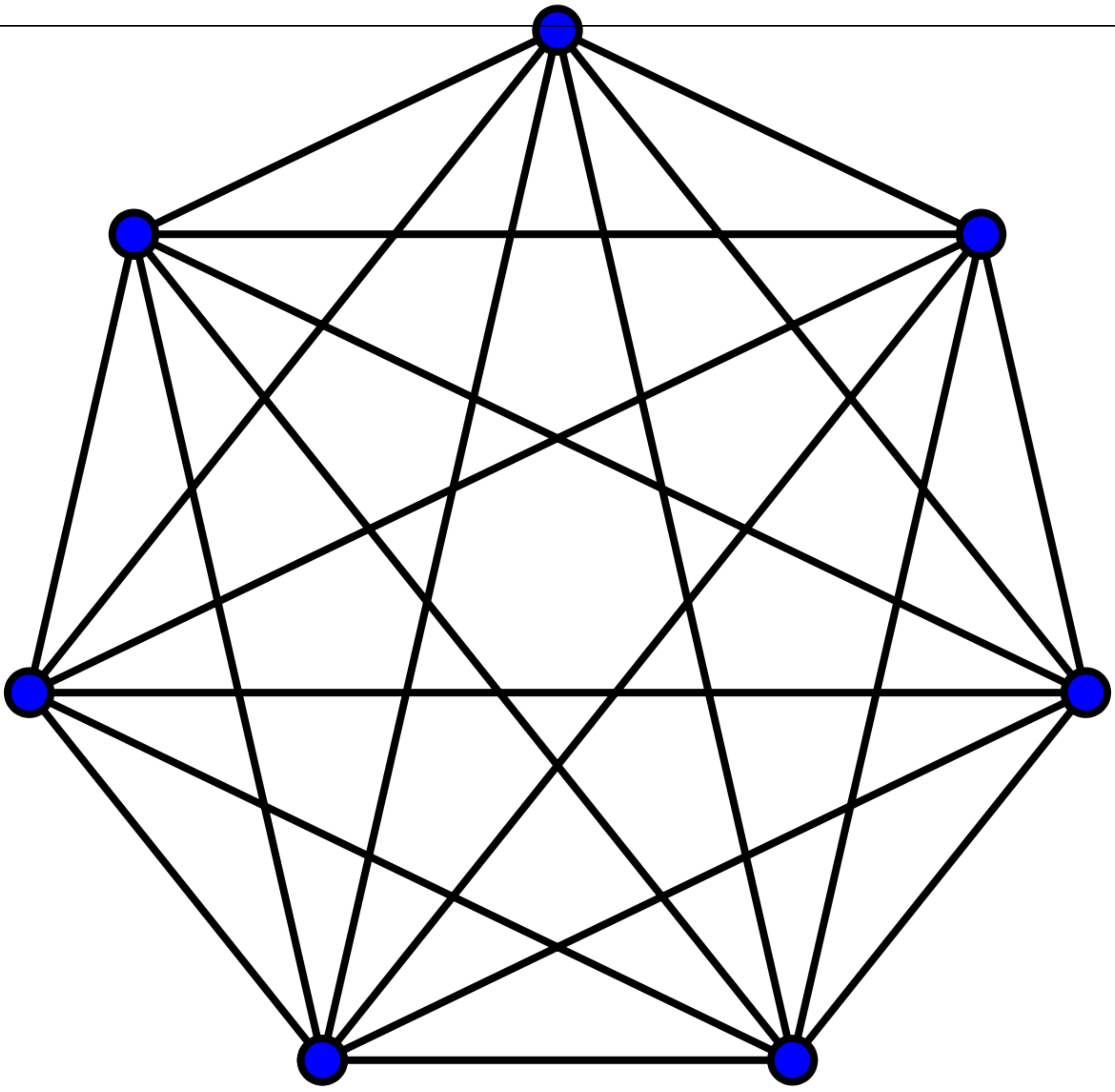


# Decentralized Cluster-Based NoSQL DB System

Atypon Java & DevOps Training

Student: Hamza alali

## Contents

# 1. Introduction

This report will discuss how I built a decentralized cluster-based NoSQL database system, this system includes a bootstrapper which initialize the database nodes and give them the initial configuration and includes a client to communicate with the bootstrapper and the database nodes, and a demo to show how someone will use the client to create their own projects with the database.

The database is in disk database, so the system I have built using fast read and write operations trying to compromise the database being in disk, also the database is multithreaded by nature so any member of users can access them.

Also, I will discuss how I handled load balancing in many areas in the system such as the bootstrapper and how he distributes users on nodes, the node and affinity distributing mechanism and the load balancing mechanism when the node is overloaded with requests.

I will also discuss the low-level design of the system and how I utilized some design patterns and clean code principles to keep the system flexible for any future modifications.

Finally, I will discuss clustering part of the system and how I utilized docker and docker network to help me achieve decentralization.

# 2. System Design

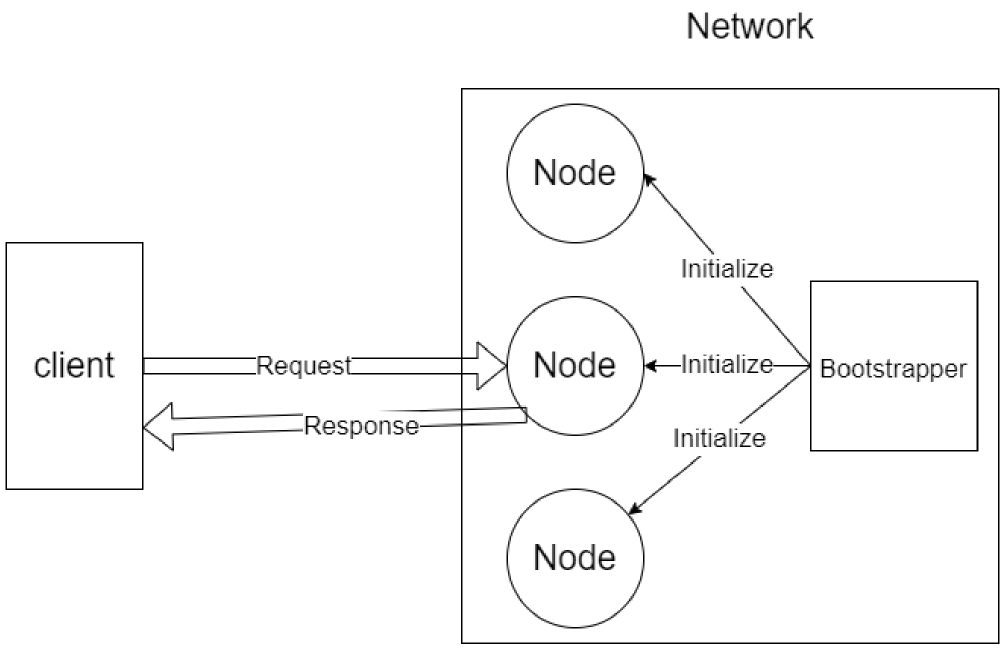
## 2.1.High-Level System Design

when designing any system one of the most important things is scalability, in short, you either scale horizontally or vertically, vertically is buying stronger computers to run the system, vertically is to buy more computers to scale the system and let them work in parallel, but the best approach will be horizontal - vertical scaling, decentralized system helps us achieve the horizontal scaling very well, they are made for this.

when designing a decentralized database, we need to take into consideration consistency the database across multiple "nodes" of our system and load balancing.

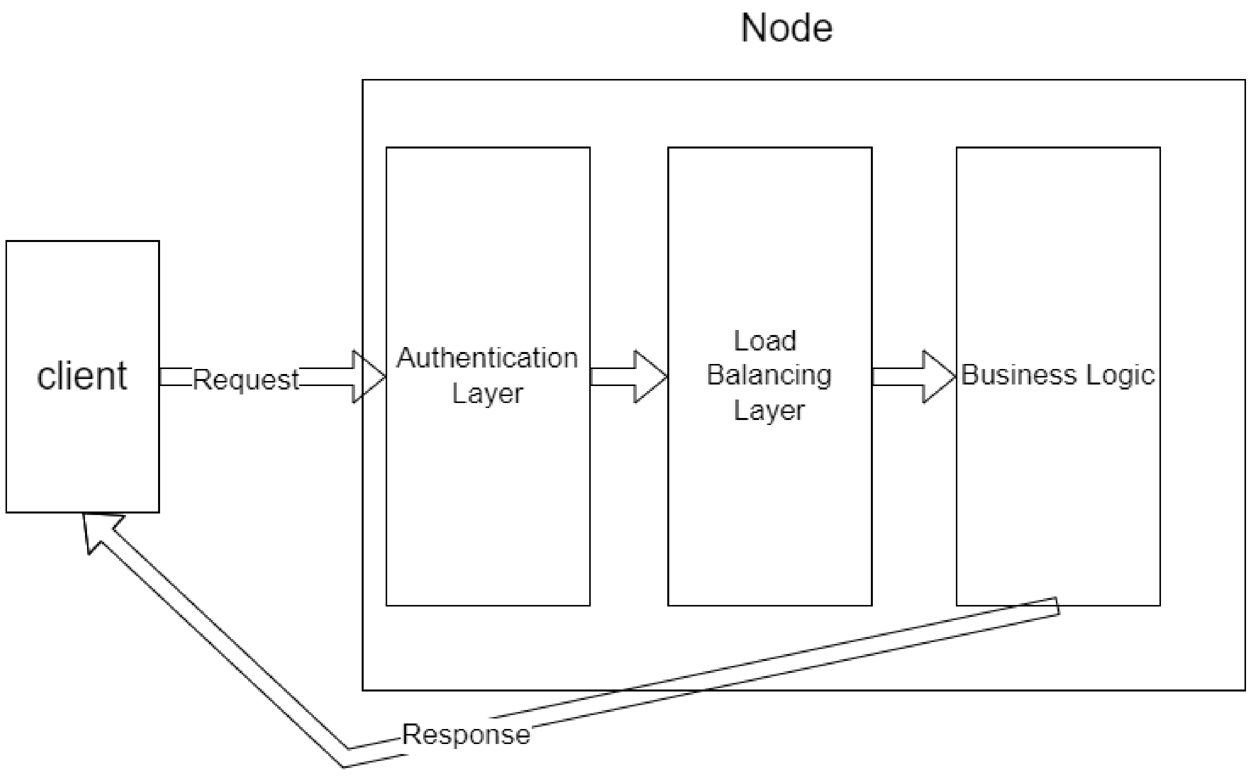
The parts of our system will be, clients, bootstrapper, database nodes.

database node should be in one network to connect with each other and synchronizing data between them, but clients connect to only one node at a time, our system design will look something like this:



and each node itself has some layers before the request is executed, first the authentication layer, user should connect to the node with his authentication information, second the load balancing layer, the bootstrapper node decides the load balancing variables such as max request per time window

and each node design will look like this:



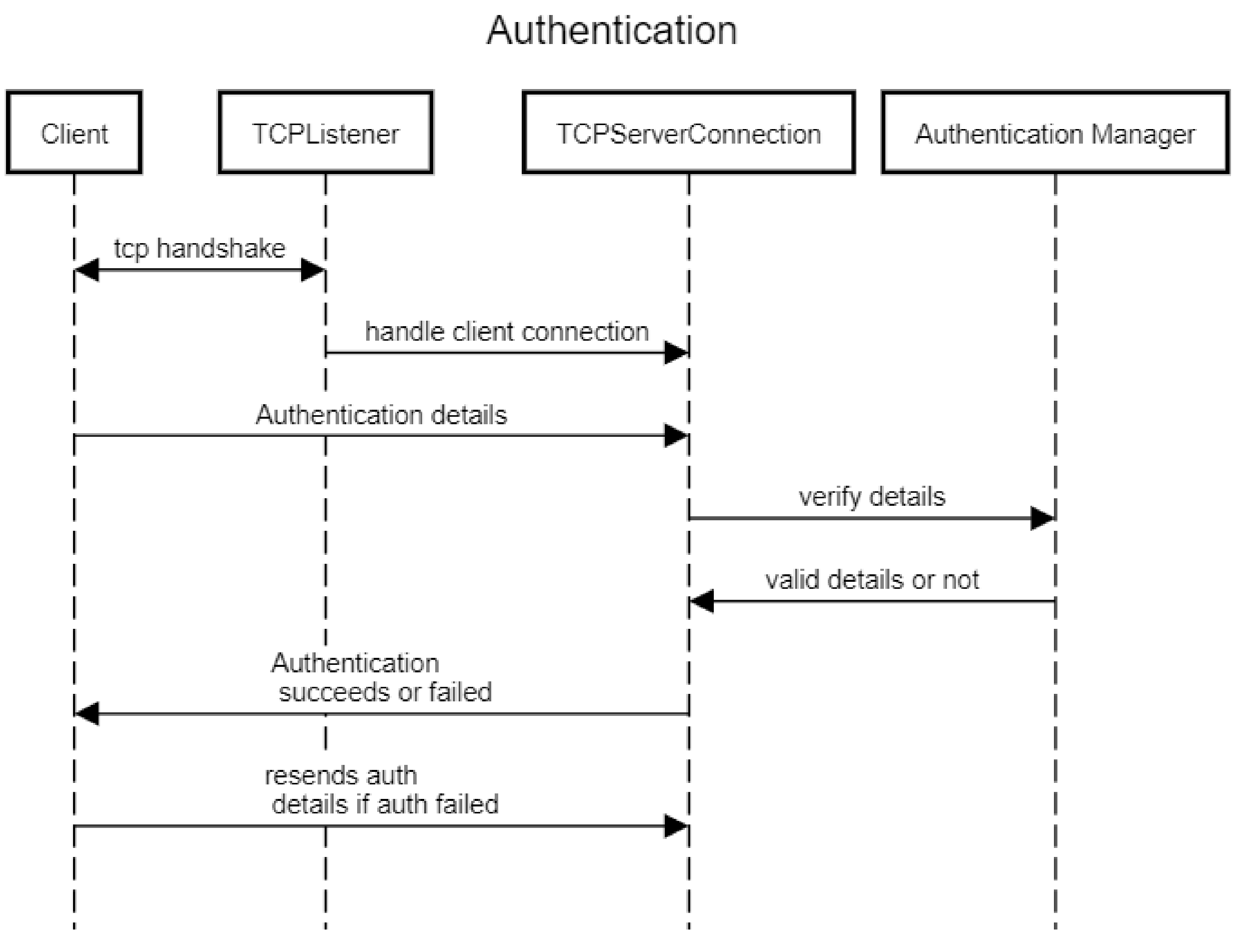
The business logic layer is where all our database operations will be executed.

## 2.2.Low-Level System Design

The first step of designing a software is high level design, the second is low level design.

### 2.2.1. Connection

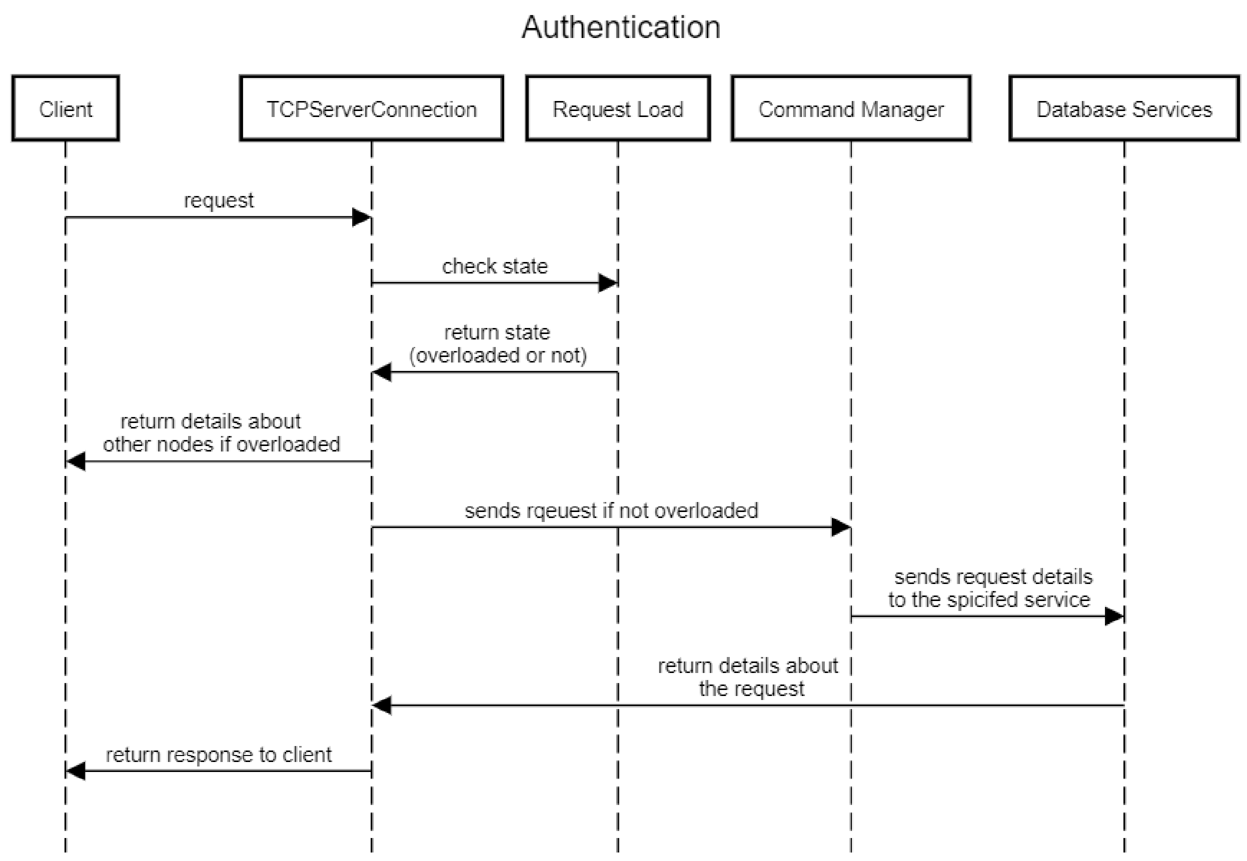
when client wants to connect to our server-through TCP – the client connects to our server using TCP handshake, then our server will make a new thread with TCPServerConnection class to handle this client connection, then the client will send his login details to our server and our server will response either login failed or succeed, sequence diagram:



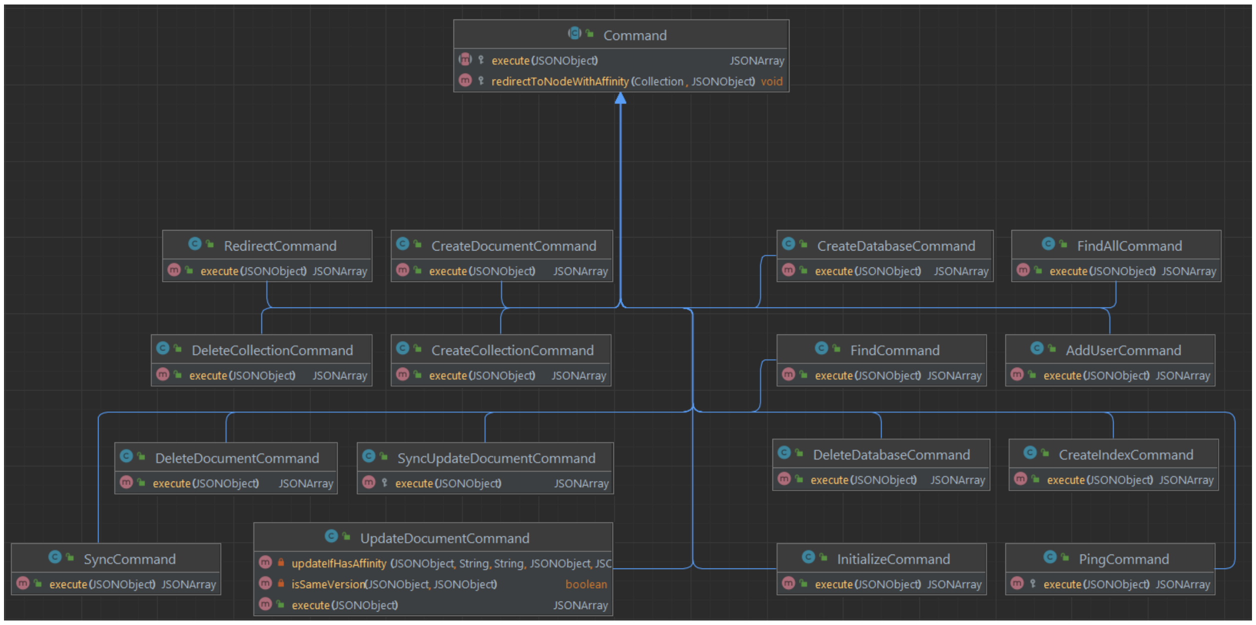
Here the TCPServerConnection class servers as a Faced Design patterns for the client, the client will send his requests to our server and this class will handle every thing and return a response to the client, Authentication Manger is a singleton class that verifies users.

### 2.2.2. Requests

Handling client request should be very efficient in every layer the request goes through, because I if one layer was slow, the response time for this request will be slow no matter the speed of other layers, so each layer should be fast and efficient, my system has authentication layer, which each client across it only one time, then it has load balancing layer, which the server check if it's overloaded or not, then the business logic layer which is the database operations from create to delete update and read, sequence diagram:


Authentication
Client
TCPServerConnection
Request Load
Command Manager
Database Services
request
check state
return state (overloaded or not)
return details about other nodes if overloaded
sends rqeuest if not overloaded
sends request details to the spicifed service
return details about the request
return response to client

Here the Request Load class is a singleton, the Command Manager is Mediator class between the Database Services and the Command class, the Command class is a Command design patterns which is used to differentiate between different requests and handling the request before it reach the database services, here is a UML diagram for the Command abstract class and his concrete classes:


Command
execute(JSONObject) JSONArray
redirectToNodeWithAffinity(Collection, JSONObject) void
RedirectCommand
execute(JSONObject) JSONArray
CreateDocumentCommand
execute(JSONObject) JSONArray
CreateDatabaseCommand
execute(JSONObject) JSONArray
FindAllCommand
execute(JSONObject) JSONArray
DeleteCollectionCommand
execute(JSONObject) JSONArray
CreateCollectionCommand
execute(JSONObject) JSONArray
FindCommand
execute(JSONObject) JSONArray
AddUserCommand
execute(JSONObject) JSONArray
DeleteDocumentCommand
execute(JSONObject) JSONArray
SyncUpdateDocumentCommand
execute(JSONObject) JSONArray
DeleteDatabaseCommand
execute(JSONObject) JSONArray
CreateIndexCommand
execute(JSONObject) JSONArray
UpdateDocumentCommand
updateIfHasAffinity(JSONObject, String, String, JSONObject, JSC
isSameVersion(JSONObject, JSONObject) boolean
execute(JSONObject) JSONArray
SyncCommand
execute(JSONObject) JSONArray
InitializeCommand
execute(JSONObject) JSONArray
PingCommand
execute(JSONObject) JSONArray

Using the Command design pattern here is very important to handle multiple request types.

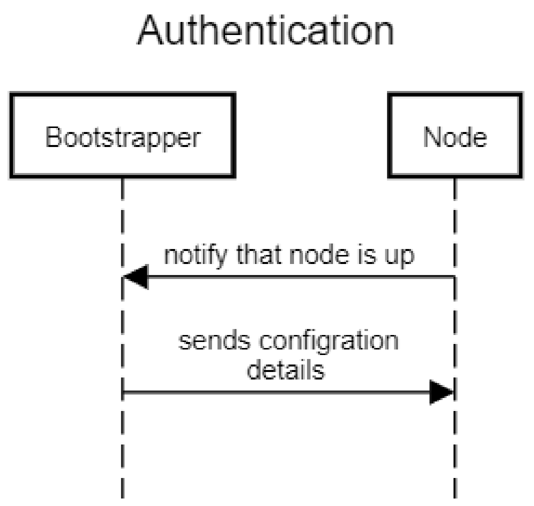
# 3. The Server

## 3.1.Protocols

We will use two protocols TCP and UDP, TCP will be used for client communications, UDP will be used for commination between nodes, so we have two listeners in each node, TCP and UDP, each of them runs on a different thread.

## 3.2.Initialize

When each node starts, it will send a UDP message to the bootstrapper to notify the bootstrapper that the node is up, then the bootstrapper will send the configuration details to this node, simple sequence diagram:



## 3.3.Authentication

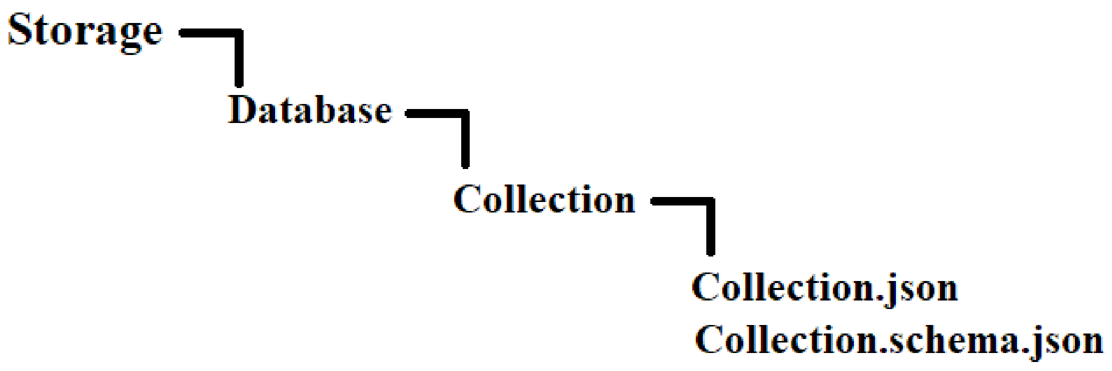
Users in our system comes from the bootstrapper, either predetermined when the initial configuration or registered while the node is running, when a client registers in the bootstrapper using username and password , the bootstrapper will give him and port for a specific node to connect to, and the bootstrapper will sends this user information to this node.

# 4. The Database

This chapter will talk about the type of database we used which is NoSQL and will talk about the load balancing used file system, indexes, database operations and database design in general.

## 4.1.Database Type And Design

Our database type is NoSQL, and we chose document-based NoSQL, the organization we chose is database, each database has different collection(table), each collection consists of documents(record).

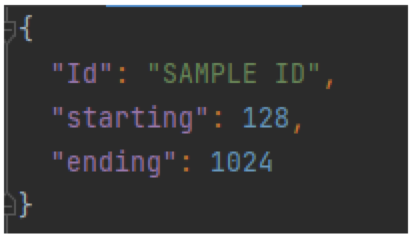


## 4.2.Database File System

### 4.2.1. Collection Organization

Our focus will be on collection and documents, since they are the part that hold the data and will be mostly used in our database, some databases save data on the disk in block, each time a user writes or reads from a record, the database will access the block associated with this record, this technique is used for fast read and writes operations, I will use a similar a simpler version of this technique using JSON documents, since JSON documents consists of characters that are formatted in a JSON format, we know that each document will have  N Byte size, so for each document we can save the starting byte of it in the JSON collection and the ending byte will be = starting byte + N, this way we can divide or JSON file into multiple block, each block consist of a document, and we keep the JSON format valid through the whole collection.

Sample of an information about a document block:



```
{
  "Id": "SAMPLE ID",
  "starting": 128,
  "ending": 1024
}
```

Id: the id of the document.

Starting: the starting byte for this document.

Ending: the ending byte of this document.

### 4.2.2. Writing

Writing will depend on the block organization we developed, normally to write a JSON file you will use these operations:

1.open the JSON file.

2.read all of it and store it in memory.

3.update it with some information.

4.rewrite it to disk.

For every write operation we will need to deal with the whole file at once, this is very expensive with large JSON files.

Using blocks we can shorten the writing time for each document using Random access file and File Channel in Java, our operations will be like this:

1.opening the file with random access file.

2.create the file channel through random access file.

3.get the size of the file.

Let’s say the size of the file is N.

4.skip to byte N-1

5.append “,” if it’s not the first document then append the document.

6.append “]” to maintain JSON format

This way it’s very fast for us to append new documents, each time we append a document we will save the starting byte which is N-1 and the ending byte which is N-1+document size, for reference each character in JSON is usually 1 byte since JSON use UTF-8 encoding.

#### 4.2.3. Reading

Since the one of the requirements is that reading operation are the most frequently operations we have, we will use the data that we saved for each document after writing it to read it fast.

Reading operations:

1.retreive the information about the indexing or this document.

2.skip to the starting byte.

3.read till the ending byte.

#### 4.2.4. Deleting

Deleting a document here will be a soft delete, we will simply delete the block information for this document, so it will no longer exist, and our program doesn't know about it.

#### 4.2.5. Updating

Updated will have 3 parts and it's the heaviest operations since it will use read delete and write operations, it will be like this:

1.read the document

2.delete the document (soft delete)

3.update its information

4.write the updated information back to the file

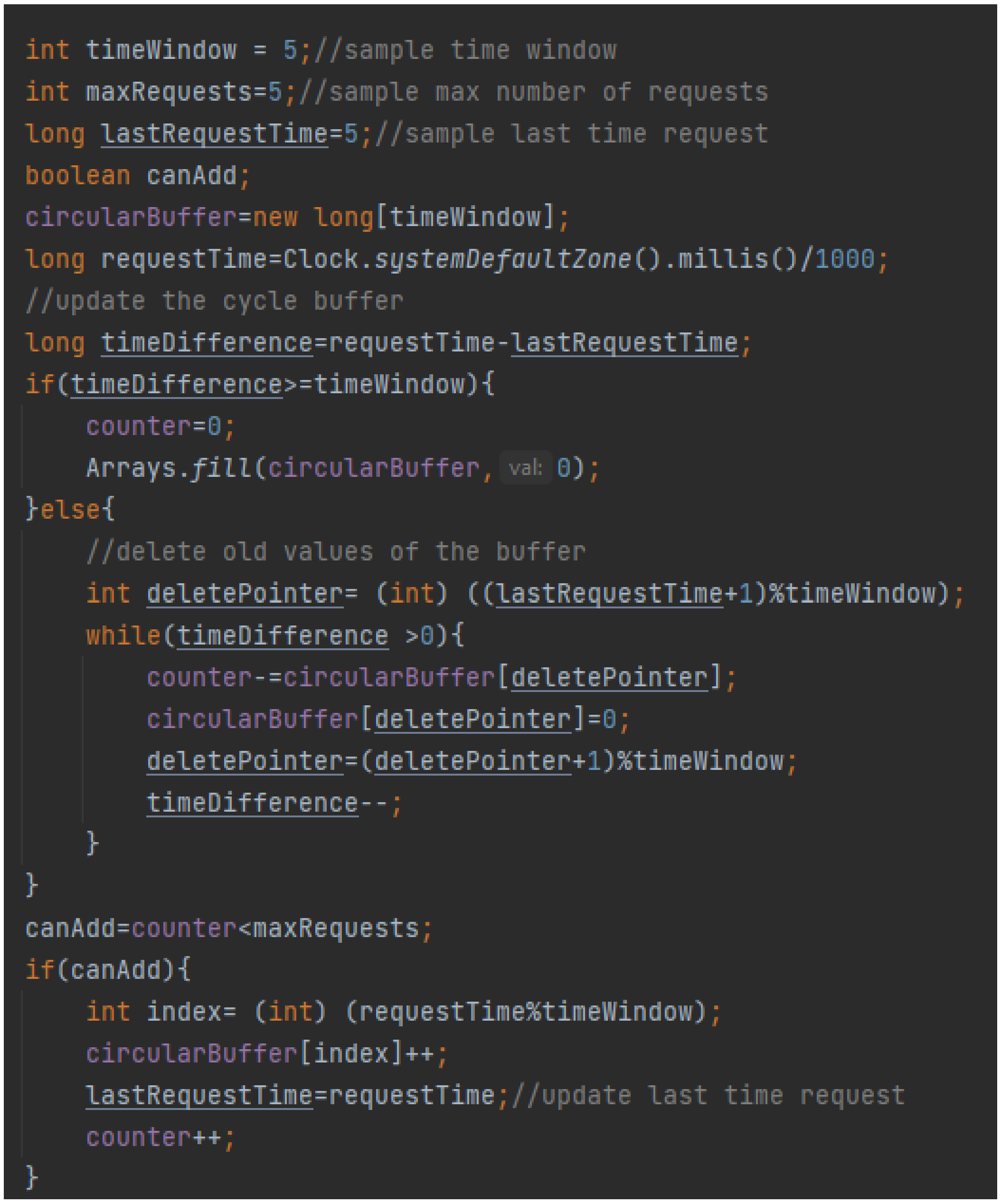
### 4.3.Server Load And Load Balance

#### 4.3.1. Request Load

Since we are implementing a decentralized system, one of the most important aspects is the load balancing at the node level, we don't want one node to have 10 request and other to have 10000 request, we want to achieve balance across multiple node as much as we can.

I implemented a class that is responsible for calculating the load on the server for each node, I called it Request Load, it's a Singleton class that all TCPServerConnections(class that handle each client connection) has access to it, each time a client sends request a server will check if it's overloaded with requests or not, if not it will pass the request, if it's overloaded it will return to the client a message containing other nodes ports so that the client can be redirected to any of them, and the redirection operation will be on the client side.

Calculating the load on the server is done via algorithm I created using cycle buffer idea, cycle buffer is like a buffer but it's starting point and ending point are connected, this is what I need since I will be calculating a load over a time window, this time windows is decided from the bootstrapper, here is a sample code for the algorithm:



```java
int timeWindow = 5;//sample time window
int maxRequests=5;//sample max number of requests
long lastRequestTime=5;//sample last time request
boolean canAdd;
circularBuffer=new long[timeWindow];
long requestTime=Clock.systemDefaultZone().millis()/1000;
//update the cycle buffer
long timeDifference=requestTime-lastRequestTime;
if(timeDifference>=timeWindow){
    counter=0;
    Arrays.fill(circularBuffer, val: 0);
}else{
    //delete old values of the buffer
    int deletePointer= (int) ((lastRequestTime+1)%timeWindow);
    while(timeDifference >0){
        counter-=circularBuffer[deletePointer];
        circularBuffer[deletePointer]=0;
        deletePointer=(deletePointer+1)%timeWindow;
        timeDifference--;
    }
}
canAdd=counter<maxRequests;
if(canAdd){
    int index= (int) (requestTime%timeWindow);
    circularBuffer[index]++;
    lastRequestTime=requestTime;//update last time request
    counter++;
}
```

This algorithm works In O (N) for any N seconds that pass between each different request as example,

If request 1 arrived at sec 6, it will calculate the request load for the last 6 seconds, if request 2 arrived at time 6, it will only increment the load counter, if request 3 arrived at time 8 it will calculate the load for last 2 seconds, looping over the buffer is required here for each seconds to delete old values from the buffer if they did exists.

So for N seconds the worst case will be O (N ) if the difference between each request was less than the time window.

### 4.3.2. Affinity Distribution

Affinity here is a relation between writing to a collection and a node, if a node has the affinity to write to this collection no other node can write to this collection, this is used to eliminate un synchronized writes, because if it's not used one node might update the data and other node at the same time might update the same data, both are valid, but which to pick? How can nodes handle picking one update after another? To solve this we implement affinity, if any node got request to update/write/delete for a collection, this node will send the request to the node with affinity, then the node with the affinity will broadcast the outcome of this request to the rest of the nodes.

So each node should know what node holds the affinity for each collection, to do this I implemented a global class that handle the distribution of affinities and called it Affinity Distributer, this class implements internally round robin load balancing, each time a new collection is made the affinity distributer will distribute it's affinity to the next node and so.

## 5. Indexing

Each database has indexing system, and indexes might get very complicated but, in our database, we will implement a simple indexing mechanism that will handle only one property indexing and duplication in keys, when choosing a data structure to implement the index, we have many options such as HashMap, BTree, LSM, my choice was a HashMap and BTree.

HashMap has O (1) lookup time, but it's not the best data structure when handling quires such as greater than, less than etc.… , but in my use case , I used it for storing the index object(which contain the starting and the ending byte of a document) and the id of the document, the key will be the id and the value will be the index object, this way I can lookup index object in a fast way, since I don't need to have quires when looking up the index object.

BTree is the most used data structure for indexing, and it has very fast reading, and you can implement quires with it, although I didn't implement any query, but I used it for any indexing other than the id, I didn't implement it from scratch, I took an implementation from GitHub, and edited it so it can handle duplication of the of a key. I took also into consideration nested property indexing, so there will be different index for a property on level 2 that has same name of a property on level 1

```
{
  "firstName": "hamza",
  "fullNName": {
    "firstName": "hamza",
    "secondName": "alali"
  }
}
```

Here first name on level 1 the index property of it will be simple “firstName” But the first name on level 2 will has it as “fullName.firstName”, this is a simple solution for this problem. I also implemented an index Manager class, this class resides in main memory, and it saves the database names, collection names, and the indexes.

## 6. Database operations

To use any database, you need to use quires, I implemented basic quires for the database to be functional which are:-

-Create database: - creates a database directory

-Delete database: - deletes the database directory with everything it has

-Create collection: - creates the collection directory and JSON files

-Delete collection: - deletes the whole collection

-Create document: - creates a document inside a collection and add it to all existing indexes

-Delete document: - delete the document from all indexes.

-Update document: - updates the document in all current indexes

-Find: - check if the id is given, if yes then take the index object from the id HashMap, if the id is not given, search inside the index with the property that is given, if it has no index, then preform full search on all documents (the costliest)

-Find all: - return the documents inside a collection

-Create index: - creates an index and insert all documents in this collection inside it

-ping: - pings the database to check if it’s overloaded or not

## 7. Locks and Synchronization

To handle different users request we will create a different thread for each different user to respond for his requests.

To solve the race condition in disk operations I implemented a simple locking mechanism with a Reentrant Lock on three levels, on the database level, collection level and document level.

1.database level: if a user tried to create a database, the database lock will be triggered and no other user can create or delete a database till the creation is done

2.collection level: when a user creates or deletes a collection lock inside this database will be triggered and no other user can create or delete a collection inside this database.

3.document level: when a user creates or update or delete a document inside a collection, the document lock will prevent any other user from accessing this collection, this way I can guarantee that no two users will enter in a race condition.

Also, I have a lock on Request Load singleton object since it’s accessed buy all threads at the same time.

# 8. The Cluster

Before talking about the bootstrapper, I will talk first about the cluster and how each node talks to other node and when they talks.

## 8.1.The cluster

As a decentralized database we achieve this by implementing a cluster of nodes, each node in the cluster is a VM itself, we implemented this using docker and docker network, each node will run on a docker container inside a docker custom bridge network.

Each network will have its own subnet inside the host subnet, and all nodes in the network can communicate with each other, and they can communicate with the host via versa using port mapping.

## 8.2.Broadcasting

We use UDP broadcasting in inside the docker network as the protocol for nodes to communicate between each other and this is done via broadcast address, this address will be calculated by the bootstrapper and given to each node as a system variable, when a node want's to sends a message to all other nodes it sends the UPD datagram to the broadcast address, if it's want to send the message to a specific node it will sends it to the address for this node.

We used UDP here since we don't want to maintain a separate connection for each node with other nodes in the system, because if we do this, we will give each node an overhead for this connection, to maintain the connection and so, and to achieve decentralization since it's a decentralized system.

# 9. The Bootstrapper

The bootstrapper runs on a host machine/VM, not inside a container,

It creates and network and gives each node the initial configurations such as info about other nodes, broad cast address, users, load balancing information's.

## 9.1. User Load Balancing

Users in the cluster are distributed in a round-robin manner, when a new user is registered, the bootstrapper will sends its info to the specified node by the load balancer.

## 9.2. Executing Commands

Executing command form the bootstrapper is done via Process Builder class

## 9.3. Configuration

The bootstrapper handles the configurations from exposing ports and calculating the broadcast address for a network

# 10. The Client

The client is the classes that users of our database will deal with and configure and user in their projects, it's a simple implementation to connect to any node and provides a connection detail and sending queries to the nodes and taking back response.

## 11. Security issue

Although our system uses username and password for the client to connect to any node, it has some issues when validating inner communications between nodes, there is no validation that validate each message between nodes or messages between bootstrapper and nodes, if I can provide a simple solution for this will be using tokens like JWT or any other kind of token, each node will send a token with the message and each node will verify this token before accepting the message or not, and the secret for this JWT tokens will be provided via the initial configuration by the bootstrapper, although I didn't handle this issue in database but this is a solution I propose to solve this problem.

## 12. Clean Code

I will talk briefly about some of the clean code principles I followed when writing the code of this project and provide examples for them.

### 12.1.Meaningful Names

Variables:

```
private User authenticatedUser;
```

Functions:

```
private void broadcastUser() throws IOException {
        JSONObject userJson=authenticatedUser.toJsonObject();
        userJson.put("commandType", CommandTypes.ADD_USER.toString());
        UDPCommunicator.broadcastCommand(userJson);
}
```

Classes:

```java
public abstract class Listener implements Runnable {
    protected boolean isRunning=true;
    @Override
    public void run() {
        listen();
    }
    abstract void listen();

    public boolean isRunning() {
        return isRunning;
    }

    public void setRunning(boolean running) {
        isRunning = running;
    }
}
```

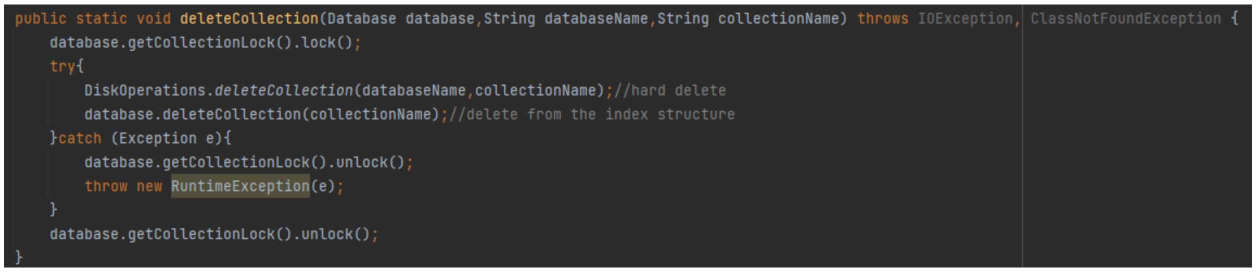
## 12.2.Functions

Small and do one thing:

```java
public void setRunning(boolean running) {
    isRunning = running;
}
```

## 12.3.Comments

Clean and only when needed:



```java
public static void deleteCollection(Database database,String databaseName,String collectionName) throws IOException, ClassNotFoundException {
    database.getCollectionLock().lock();
    try{
        DiskOperations.deleteCollection(databaseName,collectionName);//hard delete
        database.deleteCollection(collectionName);//delete from the index structure
    }catch (Exception e){
        database.getCollectionLock().unlock();
        throw new RuntimeException(e);
    }
    database.getCollectionLock().unlock();
}
```

## 12.4.Classes

When writing my classes I tried to make them small as possible using design patterns

Like the Command concert classes

# 13. Design Patterns

When writing the low-level design for a software, one of the most important aspects is design patterns, so before writing the implementation I tough about how to solve it first and the design of the code.

## 13.1.Creational

### 13.1.1.Singleton

I Used Singleton in many places in my code because many of the objects are shared by multiple threads, like authentication manager, cluster manager and so, and I tried to make them thread safe.

### 13.1.2.Factory

I implemented some factories in my code, example will be Command Factory, it gives me the wanted command based on the type of it.

Other will be bTree supplier which return the wanted bTree based on the variable type.

### 13.1.3.Builder

In client I needed to make a JSON variable in a clean way and I couldn't using Simple JSON dependency so I did create a builder class for it and for JSON Array also so it can be much easier to make Json variables in java.

## 13.2.Behavioral

### 13.2.1.Command

In command class

### 13.2.2.Mediator

In commands mediator

# 14. Effective Java

## 14.1. Item 1: CONSIDER STATIC FACTORY METHODS INSTEAD OF CONSTRUCTORS

```
public static User of(JSONObject jsonObject){
    User user=new User();
    user.setUsername((String) jsonObject.get("username"));
    user.setPassword((String) jsonObject.get("password"));
    return user;
}
```

## 14.2. Item 3: ENFORCE THE SINGLETON PROPERTY WITH A PRIVATE CONSTRUCTOR OR AN ENUM TYPE

Used In Singleton classes

## 14.3. Item 4: ENFORCE NONINSTANTIABILITY WITH A PRIVATE CONSTRUCTOR

In Utilities Classes like Disk Operations

## 14.4. Item 6: AVOID CREATING UNNECESSARY OBJECTS

I only created object when needed them

## 14.5. Item 8: AVOID FINALIZERS AND CLEANERS

When dealing with closing socket or file resources I used try with resources over finalizers

## 14.6.Item 9: PREFER TRY-WITH-RESOURCES TO TRY-FINALLY

## 14.7.Item 15: MINIMIZE THE ACCESSIBILITY OF CLASSES AND MEMBERS

Class members I write them as private and accessed outside the clase only by getters or setters

## 14.8.Item 16:IN PUBLIC CLASSES, USE ACCESSOR METHODS, NOT PUBLIC FIELDS

## 14.9. Item 18: FAVOR COMPOSITION OVER INHERITANCE

## 14.10.Item 28: PREFER LISTS TO ARRAYS

## 14.11.Item 49: CHECK PARAMETERS FOR VALIDITY

When receiving a query form a user, I check of it has the valid values for database, collection

## 14.12.Item 58: PREFER FOR-EACH LOOPS TO TRADITIONAL FOR LOOPS

Used for every time I could

## 14.13.Item 78: SYNCHRONIZE ACCESS TO SHARED MUTABLE DATA

## 14.14.Item 79: AVOID EXCESSIVE SYNCHRONIZATION

In The database only the database, collection, document had synchronization

### 14.15.Item 85: PREFER ALTERNATIVES TO JAVA SERIALIZATION

When facing a decision in sending java objects or JSON, I preferred JSON and sends it as a string

# 15. SOLID PRINCIPLES

## 15.1. Single-Responsibility Principle

I created many classes to achieve this principle, my classes each one serves for one purpose.

Such as Command classes.

## 15.2. Open-Closed Principle

You can introduce any new database command to the database by extending Command interface and add that Command type to the factory

## 15.3. Liskov Substitution Principle

Database Query, UDP Routine and their subclasses

## 15.4.Interface Segregation Principle

My classes don't implement any function that they don't need or user or just ignore it

## 15.5. Dependency Inversion Principle

My Command Mediator depends on command interfaces not concrete classes

# 16. DevOps

## 16.1.Docker

For containerization I used docker, and docker network as the private network which helped me in isolating each database node and isolating the network overall.

## 16.2.Shell Scripting

For creating the containers and the network and getting the ip I used shell commands to automate this process.

## 16.3.Git and GitHub

I used Git and GitHub in every process in creating the project

## 16.4.Maven

Used maven to packaging the database node to build the image,

And for JSON simple dependency

# 17. Using The System

Run The bootstrapper jar file and make sure docker daemon is running, user the bootstrapper cli to provide the following configurations:

**TCP ports range :** (ports used for the client to communicate with nodes)the range that the TCP ports will start from

(given range , given range + container Number)

**UDP ports range:** (ports used for the bootstrapper to talk with nodes) the range that the UDP ports will start from
(given range, given range + container Number)

**Bootstrapper ports range**: (ports used for the node to talk with bootstrapper) the range that the ports will start from
(given range, given range + container Number)

**Load balance time window**: the time window to calculate the load in seconds

**Load balance max requests**: the max request in this time window

**Number of containers**

**Bootstrapper TCP port**: port used for the client to comminate with the bootstrapper (for registering new users).

**Command to create the database docker image:**

```
docker build . --tag database-node
```

I included a demo project to explain how the client works,