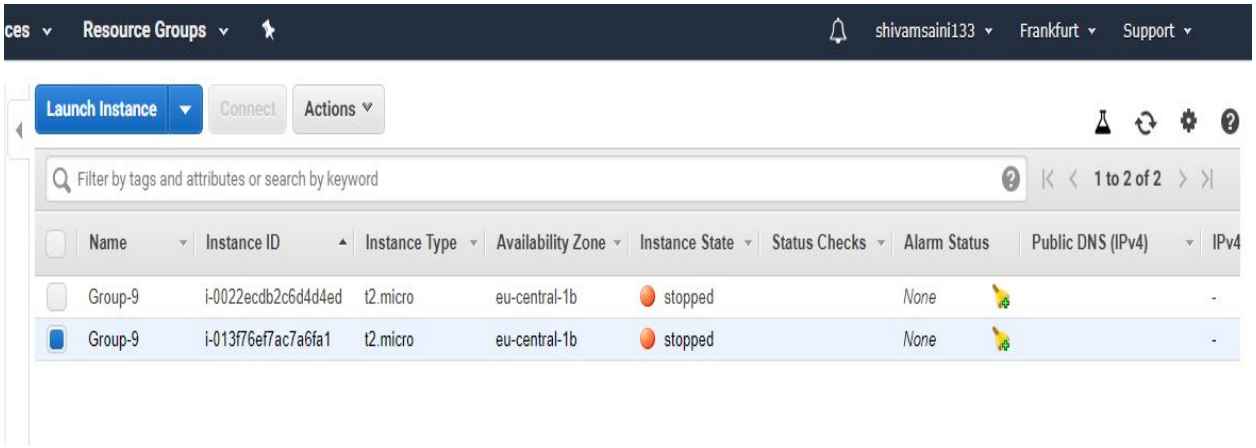
Enterprise Computing WS 2018/19

Assignment 2

By (Group 9)
- Anushka Garg (406235)
- Shivam Saini (405667)

Amazon EC2 Console with instances:



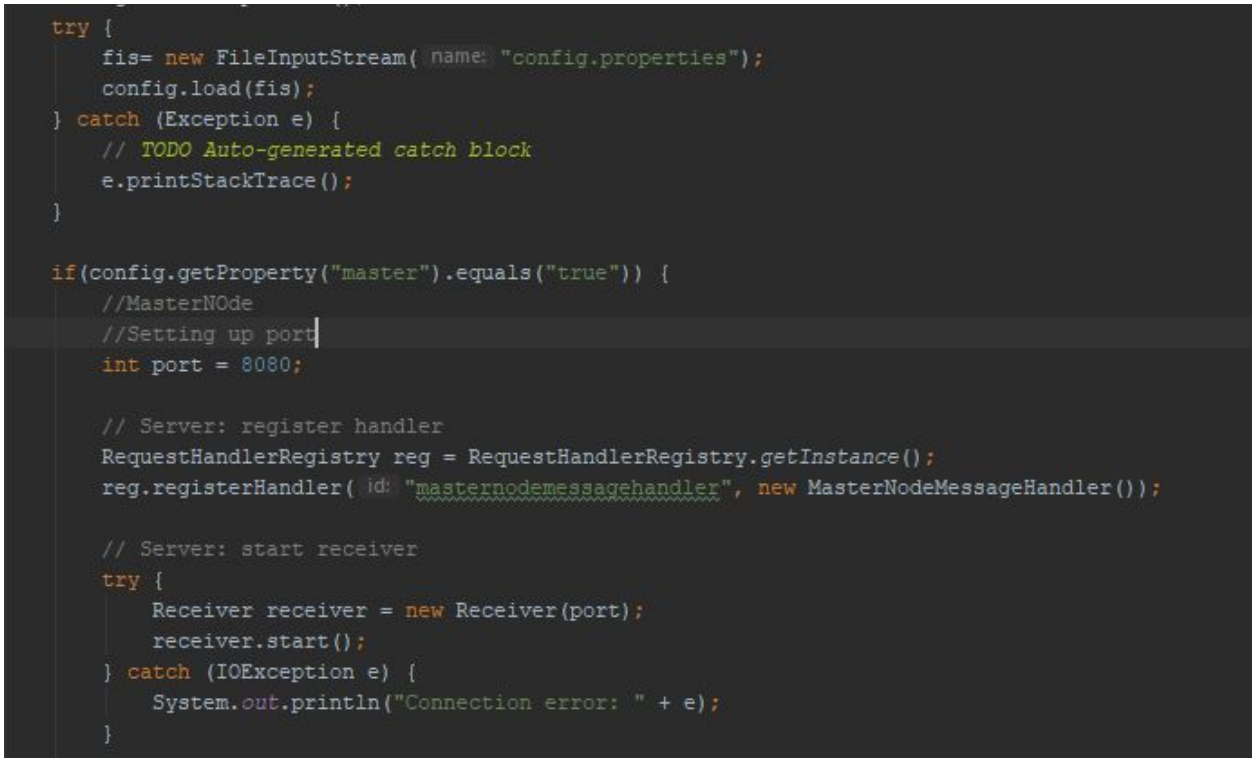
1. Distributed Key- value Store Implementation

We have created distributed key value store consisting two replicas, one is master and another one is slave. As according to the task sheet, we performed CRUD operations and noted results of benchmarking by developing the following methods:

Main.java - This is the main class of out program which can be used to set up the client, master or the slave depending on the config file. This class reads the properties from the config. Properties file and run the main method accordingly. For example:

```
if(config.getProperty(“master”).equals("true"))
```

The main method will know that it needs to run the master node. Similarly, we can set up the slave and the client using the config.properties file.



```
try {
    fis= new FileInputStream( name: "config.properties");
    config.load(fis);
} catch (Exception e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

if(config.getProperty("master").equals("true")) {
    //MasterNOde
    //Setting up port
    int port = 8080;

    // Server: register handler
    RequestHandlerRegistry reg = RequestHandlerRegistry.getInstance();
    reg.registerHandler( id: "masternodemessagehandler", new MasterNodeMessageHandler());

    // Server: start receiver
    try {
        Receiver receiver = new Receiver(port);
        receiver.start();
    } catch (IOException e) {
        System.out.println("Connection error: " + e);
    }
```

A client can perform read and write operations on master node but can only read from the slave node. The client creates an object comprising the required key , value along with other desired properties , creates a request and the sends it along with the object to the Master node using the Hermes built in methods. It then waits for the reply from the server and records the response time in a local file.

```
if(config.getProperty("client").equals("true")) {
    // HERMES TEST
    int port = 8080;  //Master node port
    String host = "3.121.184.165"; // masterNode IP

    //Writing a new key value pair
    for(int i=0;i<100;i++) {
    CrudRequest request = new CrudRequest();
    request.setKey("valueIs");
    request.setValue("hello "+i);
    request.setRequestType("CREATE");
    if(config.getProperty("isAsynchronous").equals("true")) {
        request.setAsync(true);
    }
    else{
        request.setAsync(false);
    }
    request.setMessageId("KEY-" + request.getKey() + "-" +request.getRequestType());

    // Client: create request
    Request req2 = new Request(request,  target: "masternodemessagehandler", sender: "value");

    // Client: Creating the sender
    Sender sender = new Sender(host, port);
    Long start_time = System.currentTimeMillis();

    //Sending request to the node
    Response res = sender.sendMessage(req2,  timeout: 5000);
    Long elapse = System.currentTimeMillis() - start_time;
```

MasterNodeMessageHandler.java - The request send by the client is received by the server node using the Handler. This handler implements the IRequestHandler class of the Hermes library and override its methods. This class receives the object along with the request sent by the client. It has 4 different cases for request type : Create, Read, Update , Delete

- Create - It creates the object at the master host and sends the same request to the slave to replicate the request. It checks whether the request is synchronous or asynchronous and send the create request to the slave node using the hermes library.

```
// Determine the type of request
switch (requestType) {
case "CREATE":
    //Adding the key value pair to the file using the CrudMethods Class
    obj.create(request.getKey(),request.getValue(),request.getMessageId());
    //Slave Node
    int port = 8080;
    String host = "52.59.205.51"; // Slave node IP

    // Creating the request to the slave
    Request reqSlave = new Request(request,  target: "slavenodemessagehandler", sender: "first_entry");
    // Master: send messages to Slave
    Sender sender = new Sender(host, port);

    if(request.isAsync()){
        sender.sendMessageAsync(reqSlave, new AsyncCallbackRecipient() {
            @Override
            public void callback(Response response) {
            }
        });
        // Return a dummy response
        return new Response( responseMessage: "Success",  responseCode: true, req, new ArrayList<Request>());
    }
    else{
        Response res = sender.sendMessage(reqSlave,  timeout: 5000);
    }
    break;
```

- Read - It just read the value by getting the key from the request.

```
//Reading the value using the key
case "READ":
    obj.read(request.getKey());
    break;
```

- Update - It works similar to the create request. After updating the data in its file, it sends the request to the slave node in similar way as that of the create request.

```
case "UPDATE":
    obj.update(request.getKey(), request.getValue(), request.getMessageId());

    //Slave address
    int port1 = 8080;
    String host1 = "52.59.205.51"; // slaveNode

    Request reqSlave1 = new Request(request,  target: "slavenodemessagehandler", sender: "first_entry");
    // Master: send messages to Slave
    Sender sender1 = new Sender(host1, port1);
    if(request.isAsync()){
        sender1.sendMessageAsync(reqSlave1, new AsyncCallbackRecipient() {
            @Override
            public void callback(Response response) {

            }
        });

        // Return a dummy response, we don't handle async callbacks now
        return new Response( responseMessage: "Success",  responseCode: true, req, new ArrayList<Request>());
    }
    else{
        Response res = sender1.sendMessage(reqSlave1,  timeout: 5000);
    }
    break;
```

- Delete - It deletes the object by getting the key of the object from the request.

```
        //Deleting the key value pair
    case "DELETE":
        obj.delete(request.getKey());
        break;
```

SlaveNodeMessageHandler.java - The request sent the Master node is received by this handler at the slave node. This handler also implements the IRequestHandler Class of the Hermes. It receives the object sent by master and perform the CRUD operation according to the master node request.

```java
public class SlaveNodeMessageHandler implements IRequestHandler {

    @Override
    public Response handleRequest(Request req) {

        List<Serializable> list = req.getItems();
        CrudRequest request=(CrudRequest )list.get(0);
        String requestType=request.getRequestType();
        CrudMethods obj = new CrudMethods();

        switch (requestType) {
        case "CREATE":
                obj.create(request.getKey(),request.getValue(),request.getMessageId());

            break;
        case "READ":
            obj.read(request.getKey());
            break;

        case "UPDATE":
            obj.update(request.getKey(), request.getValue(),request.getMessageId());
            break;

        case "DELETE":
            obj.delete(request.getKey());
            break;

        default:
            break;
        }
```

CrudMethods.java - It contains all the operation methods. Also we record a commit log file by saving the current time in create and update operations.

```java
public void create(String key, String value, String messageId) {
    obj.store(key, value);

    //write commit log to the file.
    String time = System.currentTimeMillis() + "";
    writeToFile(messageId, time);
}
```

FileSystemKVStore.java - This class implements the interface KeyValueInterface. This class implements the methods that help in reading and writing the data in the local file system.

2. Benchmarking Latency and Staleness -

Benchmarking is the process of evaluating performance against a known baseline. We have taken benchmarking in two ways of replication : Synchronous and Asynchronous.

- Asynchronous replication - In this master directly sends the response to client without waiting for the response from slave.
- Synchronous replication - In this client sends request to the master and then master sends it to the slave for replication. After executing the request, slave sends response back to master and then master sends the response to client.

Latency Benchmarking - How long each individual transaction takes, usually measured as the average or percentile of the latency for a sample number of transactions. In this assignment we have measured latency when a client sends a request to the server in both ways (Synchronous and Asynchronous).

Staleness Benchmarking - Staleness is defined as the period of time between the commit timestamp of the corresponding write operation (=timestamp at which the write terminates) and the point in time when the last replica is written. It is the difference between the master commit time and slave commit time in this case.

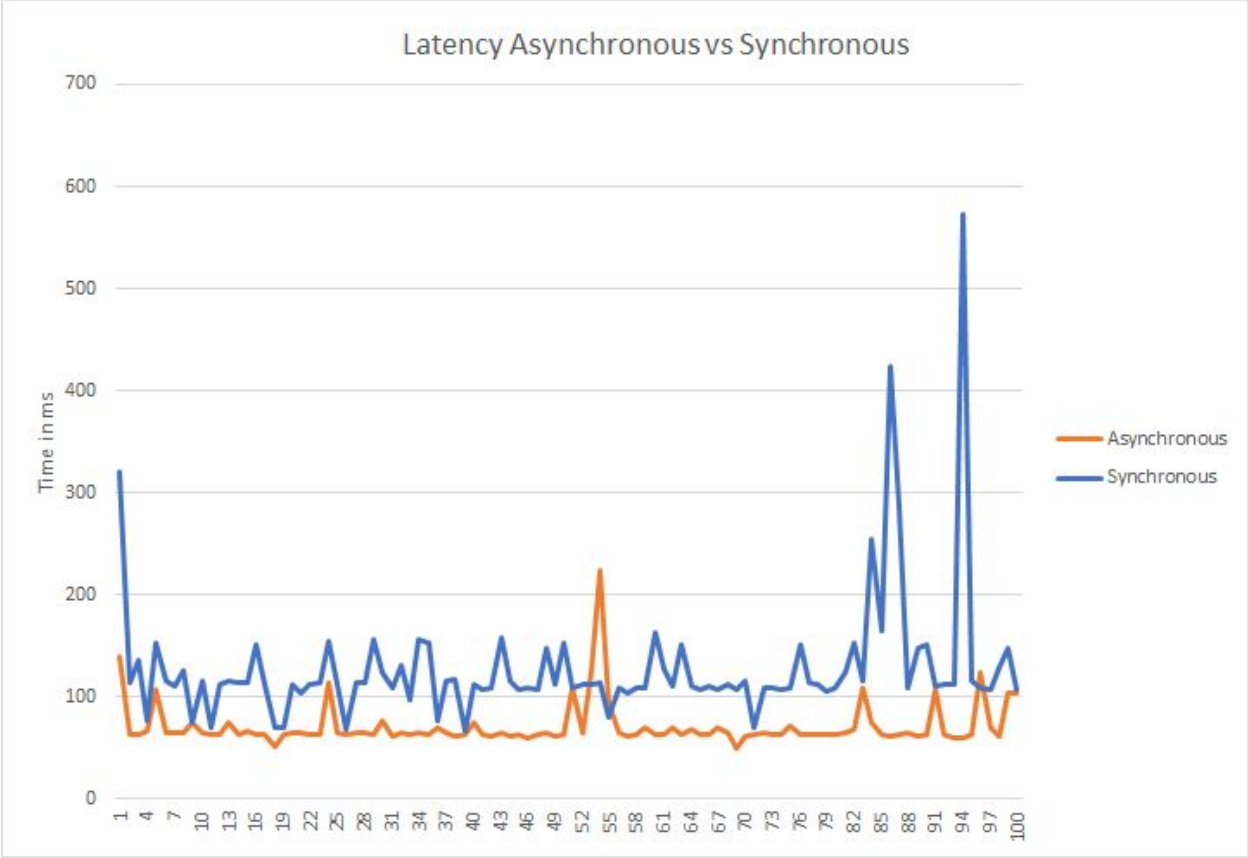
3. Analysis -

Latency measurements (Synchronous)

Minimum - 66 ms
Maximum - 572 ms
Average - 128.08 ms

Latency measurements (Asynchronous)

Minimum - 49 ms
Maximum - 223 ms
Average - 70.67



The results meet our expectation as latency in case of asynchronous is less than the latency during synchronous process. The round trip time is more in latter as the master waits for the confirmation from the slave before replying back to the client. This is not the case in asynchronous process where master reply back to the client as soon as it sends the request to slave for updating its data. However, the time difference is not very much which is also up to our expectations as both of our master and slave servers are in the same region which is in Europe.

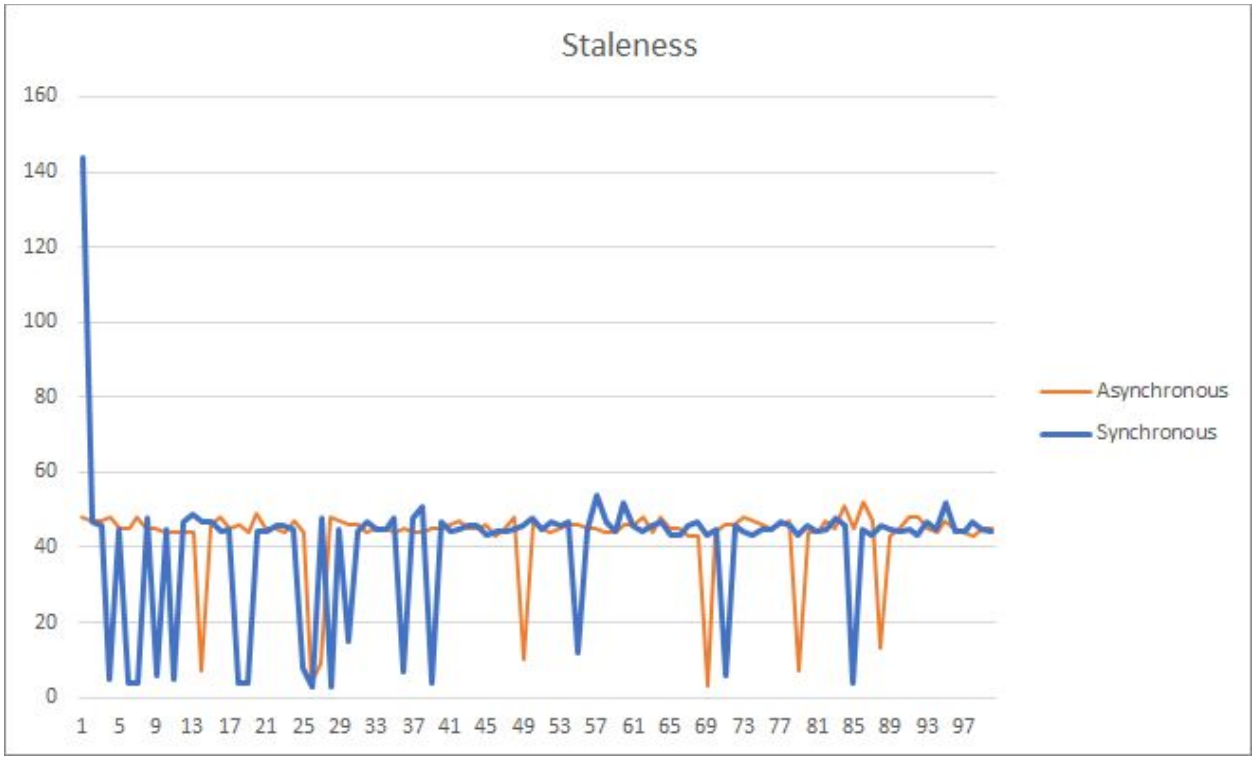
Staleness Benchmarking(Synchronous)

Minimum - 3 ms
Maximum - 144 ms
Average - 40.34 ms

Staleness Benchmarking(Asynchronous)

Minimum - 3 ms
Maximum - 52 ms
Average - 42.94 ms

The results in case of staleness also meet our expectations. As both the master server and the slave server are in the same region, there must be a very small difference in timestamp when master and slave update their data.


Staleness
160
140
120
100
80
60
40
20
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Asynchronous
Synchronous