MCC: A MICROSERVICES-BASED SYSTEM FOR CROWD COMPUTING

by

Sri Devi Rella

Supervised by: Dr. Ahmed Khaled

In Partial Fulfillment
of the Requirements for the Degree of
MASTER'S IN SCIENCE

Northeastern Illinois University
The Computer Science Department

MAY 2021

TABLE OF CONTENTS

ABSTRACT

The ever-growing inordinate capabilities of smart devices (smartphones, personal computers, etc.,) in terms of CPU, memory, sensory capability and autonomy, have the potential to enable the development of many crowd computing systems. Crowd computing is a way of solving computationally large or geographically spread tasks in a distributed manner. Solving computationally intensive tasks requires high-performance computing systems and immense processing time. Buying and maintaining such high-performance computing systems are very costly. However, by pooling the voluntarily shared idle processing powers of the hundreds and thousands of individual smart devices owned by the individuals, we could solve numerous computationally intensive tasks in a distributed manner.

In this research project, we propose and present a framework named MCC (Microservices-based System for Crowd Computing), a scalable distributed system that enables crowd computing paradigm on individual smart devices. Our framework provides a platform to the users to initiate tasks with a set of properties and files allowing the workers (Android or Desktop Clients) to volunteer for running such tasks on their devices. The framework divides a task into sub-tasks, assigns appropriate worker for each sub-task, accumulates the sub-results received from the workers, and sends the final result back to the user.

CHAPTER 1. INTRODUCTION

1.1 The research topic

It is an established fact that all the smart devices that we use (smartphones, personal computers etc.,) possess huge processing capabilities and resources. Solving computationally excessive tasks on a single device can be inefficient or computationally impossible to be accomplished. Also, the cost involved in the purchase and maintenance of high-performance computing systems tends to be excessive for several individuals, small businesses, and researchers with budget constraints. This is attained by optimally utilizing the voluntarily shared idle processing powers and resources of thousands of smart devices owned by the individuals. Such a crowd of individual smart devices and their processing cycles and resources remain untapped throughout the day. By combining the unused CPU cycles of mobile phones, small, mid, and big computers, we can resolve many large computing problems in an effective and distributed manner through crowd computing. Crowd computing, is fundamentally a distributed computing system where a rigorous task is divided into numerous sub tasks that are distributed over multiple computing devices for processing.

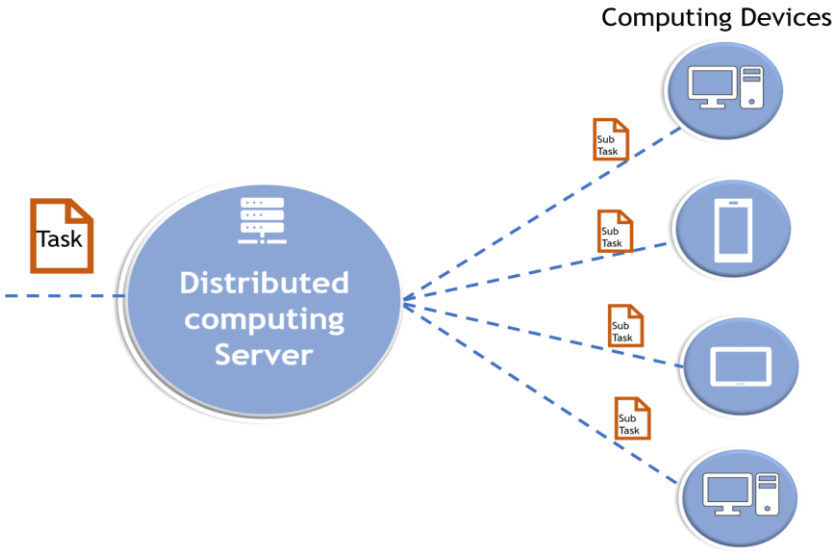


Figure 1 – Distributed computing model.

The advantages of distributed computing are

- **Scalability:** These computing devices communicate with each other by passing messages through a broker. There are no limitations on the number of devices that could be handled by the messaging broker which connects them and the memory. That allows us to add a new device to the broker when we need to scale up in case of high loads.

- **Performance:** A computing device can achieve high levels of performance by parallelly computing given sub-task (a subset of an overall task).

- **Fault tolerance:** One device failure doesn't affect other computing devices in the network.
- **Resource sharing:** Computing devices have their own memory and processors.

1.2 The Motivation and problem statement

Consider the following scenario as explained in [1]:

Rendering 3D models into a completely developed animation is quite CPU intensive and very tedious. For instance, if a small animation business owner or a student wishes to develop a two-minute full-fledged 3D animation movie, they would need to process 2880 images. It takes at least 24 hours of time in order to process a single image on one's personal computer (single device). So, a total of 2880 images would take 2880 days or roughly about 8 years. This is simplified by making use of super computers that are capable of rendering images in a much faster manner and mostly deployed by big animation render farms.

However, for a common man, Crowd Computing comes to the rescue. The rendering time of 8 years in this case could be reduced to one day or even hours by processing each image on one device thus processing all 2880 images on various computing devices in a distributed manner.

Take in the case of a researcher. The drug discovery and design problems involve screening of 180,000 compounds. Analysis of each drug compound on a personal computer can take 3 hours of time. So, the analysis of 180,000 compounds would take roughly 61 years in a traditional way. Instead, we can use the crowd computing system to execute 180,000 compounds parallelly on large collections of computing devices.

For anyone having a computationally big problem to solve, Crowd computing can be a vital solution by eliminating the need of supercomputers, immoderate costs, and huge processing time thus making it easier for anyone to access it.

1.3 Summary of the proposed solution

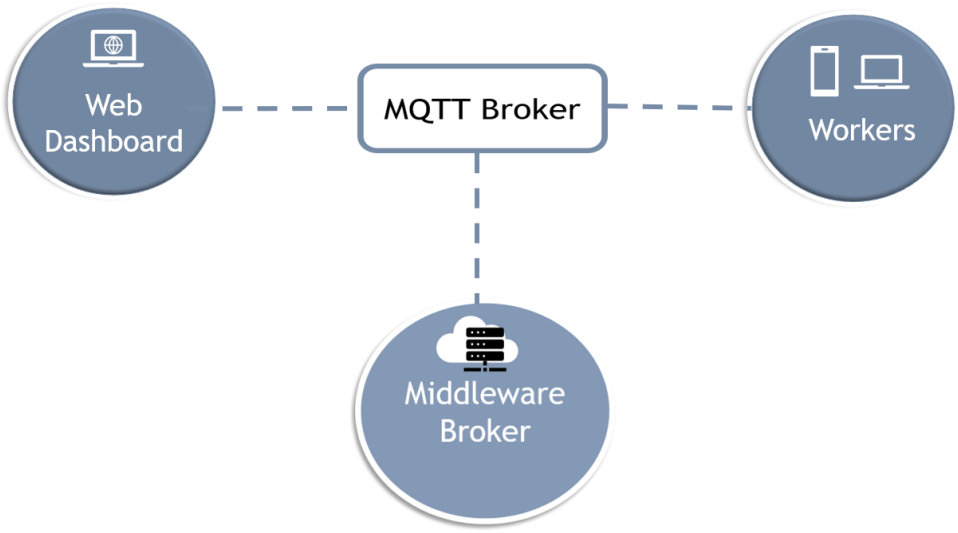


Figure 2 – Main Components of MCC.

MCC provides a web dashboard through which the user defines a task through a set of properties and a file that is to be sent to a middleware broker. All workers (devices owned by the people) who are ready to receive the tasks have to subscribe to the middleware

broker. The broker will receive the task from the web dashboard application and send the task description to all subscribed workers. The workers would need to decide if they are willing to run the task on their machine and send their approval message to the middleware broker before the task due date expires.

The broker then accepts the worker's response before the task due date expires. After the due date passes, the broker divides the task into multiple smaller sub-tasks (only if the task is divisible, otherwise, the task will not be divided) based on the number of workers who agreed to run the task and keeps them in the processing pool. The broker assigns appropriate worker (Android or Desktop Clients) for each sub-task depending on the operating system of the worker's device. The workers execute sub-tasks in the background and send computed results back to the broker. The broker accumulates the sub-results and sends the results back to the user, which will be displayed in the user's dashboard.

MCC uses MQTT (Message Queuing Telemetry Transport), a lightweight communication protocol that runs over TCP/IP to send and receive messages. The current version of MCC supports the following format tasks: pre compiled (.txt file with Java code), compiled (.apk or .jar).

1.4 Literature Review

The tradeoffs between existing crowd computing systems and MCC:

Berkeley Open Infrastructure for Network Computing (BOINC) is a distributed computing infrastructure that helps users all over the world in solving scientific problems that require huge computing power and was developed by the University of California, Berkeley. Currently, 791,443 smart devices are registered with the BOINC to contribute to the research projects in the field of science.

The World Community Grid organization developed a distributed computing system that is used for multiple voluntary computing initiatives. Cloud computing provides us a

distributed computing infrastructure that allows us to use virtual machine instances for computational purposes. But these crowd computing services are too costly to avail.

The existing crowd computing systems are maintained within organizations or institutions with restrictions that apply to the type of tasks and who can initiate the task. As compared to existing systems like BOINC, The World grid community, and Cloud Computing, our MCC framework is much more feasible and allows anyone to initiate a task through a web dashboard and collect results with minimal cost.

CHAPTER 2. PROPOSED SOLUTION AND RESULTS

2.1 Overall description of the proposed system/solution/algorithm

MCC has three components. 1) The web dashboard, 2) The middleware broker, and 3) The workers. All three components communicate with each other through a MQTT broker.

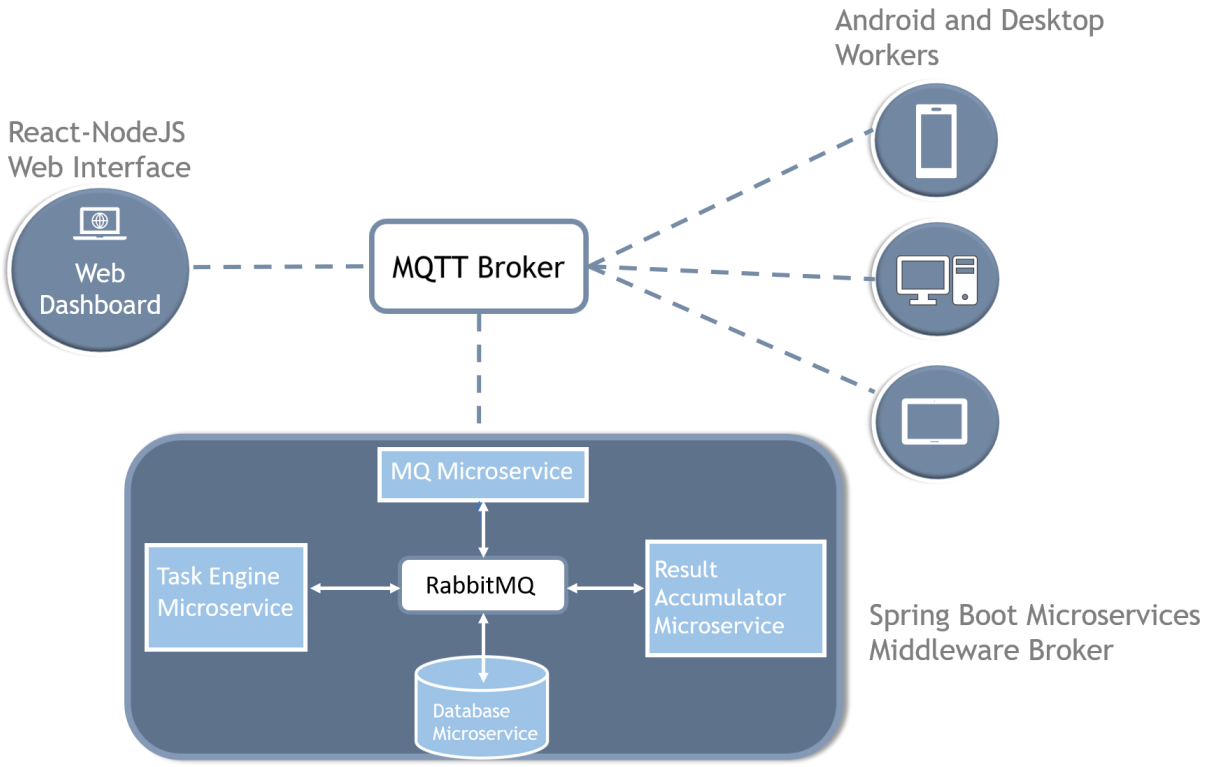


Figure 3 – MCC Architecture.

1) **Dashboard:** It is a React, NodeJS based web application that allows the user to define a task with a set of properties and a file. That also displays the computed task results to the user. The React and NodeJS server use web sockets for the communication.

- **Modules:**

    i. **MQTT connection**: This module establishes a connection with the MQTT broker through an MQTT client library to publish and receive the messages (tasks or results) from the middleware broker.

    ii. **Define Task**: This allows the user to define a task through a set of properties such as a short name for the task, a brief description about the task, due date to run the task on the worker device, the actual size of the task, the author who initiated the task, and the rewards to the worker for executing the task.

    iii. **Task Result:** This module displays the computed result of the task that received by the middleware broker.

2) **The Middleware Broker:** A Spring Boot microservices server application that acts as a middleman between user dashboard application and worker application. It has 4 main modules or microservices. All four microservices register with Eureka service discovery and uses RabbitMQ messaging queue for their inter-service communication.

    - **Modules:**

        i. **MQ Microservice:** This module establishes a connection with the MQTT broker through an MQTT client library to publish and receive the messages. It acts as a gateway between the MQTT broker and the other microservices.

        It is responsible for registering subscribed workers, receiving the task details from the user dashboard app, sending task description to the workers, receiving the acceptance response by the workers who agreed to run the task on their device, sending sub-tasks to

workers, collecting sub-task results that send by the workers, and sending accumulated single result back to the user dashboard application.

ii. **DB Microservice:** It Provides data persistence by storing data in the database. It is responsible for applying CURD operations on received tasks, worker responses, sub-task results, and accumulated results.

iii. **Task Engine Microservice:** It takes a task and subscribed worker details from the DB service to select an appropriate worker for that task. This service divides the task into multiple sub-tasks based on the number of workers who agreed to run the task on their device and selects the appropriate worker per sub-task based on the worker device OS.

iv. **Result Engine Microservice:** This module accumulates the sub-task results that brought in by each worker into a single result which will be send to display on the user's web dashboard.

3) **The Workers:** The workers could be android devices or desktop devices. It has two sub components. 1) The mobile client, and 2) The desktop client.

- **The Mobile Client:** It is an android application that runs on any android-based mobile device to receive tasks in the form of .apk from the middleware broker, accepting or rejecting the tasks, running .apk sub-tasks in the background, and broadcasting the computed result back to the main application. The main application sends the computed result back to the middleware broker through MQTT.

- **The Desktop Client:** It is a JavaFX application that runs on any desktop machine to receive tasks in the form of .jar from the middleware broker, accepting or rejecting the tasks, running .jar sub-tasks in the background and, sends the computed result back to the middleware broker through MQTT.

MQTT Broker: A **"Mosquitto"** messaging broker that Implements a lightweight protocol called MQTT (Message Queueing Telemetry Transport) protocol. The MQTT broker follows a topic-based publish-subscribe messaging model. In this model, a producer publishes messages to the topic and the consumer who is subscribed to the topic receives the messages which are published to the topic by the producer. It is responsible for receiving all messages from clients, filtering the messages, determining who is subscribed to each message, and sending the messages to those clients.

2.2 Algorithmic description of the system

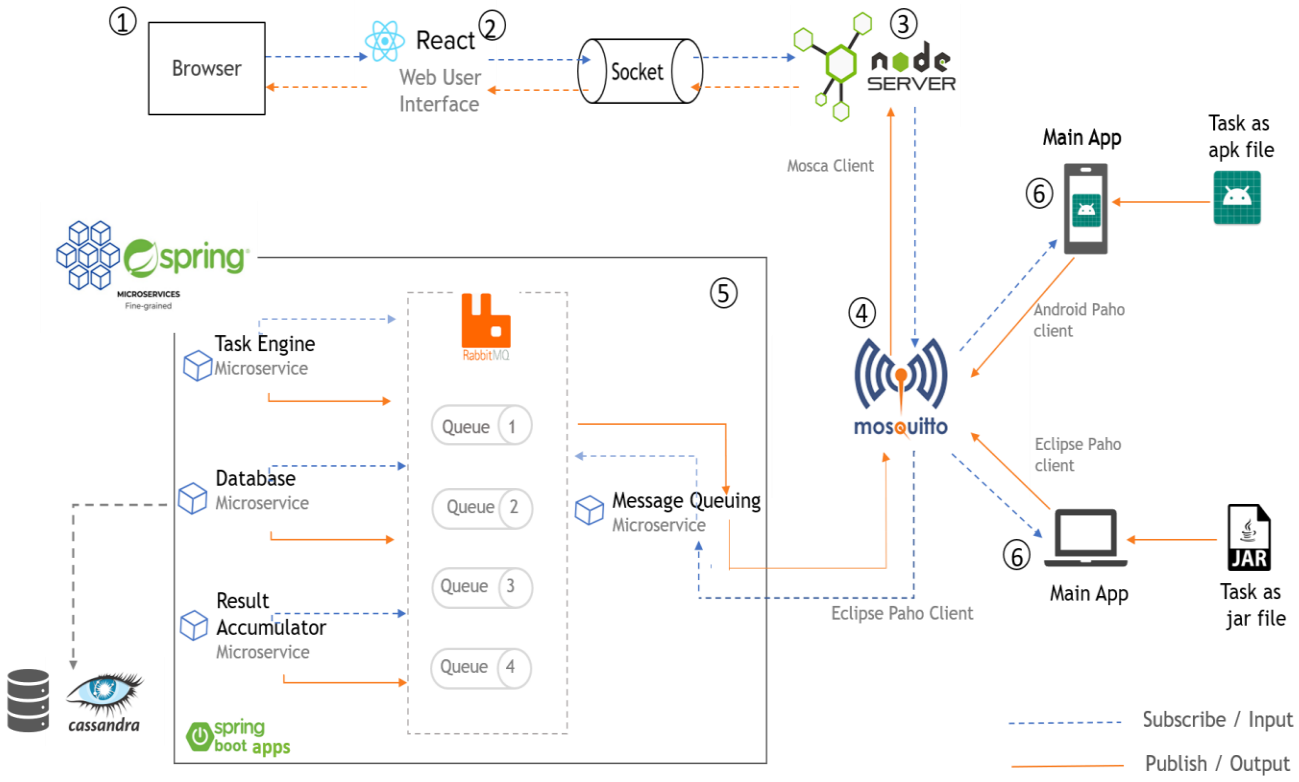


Figure 4 – MCC Data-flow view architecture.

Figure 4 Represents the data-flow view architecture and the technology used in the system.

- This system accepts inputs from a user interface accessible through a browser. A user submits a task through this user interface which post a http request with task as payload to a react application.
- This application consumes an API end point exposed by the node server through axios.post () method to post the request input sent by user from user interface.
- The node server implements a mosca client to publish and receive messages from the MQTT broker, publishes received request message on the corresponding topic hosted on the Mosqutto broker.
- The message queuing microservice in the backend server implements Eclipse Paho client to publish and receive messages from the MQTT broker.
- The message queueing micro service receives the message on the corresponding topic and forwards the message to the database.
- Once database receives message, it sends available worker's details along with task description only to Task Engine microservices through RabbitMQ message queue.
- The task engine microservices will process the database request to map task description details to available workers.
- It then sends the processed message to the messaging queue microservice. This microservice in turn publishes the message to the Mosquitto broker.
- Once this message is received by Mosquitto broker broadcasts the message to the available workers. In our context we are currently using two available workers viz., android and desktop worker. Both android and desktop will subscribe to receive task details. They have an option to accept or decline this request.
- If the workers accept the request a response message is sent back to the spring boot microservices backend server by the accepted workers.
- The message queueing microservice publishes the received response messages by the accepted android and/or desktop workers to the database microservice through corresponding RabbitMQ queue for further processing.

- This process creates corresponding build files (ex: .apk for android, .jar for desktop). Once these build files are received by accepted workers, they compute the task based on inputs provided in their respective build files.
- The workers on completing the computation, publishes the computed result as response message to the Mosquitto broker.
- The result accumulator microservice accumulates the received results and send it back to user interface all the way through, message queueing microservice, Mosquitto broker and Node server.
- The user can access these accumulated results under the result tab on user interface.

Task division and assignment implementation by Task engine microservice

There are two kinds of inputs a user can input for a task through the user interface. A .txt file or an executable file (ex: .apk, .jar) when an executable file is sent as input, it is sent to corresponding available accepted workers depending on the worker's operating system. For instance, If the input is a .apk, the task engine microservice selects workers with type "Android" from the pool of available accepted workers and sends the .apk file to them.

On contrary if an .txt file is sent as an input, it is divided into sub tasks by Task Engine microservice. The following steps are performed as part of task division and assignment.

Step1: All subscribed workers are sent an invite containing a task description to accept or decline.

Step2: Once accepted a response is sent back to Task Engine microservices.

Step3: The microservice then divides the input based on number of accepted workers.

Below is an example task and its process.

```
className:Primes;;
inputRange:1,2000;;
code:import java.util.ArrayList;
import java.util.List;

public class Primes {

    private String result;

    public Primes(int startIndex, int endIndex) {
        this.result = calculatePrimes(startIndex, endIndex);
    }
    public String calculatePrimes(int startIndex, int endIndex) {

        List<Integer> primes = new ArrayList<>();

        for (int i = startIndex; i < endIndex; i++) {
            boolean isPrimeNumber = true;

            for (int j = 2; j < i; j++) {
                if (i % j == 0) {
                    isPrimeNumber = false;
                    break;
                }
            }
            if (isPrimeNumber) {
                primes.add(i);
            }
        }
        return primes.toString();
    }

    public String getResult() { return result;}
}
```

Figure 5 – Input .txt file.

- Task: Find all prime numbers between 1 and 2000.
- Number of accepted workers: 2
- Accepted worker types: Android and Desktop.

a. The task engine parses the .txt file to get input range, class name, and code corresponding tag values.

b. For the above example it divides the task input range 1 to 2000 into two distinct inputs, (1,1000) and (1001, 2000) respectively.

c. Task engine uses these inputs, along with java code available under the tag “code” to generate classes TaskImplementaion.java, TaskName.java programmatically.

d. Depending on the worker’s type (Android or Desktop) it adds generated classes into the respective template project folder (Android project source code or Java project source code) and generates respective executable file by using Gradle tooling API.

Below are some of the tags and their description.

- **Input Range**: The tag specifies the input range for the task. It is used by Task Engine to divide the input range depending on the number of available accepted workers.

- **Code**: The tag specifies the computational task in the form of java code. It is used by Task Engine to generate a TaskName.java class using the java code available under this tag.

- **Class Name**: This tag is used by Task Engine to initialize TaskName.java class constructor with input ranges from TaskImplementation.java class.

Some important functionality code snippets.

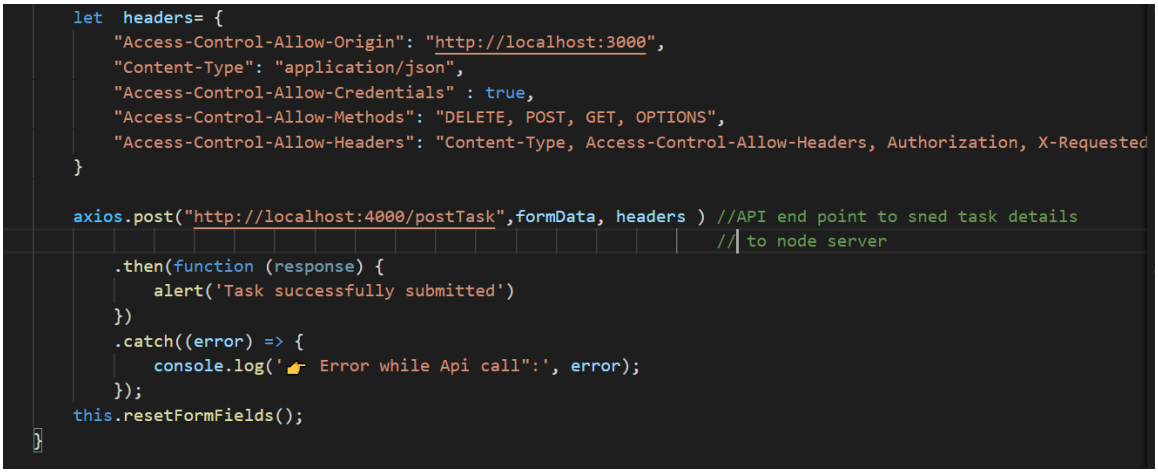


Figure 6 – Consuming Node API end point to post task details.

```
let mosca = require('mosca');
let settings = {
    port: 1883
}

//Setup mosca Mqtt server
let server = new mosca.Server(settings);
server.on('ready', function () {
    console.log("ready");
});
```

Figure 7 – Mosca MQTT server setup at NodeJS level.

```
//Setup connection listener on react-node web socket.
io.on("connection", (socket) => {
    reactSocket = socket;
    console.log("New client connected");
});
```

Figure 8 – Web socket connection listener setup at React level.

```
//API endpoint to receive task details from React and also publish tasks to the MQTT.
app.post('/postTask', (req, res) => {

    let taskDetailsObj = JSON.parse(req.body.taskDetails)
    let finalResult  = {};
    let taskResult = {
        "executableFile": {
            "name": req.files.executableFile.name,
            "data": encode(req.files.executableFile.data), // encode with Base64
            "size": req.files.executableFile.size,
            "mimetype": req.files.executableFile.mimetype,
            "md5": req.files.executableFile.md5
        },
        "taskProperties": {
            "shortName": taskDetailsObj.shortName,
            "description": taskDetailsObj.description,
            "dueDate": taskDetailsObj.dueDate,
            "size": taskDetailsObj.size,
            "author": taskDetailsObj.author,
            "rewards": taskDetailsObj.rewards
        }
    };
    finalResult.task =taskResult;
    finalResult= JSON.stringify(finalResult);
    client.publish(topic_initiate_task, finalResult);
    console.log("Successfully submitted the task")
    res.send().status(200);
});
```

Figure 9 – Code for publishing task details to the MQTT.

```java
private void connectWithOptions(MqttConnectOptions options) throws MqttException {

    if (!mqttClientInstance.isConnected()) {
        mqttClientInstance.connect(options);
        System.out.println("connected to: "+ brokerUrl);
        mqttClientInstance.subscribe(TopicName.MAIN_TOPIC.getTopicName());
    }
}

/**
 * Create MQTT connection options.
 * @return connection options of type MqttConnectOptions.
 */
private MqttConnectOptions getMqttConnectOptions() {

    MqttConnectOptions options = new MqttConnectOptions();
    options.setAutomaticReconnect(true);
    options.setCleanSession(false);
    options.setConnectionTimeout(10);
    mqttClientInstance.setCallback(mqttCallBack);
    return options;
}
```

Figure 10 – Code to establish a connection with MQTT broker through IMqttClient.

```java
private void inputRangeDivision(JSONObject textFileContentJson, int numberOfDivisions,
                                List<String> subTaskInputRanges) throws JSONException {

    String[] inputRange = textFileContentJson.getString( s: "inputRange").split( regex: ",");
    int divisionValue = Integer.parseInt(inputRange[1]) / numberOfDivisions;
    final int[] previousValue = {divisionValue};
    subTaskInputRanges.add(inputRange[0] +","+divisionValue);

    if(numberOfDivisions > 1) {

        IntStream.rangeClosed(2, numberOfDivisions - 1).forEach( inputRangeValue -> {

            int startIndex = previousValue[0] + 1;
            int endIndex = inputRangeValue * divisionValue;

            subTaskInputRanges.add(startIndex + "," + endIndex);
            previousValue[0] = endIndex;
        });
        subTaskInputRanges.add(previousValue[0] + 1 + "," + inputRange[1]);
    }
}
```

Figure 11 – Task input range division based on the number of workers who agreed to run the task on their device.

```
private void performGradleTasks(String path, String ... tasks)
{
    GradleConnector connector = GradleConnector.newConnector(); // initiate gradle connector
    connector.forProjectDirectory(new File(path));
    ProjectConnection connection = connector.connect();  // create project connection

    try {
        // configure the build
        BuildLauncher launcher = connection.newBuild();
        launcher.forTasks(tasks);
        launcher.setStandardOutput(System.out);
        launcher.setStandardError(System.err);
        // Run the build
        launcher.run();
    } finally {
        //Clean up
        connection.close();
    }
}
```

Figure 12 – Code for generation of executable files (.apk and .jar).

```
private void establishConnection(MqttConnectOptions options) {

    if ( !mqttClientInstance.isConnected() ) {
        try {
            mqttClientInstance.connect(options);
            connectionStatus.onStatusChange("Successfully connected to MQTT\n\nTo receive tasks
            System.out.println("connected to: " + Constants.URI.getConstant());
        } catch (Exception e) {
            connectionStatus.onStatusChange("Failed to connect MQTT");
        }
    }
}

/**
 * MQTT connection options.
 */
private MqttConnectOptions getMqttConnectOptions() {

    MqttConnectOptions options = new MqttConnectOptions();
    options.setAutomaticReconnect( true );
    options.setCleanSession( false );
    options.setConnectionTimeout( 10 );
    return options;
}
```

Figure 13 – Desktop and MQTT connection.

```
Process p = Runtime.getRuntime().exec( command: "java -jar " + Paths.get(FilePaths.getTaskPath().toString(),
                                        Constants.EXECUTABLE_FILE_NAME.getConstant()).toString());
BufferedInputStream bis = new BufferedInputStream(p.getInputStream());

System.out.println("Jar execution initiated");
int err = 0;
BufferedInputStream errBis = new BufferedInputStream(p.getErrorStream());
while ((err = errBis.read()) > 0) {
    System.out.print((char) err);
}

synchronized (p) {
    p.waitFor();
}
System.out.println(p.exitValue());
int res = 0;
StringBuilder subTaskResult = new StringBuilder();

while ((res = bis.read()) > 0) {

    System.out.print((char) res);
    subTaskResult.append((char) res);
}
```

Figure 14 – Code for executing the .jar file (sub-task) on the desktop device.

```
/**
 * Install .apk on the device from the given file path .
 */
public void initiateInstallation(String downloadFile, String taskId) {
    try {
        File f1 = new File( pathname: external_dir_path+"/"+downloadFile);
        Intent intent = new Intent(Intent.ACTION_VIEW);

        TaskDetails.setTaskId(taskId);

            if (android.os.Build.VERSION.SDK_INT >= Build.VERSION_CODES.N) {
                intent.setDataAndType(FileProvider.getUriForFile(context,  authority: BuildConfig.APPLICATION_ID +
                                      ".provider", new File( pathname: "/data/data/com.android.mqttclient/files/" + downloadFile)),
                                    type: "application/vnd.android.package-archive");
                intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
            } else {
                intent.setDataAndType(Uri.fromFile(new File
                ( pathname: "/data/data/com.android.mqttclient/files/"+downloadFile )),  type: "application/vnd.android.package-archive");
            }

        context.startActivity(intent);                    // initiates activity with the given intent
    }catch (Exception e){
        Log.d( tag: "Apk Installation",  msg: "Exception = "+e.getMessage());
    }
}
```

Figure 15 – Code for running the .apk file (sub-task) on the Android device.

```
 */
public static class MyBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent != null) {
            String stringIntentAction = intent.getAction();

            if (stringIntentAction != null && stringIntentAction.equals("RESULT_ACTION")) {  // checks for intent action identifier
                try {
                    JSONObject taskResultJsonObj = getTaskResultJsonObj(Constants.WORKER_ID.getConstant());
                    JSONObject taskResult = new JSONObject(intent.getStringExtra( name: "TaskResultData"));
                    taskResultJsonObj.accumulate( name: "subTaskResult", taskResult.getString( name: "result")); // append  sub task result to sub ta
                    String taskId = taskResult.getString( name: "taskId");
                    taskResultJsonObj.accumulate( name: "taskId", (taskId == null) ? TaskDetails.getTaskId() : taskId);
                    launchViewModel.getRwfInstance().unInstallApk();

                    if (taskResultJsonObj != null) {
                        String responseTopic = Constants.WORKER_SUBTASK_RESPONSE.getConstant();
                        launchViewModel.onExecutedTask( description: "Executed response topic description");
                        launchViewModel.getMqttServicesInstance().publishMessage(responseTopic, taskResultJsonObj.toString() ); // publish sub task
                        launchViewModel.getRwfInstance().deleteFile(null);                  // delete sub task apk file from device internal memory
                        //rwfInstance.unInstallApk();                                       // uninstall sub task apk on the device
                    }
                } catch (JSONException e) {
                    e.printStackTrace();
                }
```

Figure 16 – Code for receiving the computed result by the main app from the executed .apk file.

2.3 Technical details of the components

Hardware Requirements:

- RAM: 512 MB (minimum), 1 GB (recommended).
- Hard disk: up to 4 GB of available space may be required.
- Processor: x86 or x64.

Software Requirements:

- Windows XP/Vista/7/8/10
- IDE: Intellij/Eclipse/WebStrom/Android Studio
- Project GitHub Repository:
  https://github.com/sridevirella/crowdcomputing-masters-main-project

The technical details of each component

- **Dashboard:**

  - **Used Technologies:** React, NodeJS, Express, Mosca MQTT client, Bootstrap, Material-UI, and Socket.io for the communication between React and Node server.

  - **Installation:**

    - Download and install Node.js version v14.16.0 or higher and NPM through a Node version manager or the installer from https://nodejs.org/en/download/.

    - For installation steps, you can refer https://docs.npmjs.com/downloading-and-installing-node-js-and-npm.

    - Through the terminal, download and install React using the command "**npm install react**".

    - From GitHub, clone both repositories (**/module1-dashboard/mcc-dashboard-nodejs/, /module1-dashboard/mcc-dashboard-react**/) into your local machine.

  - **Run:**

    Note: You need to run the NodeJS app before running the React app.

- For the NodeJS app, open a terminal, navigate to the project folder and run the command "**npm install**" to install all project dependencies into the node_modules folder. Then start the application by running the command "npm start". Alternatively, you can import the project into IDE (e.g., Visual studio code, WebStorm) and run the application.

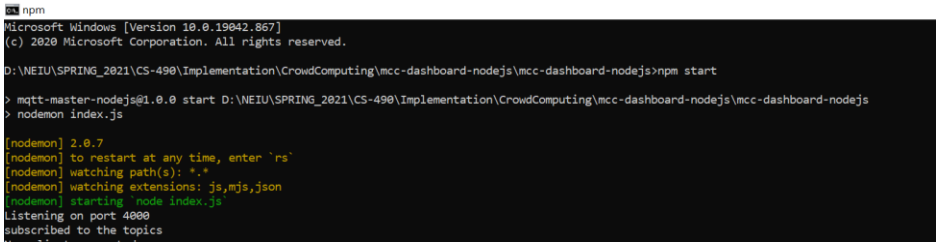


Figure 17 – Running NodeJS Application.

- For the React app, open a new terminal, navigate to the project folder and run the command "**npm install**" to install all project dependencies into the "node_modules" folder. Then start the application by running the command "npm start". Alternatively, you can import the project into IDE (e.g., Visual studio code, WebStorm) and run the application.

Figure 18 – Running React Application.

Middleware Broker:

- **Used Technologies:** Java 8, Java Spring Boot (v.2.3.5) Microservices, Netflix Eureka Server, Async Eclipse Paho Java Client (v1.1.0), Apache Cassandra Database, Spring Data Cassandra, RESTAPI, Spring Cloud Stream, RabbitMQ, Maven Build Tool.

- **Installation:**
  - Download and install JDK 8 from https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html.

- For installation, you can refer https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html.
- From GitHub, clone repository **/module2-middleware-broker**/ into your local machine.

- Download and install Apache Cassandra from https://cassandra.apache.org/doc/latest/getting_started/installing.html

- Download and install RabbitMQ from https://www.rabbitmq.com/download.html.

➢ **Run:** To run the Spring Boot Microservices application follow below steps

- Navigate to the **bin** folder located within the Apache Cassandra folder, open a terminal and run the command "**Cassandra**" to start the Cassandra database instance.

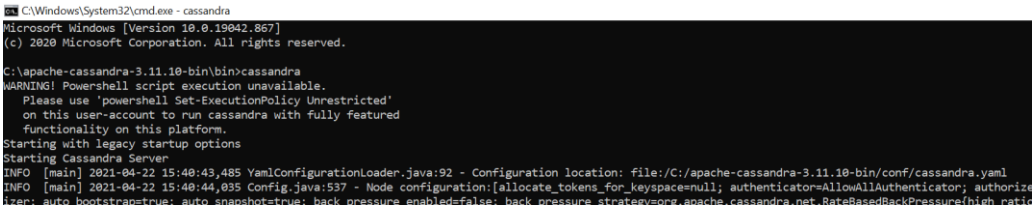


Figure 19 – Running Cassandra Instance.

- Open a new terminal and execute the command "**cqlsh**" to start the CQL shell. In the shell, run the CQL commands or schemas from the file **(/module2-middleware-broker/Cassandra-sql-queries.txt)** to create key spaces and the tables in the database.

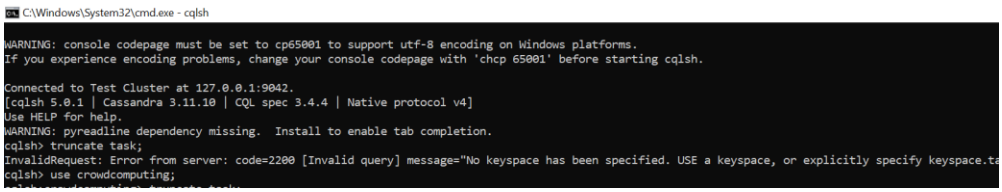


Figure 20 – CQL shell.

- After successful installation of RabbitMQ, in a browser open localhost:15672, you should be able to see running RabbitMQ instance.

- In path **/module2-middleware-broker/microservices-app/** navigate to each microservice application folder (e.g., **/microservices-app/eurekaserver**) and open a separate terminal (or) git bash for each microservice and run the command **"./mvnw spring-boot: run".** The "**eurekaserver**" needs to be run before running any other microservices. Alternatively, you can import the project into IDE (e.g., IntelliJ, Eclipse) and run the application.

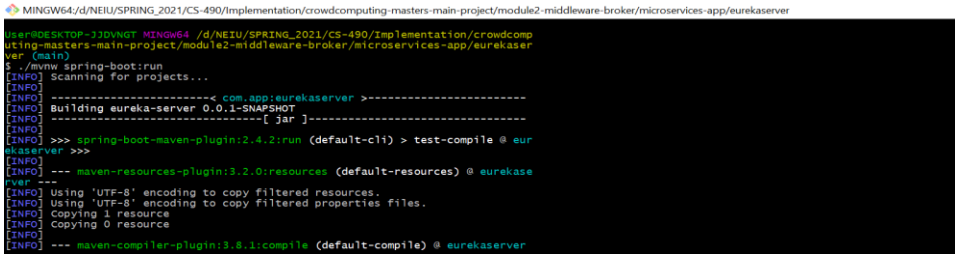


Figure 21 – Running Eureka Server Instance.

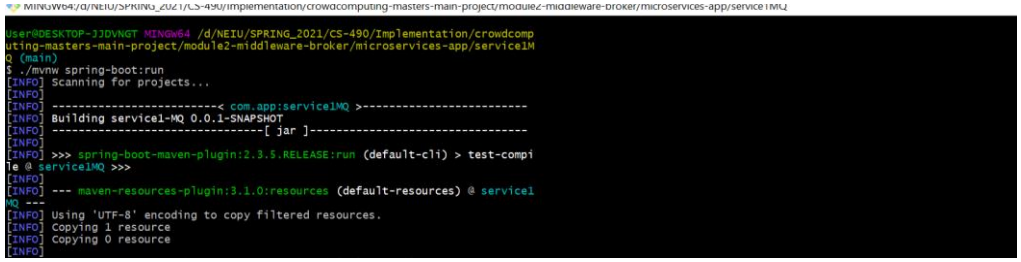


Figure 22 – Running MQ Microservice Instance.

Figure 23 – Running DB Microservice Instance.

Figure 24 – Running Task Engine Microservice Instance.

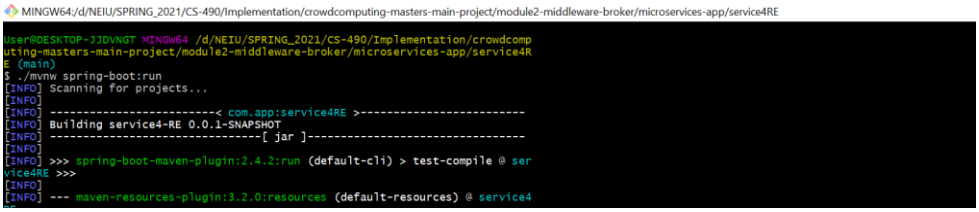


Figure 25 – Running Result Accumulator Microservice Instance.

- **Workers:**

  - **Used Technologies:** Java 8**,** Android, Async Eclipse Paho Android Client, Broadcast Receiver, Java 11 for JavaFX, Async Eclipse Paho Java Client, and Gradle Build tool.

  - **Installation:**

    - Download and install JDK 8 and JDK11 from https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html.

    - For installation, you can refer https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html.

    - From GitHub, Clone both repositories **(/module3-workers/MqttClient/, (/module3-workers/cc_desktop_client/)** into your local machine.

➢ **Run:**

- For the mobile client, import the project into android studio IDE and select **'app'** from the run/debug configurations drop-down menu in the toolbar. Enable developer options in your android mobile device and connect it to your computer. In the toolbar, select your connected mobile device to run the app from the target device drop-down menu. Alternatively, you can run the application on the emulator as well.

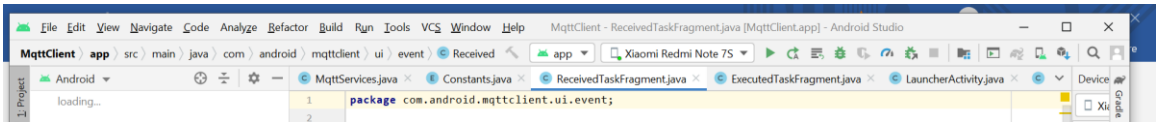


Figure 26 – Installation of Application on Android Device (Redmi note 75).

- Import the project into an IDE (IntelliJ, Eclipse). To run the JavaFx application for the first time from the IDE follow the following steps:
  Go to Gradle -> cc_desktop_client -> Tasks -> application -> run.

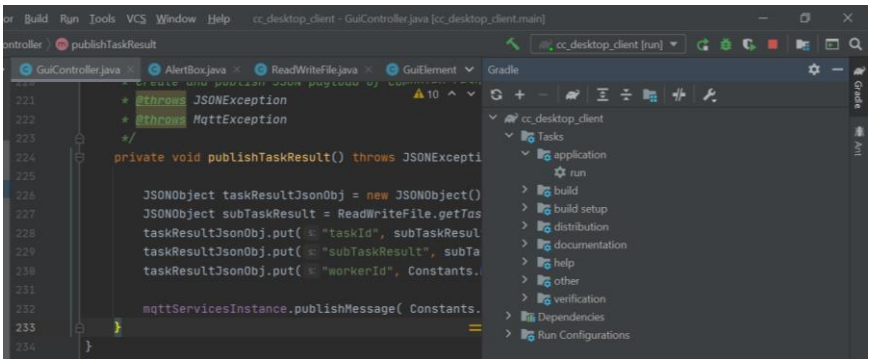


Figure 27 – Running Desktop Application through IntelliJ IDE.

Steps to run complete system

(To run any component, refer to the component's Run section for clear instructions.**)**

- Open a terminal in administer mode and start the Mosquitto broker using the command "**net start mosquitto"**.

- Open a new terminal and start all the microservices instances of the middleware broker using command **"./mvnw spring-boot: run".**

- Now, go to your browser and type the Eureka server URL **"localhost:8761"**, you should be able to see all four running microservices instances in eureka server page.

- To start the dashboard app, open one terminal for each React and nodeJS application then run the command **"npm start"** for both.

- To open the dashboard, go to your browser and type the URL "**localhost:3000**", you should be able to see the login page of the dashboard.

- Install and run the Android or the desktop application on your device as mentioned under the workers component (see the "**run**" section).

2.4 Results and discussion

- The middleware broker's (Spring boot microservices) all microservices are up and running.

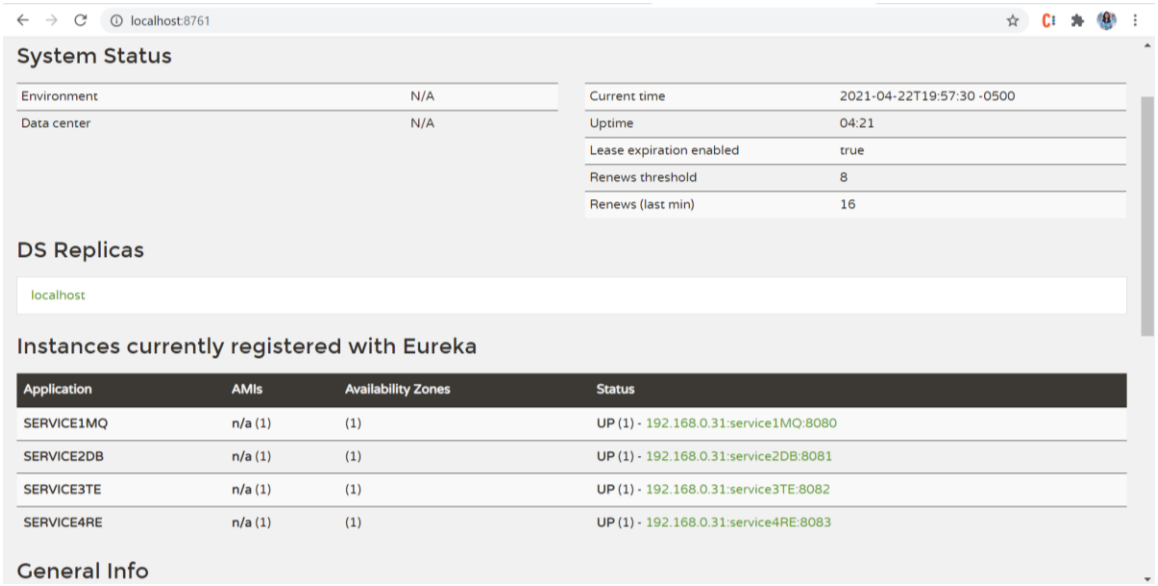


Figure 28 – Eureka Server Console.

- The RabbitMQ is up and running.

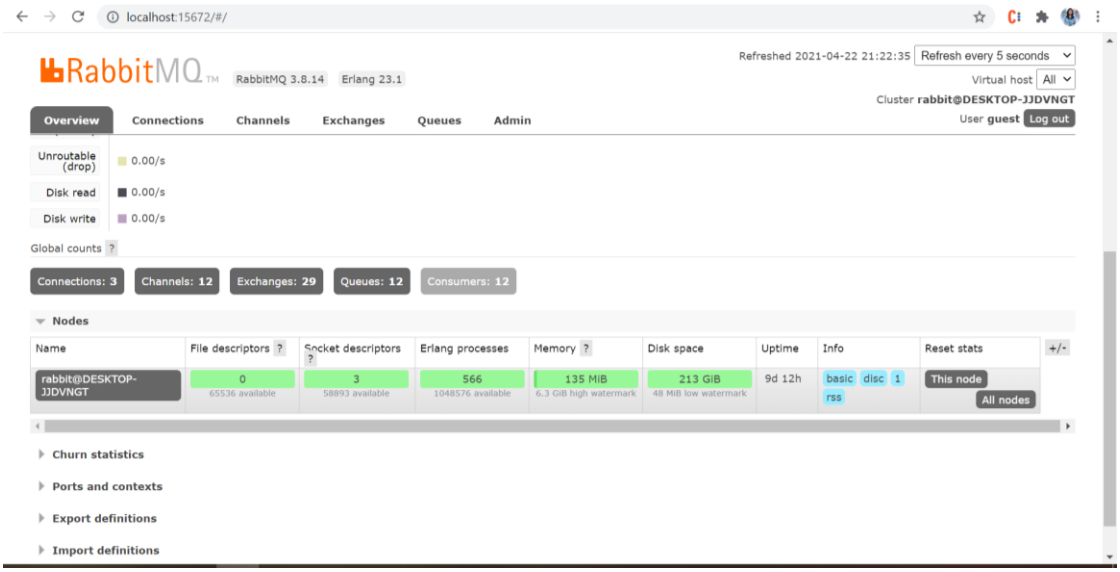


Figure 29 – RabbitMQ Console.

- Android mobile worker application connected to MQTT Broker.

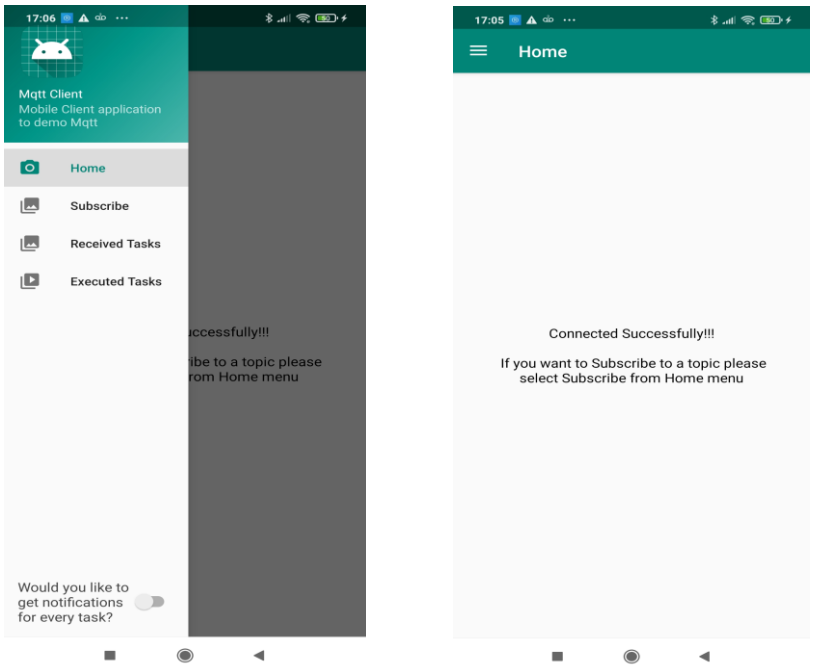


Figure 30 – Android worker MQTT connection.

- Android worker subscribed to receive the task from the middleware broker.

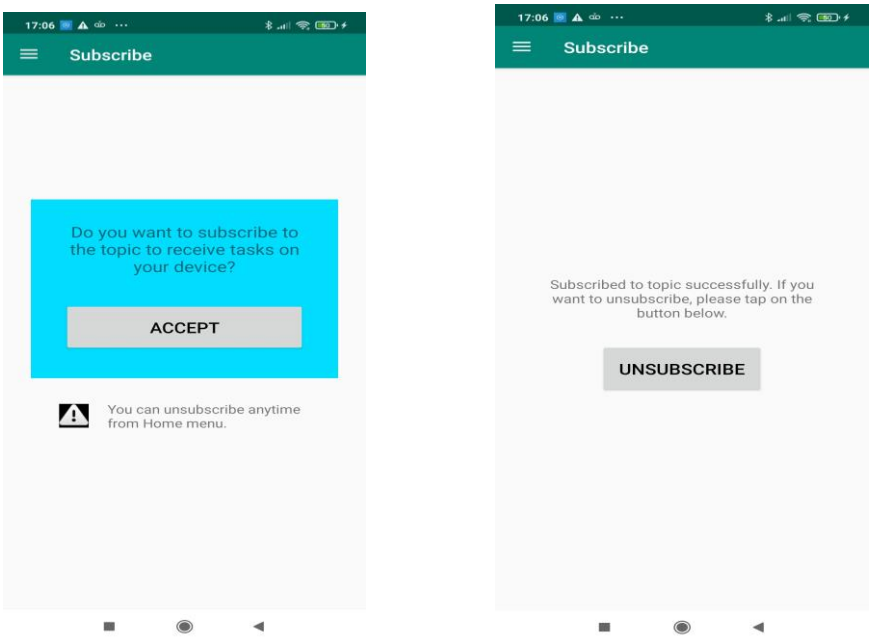
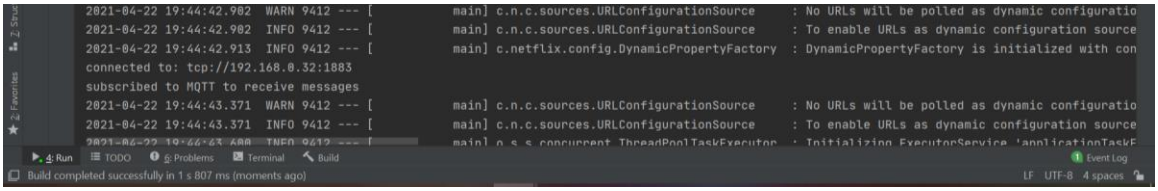


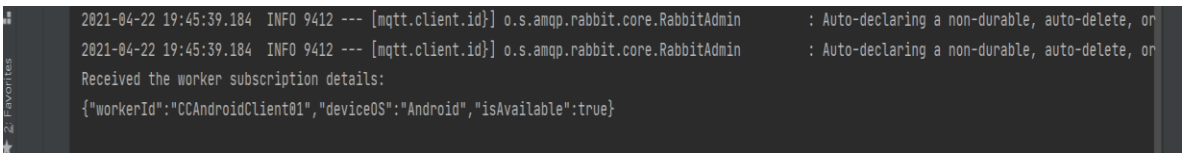
Figure 31 – Android worker subscription.

- The middleware broker MQ microservice connected to the MQTT broker.



- **Figure 32 – The middleware broker MQTT connection.**

- The Middleware Broker received subscribed Android worker details.



- **Figure 33 – Details of desktop worker received by the broker.**

- Desktop JavaFX worker application connected to MQTT Broker.

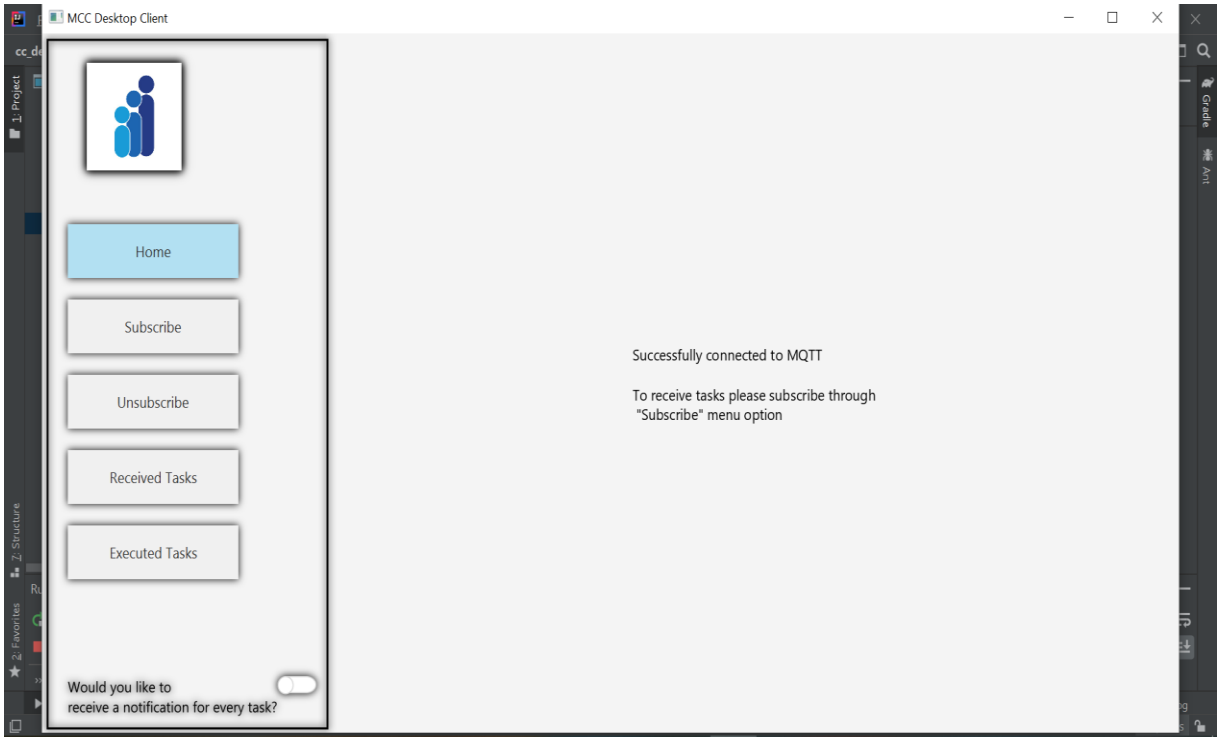


Figure 34 – Desktop worker MQTT connection.

- Desktop worker subscribed to receive the task from the middleware broker.

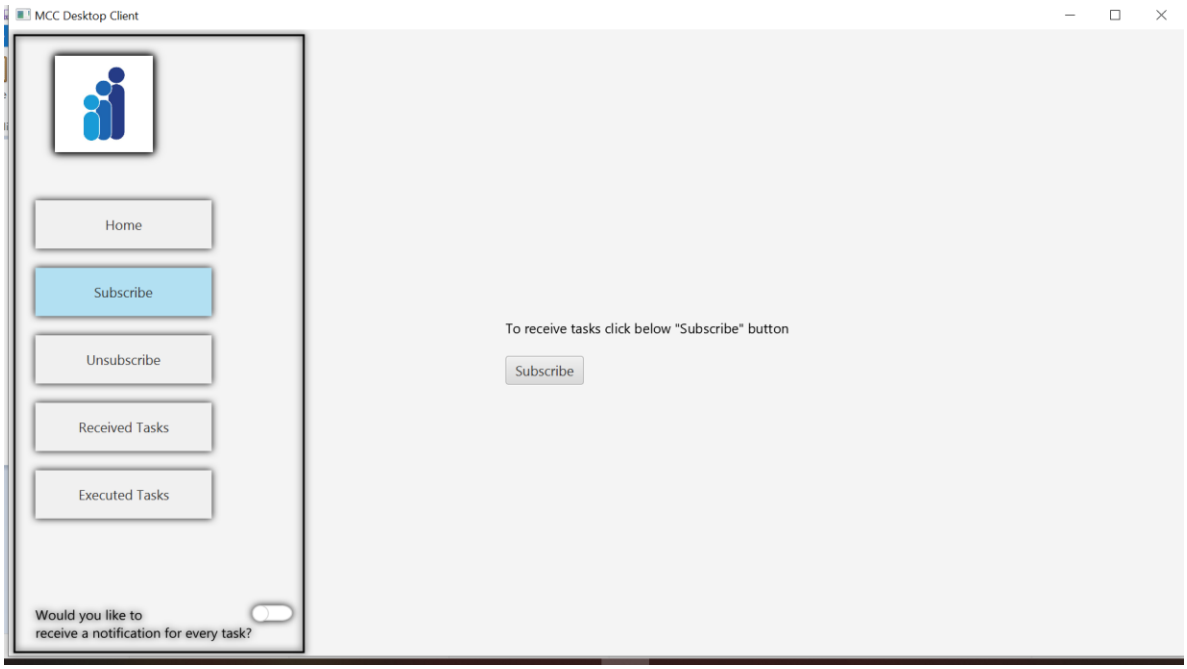


Figure 35 – Desktop worker subscribe tab.

Figure 36 – Desktop worker subscription.

- The Middleware Broker received subscribed Desktop worker details.

Figure 37 – The received details of the desktop worker.

- Login into web dashboard with credentials username:

Figure 38 – Dashboard login page.

- Initiated task to calculate prime numbers from 1 to 2000.

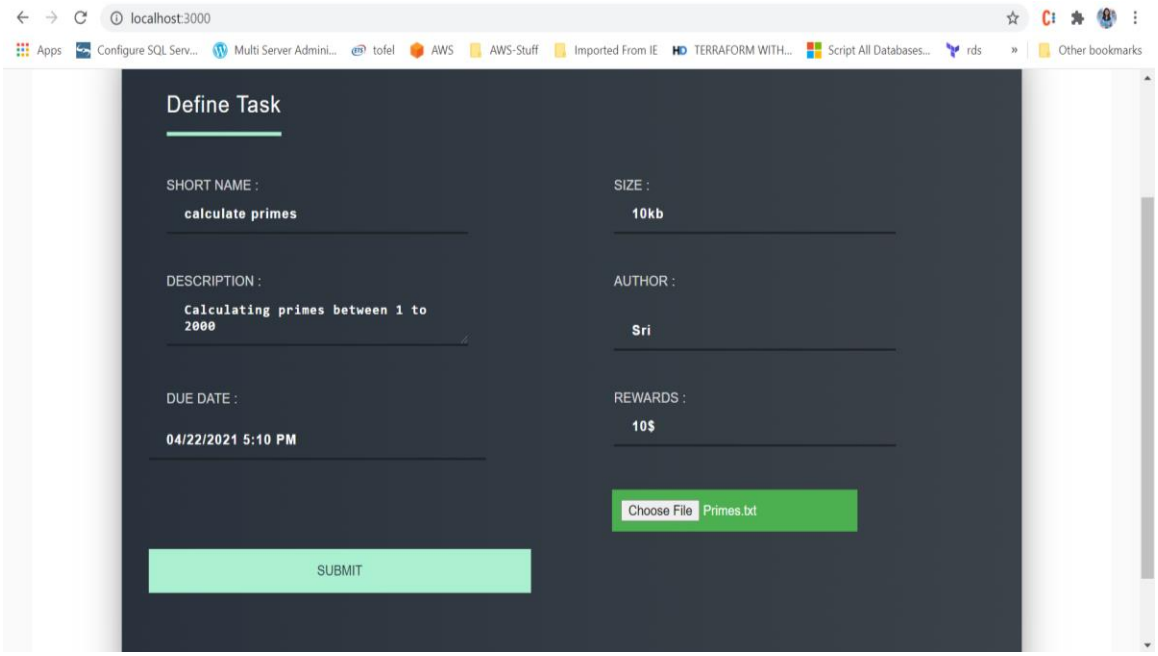


Figure 39 – Task Initiation.

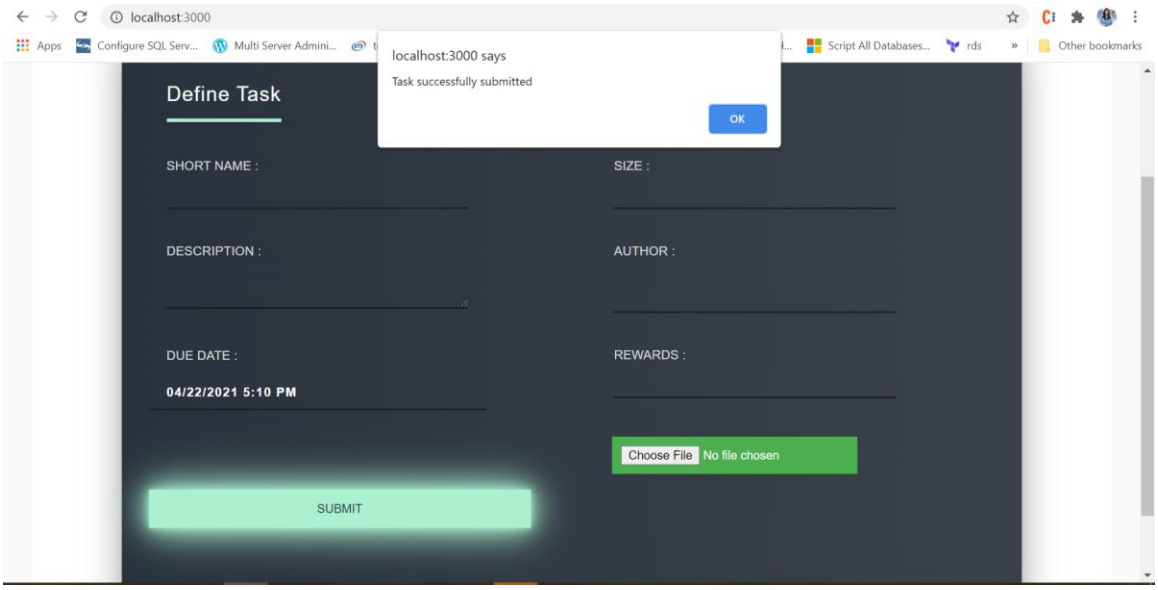


Figure 40 – Task Submission.

- The middleware broker MQ microservice received the task and forwarded it to DB and TE microservice to process. TE micro service assigns the task description to all subscribed workers. Then the MQ microservice sends the assigned task to the respective workers.

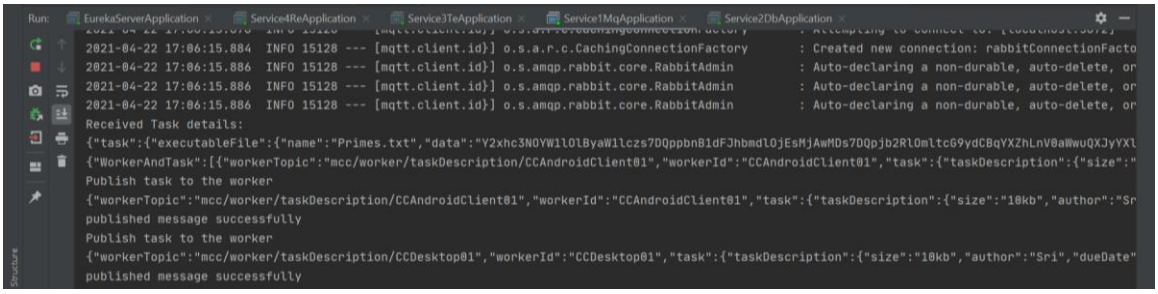


Figure 41 – The broker received the task and assigned it to the workers.

- The Android worker received the task description and accepted the task by clicking “ACCEPT” button.

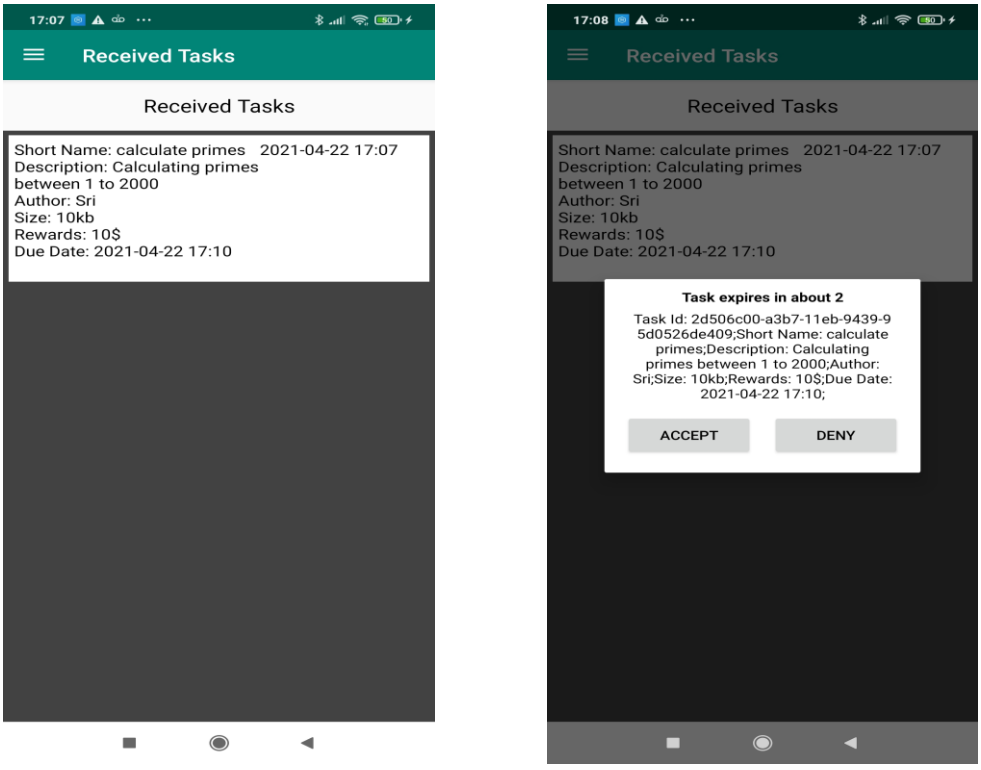


Figure 42 – The Android worker has received the task and accepted it.

- The Desktop worker received the task description and accepted the task by clicking “Accept” button.

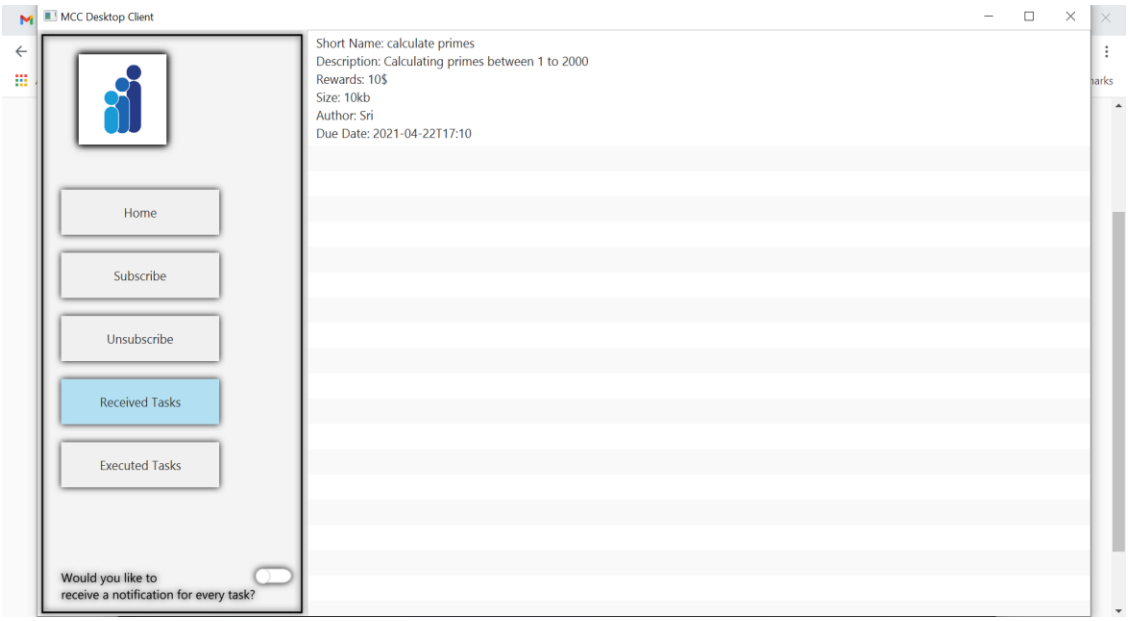


Figure 43 – The Desktop worker received the task.

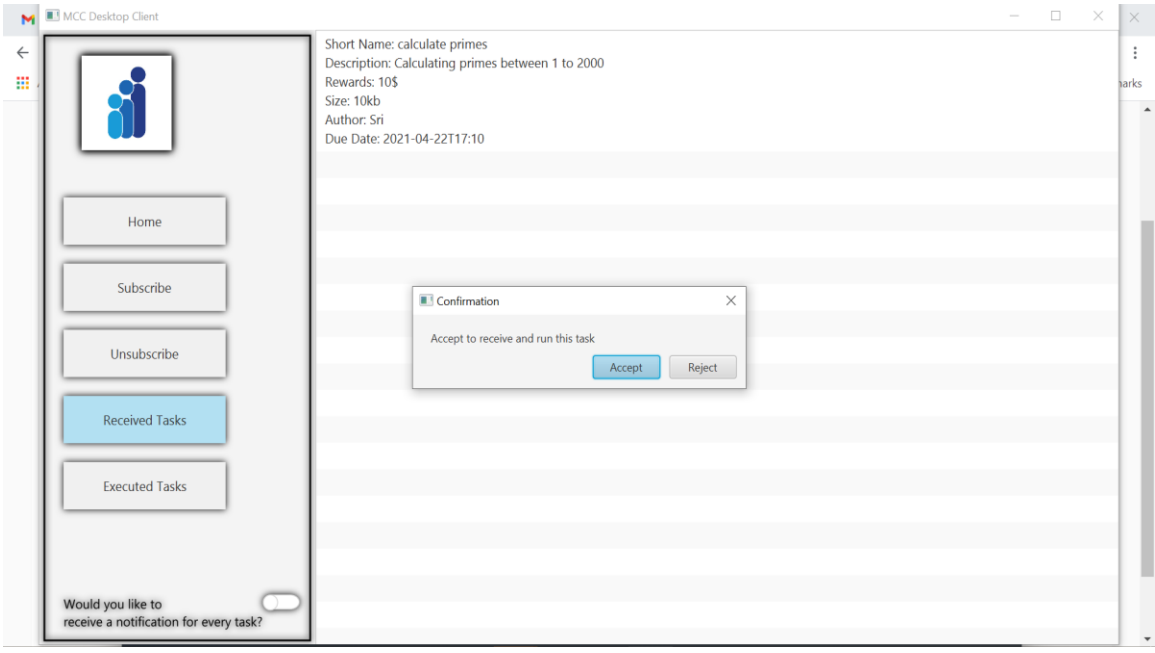


Figure 44 – The Desktop worker accepted the task.

- The middleware broker MQ microservice received the task accepted worker's details.

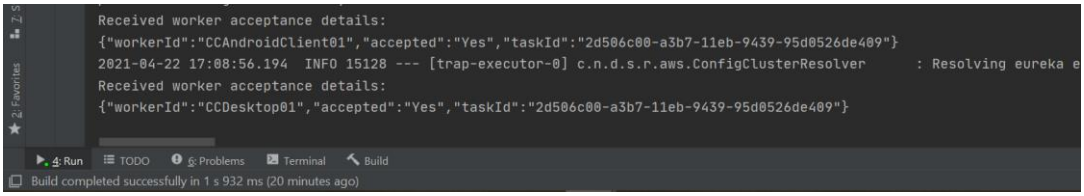


Figure 45 – Task accepted worker's details.

- DB microservice sent task accepted worker's details and .txt file to the Task Engine microservice.

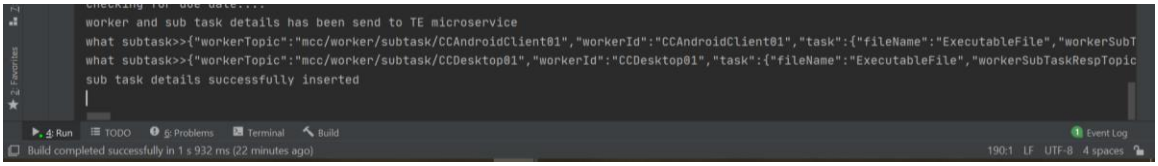


Figure 46 – Task accepted worker's details and task file.

- The Task Engine microservice divided the task (calculation of primes) into two sub-tasks with input range 1-1000 and 1001-2000. It generated a sub-task .apk file for Android workers with an input range 1-1000 and a sub-task .jar file for Desktop workers.

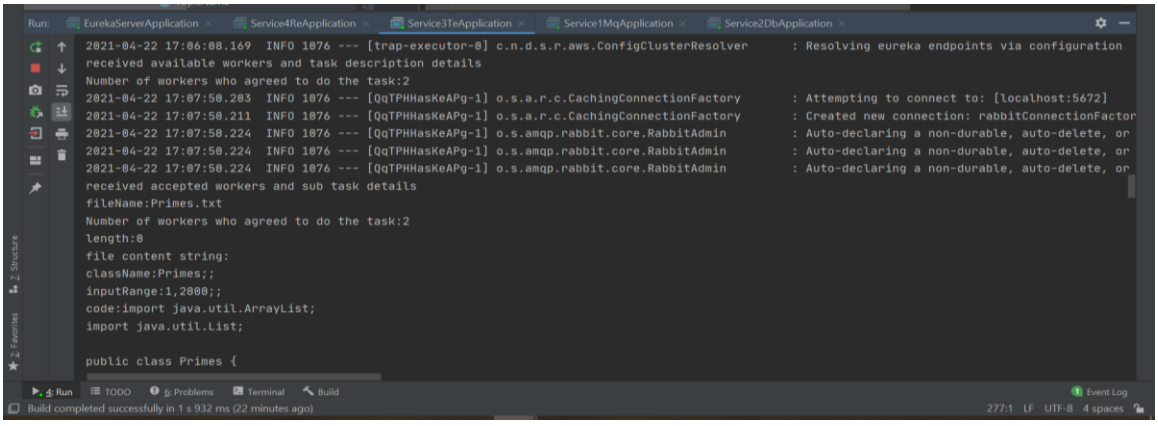


Figure 47 – The DB microservice received accepted workers and task details.

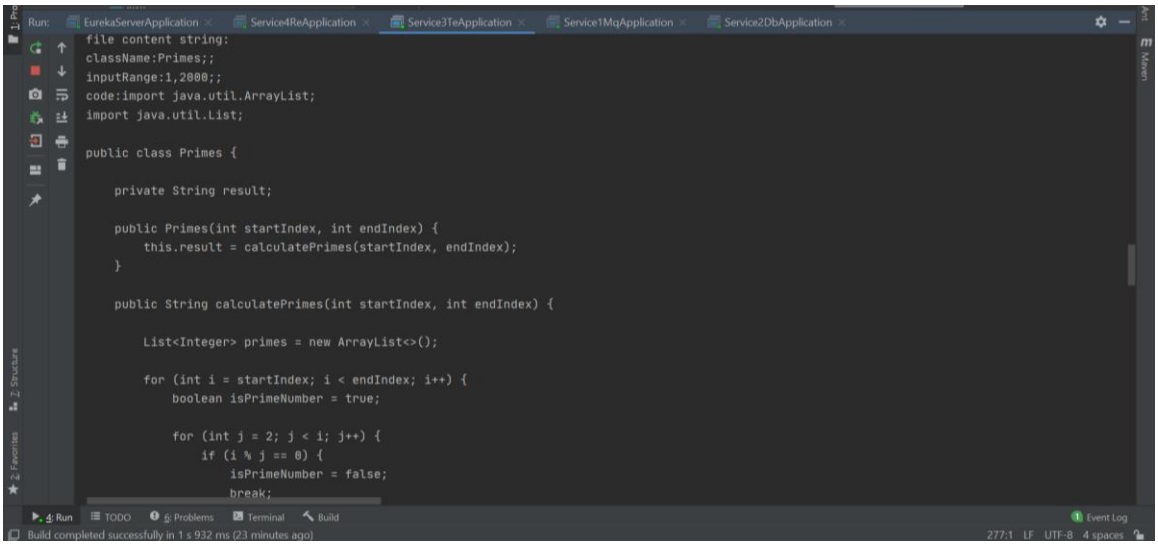


Figure 48 – Task content.

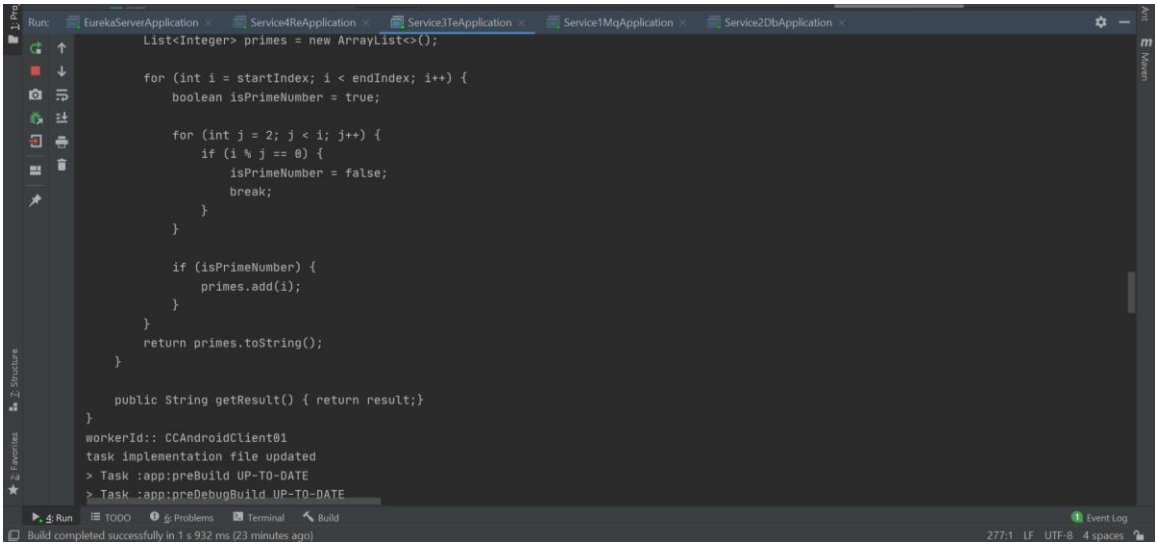


Figure 49 – The Gradle tasks have been initiated for .apk generation.

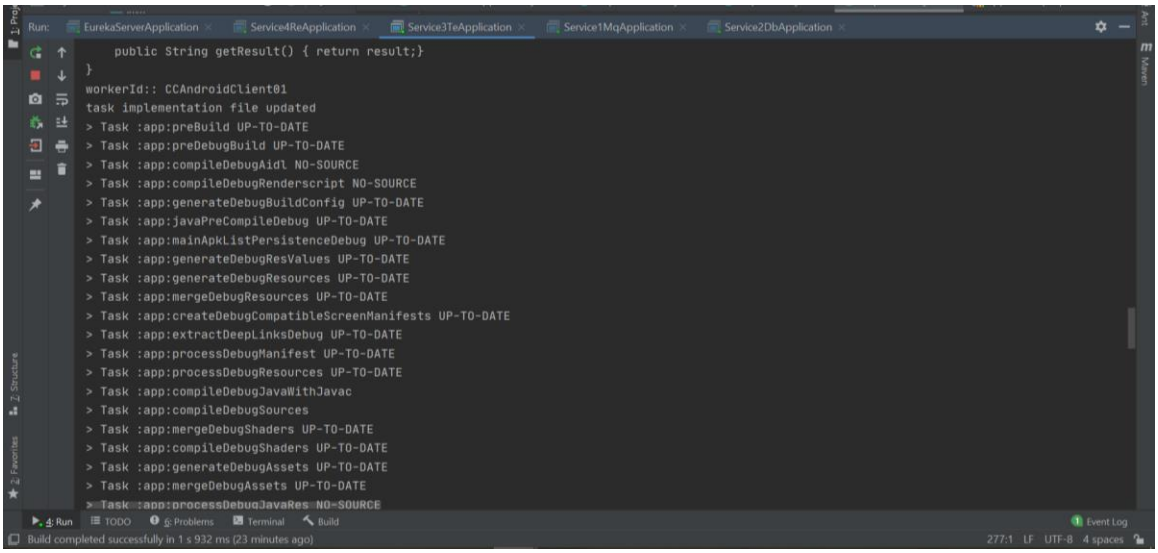


Figure 50 – Gradle tasks execution.

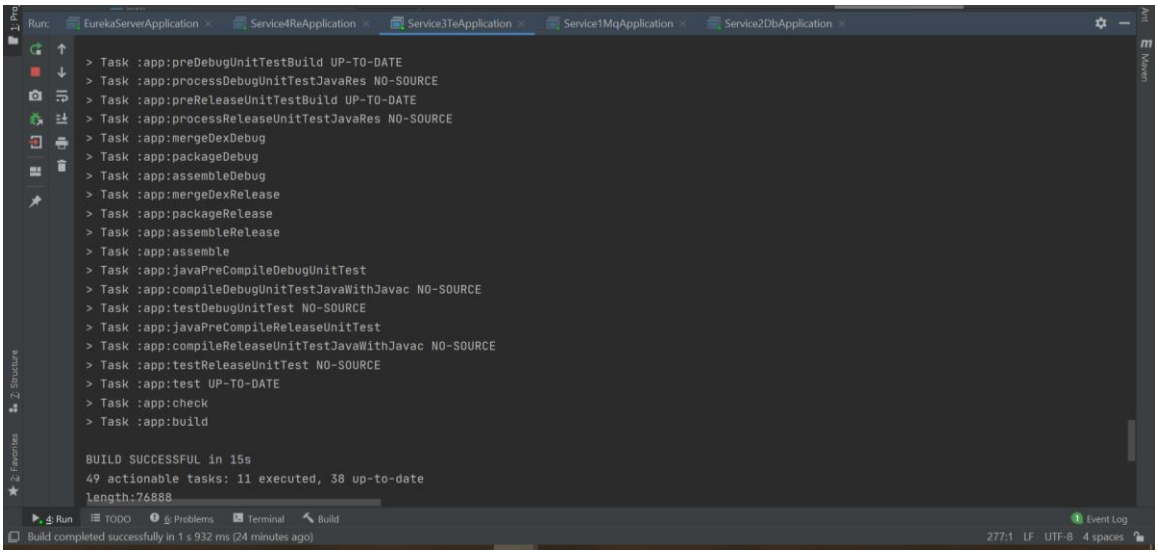


Figure 51 – The Gradle build was successful and generated .apk.

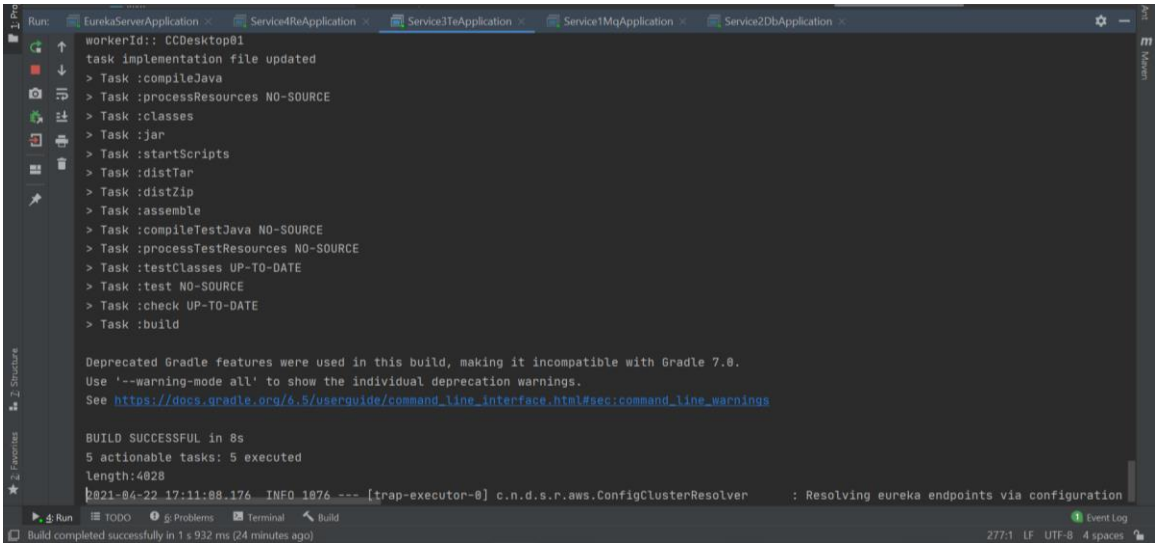


Figure 52 – The Gradle build was successful and generated .jar for desktop worker.

- The android worker received a sub-task (calculation of primes from 1 to 1000) in the form of .apk file.

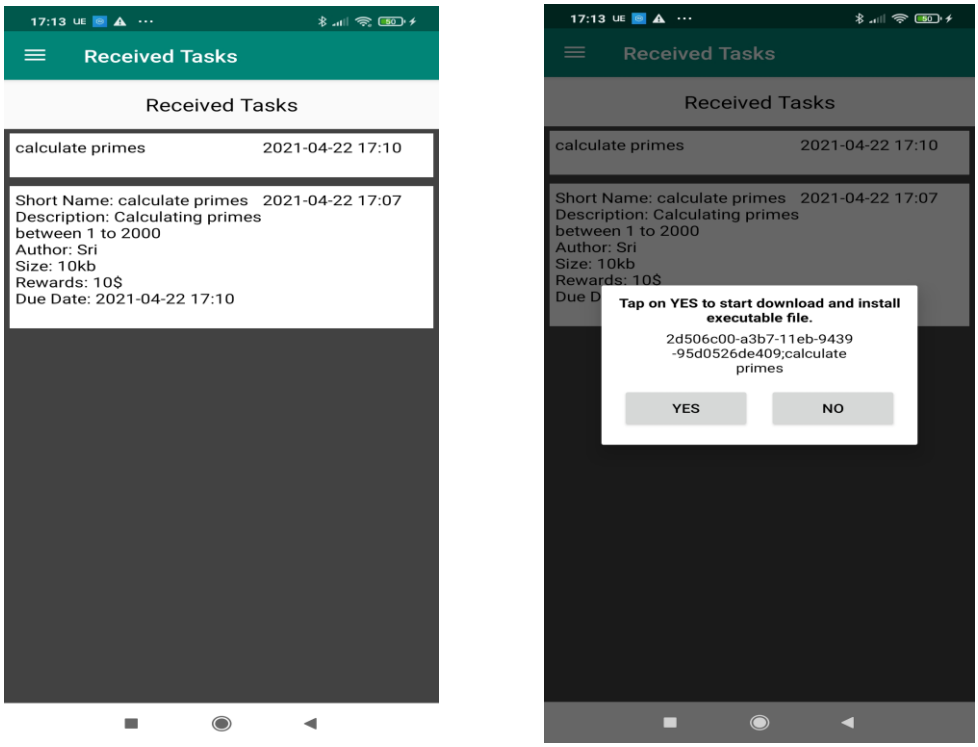


Figure 53 – The worker received .apk file and executed the task.

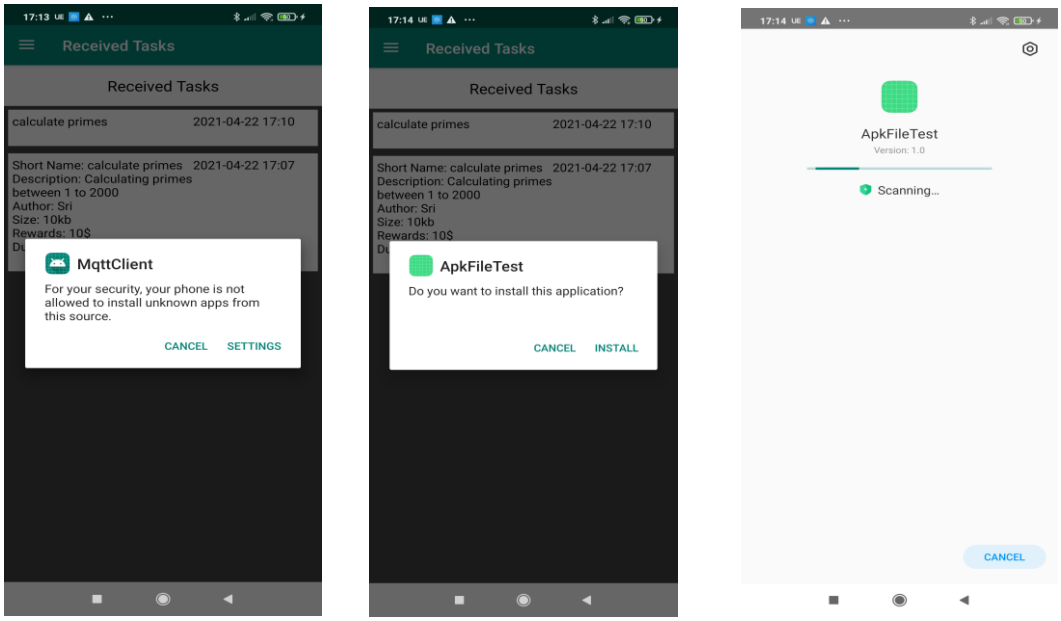


Figure 54 – The subtask file executed and broadcasted the result back to main app.

- The desktop worker received a sub-task (calculation of primes from 1001 to 2000) in the form of .jar file.

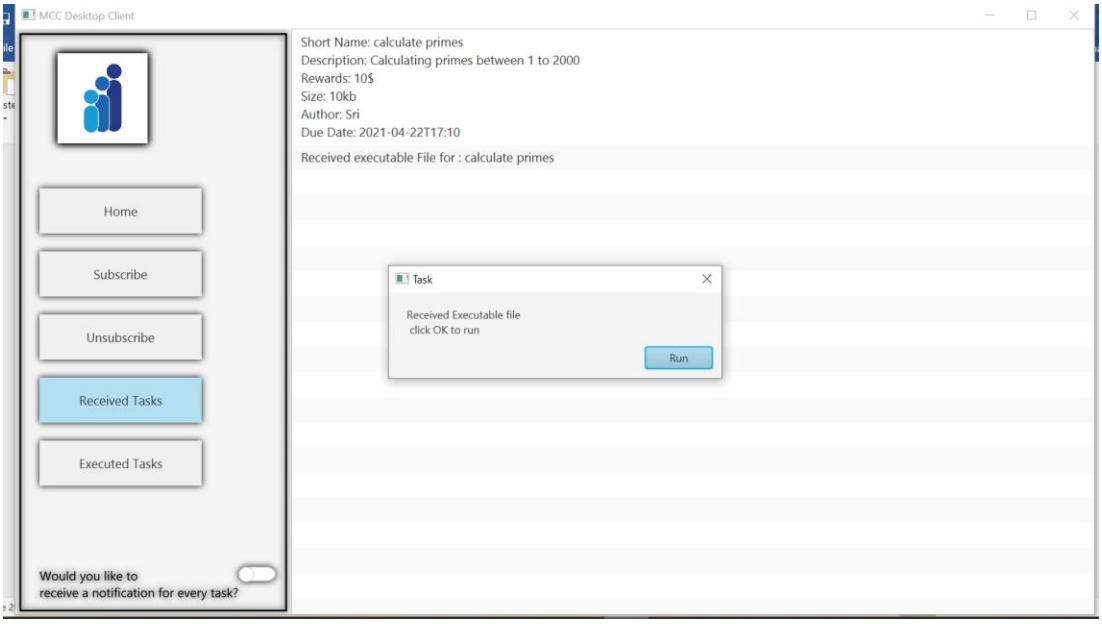


Figure 55 – The worker received the .jar file and executed the task.

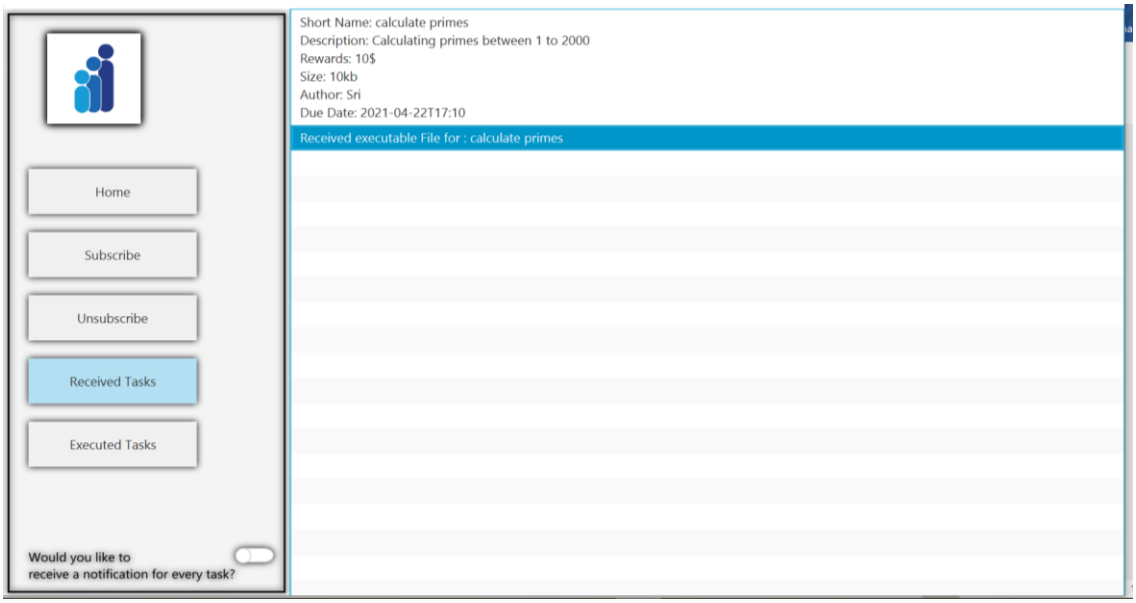


Figure 56 – The .jar file was executed and sent the result back to the main app.

- The middleware broker MQ microservice received the executed results from the android worker.

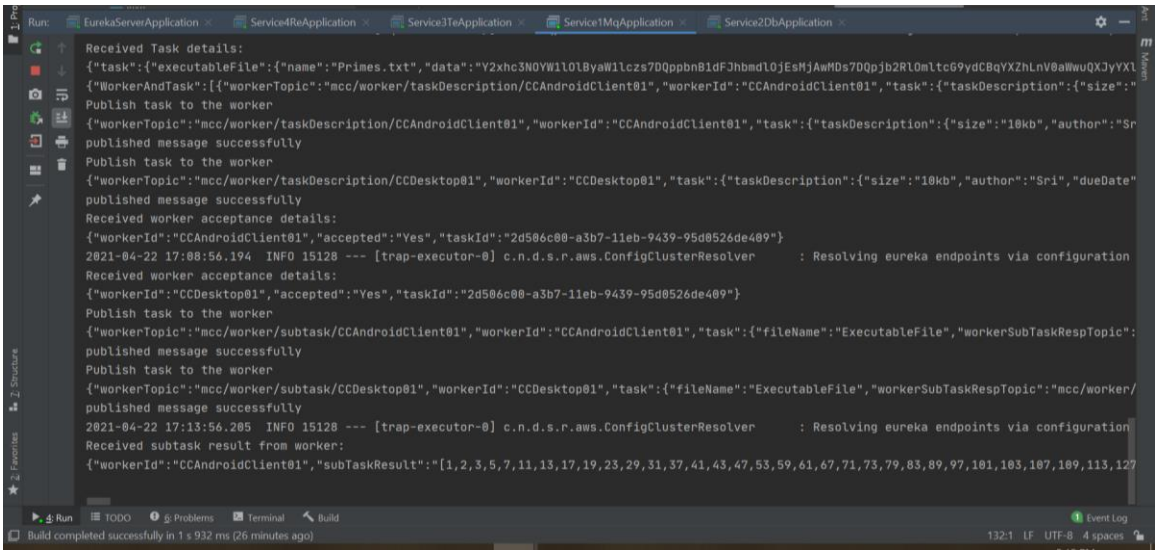


Figure 57 – Received computed subtask results from the android worker.

- The middleware broker MQ microservice received the executed results from the desktop worker.

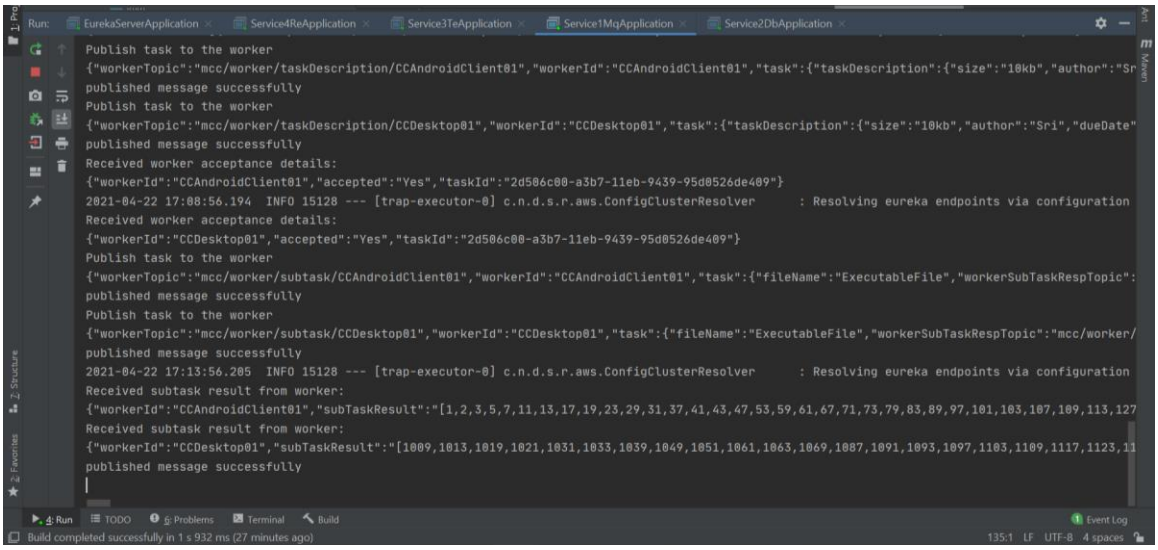


Figure 58 – Received computed subtask results from the desktop worker.

- The result engine accumulated the sub-task results and send to the MQ service. The MQ service has published to the MQTT.

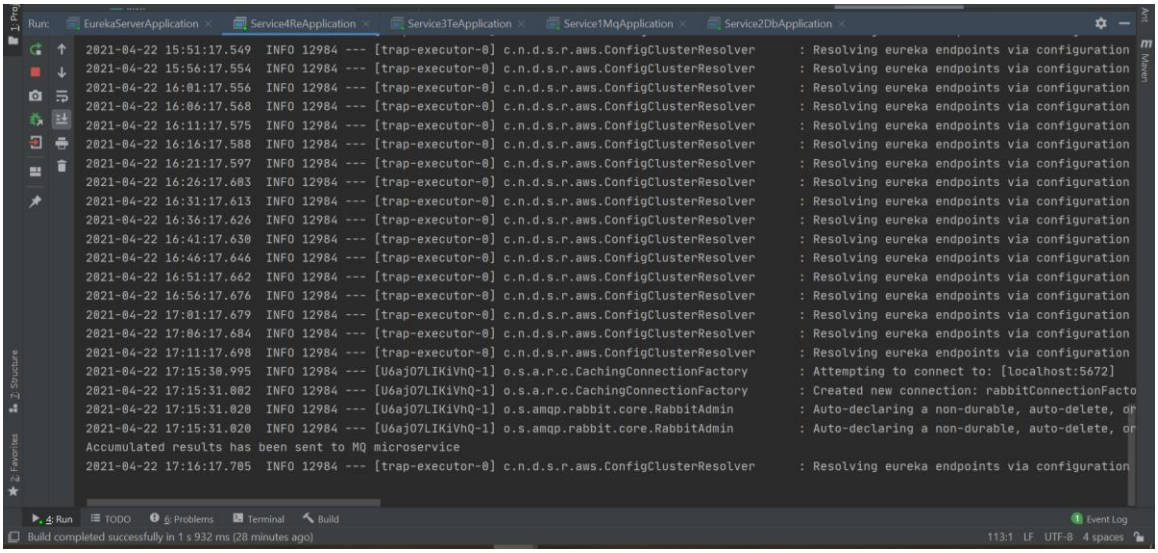


Figure 59 – Result engine accumulated the two sub-task results.

- The NodeJS application received the computed results i.e primes between 1 to 2000.

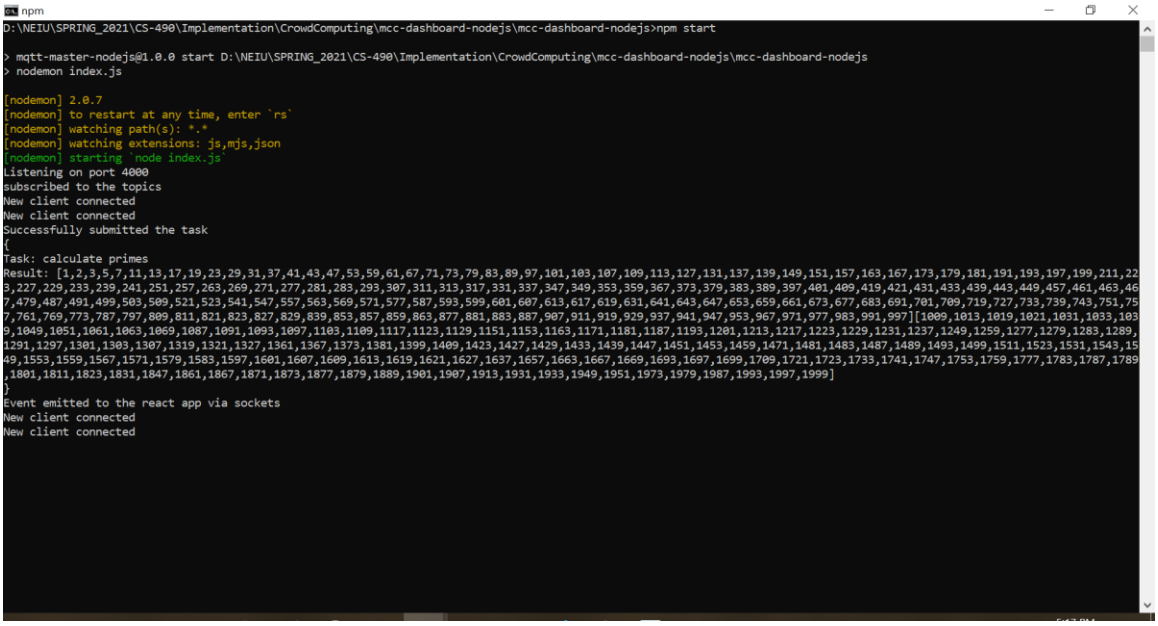


Figure 60 – Received accumulated results at the NodeJS level.

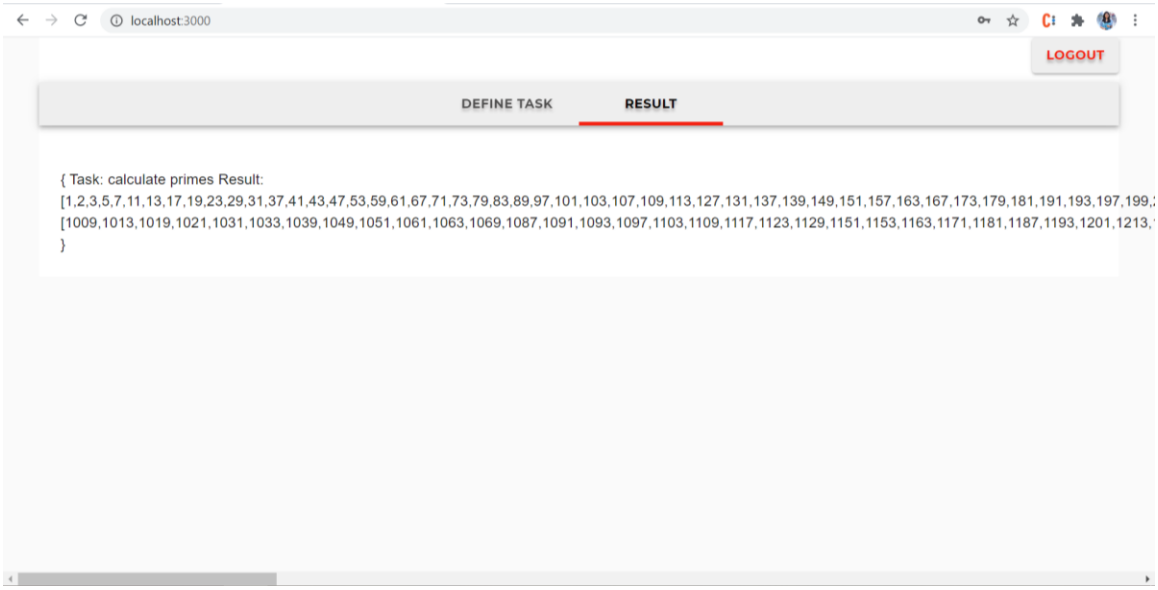


Figure 61 – The user dashboard with computed results.

Figure 62 – The user dashboard with computed results.

CHAPTER 3. CONCLUSTION AND FURTURE WORK

MCC can become a vital source for providing solutions to the people who need to solve problems that require huge computational resources and processing time. Unlike the existing crowd computing systems, the MCC provides crowd computing as a service that allows anyone to initiate a task at any time through the web interface. We have achieved this by giving an option to the user to define the task through a set of properties. This in turn allows the user to send the task in different formats such as precompiled (e.g., JSON, .txt) or compiled formats (e.g., .apk, .jar). It also provides a platform to the workers where the tasks can be run with little to zero human intervention in the background by not impacting any other workers and also obtaining the results.

The future work will be about providing a secure platform for task initiators and workers participation, the quality validation of subtask results sent by the workers. As the task initiators and workers are connected to the middleware broker through the internet, there is a high possibility of these machines being infected by viruses and botnets. We also cannot ignore the prospects of bringing in a participant who might have harmful intentions! Such scenarios need careful assessment and require strong encryption-decryption mechanisms to ensure a secure process.

REFERENCES

[1] Dutta Pramanik, P., Pal, S., Pareek, G., and Dutta, S., Choudhury, P., “Crowd Computing: The Computing Revolution,” 2018/07/24, 10.4018/978-1-5225-4200-1.ch009.