VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

SWE-4002 CLOUD COMPUTING

Slot- A2

J-COMPONENT

FINAL REVIEW

MOBILE OFFLOADING

Under the guidance of

Prof Vani M P

Submitted by

19MIS0137 Sabrina Manickam

19MIS0138 Laasya Yarlagadda

19MIS0266 Sirigiri Sri Sai Sharanya

Abstract

The method of offering computing resources and services is called cloud computing. It speaks of an infrastructure that is available when needed and enables users to use computer resources from anywhere at any time. In its most basic form, Mobile Cloud Computing refers to a system in which data processing and storage take place outside of mobile devices. Mobile cloud applications deliver applications and mobile computing to not just smartphone users but a much wider spectrum of mobile subscribers by moving the computational power and data storage away from mobile phones and into the cloud. The ability of an offloading system to recognize where the execution of code (locally or remotely) represents less computing work for the mobile is what determines how successful the system is. By properly choosing what, when, where, and how to offload, the device benefits. Code offloading is advantageous when it reduces the device's energy consumption without impairing the applications' typical reaction times. It is detrimental when the device expends more energy when a computational operation is carried out remotely as opposed to locally. With tools like code offloading, the mobile cloud is expanding the possibilities for creating and deploying applications. Offloading has been extensively advocated as a way to reduce energy consumption and improve the responsiveness of mobile devices, but its actual use still has several issues. In this position paper, we examine the current status of code offloading for mobile devices and emphasize the key procedures for developing a cloud-based offloading system that is more effective.

Keywords: Mobile Oflloading, Android Offloading, Performance, Resource Usage, Distributed Computing, Mobile Edge Computing, Computation Offloading, Offloading Strategy.

1. Problem Statement

By offloading the task to more powerful systems with superior performance and resources, computation offloading addresses the limits of Smart Mobile Devices (SMDs), such as limited battery life, restricted processing power, and limited storage capacity. Mobile cloud applications extend applications and mobile computing to a much wider range of mobile subscribers rather than just smartphone users by moving the computational power and data storage away from mobile phones and into the cloud. Utilizing resource-hungry mobile applications, such as media processing, online gaming, augmented reality (AR), and virtual reality (VR), play a crucial role in both enterprises and entertainment due to the rapid development of innovative mobile technology. Distributed Mobile Edge Computing (MEC) has been developed in order to lessen the burden of the complexity brought on by the rapid development of such serving technologies. MEC is intended to bring the computation environments close to the end-users, typically in a single hop, in order to meet predefined requirements.High computational applications which cannot be effectively and solemnly run on the mobile within reasonable time and power are offloaded to the cloud for performing the computation and the result is pushed back into the mobile device which is displayed on the mobile device.

1.1 Relevance to Mobile Cloud Computing

It is likely that a portion of a difficult computing activity could be run on a smartphone, given the computational capacity that the average smartphone offers. To build on this idea, we attempt to develop a distributed cloud system using just smartphones in order to solve the challenging mathematical issue of matrix multiplication. The distributed cloud infrastructure works in master-slave architecture. Slaves also known as workers share their battery stats, location information, and charging status. The quantity of work assigned to the worker node is determined by the master note based on these criteria. If a small number of worker nodes are unable to complete the computation due to network connectivity issues, the master node reassigns the failing compute workloads to available worker nodes, resulting in a system that is very fault resistant and resilient.

2. Literature Survey

[1] The universal user interface for cloud computing and online services will soon be mobile phones. However, there are currently only two ways to use them for this purpose: either the programs run directly on the phone, or they run on a server, and the phone connects to it remotely. The possibilities for performance optimization are also constrained by these two approaches because they do not permit personalized and flexible service engagement. They described a middleware framework in this research that can automatically distribute various application layers between a phone and a server while also optimizing a number of objective functions (latency, data transferred, cost, etc.). Their method does not call for new infrastructures and builds on distributed module management technology that is already in place. In the paper, they go over how to represent applications as a consumption graph and how to process it using a variety of cutting-edge algorithms to determine how the application modules should be distributed most effectively. After that, the application is effectively and transparently deployed dynamically on the phone. With numerous tests and two separate applications, they have examined and verified their strategy. The results showed that the methods they provide can greatly improve the functionality of cloud apps when accessed from mobile devices.

[2] This paper describes PowerBooter, an automated power model construction technique that monitors power consumption while explicitly controlling the power management and activity states of individual components. PowerBooter uses built-in battery voltage sensors and knowledge of battery discharge behavior. It doesn't need any additional measuring tools. We also go through PowerTutor, a tool that employs PowerBooter's concept for online power estimation and is based on activity state introspection and component power management. With each new smartphone variation having a different power consumption characteristic and necessitating a separate power model, PowerBooter aims to make it quick and simple for application developers and end users to produce power models. The goal of PowerTutor is to make it simpler to choose and build power-efficient software for embedded devices. By working together, PowerBooter and PowerTutor hope to make power modeling and analysis accessible to more smartphone models and their users.

[3] The paper aims to outline computation offloading, a future concern area, and its related frameworks and programming models that would be compatible with it. It also provides an overview of the recent efforts that are made toward solving the issues and identifying the efforts that are needed for a real implementation. The basic working of computation offloading and the limiting factors and their possible solutions and the benefits of computation offloading was discussed. Various programming models needed for mobile computation offloading were discussed and the characteristics of the proposed computation offloading frameworks with respect to the mobile device offloaders were analyzed in terms of implementation effort and device-to-device compatibility. The major models discussed were offloading APIs, middleware-assisted offloading, and their architectures and frameworks. And the next section was about the recent developments in offloading to mobile devices which aim to provide solutions for various concerns such as scheduling, incentive, privacy etc. The discussion section states the merits and demerits of mobile offloading in detail and the future work and conclusion are stated as one.

3. Introduction

Utilizing a server system to offload computation can increase the battery life and processing capability of mobile devices like smartphones. A mobile application is split into two parts during compute offloading, one for server-side execution and the other for mobile device execution. Such offloading framework prototypes have demonstrated longer battery life and quicker smartphone application completion times [1, 2]. Choosing which machines to utilize as servers in computation offloading is an intriguing decision. The majority of offloading framework prototypes created thus far make use of internal desktop or server equipment.

Right now, there are two different machines that can be used as servers by offloading frameworks. The first option is known as mobile cloud computing and involves offloading to commercially available cloud servers.

Mobile cloud computing is considered an appropriate solution for addressing the problem of constrained resources in mobile devices such as limitation of battery capacity and high-demand computation for some mobile applications. Mobile cloud computing is the process of offloading intensive-computation tasks and processing them in a conventional centralized cloud. Distributed computing is a mobile cloud computing model in which components of a particular software system are shared among multiple computers or nodes, even though the components are spread out on various systems, they run as one. Distributed computing helps us to improve efficiency and performance.

Briefly put, the project is the implementation of the distributed system architecture on a set of mobile devices using a mobile application that runs in master or slave mode and allows us to distribute a computing power demanding task to various Wi-Fi-enabled phones and aggregates the results back at the master at the end of the computation.

The application mainly contains two modes, the master and the slave mode. The devices in the master mode will try to find the devices which are working in the worker mode. Each of the workers will be discovered by all the masters present in the vicinity. The master can then send

connection requests to one or more workers, and it is up to the worker to accept or reject the connection.
As soon as the worker accepts the connection, the battery and the resource details are sent to the master and the master can start assigning tasks to the workers based on the details sent by the workers. The distribution of work then happens to each and every worker based on the computation power and the device battery level which are used to compute the priority cube. The master then stops with a stop.
If in case one or the other worker fails, the work done before disconnecting will be considered and the work that has to be done by the devices which got disconnected will be distributed again to the mobile devices which are connected.

4. System Requirements

The technical requirements for the software products are specified in the requirements specification. The list includes functional, performance, and security requirements for a certain software system. It is the first phase in the requirements analysis process. Additionally, usage scenarios from a user, operational, and administrative standpoint are provided in the requirements. A complete description of the software project, its parameters, and objectives is what the software requirements specification is there for. This outlines the project's user interface, hardware, and software requirements, as well as the intended audience. It describes the project's functionality and the client's, team's, and audience's perspectives on it.

4.1 Hardware Specifications

Hard Disk: 40GB and above
RAM: 512MB and above
Processor: Pentium III and above
Mobile: Dual-core preferred

4.2 Software Specifications

Android Studio with SDK version 4.3 and above
Android OS for mobile
Java with JDK 11 and above version

5. Modules

5.1 System Architecture

We created a master-slave distributed architecture where communication takes place back and forth between the master node and the slaves. There won't be any communication between

worker nodes. There is one master node and one slave node. There may be several worker nodes.

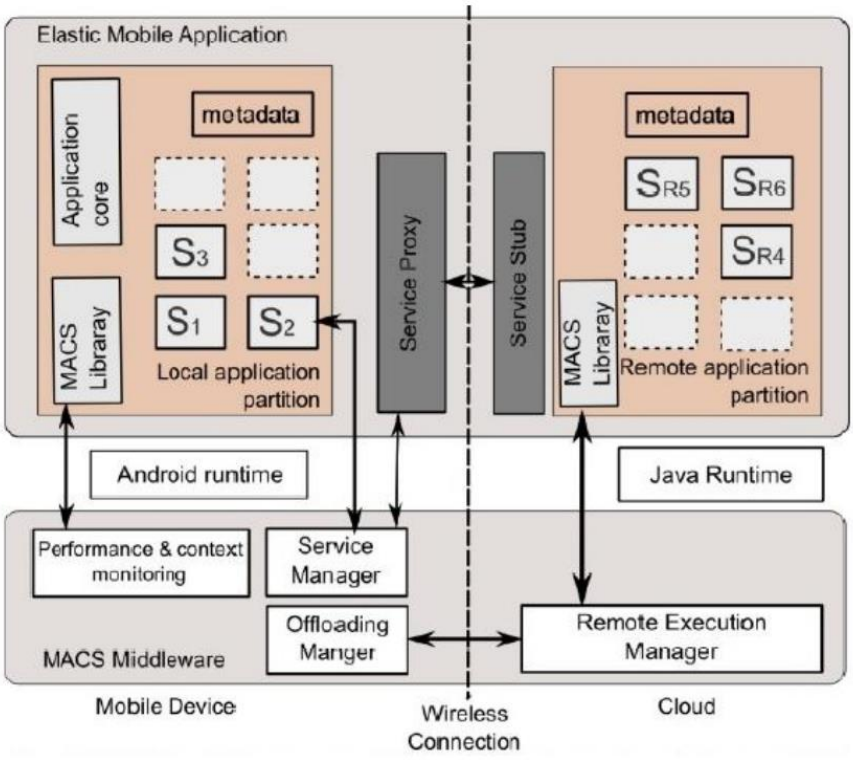
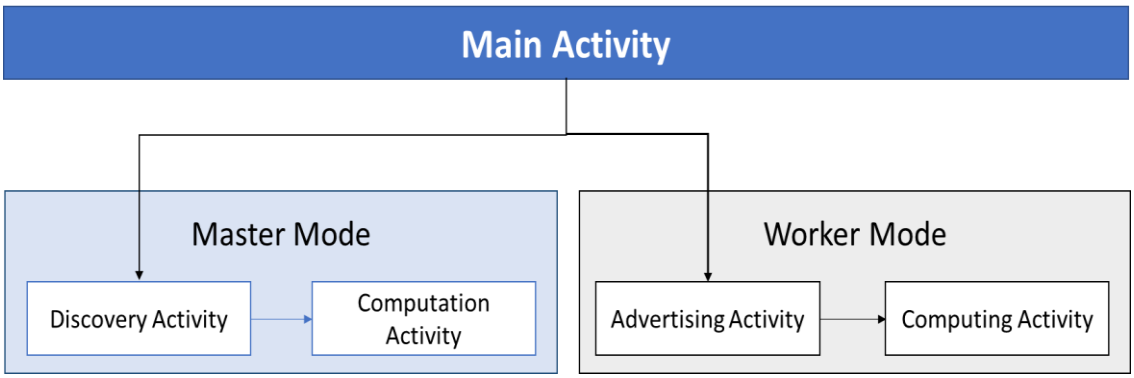
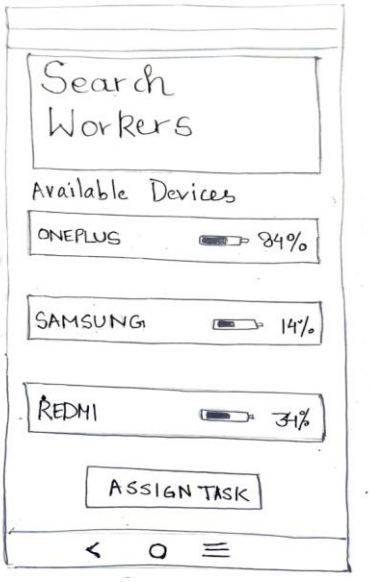


Fig 1. System Architecture Diagram

1. Setup

We have created an Android mobile application that gives the user the choice of starting the program in worker mode or master mode; choosing both options is not permitted by design. The program tries to find available workers if the user decides to set the device up as a master node. On the other hand, the application enters discovery mode and makes itself accessible to be found by the master computer running the very same application, if the user chooses to make the device operate as a worker station. To establish peer-to-peer connections between cellphones that are close to one another, we will be using the Nearby Connection API. If Wi-Fi is accessible, we use it; otherwise, Bluetooth is used. By design, the master node ignores devices that are not adjacent to one another when computing.





2. The smartphone acts as a Worker

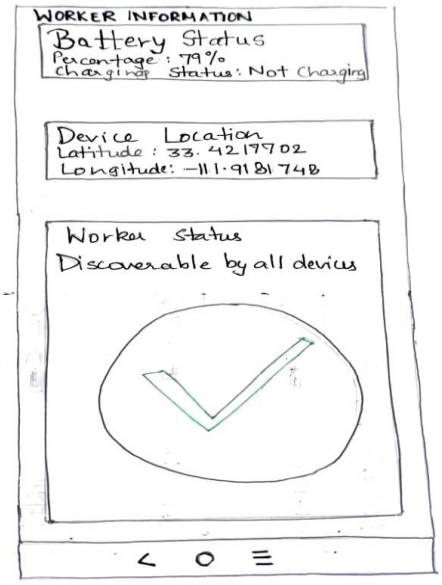
The user is initially prompted upon launching the application to select one of the two modes in the main activity. Once the user selects to employ worker mode to run the program, the program moves on to the subsequent activity, when the smartphone program announces its existence to other nearby devices using the same activity context. The "peer-to-peer CLUSTER" advertising approach has been chosen. With this approach, the numerous devices operating in worker mode may connect to the master within a 100-meter radius. Both of the gadgets have the ability to establish a connection and begin speaking.

When the device successfully broadcasts, we display a message in the user interface indicating that it is accessible to all masters in the area. We also display information on the battery life of the device, such as the battery % and whether it is plugged in or not. The user interface also displays the device's location information (latitude and longitude). Every few seconds, the statistics are updated.

Once the master has located the device, the following step will begin. An initial connection will be made between the master device and the worker. We can choose to "Accept" or "Reject" the connection in this Android notice. If the connection is refused, the worker has no effects, and the master may still see the device. Once the connection is established and the user accepts the connection, an acceptance is issued and a communication channel is created. The application moves on to the following action, "Worker Computation," after receiving acceptance. Here, the worker transmits and receives payloads using the already established communication channel.

In the worker computation phase, the worker waits for the master to send the payload data. Until then there is a busy indication shown on the screen with the message ”Waiting for work”.

Once the worker receives the work, it computes it and then sends back the result via the communication channel. Worker advertising screen this communication continues until the master sends a work complete message.



3. The smartphone acts as a Master

The application operates in a discovery state if the user launches it as a Master. The Master application can identify nearby employees who are broadcasting their presence by message payload over Wi-Fi or Bluetooth. A connection request is sent to each of these workers from the master application, which keeps a list of them. The entry of the worker device is deleted from the list kept on the Master device if the worker refuses the calculation task. On the other hand, the worker is added to the list of connected devices if they accept the computing task by clicking accept in the notice. Every time the gadget is used, the worker notifies the master of the current battery level, whether it is linked to a charging station, and its position. The master conducts a local matrix multiplication operation in this activity, records the amount of time it takes, and displays the information on the screen in the master computation activity. The identical matrix multiplication task is then disseminated by the master across the available worker devices once the task has been completed. Additionally, workers migrate to a computation activity and display a screen indicating that a distributed calculation is now running. These jobs are maintained and managed by the master, who also displays a progress indicator indicating how far along the matrix multiplication work is.

4. Matrix Multiplication Partition Process

For multiplying the two available matrices and obtaining results, logically we need to find the values of each cell of the matrix which is defined by matrix[row][column]. We call these cells partitions. Since we are using two square matrices for multiplication, we define a Hash Table

with key as partition index and value as the value of the cell in the resultant matrix. For example**,**

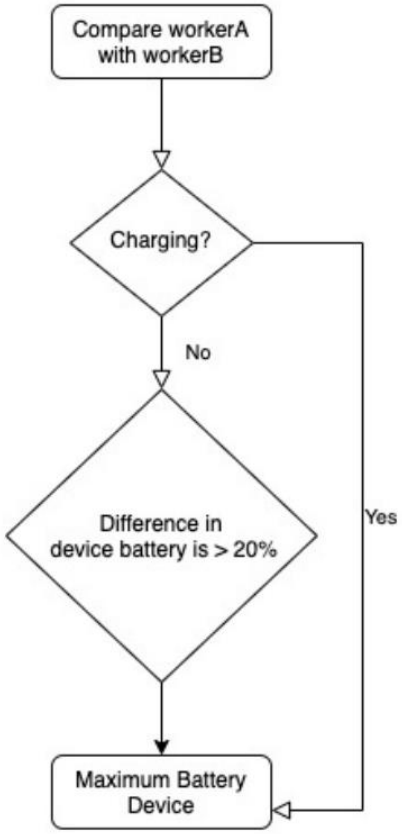
$$\begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} \times \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix} = \begin{bmatrix} z_1 & z_1 \\ z_3 & z_4 \end{bmatrix}$$

$$\begin{aligned} z_1 &= x_1 \times y_1 + x_2 \times y_3 \\ z_2 &= x_1 \times y_2 + x_2 \times y_4 \\ z_3 &= x_3 \times y_1 + x_4 \times y_3 \\ z_4 &= x_3 \times y_2 + x_4 \times y_4 \end{aligned}$$

After we determine the partitions, we split these partitions and send the data i.e. rows and columns required for calculating each individual resultant cell, across devices. We can achieve this distribution because of the mathematical property of matrix multiplication that, Dot product of the individual input columns and row results in the cell of the resultant matrix. Thus, we send the following partitions across the devices. The result of z1 is available in partition 1 and the result of z2 is available in partition 2 and so on.

$$\begin{aligned} partition_1 &= [[x_1, x_2], [y_1, y_3]] \\ partition_2 &= [[x_1, x_2], [y_2, y_4]] \\ partition_3 &= [[x_3, x_4], [y_1, y_3]] \\ partition_4 &= [[x_3, x_4], [y_2, y_4]] \end{aligned}$$

For achieving fault tolerance in our distributed system, we decided to go ahead with the system design of granular data sharing across workers using the above mentioned partitions. With this implementation, we avoid the master from sharing huge chunks of data with workers. Considering any worker can fail while computation or the worker can disconnect at any time, having a granular data sharing makes it easy to redistribute the failed tasks to other workers. This partition methodology, also avoids sending redundant data across partitions keeping the shared data across workers to minimum. We initially populate all workers in an Arraylist. From these workers Arraylist, we generate a Priority-Queue of workers with comparison condition as follows:



Using the above implementation, we achieve selection of workers with more battery and workers which are currently docked to charging stations over devices with lesser battery. Here, we also check if the difference in battery levels is greater than 20. If that is the case, we add device with more battery first in the priority queue. Also, we ignore worker devices with battery level less than 15.Once the priority queue is computed, based on the priority we send workers computation tasks. For achieving this functionality, we use the Hashmap of partition indices. One computation consists of required row and column values required for calculating the value at partition index in the resultant column. Once the data sent to these devices our implementation removes the worker from the priority queue. Once the computation result is obtained for the partition index from the worker node, we add the worker again in the priority queue. While adding worker in the priority queue, again the comparator function kicks in and adds the worker in correct desired position in the priority. Using this mechanism, we iterate over all the elements in the priority indices Hashmap and obtain values for each partition index. If the value for the partition index key is present conveys that the computation is completed for that partition index and we do not need to compute it again. If the value absent for the partition index in Hashmap, means the computation task is not complemented. So, if a worker disconnects and looses connection with the master, the computation task is lost. Since there is no entry for that partition index in the Hashmap, while iterating we send the computation task to other available workers. The advantages of this implementation are multi-fold. It allows any worker to go down in the distributed environment and still achieves task completion making the infrastructure highly fault tolerant.

a) ***Continuous Device Monitoring Strategy:*** We monitor the worker device statistics like battery level, is the device charging and longitude, latitude co-ordinates by sending a data payload from worker to master every 5 seconds. If the statistics payload or data message with computation task is not received on the master side for 10 seconds, we consider that the worker has encountered a fault and remove the worker from the list of available workers for the computation. Thus, effectively using these payload data messages we achieve heart-beating messages between all available workers and master node.

b) ***Failure Recovery:*** Once we send computation data payload to worker, it gets removed from the priority queue. Afterwards, when worker sends computation task result back to master, we add the worker again in the priority queue of available workers. In this step, if a worker gets disconnected from the network, we do not receive any message from the worker which includes device stats or data payload. If that occurs, we remove the worker from the list of available workers and that worker never appears in the priority queue which is used to send tasks to available workers. This computation task, since it does not have any value computed in the partition index Hash Map, gets calculated again using other available workers.

6. Performance Analysis

a) *Power consumption:* This metric takes into account how much battery is consumed by master and worker nodes for the actual computation and sharing data messages across the between the master node and workers. Our measurement also includes the power consumed by Wi-Fi, Bluetooth and location tracking.

b) *Time taken for computation:* This metric measures the time taken for the computation. This metric includes initialization time, time required for sending data over network and actual computation task.

We measured the time taken for computation for two cases.

1) Time taken for only computation on the master side.

2) Time taken for computation on worker side added with aggregation of results on the master side.

This metric also includes communication overhead incurred between the master and worker devices.

7. Implementation

7.1 Main Activity

```
package com.amsy.mobileoffloading;
import androidx.annotation.NonNull;
```

```
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import android.content.Intent;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;
import java.util.ArrayList;
public class MainActivity extends AppCompatActivity {
   private final int PERMISSIONS_REQUEST_CODE = 111;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
   }
   @Override
   protected void onResume() {
      super.onResume();
      String[] requiredPermissions =  checkPermissions();
      if (requiredPermissions.length > 0) {
         askPermissions(requiredPermissions);
      }
   }
   @Override
   public void onRequestPermissionsResult(int requestCode, @NonNull String[]
permissions, @NonNull int[] grantResults) {
      super.onRequestPermissionsResult(requestCode, permissions, grantResults);
      if (requestCode == PERMISSIONS_REQUEST_CODE) {
         for (int grantResult : grantResults) {
            if (grantResult == PackageManager.PERMISSION_DENIED) {
               Toast.makeText(getApplicationContext(), "Please provide all necessary
permissions", Toast.LENGTH_LONG).show();
               onBackPressed();
               finish();
            }
         }
      }
   }
   private String[] checkPermissions() {
      ArrayList<String> requiredPermissions = new ArrayList<>();
      try {
         String packageName = getPackageName();
```

```
            PackageInfo packageInfo = getPackageManager().getPackageInfo(packageName,
 PackageManager.GET_PERMISSIONS);
            String[] permissions = packageInfo.requestedPermissions;
            for (String permission : permissions) {
               if (ActivityCompat.checkSelfPermission(this, permission) !=
 PackageManager.PERMISSION_GRANTED) {
                  requiredPermissions.add(permission);
               }
            }
         } catch (PackageManager.NameNotFoundException e) {
            e.printStackTrace();
         }
         String[] _requiredPermissions = new String[requiredPermissions.size()];
         _requiredPermissions = requiredPermissions.toArray(_requiredPermissions);
         return _requiredPermissions;
      }
      private void askPermissions(String[] requiredPermissions) {
         ActivityCompat.requestPermissions(this, requiredPermissions,
 PERMISSIONS_REQUEST_CODE);
      }
      public void onClickMaster(View view) {
         Intent intent = new Intent(getApplicationContext(), MasterDiscovery.class);
         startActivity(intent);
      }
      public void onClickSlave(View view) {
         Intent intent = new Intent(getApplicationContext(), WorkerAdvertisement.class);
         startActivity(intent);
      }
 }
```

7.2 Discovery

```
package com.amsy.mobileoffloading;

import android.content.Context;

import android.util.Log;

import com.google.android.gms.nearby.Nearby;

import com.google.android.gms.nearby.connection.DiscoveryOptions;

import com.google.android.gms.nearby.connection.EndpointDiscoveryCallback;

import com.google.android.gms.nearby.connection.Strategy;
```

```
public class Discovery {
    private Context context;
    private DiscoveryOptions discoveryOptions;
    public Discovery(Context context) {
        this.context = context;
        this.discoveryOptions=new
DiscoveryOptions.Builder().setStrategy(Strategy.P2P_CLUSTER).build();
    }
    public void start(EndpointDiscoveryCallback endpointDiscoveryCallback) {
        Nearby.getConnectionsClient(context)
                .startDiscovery(context.getPackageName(),          endpointDiscoveryCallback,
discoveryOptions)
                .addOnSuccessListener((unused) -> {
                    Log.d("MASTER", "Finding Devices");
                    Log.d("MASTER", unused + "");
                })
                .addOnFailureListener((Exception e) -> {
                    Log.d("MASTER", "Unable to find devices");
                    e.printStackTrace();
                });
    }
    public void stop() {
        Nearby.getConnectionsClient(context).stopDiscovery();
    }
}
```

7.3 MasterActivity.java

```
import androidx.appcompat.app.AppCompatActivity;
import androidx.recyclerview.widget.LinearLayoutManager;
```

```
import androidx.recyclerview.widget.RecyclerView;
import androidx.recyclerview.widget.SimpleItemAnimator;
import android.content.Context;
import android.location.Location;
import android.os.BatteryManager;
import android.os.Bundle;
import android.os.Handler;
import android.util.Log;
import android.widget.TextView;
public class MasterActivity extends AppCompatActivity {
private RecyclerView rvWorkers;
   private HashMap<String, WorkerStatusSubscriber> workerStatusSubscriberMap = new HashMap<>();
   private ArrayList<Worker> workers = new ArrayList<>();
   private WorkersAdapter workersAdapter;
   /* [row1 x cols1] * [row2 * cols2] */
   private int rows1 = Constants.matrix_rows;
   private int cols1 = Constants.matrix_columns;
   private int rows2 = Constants.matrix_columns;
   private int cols2 = Constants.matrix_rows;
   private int[][] matrix1;
   private int[][] matrix2;
   private WorkAllocator workAllocator;

   private int workAmount;
   private int totalPartitions;
   private Handler handler;
```

```
    private Runnable runnable;
    private DeviceStatisticsPublisher deviceStatsPublisher;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_master);
        Log.d("MasterDiscovery", "Starting computing matrix multiplication on only master");
        TextView masterPower = findViewById(R.id.masterPower);
        masterPower.setText("Stats not available");
        BatteryManager mBatteryManager =
                (BatteryManager)getSystemService(Context.BATTERY_SERVICE);
        Long initialEnergyMaster =
mBatteryManager.getLongProperty(BatteryManager.BATTERY_PROPERTY_ENERGY_C
OUNTER);
        computeMatrixMultiplicationOnMaster();
        Long finalEnergyMaster =
mBatteryManager.getLongProperty(BatteryManager.BATTERY_PROPERTY_ENERGY_C
OUNTER);
    Long energyConsumedMaster = Math.abs(initialEnergyMaster-finalEnergyMaster);
        masterPower.setText("Power Consumption (Master): "
+Long.toString(energyConsumedMaster)+ " nWh");
        Log.d("MasterDiscovery", "Completed computing matrix multiplication on only
master");
        unpackBundle();
        bindViews();
        setAdapters();
        init();
        setupDeviceBatteryStatsCollector();
    }
```

```
    @Override
    protected void onPause() {
        super.onPause();
        stopWorkerStatusSubscribers();
        deviceStatsPublisher.stop();
        handler.removeCallbacks(runnable);
    }
    @Override
    protected void onResume() {
        super.onResume();
        startWorkerStatusSubscribers();
        deviceStatsPublisher.start();
        handler.postDelayed(runnable, Constants.UPDATE_INTERVAL_UI);
    }
    @Override
    public void onBackPressed() {
        for (Worker w : workers) {
            updateWorkerConnectionStatus(w.getEndpointId(),
Constants.WorkStatus.DISCONNECTED);
            workAllocator.removeWorker(w.getEndpointId());
NearbyConnectionsManager.getInstance(getApplicationContext()).disconnectFromEndpoint(
w.getEndpointId());
        }
        super.onBackPressed();
        finish();
    }
    private void init() {
        totalPartitions = rows1 * cols2;
```

```
        updateProgress(0);
        matrix1 = MatrixDS.createMatrix(rows1, cols1);
        matrix2 = MatrixDS.createMatrix(rows2, cols2);
        workAllocator = new WorkAllocator(getApplicationContext(), workers, matrix1, matrix2, slaveTime -> {
            TextView slave = findViewById(R.id.slaveTime);
            slave.setText("Execution time (Slave): " + slaveTime + "ms");
        });
        workAllocator.beginDistributedComputation();
    }
    private void updateProgress(int done) {
        ProgressWheel wheel = findViewById(R.id.wheelprogress);
        int per = 360 * done / totalPartitions;
        wheel.setPercentage(per);
        wheel.setStepCountText(done + "");
        TextView totalPart = findViewById(R.id.totalPartitions);
        totalPart.setText("Total Partitions: " + totalPartitions);
        if(per == 360) {
            deviceStatsPublisher.stop();
        }
    }
    private void bindViews() {
        rvWorkers = findViewById(R.id.rv_workers);
        SimpleItemAnimator itemAnimator = (SimpleItemAnimator) rvWorkers.getItemAnimator();
        itemAnimator.setSupportsChangeAnimations(false);
    }
    private void setAdapters() {
```

```
        workersAdapter = new WorkersAdapter(this, workers);
        LinearLayoutManager linearLayoutManager = new LinearLayoutManager(getApplicationContext());
        rvWorkers.setLayoutManager(linearLayoutManager);
        rvWorkers.setAdapter(workersAdapter);
        workersAdapter.notifyDataSetChanged();
    }
    private void unpackBundle() {
        try {
            Bundle bundle = getIntent().getExtras();
            ArrayList<ConnectedDevice> connectedDevices = (ArrayList<ConnectedDevice>) bundle.getSerializable(Constants.CONNECTED_DEVICES);
            addToWorkers(connectedDevices);
            Log.d("CHECK", "Added a connected Device as worker");
        } catch (NullPointerException e) {
            e.printStackTrace();
        }
    }
    private void addToWorkers(ArrayList<ConnectedDevice> connectedDevices) {
        for (ConnectedDevice connectedDevice : connectedDevices) {
            Worker worker = new Worker();
            worker.setEndpointId(connectedDevice.getEndpointId());
            worker.setEndpointName(connectedDevice.getEndpointName());
            WorkInfo workStatus = new WorkInfo();
            workStatus.setStatusInfo(Constants.WorkStatus.WORKING);
            worker.setWorkStatus(workStatus);
            worker.setDeviceStats(new DeviceStatistics());
            workers.add(worker);
```

```
        }
    }
    private void computeMatrixMultiplicationOnMaster() {
        matrix1 = MatrixDS.createMatrix(rows1, cols1);
        matrix2 = MatrixDS.createMatrix(rows2, cols2);
        Needle.onBackgroundThread().execute(() -> {
            long startTime = System.currentTimeMillis();
            int[][] mul = new int[rows1][cols2];
            for (int i = 0; i < rows1; i++) {
                for (int j = 0; j < cols2; j++) {
                    mul[i][j] = 0;
                    for (int k = 0; k < cols1; k++) {
                        mul[i][j] += matrix1[i][k] * matrix2[k][j];
                    }
                }
            }
            long endTime = System.currentTimeMillis();
            long totalTime = endTime - startTime;
            FlushToFile.writeTextToFile(getApplicationContext(), "exec_time_master_alone.txt",
false, totalTime + "ms");
            TextView master = findViewById(R.id.masterTime);
            master.setText("Execution time (Master): " + totalTime + "ms");
        });
    }
    private void setupDeviceBatteryStatsCollector() {
        deviceStatsPublisher = new DeviceStatisticsPublisher(getApplicationContext(), null,
Constants.UPDATE_INTERVAL_UI);
        handler = new Handler();
```

```
        runnable = () -> {
            String deviceStatsStr = DeviceStatisticsPublisher.getBatteryLevel(this) + "%"
                    + "\t" + (DeviceStatisticsPublisher.isPluggedIn(this) ? "CHARGING" : "NOT CHARGING");
            FlushToFile.writeTextToFile(getApplicationContext(), "master_battery.txt", true, deviceStatsStr);
            handler.postDelayed(runnable, Constants.UPDATE_INTERVAL_UI);
        };
    }
private void updateWorkerConnectionStatus(String endpointId, String status) {
        Log.d("DISCONNECTED----", endpointId);
        for (int i = 0; i < workers.size(); i++) {
            Log.d("DISCONNECTED--", workers.get(i).getEndpointId());
            if (workers.get(i).getEndpointId().equals(endpointId)) {
                workers.get(i).getWorkStatus().setStatusInfo(status);
                workersAdapter.notifyDataSetChanged();
                break;
            }
        }
    }
private void startWorkerStatusSubscribers() {
        for (Worker worker : workers) {
            if (workerStatusSubscriberMap.containsKey(worker.getEndpointId())) {
                continue;
            }
            WorkerStatusSubscriber workerStatusSubscriber = new WorkerStatusSubscriber(getApplicationContext(), worker.getEndpointId(), new WorkerStatusListener() {
```

```
        @Override
        public void onWorkStatusReceived(String endpointId, WorkInfo workStatus) {
            if (workStatus.getStatusInfo().equals(Constants.WorkStatus.DISCONNECTED))
{
                updateWorkerConnectionStatus(endpointId,
Constants.WorkStatus.DISCONNECTED);
                workAllocator.removeWorker(endpointId);
NearbyConnectionsManager.getInstance(getApplicationContext()).rejectConnection(endpoin
tId);
            } else {
                updateWorkerStatus(endpointId, workStatus);
            }
            workAllocator.checkWorkCompletion(getWorkAmount());
        }
        @Override
        public void onDeviceStatsReceived(String endpointId, DeviceStatistics deviceStats)
{
            updateWorkerStatus(endpointId, deviceStats);
            String deviceStatsStr = deviceStats.getBatteryLevel() + "%"
                    + "\t" + (deviceStats.isCharging() ? "CHARGING" : "NOT CHARGING")
                    + "\t\t" + deviceStats.getLatitude()
                    + "\t" + deviceStats.getLongitude();
            FlushToFile.writeTextToFile(getApplicationContext(), endpointId + ".txt", true,
deviceStatsStr);
            Log.d("MASTER_ACTIVITY", "WORK AMOUNT: " + getWorkAmount());
            workAllocator.checkWorkCompletion(getWorkAmount());
        }
    });
    workerStatusSubscriber.start();
```

```
            workerStatusSubscriberMap.put(worker.getEndpointId(), workerStatusSubscriber);
        }
    }   private int getWorkAmount() {
        int sum = 0;
        for (Worker worker : workers) {
            sum += worker.getWorkAmount();
        }
        return sum;
    }
    private void updateWorkerStatus(String endpointId, WorkInfo workStatus) {
        for (int i = 0; i < workers.size(); i++) {
            Worker worker = workers.get(i);
            if (worker.getEndpointId().equals(endpointId)) {
                worker.setWorkStatus(workStatus);
                if     (workStatus.getStatusInfo().equals(Constants.WorkStatus.WORKING)     &&
workAllocator.isItNewWork(workStatus.getPartitionIndexInfo())) {
                    workers.get(i).setWorkAmount(workers.get(i).getWorkAmount() + 1);
                    workAmount += 1;
                }
workAllocator.updateWorkStatus(worker, workStatus);
             workersAdapter.notifyItemChanged(i);
                break;
            }
        }
        updateProgress(workAmount);
    }
    private void updateWorkerStatus(String endpointId, DeviceStatistics deviceStats) {
```

```
        for (int i = 0; i < workers.size(); i++) {
            Worker worker = workers.get(i);
            if (worker.getEndpointId().equals(endpointId)) {
                worker.setDeviceStats(deviceStats);
                Location masterLocation = DeviceStatisticsPublisher.getLocation(this);
                if (deviceStats.isLocationValid() && masterLocation != null) {
                    float[] results = new float[1];
                    Location.distanceBetween(masterLocation.getLatitude(),
masterLocation.getLongitude(),
                            deviceStats.getLatitude(), deviceStats.getLongitude(), results);
                    Log.d("MASTER_ACTIVITY", "Master Location: " +
masterLocation.getLatitude() + ", " + masterLocation.getLongitude());
                    Log.d("MASTER_ACTIVITY", "Master Distance: " + results[0]);
                    worker.setDistanceFromMaster(results[0]);
                }
                workersAdapter.notifyItemChanged(i);
            }
        }
    }
    private void stopWorkerStatusSubscribers() {
        for (Worker worker : workers) {
            WorkerStatusSubscriber workerStatusSubscriber =
workerStatusSubscriberMap.get(worker.getEndpointId());
            if (workerStatusSubscriber != null) {
                workerStatusSubscriber.stop();
                workerStatusSubscriberMap.remove(worker.getEndpointId());
            }
        }
```

```
    }
}
```

7.4 WorkingComputation.java

```
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import android.content.Context;
import android.content.Intent;
import android.os.BatteryManager;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.TextView;
import java.io.IOException;
import java.util.HashSet;
import pl.droidsonroids.gif.GifImageView;
public class WorkerComputation extends AppCompatActivity {
    private String masterId;
    private DeviceStatisticsPublisher deviceStatsPublisher;
    private ClientConnectionListener connectionListener;
    private PayloadListener payloadCallback;
    private int currentPartitionIndex;
    private HashSet<Integer> finishedWork = new HashSet<>();
    BatteryManager mBatteryManager = null;
    Long initialEnergyWorker,finalEnergyWorker,energyConsumedWorker;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_worker_computation);
        extractBundle();
        startDeviceStatsPublisher();
        setConnectionCallback();
        connectToMaster();
        //start measuring the pwer consumption at Worker
        mBatteryManager = (BatteryManager)getSystemService(Context.BATTERY_SERVICE);
        initialEnergyWorker =

mBatteryManager.getLongProperty(BatteryManager.BATTERY_PROPERTY_ENERGY_COUNTER);
        Log.d("WORKER_COMPUTATION", "Capturing power consumption");
    }
    public void setStatusText(String text, boolean isWorking) {
        //UI Textview
        TextView statusText = findViewById(R.id.statusText);
        statusText.setText(text);
        GifImageView waiting = findViewById(R.id.waiting);
        waiting.setVisibility(isWorking ? View.INVISIBLE : View.VISIBLE);
        GifImageView working = findViewById(R.id.working);
        working.setVisibility(isWorking ? View.VISIBLE : View.INVISIBLE);
    }

    public void onWorkFinished(String text) {
        //UI Textview
        TextView statusText = findViewById(R.id.statusText);
```

```
        statusText.setText(text);
        GifImageView waiting = findViewById(R.id.waiting);
        waiting.setVisibility(View.INVISIBLE);
        GifImageView working = findViewById(R.id.working);
        working.setVisibility(View.INVISIBLE);
        GifImageView done = findViewById(R.id.done);
        done.setVisibility(View.VISIBLE);
        TextView powerConsumed = findViewById(R.id.powerValue);
        powerConsumed.setText("Power Consumption (Slave) : " +
Long.toString(energyConsumedWorker)+ " nWh");
    }
    public void setPartitionText(int count) {
    private void extractBundle() {
        Bundle bundle = getIntent().getExtras();
        this.masterId = bundle.getString(Constants.MASTER_ENDPOINT_ID);
    }

    private void startDeviceStatsPublisher() {
        deviceStatsPublisher = new DeviceStatisticsPublisher(getApplicationContext(),
masterId, Constants.UPDATE_INTERVAL_UI);
    }
    private void connectToMaster() {
        payloadCallback = new PayloadListener() {
            @Override
            public void onPayloadReceived(@NonNull String endpointId, @NonNull Payload
payload) {
                startWorking(payload);
            }
```

```
        @Override
        public void onPayloadTransferUpdate(@NonNull String endpointId, @NonNull PayloadTransferUpdate payloadTransferUpdate) {
        }
    };

NearbyConnectionsManager.getInstance(getApplicationContext()).acceptConnection(masterId);
  }
  private void setConnectionCallback() {
    connectionListener = new ClientConnectionListener() {
        @Override
        public void onConnectionInitiated(String id, ConnectionInfo connectionInfo) {
        }
        @Override
        public void onConnectionResult(String id, ConnectionResolution connectionResolution) {
        }
        @Override
        public void onDisconnected(String id) {
          navBack();
        }
    };
  }

  private void navBack() {
    finishedWork = new HashSet<>();
```

```
        finish();
    }
    @Override
    protected void onResume() {
        super.onResume();
NearbyConnectionsManager.getInstance(getApplicationContext()).registerPayloadListener(payloadCallback);
NearbyConnectionsManager.getInstance(getApplicationContext()).registerClientConnectionListener(connectionListener);
        deviceStatsPublisher.start();
    }
    @Override
    protected void onPause() {
        super.onPause();
NearbyConnectionsManager.getInstance(getApplicationContext()).unregisterPayloadListener(payloadCallback);
NearbyConnectionsManager.getInstance(getApplicationContext()).unregisterClientConnectionListener(connectionListener);
        deviceStatsPublisher.stop();
    }
    @Override
    public void finish() {
        super.finish();
NearbyConnectionsManager.getInstance(getApplicationContext()).disconnectFromEndpoint(masterId);
        currentPartitionIndex = 0;
    }
    public void onDisconnect(View view) {
        WorkInfo workStatus = new WorkInfo();
        workStatus.setPartitionIndexInfo(currentPartitionIndex);
```

```
        workStatus.setStatusInfo(Constants.WorkStatus.DISCONNECTED);
        ClientPayLoad tPayload1 = new ClientPayLoad();
        tPayload1.setTag(Constants.PayloadTags.WORK_STATUS);
        tPayload1.setData(workStatus);
        DataTransfer.sendPayload(getApplicationContext(), masterId, tPayload1);
        navBack();
    }
    public void startWorking(Payload payload) {
        WorkInfo workStatus = new WorkInfo();
        ClientPayLoad sendPayload = new ClientPayLoad();
        sendPayload.setTag(Constants.PayloadTags.WORK_STATUS);
        try {
            ClientPayLoad receivedPayload = PayloadConverter.fromPayload(payload);
            if (receivedPayload.getTag().equals(Constants.PayloadTags.WORK_DATA)) {
                setStatusText("Working now", true);
                WorkData workData = (WorkData) receivedPayload.getData();
                int dotProduct = MatrixDS.getDotProduct(workData.getRows(),
workData.getCols());
                Log.d("WORKER_COMPUTATION", "Partition Index: " +
workData.getPartitionIndex());
                if (!finishedWork.contains(workData.getPartitionIndex())) {
                    finishedWork.add(workData.getPartitionIndex());
                }
                currentPartitionIndex = workData.getPartitionIndex();
                setPartitionText(finishedWork.size());
                workStatus.setPartitionIndexInfo(workData.getPartitionIndex());
                workStatus.setResultInfo(dotProduct);
                workStatus.setStatusInfo(Constants.WorkStatus.WORKING);
```

```
            sendPayload.setData(workStatus);
            DataTransfer.sendPayload(getApplicationContext(), masterId, sendPayload);
        } else if (receivedPayload.getTag().equals(Constants.PayloadTags.FAREWELL));
            finalEnergyWorker
=BatteryManager.getLongProperty(BatteryManager.BATTERY_PROPERTY_ENERGY_C
OUNTER); energyConsumedWorker = Math.abs(initialEnergyWorker-
finalEnergyWorker);
            onWorkFinished("Work Done !!");
            Log.d("WORKER_COMPUTATION", "Work Done");
            workStatus.setStatusInfo(Constants.WorkStatus.FINISHED);
            sendPayload.setData(workStatus);
            DataTransfer.sendPayload(getApplicationContext(), masterId, sendPayload);
            deviceStatsPublisher.stop();
        } else if
(receivedPayload.getTag().equals(Constants.PayloadTags.DISCONNECTED)) {
            navBack();
        }
      } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
      }
   }
}
```

8. Results

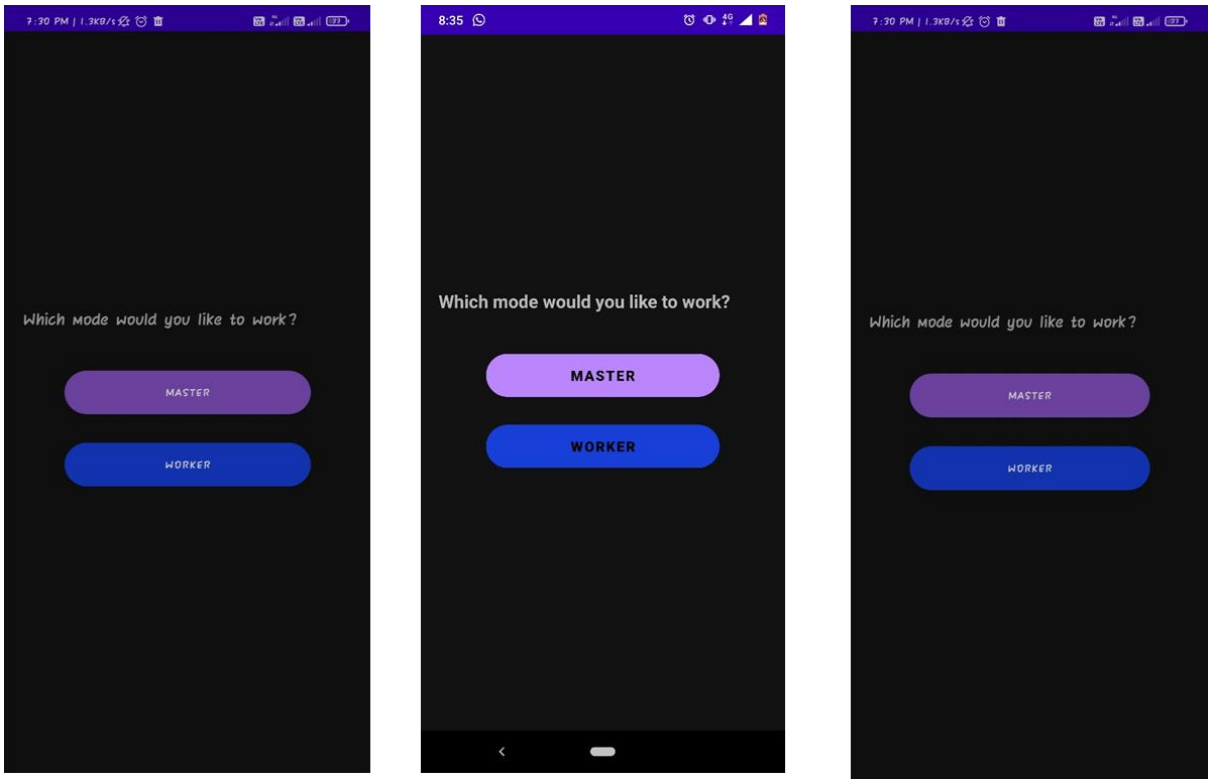


Fig 1. Screenshots of devices choosing a mode

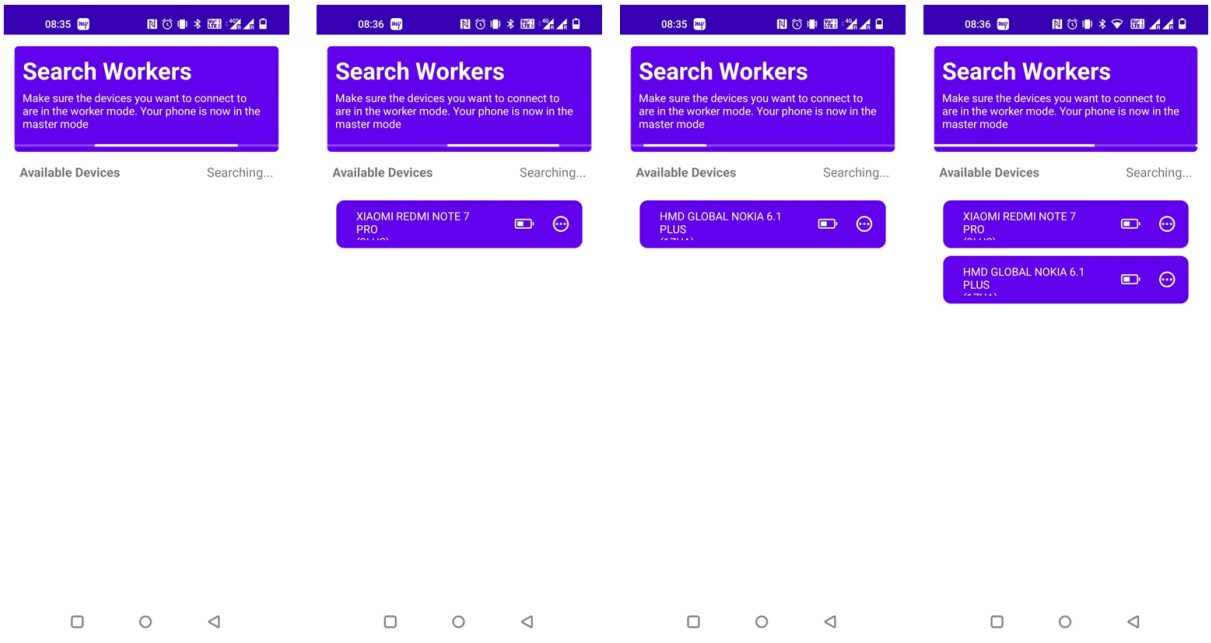


Fig 2. Screenshots of master device searching for the workers

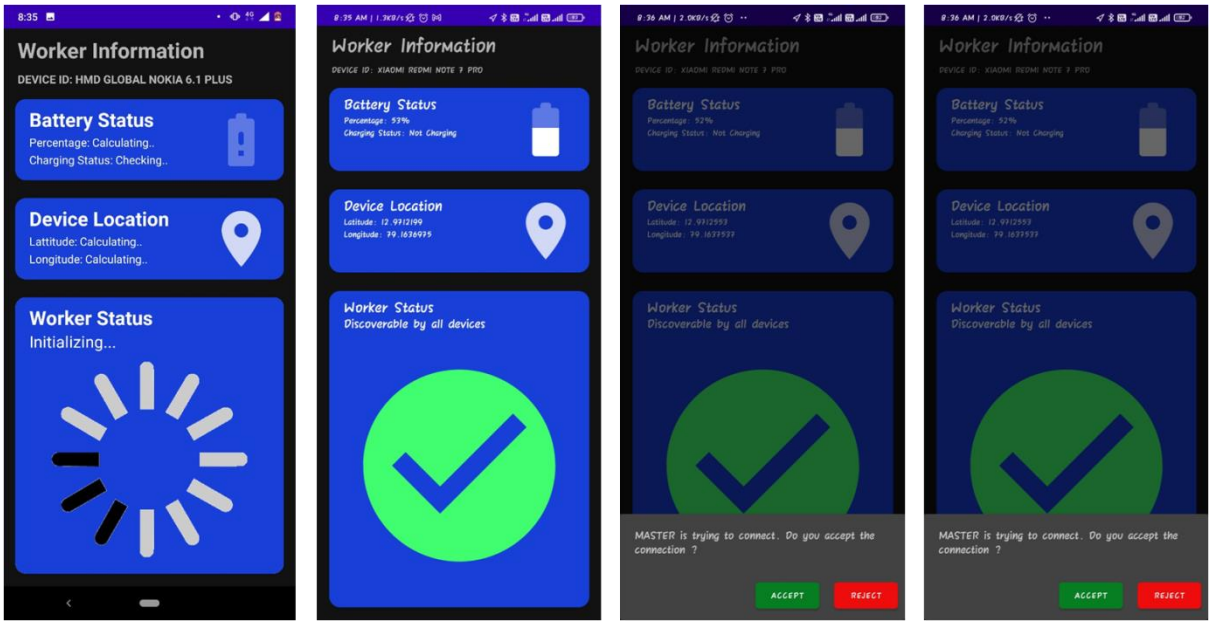


Fig 3. Screenshots of workers accepting the master request

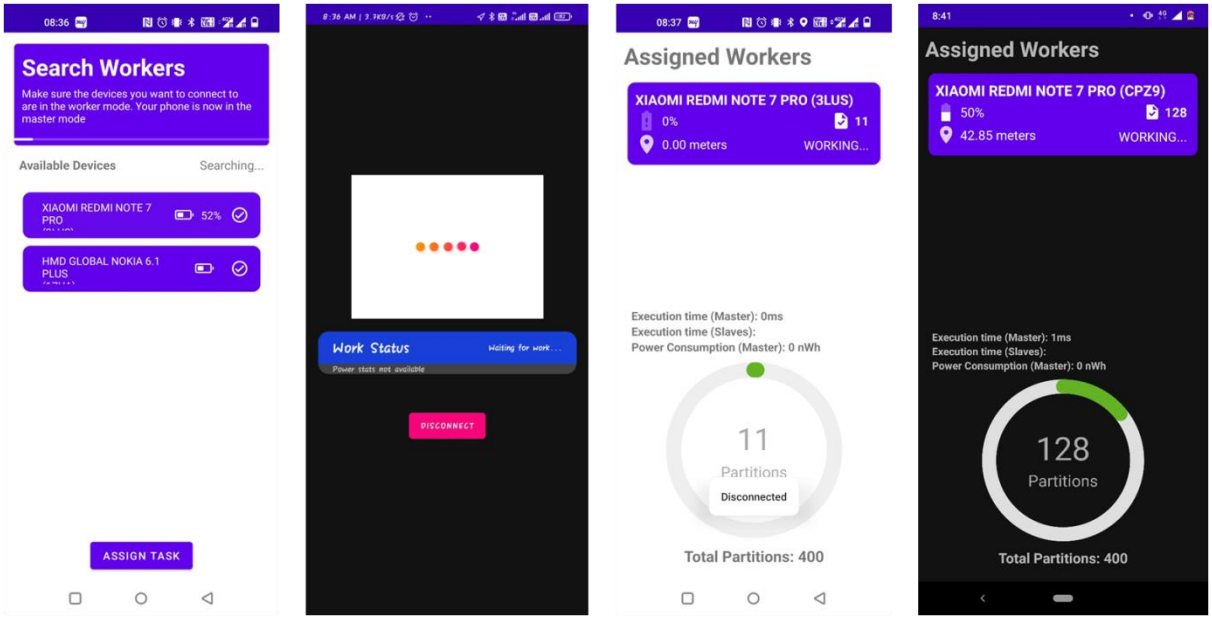


Fig 4. Screenshots of tasks getting assigned to the workers

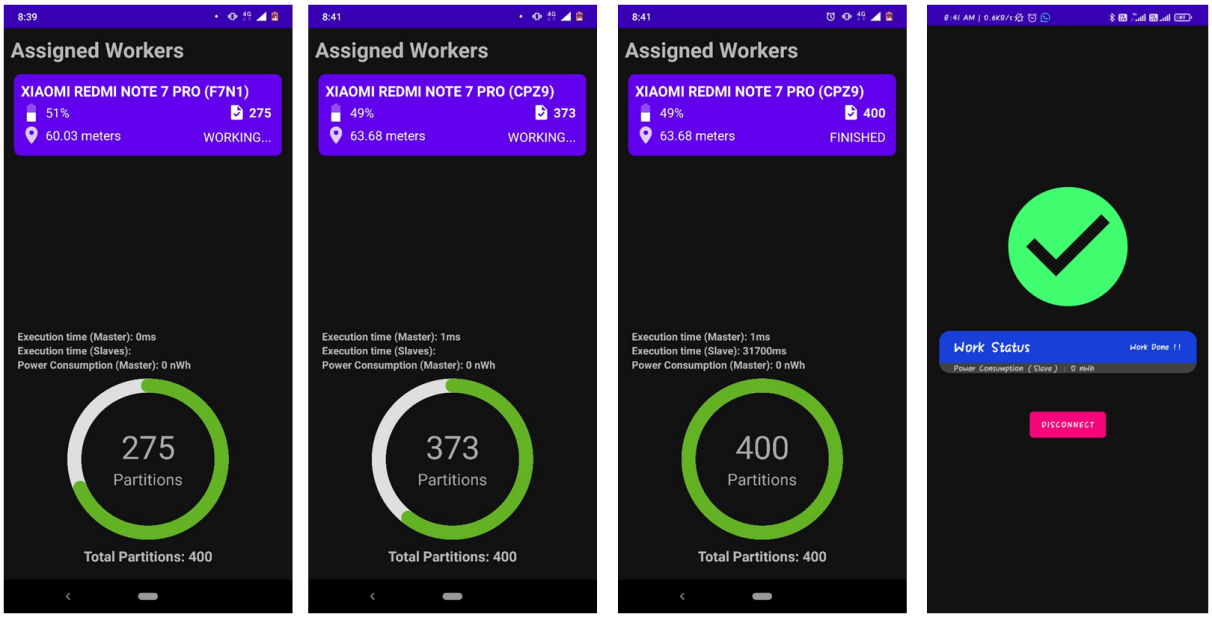


Fig5. Screenshots of tasks getting done one by one

9. Observations

Execution time	**Time (ms)**
Master only	1
Slaves	31700

We measured observations about our distributed system implementation of mobile phones using the performance analysis metric defined above.

a) Power consumption metrics: For a matrix multiplication of 20×20 matrices, the battery consumed on master node is below 1%. For Matrix multiplication done solely on the worker, no significant battery drop in the master battery was observed. For the distributed computation workers observed less than 1% battery consumption.

b) Execution time metrics:

1) Matrix multiplication of 20 × 20 size was completed on master node within 1 milliseconds.

2) Matrix multiplication on distributed environment took around 31700 milliseconds.

10. Conclusion

With this project, we achieved the implementation of one of the major cloud computing technologies, distributed computing, and a distributed system on a number of connected mobile networks which are present in the same network or in the Bluetooth reachability range of each other. This distributed system was leveraged for the implementation of the matrix multiplication across various devices. And as per the design that was implemented, it was a highly resilient and fault tolerant design. Using different matrix sizes for computation we conclude that in a distributed environment containing mobile phones, if the computation task is not heavy, then the additional overhead of communication will outweigh the benefits of distributed computing and the results would take a longer time to reach. On the other hand, if there is a considerably large matrix, distributed computing will be faster than the computing which will be done on the master device.

11. References

[1] Giurgiu, I., Riva, O., Juric, D., Krivulev, I., & Alonso, G. (2021, November). Calling the cloud: Enabling mobile phones as interfaces to cloud applications. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing* (pp. 83-102). Springer, Berlin, Heidelberg.

[2] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., & Yang, L. (2020, October). Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis* (pp. 105-114).

[3] Tali, K., & Mass, J. Recent Developments and Programming Models of Mobile Device-to-Device Computation Offloading.

[4] Android API. Battery Manager. [Online; accessed 25- March-2021]. 2021. URL: https://developer.android.com/ reference/android/os/BatteryManager.

[5] Android API. Location Data. [Online; accessed 25- March-2021]. 2021. URL: https://developer.android.com/ training/location/request-updates.

[6] Google API. Near By Connections. [Online; accessed 25- March-2021]. 2021. URL: https://developers.google.com/ nearby/connections/overview.

[7] Flores, H., Hui, P., Tarkoma, S., Li, Y., Srirama, S., & Buyya, R. (2015). Mobile code offloading: from concept to practice and beyond. *IEEE Communications Magazine*, *53*(3), 80-88.

[8] Jiao, L., Friedman, R., Fu, X., Secci, S., Smoreda, Z., & Tschofenig, H. (2013). Cloud-based computation offloading for mobile devices: State of the art, challenges and opportunities. *2013 Future Network & Mobile Summit*, 1-11.

[9] Maray, M., & Shuja, J. (2022). Computation offloading in mobile cloud computing and mobile edge computing: survey, taxonomy, and open issues. *Mobile Information Systems*, *2022*.

[10] Akherfi, K., Gerndt, M., & Harroud, H. (2018). Mobile cloud computing for computation offloading: Issues and challenges. *Applied computing and informatics*, *14*(1), 1-16.