Data Tracker
Android App Specific Data Usage Measurement

Berk Çemberci
Cristian Zara
Nicolai Colesnic
Danu Caus

13.01.16

Abstract

This report is created to give a detailed explanation of the implementation of the project Data Tracker - Android App Specific Data Usage Measurement for CM2

Contents

List of Figures

1 Introduction

The following project had as a goal the development of an application on top of the **Android Operating System** that would measure the traffic data used by each installed application on a smart phone device.

It consists of 2 important conceptual parts: **Application side programming** and **Server side programming**. The application was developed using the **Java** programming language and **XML** inside the **Android Studio** development environment.
The Server side was developed using **Python**, **Javascript**, **HTML** and **PHP**.
Along the way, several good practice programming methodologies, as well as small tricks were used to achieve the final result of a working application which fulfills the requirements.

Hopefully, the following report will be helpful in explaining all these things, from the most important facts to the smallest details and provide a proof of concept of how such an application might be implemented.

For clarity reasons, the report is written in 2 major parts: **IMPLEMENTATION** and **TEST CASES**. There are several code snippets used throughout in order to explain the main functionality. However, for the full implementation and commented code please refer to the source files.

2 Requirements

The main goal of the project is to measure the data for each application over time and acquisition of typical statistics.

It has several requirements that have to be fulfilled:

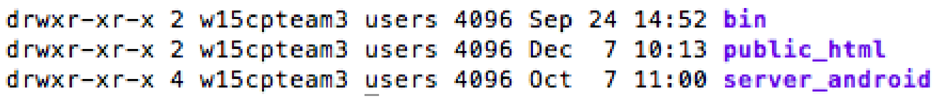
- In regular interval the application should check the data consumption per application.
- The statistics measured by the application are: cellular/wifi network traffic and also active/inactive (applications in foreground and background traffic).
- Measurements are sent to a server which collects statistics for all particular applications, but the users that sent the data are anonymous.
- The server should display the data on the webpage showing information per application.
- The user should be able to check his/her phone statistics by accessing a GUI of the application.

3 Implementation

3.1 Python Server

As mentioned earlier, the server, composed of a server socket that receives data from its clients, stores all the measurements into files. By the help of a php script, server is also capable of displaying data in form of a table with the option to plot statistical information on a web page.

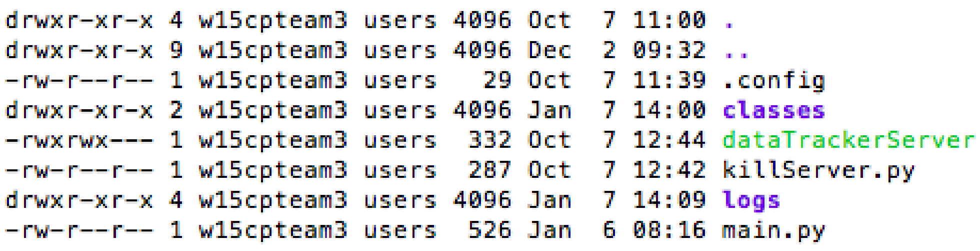
The main directory of the project is located right at */home/w15cpteam3/*, and it is composed of three sub directories which can be seen in Figure 1.



```
drwxr-xr-x 2 w15cpteam3 users 4096 Sep 24 14:52 bin
drwxr-xr-x 2 w15cpteam3 users 4096 Dec  7 10:13 public_html
drwxr-xr-x 4 w15cpteam3 users 4096 Oct  7 11:00 server_android
```

Figure 1: List of subdirectories

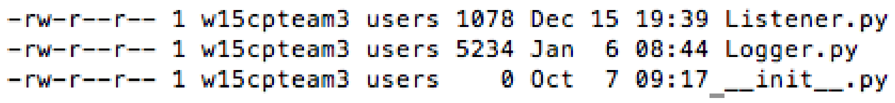
Python project files are located in the directory */home/w15cpteam3/server_ android*. The contents of this directory can be seen in Figure 2.



```
drwxr-xr-x 4 w15cpteam3 users 4096 Oct  7 11:00 .
drwxr-xr-x 9 w15cpteam3 users 4096 Dec  2 09:32 ..
-rw-r--r-- 1 w15cpteam3 users   29 Oct  7 11:39 .config
drwxr-xr-x 2 w15cpteam3 users 4096 Jan  7 14:00 classes
-rwxrwx--- 1 w15cpteam3 users  332 Oct  7 12:44 dataTrackerServer
-rw-r--r-- 1 w15cpteam3 users  287 Oct  7 12:42 killServer.py
drwxr-xr-x 4 w15cpteam3 users 4096 Jan  7 14:09 logs
-rw-r--r-- 1 w15cpteam3 users  526 Jan  6 08:16 main.py
```

Figure 2: Overview of the Python project directory

Directory with the name *logs* is where the received data is stored. And the directory *classes* is composed of the python classes for the server. As can be seen in Figure 3, there are only two classes available: Listener and Logger. The other python file *__init__.py* is necessary for Python to recognize this directory as a part of the project; so that modules, in this case classes, can be imported from the parent directories.

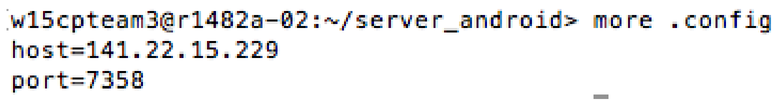


```
-rw-r--r-- 1 w15cpteam3 users 1078 Dec 15 19:39 Listener.py
-rw-r--r-- 1 w15cpteam3 users 5234 Jan  6 08:44 Logger.py
-rw-r--r-- 1 w15cpteam3 users    0 Oct  7 09:17 __init__.py
```

Figure 3: List of classes

The executable *dataTrackerServer* (Figure 2) is used to start or stop the server and the Python script *killServer.py* makes it possible to kill a running server which is also used by the executable *dataTrackerServer*. Last but not least, as the name suggests, the main of the Python code is located in *main.py*.

Finally, the hidden file *.config*, is used to define port and host used by the server socket. Contents of this config file can be seen in Figure 4.



```
w15cpteam3@r1482a-02:~/server_android> more .config
host=141.22.15.229
port=7358
```

Figure 4: Contents of .config

3.1.1 Socket Programming

As mentioned earlier, one of the two classes present in this python project is called *Listener*. This class takes care of the socket connections by working as a server socket. There are five methods available which are as follows:

- __init__() - Constructor
- bind()
- get_message() - Getter for the attribute message. It also returns a flag
- send_ack()
- listen()

Constructor, shown in Listing 1, is used to initialize the attributes of this class. The attributes that are present are as follows:

- server_socket: An object type socket
- message: A string to store the received message

```
def __init__(self):
        self.server_socket = socket.socket(socket.AF_INET,
            socket.SOCK_STREAM)
        self.message = ""
```

Listing 1: Constructor

The bind method, shown in Listing 2, expects two arguments which are the host name in the form of a string and the port number in the form of an integer. It tries to bind the server socket to the given host and port. If this is executed successfully, it tells the socket to start listening. The argument passed to the *listen()* method, defines the maximum simultaneous connections to the socket. The default maximum number allowed by python is 5, and therefore 5 is passed. This method returns a boolean together with an error message in the form of a string. If the socket successfully starts listening to the given port at the given host, it returns true. A false together with a non empty error message indicates the procedure failed.

```
def bind(self, host, port):
    try:
        self.server_socket.bind((host, port))
        self.server_socket.listen(5)
        return True, ""
    except:
        return False, "\n> Error: Couldn't establish
            connection to {host}:{port}".format(host=host, port
            =port)
```

Listing 2: bind()

The method get_message, shown in Listing 3, is a getter for the attribute message. Purpose here is to indicate the existence of a message by a flag. If the message attribute is equivalent to an empty string, this method returns false with a null string; otherwise it returns true together with the message itself.

```
def get_message(self):
    if self.message != "":
        msg = self.message
        self.message = ""
        return True, msg
    else:
        return False, ""
```

Listing 3: get_message()

The send_ack method, shown in Listing 4, expects one argument which is type socket object. The argument passed to this method is a reference to the client socket that is successfully connected. By using this reference, an acknowlegment is sent back. The length of the received message at the server socket is echod back to client so that the client can decide if the whole message was received by the server. The length of the received message is decremented by one to avoid the count for the terminator at the end of the received string.

```
def send_ack(self, conn):
    try:
        conn.send(str(len(self.message)-1))
    except:
        pass
```

Listing 4: send_ack()

The last method in this class that is worth mentioning called listen (Listing 5). Python sockets need to know the expected message length in advance; to be more precise, expected message length has to be passed as an argument to the *recv()* method of the sockets. Since the server cannot know the length of a message that is about to be received; idea here is to recieve the message byte by byte until there is no further incoming byte. This is achieved by setting a timeout. Once this method is called, it waits for an incoming connection to the server socket. The moment a connection is established, it creates a reference to the client socket that is connected; and sets its timeout for the receiving process to be 0.5 seconds. Afterwards, it starts reading the message byte by byte as the *conn.recv(1)* indicates. The moment receiving process of the next byte takes longer than 0.5 seconds, *recv()* method stops blocking by throwing an exception, and the loop is broken. This indicates the end of the received message. Listen method returns a boolean indicating if the received message is *"exit"*. If the received message is not *"exit"*, it is saved into the attribute message, and the method *send_ack()* is called with a reference to the received client socket. Once all these are completed, method returns true to indicate that a message is successfully received.

```
def listen(self):
    conn, addr = self.server_socket.accept()
    conn.settimeout(0.5);
    message = ""
    try:
        while True:
            byte = conn.recv(1)
            if byte=="":
                break
            else:
                message+=byte
    except:
        pass
    if message=="exit":
        return False
    elif message != "":
        self.message = message
        self.send_ack(conn)
    conn.close()
    return True
```

Listing 5: listen()

3.1.2 File Handling

The information sent by the client to the server is necessary to parse, and store on the server. This information contains stats which refer to each app installed on the clients device:

- wifiActiveHours
- wifiActiveMb
- wifiInActiveHours
- wifiInActiveMb
- cellularActiveHours
- cellularActiveMb
- cellularInactiveHours
- cellularInactiveMb

The information is handled by the Logger class, more specifically the updateLog() method. Each time this method is invoked, it processes the string passed to it, so that it is stored on the server.

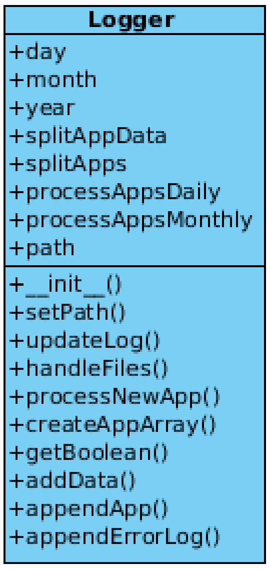


Figure 5: Logger Class Diagram

The Class diagram 5 depicts all the attributes and methods necessary to process the client app information. The only methods necessary by the mail are updateLog(), and updateErrorLog(), everything else is used for local purposes.

```
#The class holds information like date, path, string splitters
    and flags
   #so that it will be available from all methods
   def __init__(self):
      self.day = str(datetime.datetime.now().date()).split("-"
         )[2]
      self.month = str(datetime.datetime.now().date()).split("
         -")[1]
      self.year = str(datetime.date.today().year)
      self.path = os.getcwd() + "/logs/"
      self.setPath()
      self.splitAppData = "==="
      self.splitApps = "###"
      self.processAppsDaily = 0
      self.processAppsMonthly = 1
```

Listing 6: Logger Class Constructor

When a new object of class Logger is constructed, there is a necessity to populate its attributes, so that they would be available for the updateLog() 8 and appendErrorLog() 14 methods, parameters like:

- year/month/day - it is important to get these values from the servers system clock in order to be able generate the path of the files to be written, to know exactly for what period of time the new data is referred to and for appending a time stamp for an eventual error message
- path - it is generated directly from the relative path of the main.py file, plus the year. All files are stored in the logs folder. They are separated by year folders. If it doesn't exist, a folder with the current year is created. All the files for the entire year are stored in the same file.
- string splitters - are necessary to be fixed, and the same as the ones used on the client side to form the message containing all the apps information.
- daily/monthly flags - since the processing of data designed for daily and monthly stats is similar, the same code is used. Therefore it is important to distinguish between the two cases, with these flags

The data coming from the client is stored in files according to the servers current system clock, and whether it is daily or monthly data. If a folder or a file doesn't exist, it is created automatically by the Logger class. The path used within a call of the updateLog() method is created by calling the setPath() method (7). Since the data within the logs folder is separated by years, the year parameter is appended to the path.

```
def setPath(self):
        if not os.path.isdir(self.path + self.year):
            os.makedirs(self.path + self.year)
        self.path =  self.path + self.year + "/"
```

Listing 7: setPath() method

The same client app data is processed for getting the daily and the monthly usage. The year folder contains files for all the months , with the names corresponding to a months number : January is 01. A day file has the name formed by the month number and day of the month number: December-31 is 12.31

- updateLog() - it is called in the main class, and it expects the whole data string from the cient.

```
    def updateLog(self, text):
        #Set the current month, and day
        monthPath = self.path+self.month
        #Since all files are in the same folder, the days
            contain the month to which they belong
        #format mm.dd
        dayPath = self.path+self.month + "." + self.day

        #Split the client string in order to get a separate
            string for each App
        try:
            clientData = text.split(self.splitApps)
        except:
            return self.appendErrorLog("Corrupt Data: Missing
                or wrong app strig splitter.")


        #The daily and monthly data come within the same
            information, but they are handeled according
        #to their flags separately
```

```
successfullMonthHandled = self.handleFiles(
    clientData, monthPath, self.processAppsMonthly)
successfullDayHandeled = self.handleFiles(clientData
    , dayPath, self.processAppsDaily)

if (successfullMonthHandled and
    successfullDayHandeled): return self.
    appendErrorLog("Success")
else : return
```

Listing 8: updateLog() method

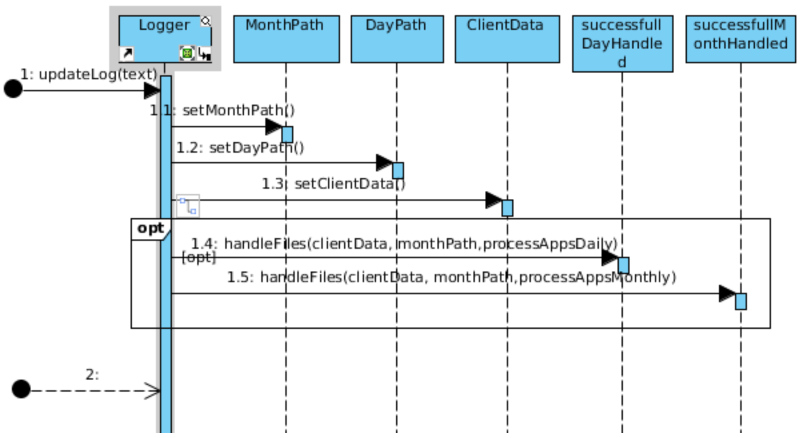


Figure 6: The flow of the updateLog() method

The sequence diagram (6) depicts how it manages all other methods in order to write the new data to the server files. Since all methods return success flags, the success or failores are propagated to this method as well. So it is possible to see even from this scope wheather there were or no any errors.

- handleFiles() method takes care of all the file handling, reading, writing the new data and clearing the temporary file. Since it is made to be used for calculating the data for monthly and daily usage it is important to differentiate between the two cases. This is being accomplished by the passed integer as processFlag. All the file handling is done according to the current day or month, which would correspond to certain files in the logs folder.

```
def handleFiles(self,clientData, filePath, processFlag)
    :
```

Listing 9: handleFiles() method

- processNewApps() takes as parameters:
  1. oldFile - which corresponds to the current day or month on the server
  2. newFile - which is a temp file to which the data from the oldFile is merged with the new data
  3. clientData - new data from the client
  4. processFlag - index which corresponds to the type of data to be updated

```
def processNewApps(self, oldFile, newFile, clientData,
    processFlag):
        #It is important here to make a deep copy of the
            string, so that this method could be called
        #multiple times
        tempClientData = list(clientData)
```

Listing 10: processNewApps() method

The clientData string is passed by reference, and since it is used within multiple function calls of the above (10) method, it is necessary to make a hard copy of it. The algorithm inside the method reads line by line from the server file and compares the with each app from the client data. If there is a match, the data is added together and appended as a line to the temp file. If it is not found in all the apps sent by the client, then that current server app is appended to the temp file, and the program continues with the next app in the server file. This way all matches are added together, and all mismatches are simply appended to the temp file. In case of success the temp file overwrites the current server file, containing the old data. In case of failure, the temp file is removed from the file system. Since the file handles are passed by reference, the files are closed and further handled after the method returns to the (9) method.

- createAppArray() has the purpose of splitting an app string into an array, and retrieving the daily/ monthly data usage flags. If an error occurs, a False flag is returned to mark the failure

```
def createAppArray(self, newAppData, newFile, oldFile):
```

Listing 11: createAppArray() method

- addData() has the purpose of adding the contents of the an app from the server file to the same app sent from the client. The two main purposes of the method is to add the contents with the correct format (long integer), and make sure that the number of users is updated accordingly. If the sent flag is false, the number of users remains the same as the one from the server file. Only when the flag is true, the number of users is incremented by one. This method is written so that there is no difference for which data period these calculations are referred.

```
def addData(self, fileAppArray, flag, newAppArray):
```

Listing 12: addData() method

- appendApp() has the purpose to convert the newAppArray into an app string that can be appended to the server file. This method is called only for apps coming from the client that are not yet present on the server storage file. It means then that if the passed boolean flag (which referrers to the daily or monthly period flags)is true, the the number of users is set to "1".

```
def appendApp(self, flag, newAppArray):
```

Listing 13: appendApp() method

- It is important to mention at this point the script for the Logger Class (5) is divided into smaller methods, each returning boolean success flags. This means that in all known cases exceptions are caught, and handled. In case of one, an appropriate error message is appended (14) to the error log.

```
def appendErrorLog(self, error):
        date = str(datetime.datetime.now().date())
        time = str(datetime.datetime.now().time()).split('.'
            )[0]
        if not os.path.isfile(self.path+".log"+self.month):
            open(self.path+".log"+self.month, "w").close()
        open(self.path+".log"+self.month, "a").write("["+
            date+":"+time+"]: "+error+"\n")
```

Listing 14: appendErrorLog() method

Within a month a file is created which stores all the error messages passed to it. The name format of these files is : ".log01" which means that it is the error log file for January, while the dot in front means that the file is hidden.

3.1.3 Script dataTrackerServer

The executable with the name *dataTrackerServer* is a bash script. It is designed to start or stop the Python server. In order to differentiate between these two tasks, it requires a commandline argument in the form of a string being either *start* or *stop*. As the names suggest, if the executable is called with the argument *start*, it starts the server and if it is called with the argument *stop*, it stops the server. Corresponding code of the bash script can be seen in Listing 15.

```
#!/bin/bash

#checks if an argument is given
if [ "$#" -ne 1 ]; then
    printf "\n> Please run with an argument, either 'start' or
        'stop'\n\n"
    exit
fi

arg1=$1

#if the argument is start, executes the main.py
#if it is stop, executes the python script killServer.py
if [ $arg1 == "start" ]; then
   exec python main.py &
   printf "\n> Server is started\n\n"
else
   if [ $arg1 == "stop" ]; then
      exec python killServer.py &
      printf "\n> Server is killed\n\n"
   fi
fi
```

Listing 15: dataTrackerServer bash script

3.1.4 Script killServer.py

It is also worth mentioning what exactly the script *killServer.py* does. First of all, it reads the port number defined in *.config* file. Once that is read, it tries to connect to that port at the localhost by the help of a socket. If the connection fails, which means the server is down, it terminates. If it connects to the port successfully, sends the string *"exit"* over to initiate termination of the server. (Listing 16)

```
import socket,re
from sys import exit

#gets the port number from the config file
port = 0
f = open(".config", "r")
for line in f:
    if re.search("port", line):
        port = int(line.split("=")[-1].strip())
        break
f.close()
#creates a client socket
#tries to connect to the given port at localhost
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
    client.connect(('',port))
except:
    #if connection fails, exits
    exit()
#if connection is successful
#sends the command 'exit' to the server to kill it
client.send("exit")
```

Listing 16: killServer.py python script

3.1.5 Main.py

Main acts as some kind of a controller. The purpose here is to read the port number from the *.config* file, instantiate objects of the two classes that are mentioned earlier, *Listener* and *Logger*; listen to the port by the help of the *Listener* object, and log the received messages by the help of the *Logger* object. In order to achieve all these, it is necessary to import some libraries as well as the classes *Listener* and *Logger*. (Listing 17)

```
from sys import exit
from classes.Listener import Listener
from classes.Logger import Logger
import re
```

Listing 17: Import statements

Inside the main, first thing to do is to retrieve the port number from the *.config* file. (Listing 18).

```
port = 0
f = open(".config", "r")
for line in f:
    if re.search("port", line):
        port = int(line.split("=")[-1])
          break
f.close()
```

Listing 18: Reading the port number from the config file

Once the port number is read, both objects are instantiated and main tells the *Listener* object to bind to the given port. Success of this process is indicated by a flag and a error message returned by this function call; if the procedure fails, this error is printed into the error log by the help of the *Logger* object, and the process terminates. (Listing 19)

```
listener = Listener()
logger = Logger()
flag, message = listener.bind("", port)
if not flag:
    logger.appendErrorLog(message)
    exit()
```

Listing 19: Instantiation of objects and binding to the read port

If all the above procedures are executed successfully, program goes into a while loop so that it can continuously listen to the given port and log the retrieved data. (Listing 20)

```
while True:
    if not listener.listen():
        break
    flag, message = listener.get_message()
    if flag:
        logger.updateLog(message)
```

Listing 20: While loop to listen and retrieve data

The method *listen()* of the *Listener* object, returns *false* if the message received is *"exit"*. The while loop is broken only if that message is received. Also, it is worth mentioning that this while loop doesn't consume any cpu power since the *listen()* method is blocking the further execution inside the loop. Once it is called, it waits for an incoming message through the given port, and as long as there is no message being received, it continues waiting. This is the main reason why, *killServer.py* script (Section 3.1.4) is written; which sends the message *"exit"* to stop this blocking call so the server can be killed.

If any other message is received by the *Listener* object, the message is retrieved by the *get_message()* call and written into the log files by the help of the *Logger* object's *updateLog()* method.

3.1.6 Data Table

In order to avoid an admin to manually go through the log files and try to figure out the usage statistics, a simple php script is created. Php script is able read in a variable called *period* defined through the URL that calls the script. For example, the URL demonstrated in Listing 21, tells the php script to populate its table with the data from the November logs of the year 2015.

```
http://141.22.15.229/~w15cpteam3/dataTracker?period=2015/11
```

Listing 21: URL example for November logs of the year 2015

In a similar fashion, any other month as well as individual days can be displayed by manipulating the variable *period*. For example, by changing the *...?period=2015/11* to *...?period=2015/12* in Listing 21, user can see December logs of the year 2015.

In order to see individual days, again it is required to manipulate the *period* variable. By changing *...?period=2015/11* to *...?period=2015/01.11*, php script can show 01.11.2015 logs in a table. (Listing 22)

```
http://141.22.15.229/~w15cpteam3/dataTracker?period=2015/01.11
```

Listing 22: URL example for 01.11.2015

The way the php script works is quite simple. First the headers for the html table are defined in an *EOT* block assigned to a variable called *html* as can be seen in Listing 23.

```
$html = <<<EOT
    <script src="sorttable.js"></script>
    <html>
        <table class="sortable" style="width:100%">
            <tr>
                <td><p><b>AppName</b></p></td>
                <td><p><b>#Users</b></p></td>
                <td><p><b>Wifi-<br>Active Hours <i>[h]</i></b></p
                    ></td>
                <td><p><b>Wifi-<br>Active <i>[mb]</i></b></p></td>
                <td><p><b>Wifi-<br>InActive Hours <i>[h]</i></b></
                    p></td>
                <td><p><b>Wifi-<br>InActive <i>[mb]</i></b></p></
                    td>
                <td><p><b>Cellular-<br>Active Hours <i>[h]</i></b
                    ></p></td>
                <td><p><b>Cellular-<br>Active <i>[mb]</i></b></p
                    ></td>
                <td><p><b>Cellular-<br>InActive Hours <i>[h]<i/></
                    b></p></td>
                <td><p><b>Cellular-<br>InActive <i>[mb]<i/></b></p
                    ></td>
            </tr>
            <tr>
            </tr>
EOT;
```

Listing 23: EOT block for table headers

In order to make individual columns sortable, an open source Javascript script is used, [2]. Once defined in an html code, this script is able to add sortability to each table that is defined with its class set to *"sortable"*. This is exactly what's done with the table defined in the EOT block shown in Listing 23.

Once the html headers are defined, next thing on the line is the reading in process of the desired log file. As mentioned earlier, the desired log file is defined by the help of the variable *period* via the URL. Php scripts reads in that variable and tries to open the corresponding log file as can be seen in Listing 24.

```
$handle = fopen("../server_android/logs/".$_GET['period'], "r"
    );
```

Listing 24: Reading the defined variable 'period' and checking the log file

$_GET[variable_name] is how the Php script access the variable defined in the URL. If the variable refers to an existing log file in the logs directory, reference to the opened file is assigned to the *handle* variable. If the file doesn't exist, the variable simply points to null. Therefore by the help of an if statement content of the variable *handle* is checked, and the code then reads in the defined file. (Listing 25)

```
if ($handle) {
    while (($line = fgets($handle)) !== false) {
        $html .= "<tr>\n";
        $parts = split("===", $line);
        foreach ($parts as $part) {
            $html .= "<td>".$part."</td>\n";
        }
        $html .= "</tr>\n";
    }
    fclose($handle);
}
```

Listing 25: Appending the html table by iterating through the file line by line

As can be seen in Listing 25, by the help of a while loop and the built in php function *fgets()*, the file is read line by line and each line is assigned to the variable *line*. When the end of line is reached, *fgets()* method simply returns false which results in the while loop being broken. For each line in the file, a *<tr>* is added to the html code. Each line in the log files represent a table row with its columns seperated by *"==="*. Therefore line is split by *"==="* and each resulting part is surrounded by *<td>...</td>* and added to the earlier set *<tr>*. Once all the columns (<td>) are added to the row (<tr>), the row is closed by appending the html string by *</tr>* to close the *<tr>* block. Once the entire table is populated according to the lines set in the desired log file, earlier opened file is closed.

At this point, all that is left is to signal the end of table block as well as the html block in the html string which is done by appending the string with *</table></html>*. This means the html string is ready to be displayed on the web page, and the script tells the browser to do exactly that by the *echo* call. (Listing 26)

```
$html .= "</table></html>";
echo $html;
```

Listing 26: End of the table and html block together with the echo call

3.1.7 Data Plotting

The information stored in the /logs/ folder is a raw representation data. It refers to general statistics regarding all apps of all the clients using this dataTracker app. These statistics are:

- wifiActiveSeconds
- wifiActiveBytes
- wifiInActiveSeconds
- wifiInActiveBytes
- cellularActiveSeconds
- cellularActiveBytes
- cellularInactiveSeconds
- cellularInactiveBytes

This data can be used by different entities:

- app developers
- internet providers
- individual clients

So in order to show the possible uses for this raw data, besides visualising the raw on the dataTracker web page, a small charting web page was created for this purpose (Listing 27). The webpage itself is a simple html design that is able to create either pie charts of bar charts, from the raw data stored on the server.

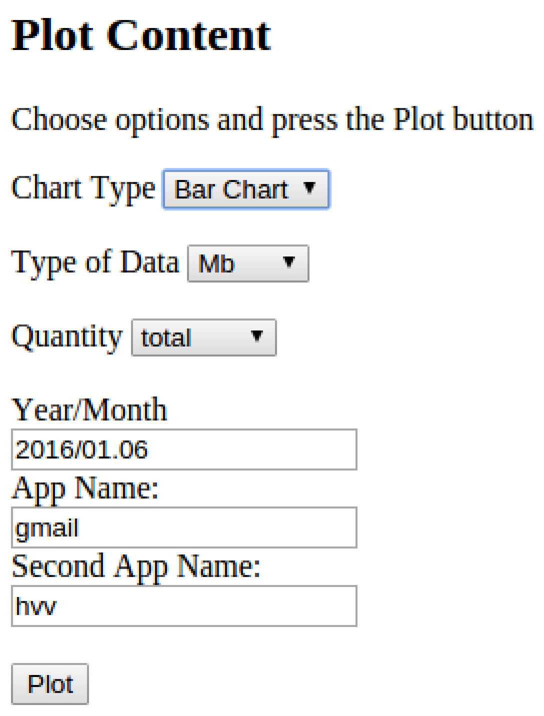
```
http://141.22.15.229/~w15cpteam3/plot.html
```

Listing 27: Charting web page link

The plotting web page is a simple html UI, that can be run in a browser. It contains :

- plot.html - the file describing the web page
- script.js - a script that handles all user inputs
- read.php - a script that reads files on the server
- Chart.js - a open source script(Citation [4] that creates the plots
- jquery.js - an open source script (Citation [5] used to make a connection from the client side script, and the server read.php script

Loading the plot.html (Listing 27), would show in the browser the GUI, with which the user can make a chart from the apps raw data stored on the server.


Plot Content

Choose options and press the Plot button

Chart Type: Bar Chart

Type of Data: Mb

Quantity: total

Year/Month: 2016/01.06

App Name: gmail

Second App Name: hvv

Figure 7: Html Page for Plotting the raw data

All html inputs from the page are handled by the script.js file, which is located in the same folder as plot.html. Since all html elements have IDs, it is easy to access them from the script.

```
Second App Name:<br>
      <input type="text" id="2ndAppName"><br> <br>

      <button type="button" id="plotButton" onclick="plot()">
         Plot</button> <br>
```

Listing 28: Html Element

Example 28 depicts multiple html elements like: text area, button. In the case of the "Plot" button, it contains the:

- type - button
- id - "plotButton"

- onclick method - plot(), implemented in the script.js file
- name - Plot

Due to the fact that the html page has multiple options , for multiple plot types, it is important to limit the user as which inputs should he be allowed to trigger. The default plot type is the Pie Chart. Therefore the user should not be able to choose a second app name, or calculations by the number of users.

```
Second App Name:
document.getElementById("dataQuantity").disabled = true;
document.getElementById("2ndAppName").readOnly = true;
```

Listing 29: Initial settings of the plot.html page UI

Since these are the first lines in the first script lines executed, it makes sure that the conditions above are met. The script directly references the appropriate html elements by their id tag.

However, as soon as the Chart Type selector element is changed to "Bar Chart", it is necessary to make all the options available.

```
function changeAccordingToType(){
    if(document.getElementById("chartType").value == "Bar Chart
        "){
        document.getElementById("2ndAppName").readOnly = false;
        document.getElementById("dataQuantity").disabled = false
            ;
    }
    else{
        document.getElementById("2ndAppName").readOnly = true;
        document.getElementById("dataQuantity").disabled = true;
    }
}
```

Listing 30: onClick method of the chartType html selector

By clicking on it, the plot() 31 method is invoked. This function handles all the actions in order to plot the required data, or in case of an error, print an error message in the browser console

```
//The onclick method for the Plot button html element
function plot(){
    //get the file name
    var path = document.getElementById("period").value;
    var chartData;
    //Post methon using jquery, which invokes the read.php
       script
    //Since the its callback method is infoked assynchronously,
        the most logic resides inside
    $.post('read.php', {postpath:path},
    function(data){
            /*
            ...
            */
    });
    return false;
}
```

Listing 31: plot() method snipped

The creation of charts is done mainly using the Chart.js library [4].

- First, a canvas is required, onto which the graph is going to be plotted, by accessing it by its id "chart-area".
- Then it the html <div>, containing all the user inputs is made hidden, to make space on the page for the graph.
- Get the chart data inputs(values, labels) according to the type of plot. Also the data for the plots are in Mb, and Hours, which is different fromm the raw data datatypes stored in the /logs/ folder files.
- Create the chart object using the precalculated chart data.
- generate labels so that it is possible to distinguish between the values plotted

```
            var ctx = document.getElementById("chart-area").
               getContext("2d");
            document.getElementById("input").hidden = true;
            if(document.getElementById("chartType").value == "Pie
                Chart"){
               chartData = createPieChartData(firstResult);
               window.myChart = new Chart(ctx).Pie(chartData);
               document.getElementById('js-legend').innerHTML =
                  window.myChart.generateLegend();
            }
            else{
               chartData = createBarChart(firstResult,
                  secondResult);
               window.myChart = new Chart(ctx).Bar(chartData, {
                  responsive : true});
            document.getElementById('js-legend').innerHTML =
                window.myChart.generateLegend();
        }
```

Listing 32: Creation and outputing the chosen plot

An important part of this mechanism is that the browser loaded script makes a POST request to the server, in order to retrieve the data from a certain file. For this, the script makes use of the jquery library (Listing 27). On the server side a php script is executed and the contents of the file, if found are echoed to the client.

```
$path = $_POST['postpath'];
$handle = fopen("../server_android/logs/".$path, "r");
```

Listing 33: POST method that reads the file on the server

The file is read, according to the passed file name only 33. This way, the client doesn't have any idea about the file storage on the server side.

```
if ($handle) {
        //read all lines and append them to the output string
        while (($line = fgets($handle)) !== false) {
            $result .= $line;
            }
        //close file
        fclose($handle);
    }
    echo $result;
```

Listing 34: Read file and echo the result to the client

Since this script 34 is executed only on the client side, it also means that it is executed in asynchronous manner in reference to the client side browser script. This then implies that all data manipulations regarding data read from the server files needs to happen in the POST request callback method.

3.2 App

3.2.1 Interrupts

This application makes use of both: interrupts, and polling.

There are two polling states:

1. When phone screen is on, the polling is done more frequently (once in 0.5 seconds)
2. When the phone is sleeping, the polling frequency is user selectable (15 minutes, 30 minutes, 1 hour, 6 hours)

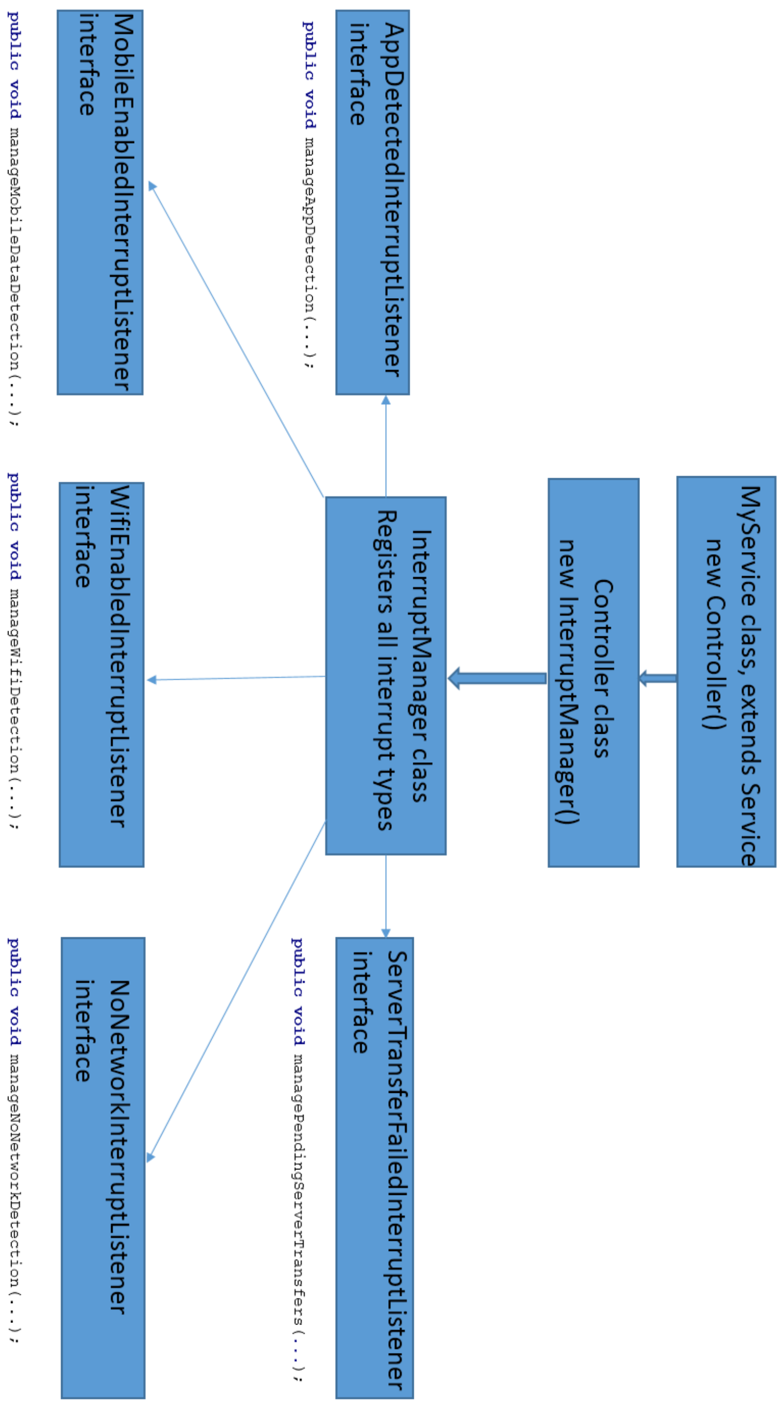
Fortunately, some of the information can be received using an interrupt based approach. This makes it possible not to drain the battery of the phone too much and make the measurements as accurate as possible, without an insanely high polling frequency.

The following events are detected using interrupts:

1. Screen goes On
2. Screen goes Off
3. WiFi gets enabled
4. Mobile Data gets enabled
5. Network Connection becomes unavailable
6. Transfer to server succeeded
7. Transfer to server failed
8. New application active

In order to create a scalable system, this application makes use of an MVC type structure which will be presented subsequently.

To gain a better understanding of the overall system, here is a top view architecture:


MyService class, extends Service
new Controller()
Controller class
new InterruptManager()
InterruptManager class
Registers all interrupt types
AppDetectedInterruptListener interface
MobileEnabledInterruptListener interface
WifiEnabledInterruptListener interface
NoNetworkInterruptListener interface
ServerTransferFailedInterruptListener interface

```
public void manageAppDetection(...);
public void manageMobileDataDetection(...);
public void manageWifiDetection(...);
public void managePendingServerTransfers(...);
public void manageNoNetworkDetection(...);
```

Figure 8: Top View System structure

There is a class called **InterruptManager** where all the types of interrupts are registered.

There are also 5 interfaces (corresponding to 5 types of interrupts), each having one declared method. Of course, more methods can be added if need be. Here is the list of all the interfaces with their declared methods respectively:

1. AppDetectedInterruptListener
   - public void manageAppDetection(String appName, String appPackageName, int id);
2. MobileEnabledInterruptListener
   - public void manageMobileDataDetection();
3. NoNetworkInterruptListener
   - public void manageNoNetworkDetection();
4. WiFiEnabledInterruptListener
   - public void manageWifiDetection(String currentAppName);
5. ServerTransferFailedInterruptListener
   - public void managePendingServerTransfers(boolean uploadStatus);

The advantage of this structure is that, it is very easy to create a **Controller** class in which the **InterruptManager** is instantiated and then all the interface methods are implemented. Please refer to the attached **.java** files in order to see the full implementation.

Interrupt handlers inside **Controller** class:

```
        interruptManager = new InterruptManager();

        interruptManager.setAppDetectedInterruptListener(new
            AppDetectedInterruptListener() {
             @Override
             public void manageAppDetection(String appName,
                 String appPackageName, int id) {
                  Log.i("Log", "An app was detected with name: "
                      + appName);
               // All the code goes here
            }
        });

         interruptManager.setWifiEnabledInterruptListener(new
            WifiEnabledInterruptListener() {
            public void manageWifiDetection(String appName) {
                Log.i("Log", "wifi enabled");
                // All the code goes here

            }
        });

        interruptManager.setMobileDataEnabledInterruptListener
           (new MobileEnabledInterruptListener() {
            public void manageMobileDataDetection() {
                Log.i("Log", "mobile data enabled");
                // All the code goes here
            }
        });


        interruptManager.
           setNoNetworkInterruptInterruptListener(new
           NoNetworkInterruptListener() {
            public void manageNoNetworkDetection() {
                Log.i("Log", "not connected to network");
                //All the code goes here
            }
        });

        interruptManager.
           setServerTransferFailedInterruptListener(new
           ServerTransferFailedInterruptListener() {
            @Override
            public void managePendingServerTransfers(boolean
               uStatus) {
                //All the code goes here
            }
        });
```

This kind of structure makes it very easy to have all the **"Controller Path"** code inside the **Controller** class.

Note that some interrupts, for example: "Screen goes on" and "Screen goes off" do not have a corresponding interface. This is because these events are detected and handled inside the **MyService** class and not in the **Controller** class. The interrupts that have interfaces however, are detected inside **MyService**, but handled inside **Controller**.

The last remaining part is to register all these interrupts in the **InterruptManager** class. This is done simply by using setter methods:

Code for **InterruptManager** class:

```
public class InterruptManager {

    protected AppDetectedInterruptListener
        appDetectedinterruptListener;
    protected WifiEnabledInterruptListener
        wifiEnabledInterruptListener;
    protected MobileEnabledInterruptListener
        mobileEnabledInterruptListener;
    protected NoNetworkInterruptListener
        noNetworkInterruptListener;
    protected ServerTransferFailedInterruptListener
        serverTransferFailedInterruptListener;

    public InterruptManager(){

    }

    public void setAppDetectedInterruptListener(
        AppDetectedInterruptListener
        appDetectedinterruptListener){
         this.appDetectedinterruptListener =
            appDetectedinterruptListener;
    }

    public void setWifiEnabledInterruptListener(
        WifiEnabledInterruptListener
        wifiEnabledInterruptListener){
         this.wifiEnabledInterruptListener =
            wifiEnabledInterruptListener;
    }

    public void setMobileDataEnabledInterruptListener(
        MobileEnabledInterruptListener
        mobileEnabledInterruptListener){
         this.mobileEnabledInterruptListener =
            mobileEnabledInterruptListener;
    }

    public void setNoNetworkInterruptInterruptListener(
        NoNetworkInterruptListener noNetworkInterruptListener){
         this.noNetworkInterruptListener =
            noNetworkInterruptListener;
    }

    public void setServerTransferFailedInterruptListener(
        ServerTransferFailedInterruptListener
        serverTransferFailedInterruptListener){
         this.serverTransferFailedInterruptListener =
            serverTransferFailedInterruptListener;
    }

}
```

In order to detect whether the screen goes on or off, a broadcast receiver is used:

```
public class ForegroundBroadcastReceiver extends
    BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent)
            {
             if (intent.getAction().equals(Intent.
                ACTION_SCREEN_OFF)) {
                 Log.i("Log", "Screen went OFF");
                 Controller.SCREEN_ON = 0;
                 controller.interruptManager.
                     appDetectedinterruptListener.
                     manageAppDetection("", "", 0);
                 if(mForegroundTimer != null){
                      mForegroundTimer.cancel();
                 }
```

```java

            } else if (intent.getAction().equals(Intent.
                ACTION_SCREEN_ON)) {
                Log.i("Log","Screen went ON");
                Controller.SCREEN_ON = 1;
                mForegroundTimer = new Timer();
                mForegroundTimer.scheduleAtFixedRate(new
                    CheckForegroundAppsTimerTask(), 0,
                    FOREGROUND_INTERVAL);
            }
        }
    }
```

In order to detect whether WiFi connection appears, or detect mobile data network, again a broadcast receiver can be implemented:

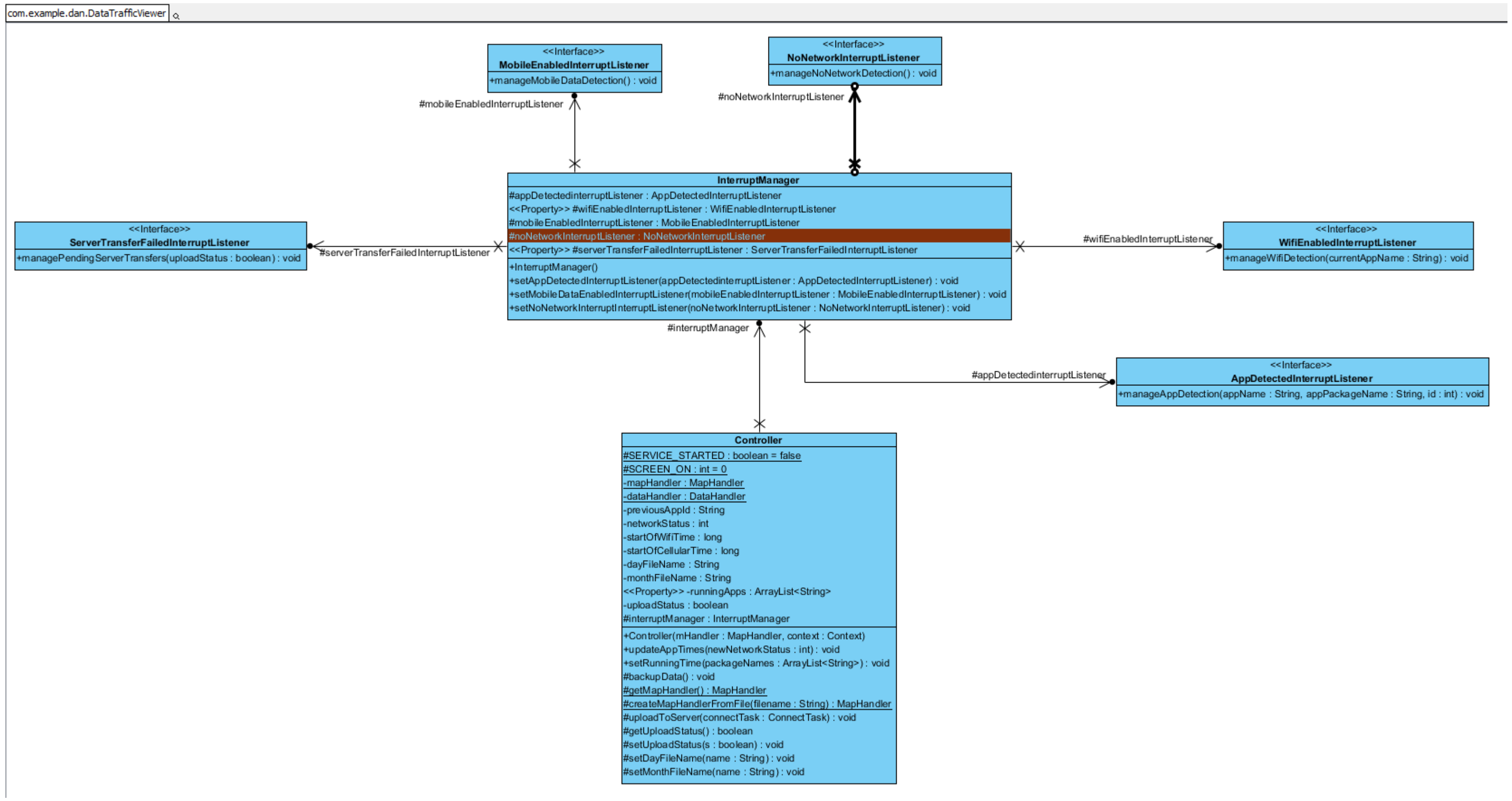
```java
    public class NetworkBroadcastReceiver extends
        BroadcastReceiver{
        @Override
        public void onReceive(Context context, Intent intent)
            {
            String status = NetworkUtil.
                getConnectivityStatusString(context);
            if(status.equals("Wifi enabled")){
                if( (controller.interruptManager.
                    wifiEnabledInterruptListener != null)){
                    controller.interruptManager.
                        wifiEnabledInterruptListener.
                        manageWifiDetection(
                        currentRunningAppName);
                }
            }
            if(status.equals("Mobile data enabled")){
                if( (controller.interruptManager.
                    mobileEnabledInterruptListener != null)){
                    controller.interruptManager.
                        mobileEnabledInterruptListener.
                        manageMobileDataDetection();
                }
            }
            if(status.equals("Not connected to Internet")){
                if( (controller.interruptManager.
                    noNetworkInterruptListener != null)){
                     controller.interruptManager.
                         noNetworkInterruptListener.
                         manageNoNetworkDetection();
                }
            }
        }
    }
```

Note that these broadcast receivers must be set up when the service starts (aka in the public void **onCreate()** method of **MyService** class which extends **Service**):

Registering the Broadcast Receivers:

```
        mScreenStateReceiver = new ForegroundBroadcastReceiver
            ();
        mNetworkBroadcastReceiver = new
            NetworkBroadcastReceiver();

        IntentFilter screenStateFilter = new IntentFilter();
        screenStateFilter.addAction(Intent.ACTION_SCREEN_ON);
        screenStateFilter.addAction(Intent.ACTION_SCREEN_OFF);
        registerReceiver(mScreenStateReceiver,
            screenStateFilter);

    //-----------------------------------------------------

        final IntentFilter networkFilters = new IntentFilter()
            ;
        networkFilters.addAction("android.net.wifi.
            WIFI_STATE_CHANGED");
        networkFilters.addAction("android.net.conn.
            CONNECTIVITY_CHANGE");
        registerReceiver(mNetworkBroadcastReceiver,
            networkFilters);
```

Getting back to the core structure of the application, please refer to the figures below for a more detailed view of the relation between the **Controller**, **InterruptManager** and **MyService**. These 3 classes form the backbone of the entire program.


com.example.dan.DataTrafficViewer
<<Interface>>
MobileEnabledInterruptListener
+manageMobileDataDetection() : void
<<Interface>>
NoNetworkInterruptListener
+manageNoNetworkDetection() : void
#mobileEnabledInterruptListener
#noNetworkInterruptListener
InterruptManager
#appDetectedinterruptListener : AppDetectedInterruptListener
<<Property>> #wifiEnabledInterruptListener : WifiEnabledInterruptListener
#mobileEnabledInterruptListener : MobileEnabledInterruptListener
#noNetworkInterruptListener : NoNetworkInterruptListener
<<Property>> #serverTransferFailedInterruptListener : ServerTransferFailedInterruptListener
+InterruptManager()
+setAppDetectedInterruptListener(appDetectedinterruptListener : AppDetectedInterruptListener) : void
+setMobileDataEnabledInterruptListener(mobileEnabledInterruptListener : MobileEnabledInterruptListener) : void
+setNoNetworkInterruptInterruptListener(noNetworkInterruptListener : NoNetworkInterruptListener) : void
<<Interface>>
ServerTransferFailedInterruptListener
+managePendingServerTransfers(uploadStatus : boolean) : void
#serverTransferFailedInterruptListener
#wifiEnabledInterruptListener
<<Interface>>
WifiEnabledInterruptListener
+manageWifiDetection(currentAppName : String) : void
#interruptManager
#appDetectedinterruptListener
<<Interface>>
AppDetectedInterruptListener
+manageAppDetection(appName : String, appPackageName : String, id : int) : void
Controller
#SERVICE_STARTED : boolean = false
#SCREEN_ON : int = 0
-mapHandler : MapHandler
-dataHandler : DataHandler
-previousAppId : String
-networkStatus : int
-startOfWifiTime : long
-startOfCellularTime : long
-dayFileName : String
-monthFileName : String
<<Property>> -runningApps : ArrayList<String>
-uploadStatus : boolean
#interruptManager : InterruptManager
+Controller(mHandler : MapHandler, context : Context)
+updateAppTimes(newNetworkStatus : int) : void
+setRunningTime(packageNames : ArrayList<String>) : void
#backupData() : void
#getMapHandler() : MapHandler
#createMapHandlerFromFile(filename : String) : MapHandler
#uploadToServer(connectTask : ConnectTask) : void
#getUploadStatus() : boolean
#setUploadStatus(s : boolean) : void
#setDayFileName(name : String) : void
#setMonthFileName(name : String) : void

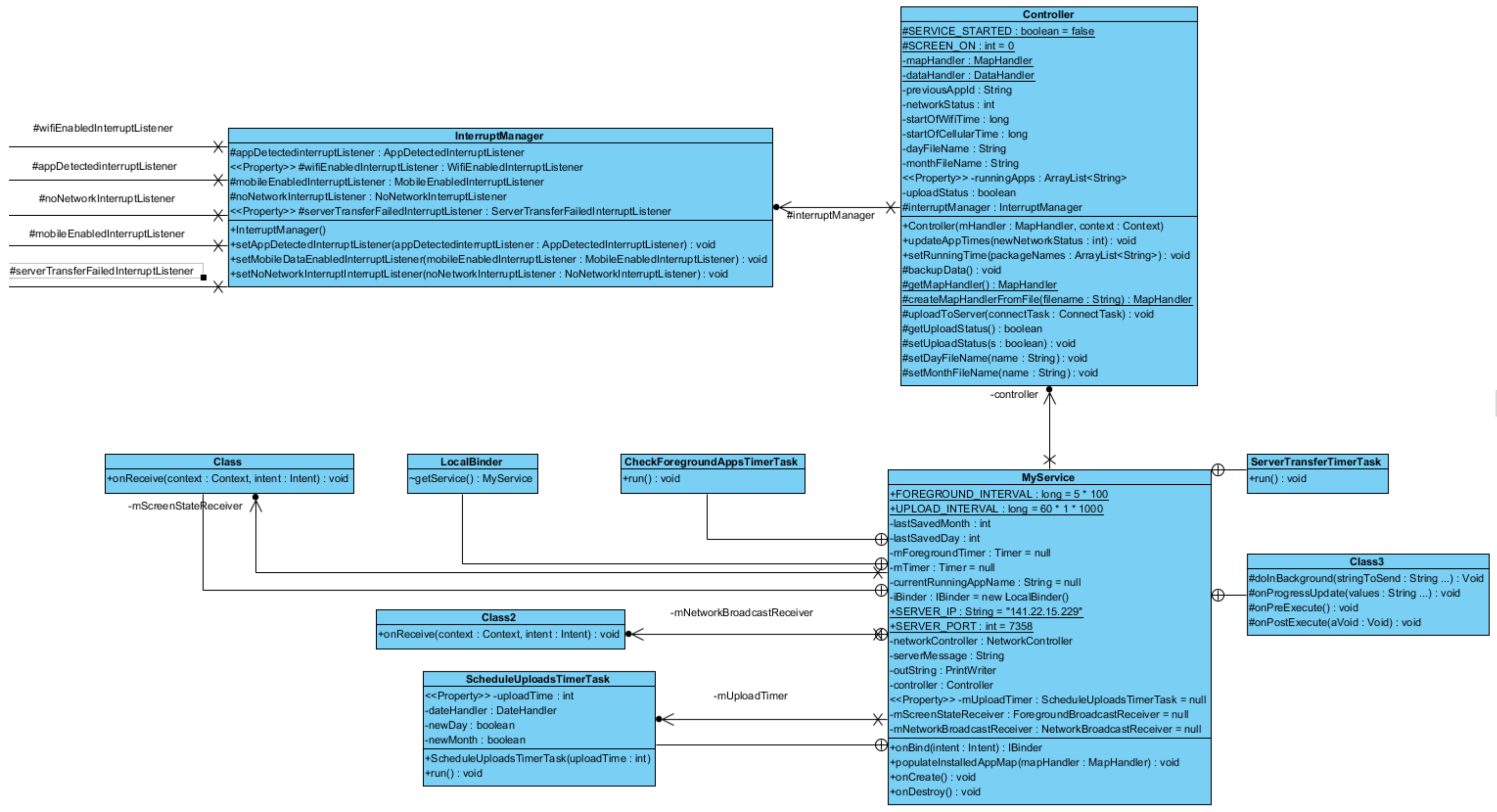

InterruptManager
#appDetectedinterruptListener : AppDetectedInterruptListener
<<Property>> #wifiEnabledInterruptListener : WifiEnabledInterruptListener
#mobileEnabledInterruptListener : MobileEnabledInterruptListener
#noNetworkInterruptListener : NoNetworkInterruptListener
<<Property>> #serverTransferFailedInterruptListener : ServerTransferFailedInterruptListener
+InterruptManager()
+setAppDetectedInterruptListener(appDetectedinterruptListener : AppDetectedInterruptListener) : void
+setMobileDataEnabledInterruptListener(mobileEnabledInterruptListener : MobileEnabledInterruptListener) : void
+setNoNetworkInterruptInterruptListener(noNetworkInterruptListener : NoNetworkInterruptListener) : void
#wifiEnabledInterruptListener
#appDetectedinterruptListener
#noNetworkInterruptListener
#mobileEnabledInterruptListener
#serverTransferFailedInterruptListener
#interruptManager
Controller
#SERVICE_STARTED : boolean = false
#SCREEN_ON : int = 0
-mapHandler : MapHandler
-dataHandler : DataHandler
-previousAppId : String
-networkStatus : int
-startOfWifiTime : long
-startOfCellularTime : long
-dayFileName : String
-monthFileName : String
<<Property>> -runningApps : ArrayList<String>
-uploadStatus : boolean
#interruptManager : InterruptManager
+Controller(mHandler : MapHandler, context : Context)
+updateAppTimes(newNetworkStatus : int) : void
+setRunningTime(packageNames : ArrayList<String>) : void
#backupData() : void
#getMapHandler() : MapHandler
#createMapHandlerFromFile(filename : String) : MapHandler
#uploadToServer(connectTask : ConnectTask) : void
#getUploadStatus() : boolean
#setUploadStatus(s : boolean) : void
#setDayFileName(name : String) : void
#setMonthFileName(name : String) : void
-controller
Class
+onReceive(context : Context, intent : Intent) : void
-mScreenStateReceiver
LocalBinder
~getService() : MyService
CheckForegroundAppsTimerTask
+run() : void
MyService
+FOREGROUND_INTERVAL : long = 5 * 100
+UPLOAD_INTERVAL : long = 60 * 1 * 1000
-lastSavedMonth : int
-lastSavedDay : int
-mForegroundTimer : Timer = null
-mTimer : Timer = null
-currentRunningAppName : String = null
-iBinder : IBinder = new LocalBinder()
+SERVER_IP : String = "141.22.15.229"
+SERVER_PORT : int = 7358
-networkController : NetworkController
-serverMessage : String
-outString : PrintWriter
-controller : Controller
<<Property>> -mUploadTimer : ScheduleUploadsTimerTask = null
-mScreenStateReceiver : ForegroundBroadcastReceiver = null
-mNetworkBroadcastReceiver : NetworkBroadcastReceiver = null
+onBind(intent : Intent) : IBinder
+populateInstalledAppMap(mapHandler : MapHandler) : void
+onCreate() : void
+onDestroy() : void
ServerTransferTimerTask
+run() : void
Class3
#doInBackground(stringToSend : String ...) : Void
#onProgressUpdate(values : String ...) : void
#onPreExecute() : void
#onPostExecute(aVoid : Void) : void
Class2
+onReceive(context : Context, intent : Intent) : void
-mNetworkBroadcastReceiver
ScheduleUploadsTimerTask
<<Property>> -uploadTime : int
-dateHandler : DateHandler
-newDay : boolean
-newMonth : boolean
+ScheduleUploadsTimerTask(uploadTime : int)
+run() : void
-mUploadTimer

Figure 9: Controller-MyService-InterruptManager class diagram

It should be mentioned that inside **MyService** class, as it can also be seen in figure 9 there are some inner classes defined, namely: Class, Class2 and Class3. These correspond to:

1. Class: NetworkBroadcastReceiver

2. Class2: ForegroundBroadcastReceiver

3. Class3: ConnectTask

The reason for these inner classes is mainly the fact that they all need access to the **controller** object which is instantiated only once inside the **onCreate()** method of **MyService**.

An alternative would be to bind to the service in order to gain access to this object and split these classes, but this further complicates the code structure with no important trade-off gains (like better architecture and clarity).
In order to see a more appropriate and useful example of service binding please refer to the section about **UpdateFrequencyActivity** 20 . There, it makes much more sense to bind to the service, since it is definitely inappropriate to declare the **GUI** classes as inner classes of the controller path.

3.2.2 Api Dependency Workarounds

Due to several existing Android versions, some built in functions has either their names changed or they became redundant in the newer versions. In order to have an App that is executable for almost any version of Android, some switches and workarounds are implemented. These workarounds are needed for the following procedures which will be explained in detail throughout this section:

- Screen on detection
- Detection of running apps and the active app window
- App specific traffic stats

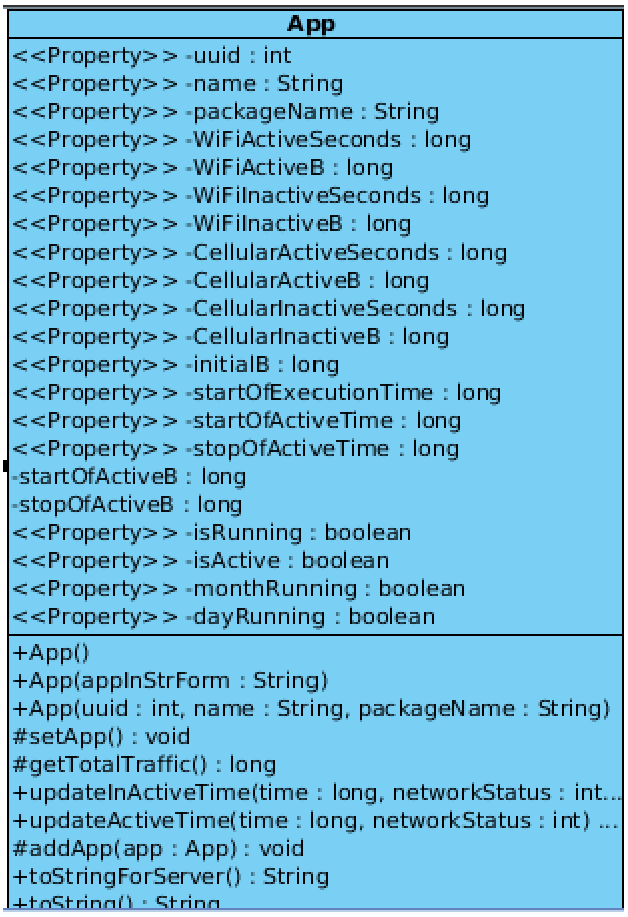
Determining the screen status when the app starts is required. This is necessary because there are specific procedures to be done dependent on the screen status which will be explained in the upcoming sections. Problem here is caused by a change in the function name that returns the screen on/off status. In the older APIs, to be precisice in the api's older than Kitkat, the class *PowerManager* has a method called *isScreenOn()* which returns a boolean indicating the screen on/off status as the method name suggests. Unfortunately, in the APIs that are Kitkat or newer, this method has been renamed to *isInteractive()*. Therefore an if block is implemented to perform the API version check and proceed accordingly. (Listing 53)

As can be imagined already, an app that isn't running shouldn't have any of its statistics changed. Even though this might sound trivial, which indeed it is for traffic usage because the traffic stats don't really change when an app isn't running; unfortunately it isn't that trivial for time measurement. The reason behind this is that there is not a single method provided by the Android libraries which returns the running time of an app. Therefore the time measurements had to be done manually which will be explained in detail later. In short, in order to measure the up time of an app, the program needs to mark the time stamp when an app starts to run, as well as the time stamp when it stops running. To do this, a list of running apps is required. Program iterates through the list of installed apps and stops the time measurements of the not running apps. Unfortunately again, a conflict is present between the new and old APIs. APIs that are Lollipop or newer, for security reasons, available permissions retrieve information about other running apps are taken away; therefore the method *getRunningTasks()* of the class *ApplicationManager* is deprecated. Fortunately, a workaround is available online. An open source project available on github, is able to retrieve a list of running apps as well as determining the active app at the time [3]. Even though this project claims to be working also for the older APIs, unfortunately it cannot determine the active app window, which is also required in order to correctly active and inactive usage statistics. Therefore again an if block is implemented to determine the API version and proceed accordingly. For newer APIs, open source project is used and for the older APIs program depends on the class *ApplicationManager* and its method *getRunningTasks().* (Listing 49 and 50)

The last version dependency to mention is unfortunately related to the class *TrafficStats.* Methods *getUidTxBytes()* and *getUidRxBytes* seem to be working fine with the newer APIs but unfortunately they sometimes malfunction in the older APIs. In order to avoid these problems, it was decided to do manually what these methods do. It turns out, these methods read the files *tcp_rcv* and *tcp_snd* under the directory */proc/uid_stat/<uid_of_app>/* and therefore a function is implemented to read the contents of the above mentioned files which seem to solve the issue in the older APIs. (Listing 39)

3.2.3 Measurement Techniques

The core task of this project was to implement a technique in order to get stats (3.1.2) of all the Apps of a client. A good way to start is to describe what an app is:



```
App
<<Property>> -uuid : int
<<Property>> -name : String
<<Property>> -packageName : String
<<Property>> -WiFiActiveSeconds : long
<<Property>> -WiFiActiveB : long
<<Property>> -WiFiInactiveSeconds : long
<<Property>> -WiFiInactiveB : long
<<Property>> -CellularActiveSeconds : long
<<Property>> -CellularActiveB : long
<<Property>> -CellularInactiveSeconds : long
<<Property>> -CellularInactiveB : long
<<Property>> -initialB : long
<<Property>> -startOfExecutionTime : long
<<Property>> -startOfActiveTime : long
<<Property>> -stopOfActiveTime : long
-startOfActiveB : long
-stopOfActiveB : long
<<Property>> -isRunning : boolean
<<Property>> -isActive : boolean
<<Property>> -monthRunning : boolean
<<Property>> -dayRunning : boolean
+App()
+App(appInStrForm : String)
+App(uuid : int, name : String, packageName : String)
#setApp() : void
#getTotalTraffic() : long
+updateInActiveTime(time : long, networkStatus : int...
+updateActiveTime(time : long, networkStatus : int) ...
#addApp(app : App) : void
+toStringForServer() : String
+toString() : String
```

Figure 10: App Class Diagram

An object of Class App, has attributes relates to all the information describing an app:

- app name
- package name - it is required due to the fact that it is unique
- uuid - app, process name
- time measurements is seconds - Since current measurements are done in millisec-

onds, and they have small values it makes since to store them in a long integer data type in order not to loose information

- data measurements in bytes - Again in order to save information. Apps usually can use small amounts of data, which would be lost if this information would be converted to float, or in Mb
- running flags - In order to get accurate calculations on the server side, it is important to distinguish between apps that run a full day for example, and apps that were installed in the middle of a day. Since calculations like Mb/Day/User are necessary to be made and since it is not allowed to save any information about the clients on the server (some identifiers), there needs to be an acknowledgement by the client that a certain app has run for the whole day, in this case.
- isActive flag - necessary to differentiate the active and inactive data calculations
- isRunning flag - necessary to distinguish whether to make calculations or not
- initial values - each time a new app is instantiated, it is important to store initial values, in order to be able to make all measurements

The App Class has also all the necessary constructors, getters, setters, toString methods, that the calling objects would require.

- default constructor is called when there is a necessity to create an empty app.

```
public App() {
    this.setApp();
    dayRunning = false;
    monthRunning = false;
}
```

Listing 35: App Default Constructor

- app constructor that generates an am from a passed String

```
public App(String appInStrForm) {
    this.setFromString(appInStrForm);
}
```

Listing 36: Revert from string App() constructor

- app constructor that creates an empty app, only setting the passed uuid, app-Name, package name

```
public App(int uuid, String name, String packageName) {
    /**/}
```

Listing 37: Empty App constructor with the passed uuid and names

- resetApp() sets all calculation attributes to 0

```
protected void resetApp() {/**/}
```

Listing 38: Resets the apps parameters to 0

- getTotalTraffic() calculates and returns the total traffic according to the apps uuid.

```
protected long getTotalTraffic() {/**/}
```

Listing 39: Method that calculates and returns the total traffic of an app

- updateInActiveTime() ads the new data regarding the inactive time of the app. The data is calculated according the the type of network adapter used.

```
public void updateInActiveTime(long time, int
    networkStatus){/**/}
```

Listing 40: Updates inactive time of the app

- updateActiveTime() ads the new data regarding the active time of the app. The data is calculated according the the type of network adapter used.

```
public void updateActiveTime(long time, int
    networkStatus){/**/}
```

Listing 41: Updates inactive time of the app

- addApp() adds the data contents from the passed app object to the current app object

```
protected void addApp(App app){/**/}
```

Listing 42: addApp() method

- toStringForServer() packs all the data attributes necessary to send to the server into string and returns it: dayRunning, monthRunning, name, all eight measurement attributes.

```
public String toStringForServer(){/**/}
```

Listing 43: toStringForServer() method

- toString() packs all the App attributes into a string and returns it. It is used when a backup is made.

```
public String toString(){/**/}
```

Listing 44: toString() method

- setFromString() populates all the app attributes from the passed string. It is used to generate an app when reading from the backup.

```
public void setFromString(String input){
```

Listing 45: setFromString() method

The measurements for each app are done depending on multiple factors:

- is the app running
- is the app not running
- is the app in the foreground
- is the app in the background
- what network does it use

At its core the algorithm of calculating the data is as follows:

1. When the app starts it reads all the installed app, and populates a hash map with the package name as key (since is unique) and as a value an App (10) object. If there is a necessity it reads also a backup file, and merges the data from that to the already existing hash map of apps.
2. A timer triggers a handler that checks which app is in the foreground. This allready means that all other apps are in the background. And the calculations are don for the inactive data (bytes, seconds)
3. As soon as the screen changes, the program check if the app in the foreground has changed. If so the calculations for the active data are calculated for the previous app which was in the foreground.
4. All calculations are done with regard to which network adapter is being used.
   - Wifi: 2
   - Cellular: 1

- None: 0

Another point necessary to mention is that the measurements kept on the client side are in bytes and seconds, using variables of type long. This is due to the fact that an app might change fast its status, and the changes in measurements would be almost impossible to distinguish if they would be converted into Mb and Hours respectively.

3.2.4 Timers

In order to schedule certain time dependent tasks, two timers are used: *ScheduleUploadsTimerTask* and *CheckForegroundAppsTimerTask*. Unfortunately there is no provided Android interrupt that can be triggered when the active window changes; therefore one of the mentioned timers is scheduled to poll for the active window every 500ms when the screen is on. The other timer is set to schedule uploads to the server and also to back up measurements on the phone. Both of these timers extend the Android class *TimerTask* and override its *run()* method.

Before going into specifics of overriden *run()* methods, it is worth mentioning how these timers are initialized and scheduled. First a *Timer* object is instantiated, which is followed by the instantiation of a *TimerTask* object either *ScheduleUploadsTimerTask* or *CheckForegroundAppsTimerTask*. By using the *scheduleAtFixedRate()* method of the *Timer* object, the overriden *run()* methods are scheduled to be executed (Listing 46 and 47).

```
mTimer = new Timer();
mUploadTimer = new ScheduleUploadsTimerTask(uploadTime);
mTimer.scheduleAtFixedRate(mUploadTimer, uploadTime*1000*60,
    UPLOAD_INTERVAL);
```

Listing 46: ScheduleUploadsTimerTask initilization

```
mForegroundTimer = new Timer();
mForegroundTimer.scheduleAtFixedRate(new
    CheckForegroundAppsTimerTask(), 0, FOREGROUND_INTERVAL);
```

Listing 47: CheckForegroundAppsTimerTask initilization

The *Timer* method *scheduleAtFixedRate()*, expects three arguments: a reference to a *TimerTask* object, a delay in milliseconds that defines how long the inital execution of *run()* method of the provided *TimerTask* object is going to be delayed and the period again in milliseconds defining the execution period of the *run()* method.

As mentioned earlier, *CheckForegroundAppsTimerTask* is responsible of polling for the active window every 500ms and trigger an interrupt when the active window changes. In order to do this, this method needs to be able to tell the foreground app. First as shown in Listing 48, some local variables are initialized.

```
String foregroundAppName = "";
String foregroundPackageName = "";
```

```
int foregroundUid = 0;
boolean existsForegroundApp = false;
ArrayList<String> runningTaskPackageNames = new ArrayList<
    String>();
```

Listing 48: CheckForegroundAppsTimerTask local variables

As explained in Section 3.2.6, a list of all running apps is required as well and detection of the foreground app together with the retrievel process of all running apps prone to API dependencies. Therefore an if block is set to perform the detection of active window together with retrieving all the running apps. To achieve this, for new APIs, methods *getRunningAppProcesses()* *getRunningForegroundApps()* of the class *ProcessManager* from the GitHub project [3] are used. These methods return a list of objects type *AndroidAppProcess* again from the GitHub project [3]. First length of the foreground apps list is checked and if there is a foreground app present, the boolean *existsForegroundApp* is adjusted accordingly. In case the list foreground app list has more than one app, the app at the index 0 refers to the active window. Once the active app is determined, the uid, package name and the app name are retrieved and saved to the local variables (Listing 49).

```
//new api
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
     List<AndroidAppProcess> runningApps = ProcessManager.
         getRunningAppProcesses();
     List<AndroidAppProcess> foregroundApps = ProcessManager.
         getRunningForegroundApps(getBaseContext());
     if (foregroundApps.size() > 0) {
          existsForegroundApp = true;
          foregroundAppName = foregroundApps.get(0).name;
          foregroundPackageName = foregroundApps.get(0).
              getPackageName();
          foregroundUid = foregroundApps.get(0).uid;
          for (AndroidAppProcess p: runningApps)
              runningTaskPackageNames.add(p.getPackageName());
     }
} else { ... } \\old api
```

Listing 49: Detection of foreground apps and the list of running apps for the new APIs

The else statement shown in Listing 49, is responsible of executing the same tasks for the old APIs. For this purpose *ActivityManager* is required. The method *getRunningTasks()* of the class *ActivityManager*, as the name suggest, returns a list of running apps. Since this method expects the maximum size for the list it returns, to avoid loss of information, a large enough value is passed to the method as can be seen in Listing 50. Once the list is retrieved, the uid, package name and the app name are stored saved to the corresponding local variables as it is done for the new APIs.

```
ActivityManager am = (ActivityManager) MyService.this.
    getSystemService(ACTIVITY_SERVICE);
List<ActivityManager.RunningTaskInfo> runningTasksOldApi = am.
    getRunningTasks(10000);
foregroundPackageName = runningTasksOldApi.get(0).topActivity.
    getPackageName();
try {
     PackageManager pm = MyService.this.getPackageManager();
     foregroundUid = pm.getPackageInfo(foregroundPackageName,
         0).applicationInfo.uid;
     foregroundAppName = pm.getPackageInfo(
         foregroundPackageName, 0).applicationInfo.loadLabel(pm)
         .toString();
     existsForegroundApp = true;
     for (ActivityManager.RunningTaskInfo process:
         runningTasksOldApi) runningTaskPackageNames.add(process
         .topActivity.getPackageName());
} catch (PackageManager.NameNotFoundException e) {
     //do nothing, existsForegroundApp boolean is already set
         to false }
```

Listing 50: Body of the else statement of Listing 49 - Old API

Once the foreground app and the list of running apps are retrieved, these information need to be passed to the *controller* so the necessary tasks can be performed. The list of running apps is passed to the *controller* by the help of a setter. Since the *controller* expects an interrupt when the foreground app changes, the new foreground app is compared to the previous detected foreground app and if they are not the same, the interrupt *appDetectedinterruptListener* is triggered. (Listing 51)

```
if (existsForegroundApp) {
     controller.setRunningTime(runningTaskPackageNames);
     if(!foregroundAppName.equals(currentRunningAppName)) {
          currentRunningAppName = foregroundAppName;
          if( (controller.interruptManager.
              appDetectedinterruptListener != null)){
               controller.interruptManager.
                   appDetectedinterruptListener.manageAppDetection
```

```
                    (foregroundAppName, foregroundPackageName,
                    foregroundUid);
            }
        }
```

Listing 51: Passing the foreground app and the list of running apps to the *controller*

One last thing to be mentioned about *CheckForegroundAppsTimerTask* is about the way it is only executed when the screen is on. This is implemented by the help of an interrupt created using a *BroadcastReceiver*. A sub class of *BroadcastReceiver* is created and the *onReceive()* method is overriden. This overriden method is triggered every time the screen state changes between on and off. By the help of an if statement, the screen state is checked and tasks are performed accordingly. If the screen goes off, the scheduled *CheckForegroundAppsTimerTask* is simply cancelled; and if the screen goes on, it is rescheduled. (Listing 52)

```
public class ForegroundBroadcastReceiver extends
    BroadcastReceiver {
     @Override
     public void onReceive(Context context, Intent intent) {
          if (intent.getAction().equals(Intent.ACTION_SCREEN_OFF
              )) {
               if(mForegroundTimer != null){
                    mForegroundTimer.cancel();
               }
          } else if (intent.getAction().equals(Intent.
              ACTION_SCREEN_ON)) {
               mForegroundTimer = new Timer();
               mForegroundTimer.scheduleAtFixedRate(new
                    CheckForegroundAppsTimerTask(), 0,
                    FOREGROUND_INTERVAL);
          }
     }
}
```

Listing 52: ForegroundBroadcastReceiver detecting screen status

ForegroundBroadcastReceiver is only able to schedule the *CheckForegroundAppsTimer-Task* when the screen state changes; which means, until the user turns the screen on or off, the program cannot perform any measurements, which can be considered to be a bug if the app gets installed while the screen is on. In order to avoid this, the program has to manually check the screen status during its initialization state. Unfortunately, as mentioned in Section 3.2.6, detecting screen status manually is prone to API dependencies. For this reason, an if block is implemented again, to check the API version and perform tasks accordingly. (Listing 53)

As can be seen in Listing 53, if the screen is on the *CheckForegroundAppsTimerTask* is scheduled to start right away with the delay being set to *0*; which makes sure the program doesn't lose any information.

```
boolean screenOn = false;
if (Build.VERSION.SDK_INT>= Build.VERSION_CODES.KITKAT_WATCH)
    {
     if (powerManager.isInteractive()) screenOn = true;
} else {
    if (powerManager.isScreenOn()) screenOn = true;
}
if (screenOn) {
     mForegroundTimer = new Timer();
     mForegroundTimer.scheduleAtFixedRate(new
         CheckForegroundAppsTimerTask(), 0, FOREGROUND_INTERVAL)
         ;
}
```

Listing 53: Checking the API status and detecting screen status accordingly

The other timer, *ScheduleUploadsTimerTask*, as mentioned earlier, is responsible of scheduling uploads to the server; but this is not its only task. This *TimerTask* is also responsible of triggering backing up and detecting begining of a new day and a new month. In order to achieve these tasks and to make sure measurements are backed up frequently enough without overloading the CPU, it is scheduled to execute every 15 minutes by default; which is also the other reason why upload interval has to be defined by multiples of 15 minutes. The reason behind detecting begining of a new day and begining of a new month is to make sure individual statistics of a given day and a given month are correctly calculated. The measurement data of an app is only included into the day specific statistics if the app is installed at the begining of that day; the same goes for the month. As can be imagined, this requires the program to detect installed apps. For this purpose the method *populateInstalledAppMap()* is defined. This method expects a reference to a *MapHandler* object, and initializes it's *map* with the installed apps. (Listing 54)

```
public void populateInstalledAppMap(MapHandler mapHandler) {
     PackageManager pm = MyService.this.getPackageManager();
     for (ApplicationInfo app: this.getBaseContext().
         getPackageManager().getInstalledApplications(0)) {
          String appName;
          PackageInfo packageinfo = null;
          try {
               packageinfo = pm.getPackageInfo(app.packageName,
                   0);
          } catch (PackageManager.NameNotFoundException e) {
               continue;
          }
```

```
            appName = packageinfo.applicationInfo.loadLabel(pm).
                toString();
            mapHandler.put(app.packageName, new App(app.uid,
                appName, app.packageName));
        }
}
```

Listing 54: Detection of installed apps and adapting the *MapHandler* accordingly

The reason why package names are decided to be defined as keys of the *map* object (Listing 54) is because they stay to be unique between apps regardless of a reboot of the phone or an update of an app. Uid or name of an app are bad choices because an app name can change by an update of the app, and the uid is prone to change when the phone reboots.

```
if (this.dateHandler.getCurrentDayInt() != lastSavedDay) {
    MapHandler mapHandler = new MapHandler();
    populateInstalledAppMap(mapHandler);
    controller.getMapHandler().mergeMap(mapHandler.getMap(),
        false);
    controller.setDayFileName(this.dateHandler.
        getCurrentDayMonth());
    lastSavedDay = this.dateHandler.getCurrentDayInt();
    this.newDay = true;
    if (this.dateHandler.getCurrentMonthInt() !=
        lastSavedMonth) {
        controller.setMonthFileName(this.dateHandler.
            getCurrentMonth());
        lastSavedMonth = this.dateHandler.getCurrentMonthInt()
            ;
        this.newMonth = true;
        controller.getMapHandler().setDayFlag(true);
    } else {
        controller.getMapHandler().setMonthFlag(true);
    }
}
```

Listing 55: Detection of new day and new month including the trigger for back ups

First thing the *ScheduleUploadsTimerTask* does, is to check if it's a new day (Listing 55). A variable containing the saved day is compared to the current day received from the method *getCurrentDayInt()* of an object type *DateHandler*. The method *getCurrentDayInt()* simply returns the corresponding integer to the day of the current date.

For example, for 01.05.2016, this method returns 1 and for 15.06.2016, it returns 15. Every time it is a new day, the current day is saved into a variable, so the comparison can be executed again for the next day. A new day also indicates that the *map* needs to be updated according to the installed apps; so if an app is uninstalled, the key should be removed from the *map* and if there is a new app, it should be added. The method *mergeMap()* of the class *MapHandler* is able to perform the mentioned merge between installed apps and the *map* object. This method expects two arguments: a reference to an object type *map* containing the installed apps, and another argument as a boolean defining if it is going to be a deep merge meaning a merge between each *app* object inside the *maps* or just a high level merge of *map* keys. Back up file names are also adapted to a change in the day. Also every time it is detected to be a new day, it is also checked if it is a new month. Similar to new day detection, this is also done by the help of the object type *DateHandler*. Everything that applies to a new day, applies to a new month as well; meaning back up file name as well as the individual flags for apps indicating validity of measurements needs to be set accordingly. Once these checks are made and everything is sorted, measurements are backed up to files on the phone by calling the *controller* method *backupData()*. (Listing 55)

```
if (this.dateHandler.hasItBeenMinutes(this.uploadTime) && !
    controller.getUploadStatus()) {
     controller.uploadToServer(new ConnectTask());
} else {
     if (controller.getUploadStatus()) {
         controller.setUploadStatus(false);
          this.dateHandler.setPreviousTimeStamp();
          if (this.newDay) {
               if (this.newMonth) {
                    controller.getMapHandler().setMonthFlag(false)
                        ;
                    this.newMonth = false;
               }
               controller.getMapHandler().setDayFlag(false);
               this.newDay = false;
         }
     }
}
```

Listing 56: Triggering the upload process and the acknowledgement protocol between *controller* and *ScheduleUploadsTimerTask*

Next thing to be done is to determine if it is time for an upload (Listing 56). The method *hasItBeenMinutes()* of the class *DateHandler*, is able to compare a previously set time stamp to the current one and determine if the difference is equal or greater to the one passed as an argument. Every time a successful upload takes place, the time stamp stored in the *DateHandler* object is updated accordingly so the upload period starts over. Also the new day and new month flags sent to the server need to be set to false after each successful upload to make sure that this occurs only once at the beginning of a new day or a new month. This is achieved by an acknowledgement protocol between the *controller* and the *ScheduleUploadsTimerTask*. After a successful upload, the corresponding flag is set by the *controller*; this flag is checked by the *ScheduleUploadsTimerTask* and if it is false, upload is repeated. In the case that the flag is true, which means the previously attempted upload was carried successfully; *ScheduleUploadsTimerTask* acknowledges by setting the flag back to false, and running the corresponding tasks. (Listing 56)

3.2.5 File Handling

Since there is a necessity to calculate, save, and store app data readouts in a fashionable manner, each app has it's own object of Class App (10).

An object of Class App, has attributes relates to all the information describing an app:

- app name
- package name - it is required due to the fact that it is unique
- uuid - app, process name
- time measurements is seconds - Since current measurements are done in milliseconds, and they have small values it makes since to store them in a long integer data type in order not to loose information
- data measurements in bytes - Again in order to save information. Apps usually can use small amounts of data, which would be lost if this information would be converted to float, or in Mb
- running flags - In order to get accurate calculations on the server side, it is important to distinguish between apps that run a full day for example, and apps that were installed in the middle of a day. Since calculations like Mb/Day/User are necessary to be made and since it is not allowed to save any information about the clients on the server (some identifiers), there needs to be an acknowledgement by the client that a certain app has run for the whole day, in this case.
- isActive flag - necessary to differentiate the active and inactive data calculations
- isRunning flag - necessary to distinguish whether to make calculations or not
- initial values - each time a new app is instantiated, it is important to store initial values, in order to be able to make all measurements

Depending on the activity and settings the of app, files are storen the device with a certain frequency, for backup and GUI requirements considerations. For this purpose there is the DataHandler Class. It has a single attribute, which is the the absolute path of the application within the device storage. The backup is a file that is written and read, when unexpected events occur, like shut down, app crash. On the other side there are daily and monthly files for a whole year.

- writeToFile() - write the passed string to a file with the passed file name

```
public void writeToFile(String fileName, String
    content) {/**/}
```

Listing 57: Write to data to file

- readFromFile() - read the content of the file with the name passed to the function

```
public String readFromFile(String fileName) {/**/}
```

Listing 58: Read date from file and return its contents

- clearFile() - clear the file with the name passed to the function

```
public boolean clearFile(Context context, String
    fileName) {/**/}
```

Listing 59: Clear File Method

The clearFile() method also returns a boolean flag, marking the success of the file removal.

3.2.6 Api Dependency Workarounds

Due to several existing Android versions, some built in functions has either their names changed or they became redundant in the newer versions. In order to have an App that is executable for almost any version of Android, some switches and workarounds are implemented. These workarounds are needed for the following procedures which will be explained in detail throughout this section:

- Screen on detection
- Detection of running apps and the active app window
- App specific traffic stats

Determining the screen status when the app starts is required. This is necessary because there are specific procedures to be done dependent on the screen status which will be explained in the upcoming sections. Problem here is caused by a change in the function name that returns the screen on/off status. In the older APIs, to be precisice in the api's older than Kitkat, the class *PowerManager* has a method called *isScreenOn()* which returns a boolean indicating the screen on/off status as the method name suggests. Unfortunately, in the APIs that are Kitkat or newer, this method has been renamed to *isInteractive()*. Therefore an if block is implemented to perform the API version check and proceed accordingly. (Listing 53)

As can be imagined already, an app that isn't running shouldn't have any of its statistics changed. Even though this might sound trivial, which indeed it is for traffic usage because the traffic stats don't really change when an app isn't running; unfortunately it isn't that trivial for time measurement. The reason behind this is that there is not a single method provided by the Android libraries which returns the running time of an app. Therefore the time measurements had to be done manually which will be explained in detail later. In short, in order to measure the up time of an app, the program needs to mark the time stamp when an app starts to run, as well as the time stamp when it stops running. To do this, a list of running apps is required. Program iterates through the list of installed apps and stops the time measurements of the not running apps. Unfortunately again, a conflict is present between the new and old APIs. APIs that are Lollipop or newer, for security reasons, available permissions retrieve information about other running apps are taken away; therefore the method *getRunningTasks()* of the class *ApplicationManager* is deprecated. Fortunately, a workaround is available online. An open source project available on github, is able to retrieve a list of running apps as well as determining the active app at the time [3]. Even though this project claims to be working also for the older APIs, unfortunately it cannot determine the active app window, which is also required in order to correctly active and inactive usage statistics. Therefore again an if block is implemented to determine the API version and proceed accordingly. For newer APIs, open source project is used and for the older APIs program depends on the class *ApplicationManager* and its method *getRunningTasks()*. (Listing 49 and 50)

The last version dependency to mention is unfortunately related to the class *TrafficStats*. Methods *getUidTxBytes()* and *getUidRxBytes* seem to be working fine with the newer APIs but unfortunately they sometimes malfunction in the older APIs. In order to avoid these problems, it was decided to do manually what these methods do. It turns out, these methods read the files *tcp_rcv* and *tcp_snd* under the directory */proc/uid_stat/<uid_of_app>/* and therefore a function is implemented to read the contents of the above mentioned files which seem to solve the issue in the older APIs. (Listing 39)

3.2.7 Socket Programming

In order to send and receive data from the server, the application makes use of an **AsyncTask**. This is a type of background thread usefull for periodic short term activities. It is also very well suited for updating the user interface if need be. For example, it is possible to inform the user via a popup if the sending was successful or not. Basically, the main feature of an Asynk Task is that it does all the heavy operation in the background, not on the main **UI Thread** and it has the ability to bound with the UI thread and update the GUI if this is necessary.

For a better understanding, see figure 11 below:

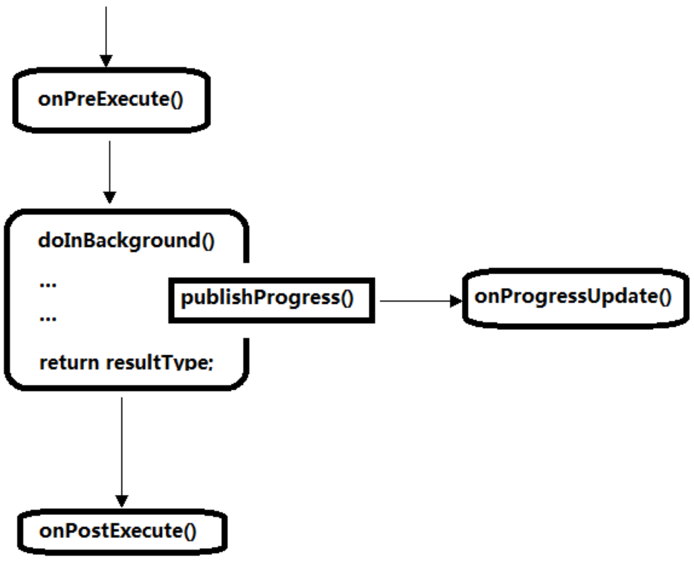


Figure 11: AsynkTask Structure

As it can be seen, there is a method **onPreExecute()**, where small instructions can be executed before the heavy work. Then **doInBackground()** is executed, where the main work is done. Afterwords, the app may execute **onPostExecute()** for small instructions after the heavy work from **doInBackground()**. It can also be seen that the AsyncTask interacts with the UI thread using **onProgressUpdate()** while it is doing the background work. Currently this server connection AsynkTask does not update the user interface. An example of updating the user interface is the AsynkTask that opens the list of all apps (**LoadApplications** class inside **AllAppsActivity** class). There, the user sees a rolling circle while waiting for the list of installed applications to load entirely (see figure 12)

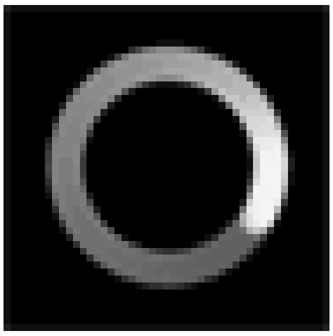


Figure 12: AsynkTask updates UI Thread

This particular server connection AsynkTask, named **ConnectTask** is executed periodically by a timer and the period can be changed by the user using the main activity menu (see figure 17)

The **doInBackground()** method simply issues an interrupt after each execution, informing the controller whether the transfer of data was successful or not.

If transfer was successfull:

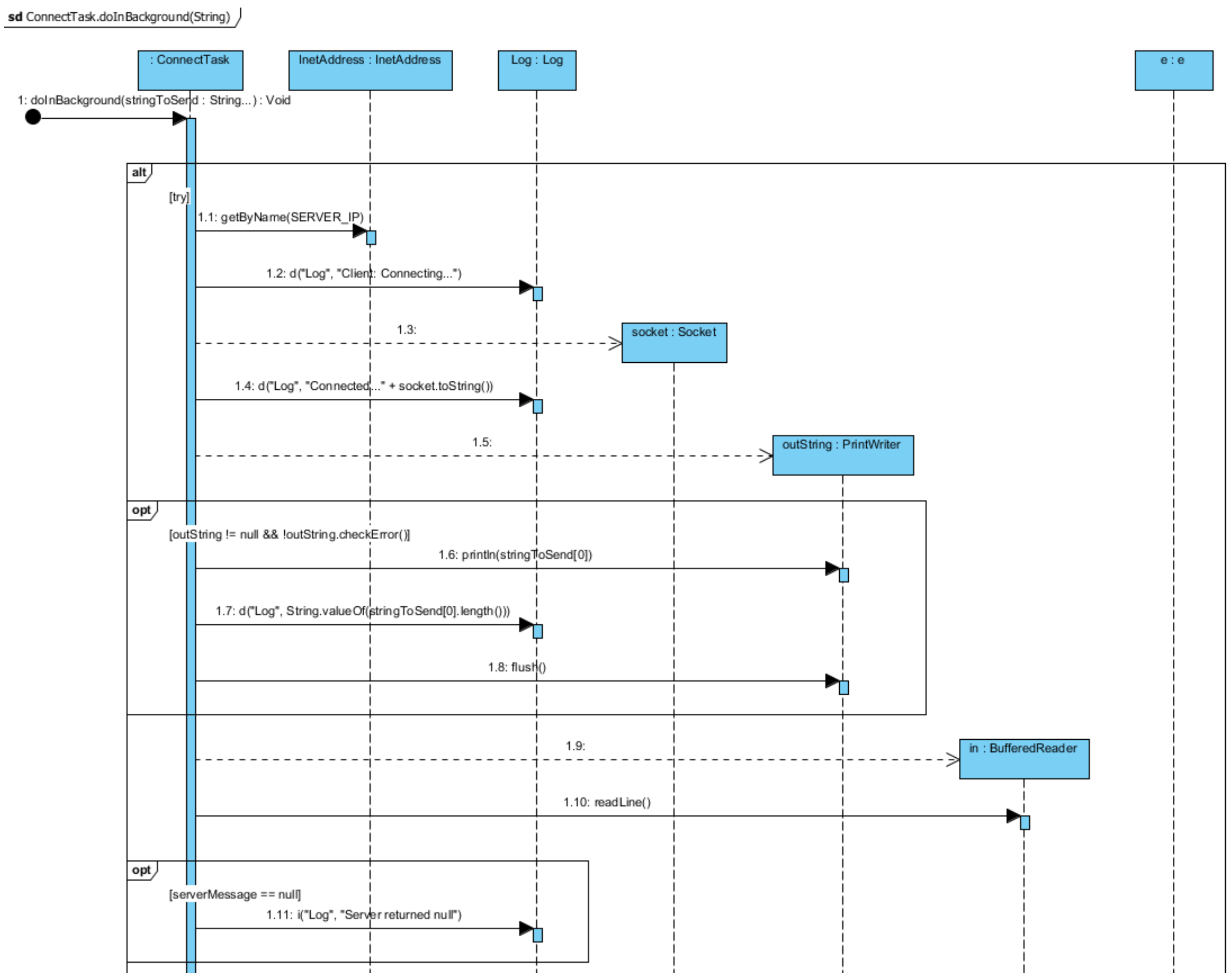
```
controller.interruptManager.
    serverTransferFailedInterruptListener.
    managePendingServerTransfers(true);
```

If transfer was not successful:

```
controller.interruptManager.
    serverTransferFailedInterruptListener.
    managePendingServerTransfers(false);
```

If server connection was not successfull, the controller registers this information and attempts a retransmission at a later time using a periodic backround thread.

For a better understanding of the entire server transmission process, here is a sequence diagram of the **doInBackround()** method of the **ConnectTask** class:


sd ConnectTask.doInBackground(String)
: ConnectTask
InetAddress : InetAddress
Log : Log
e : e
1: doInBackground(stringToSend : String...) : Void
alt
[try]
1.1: getByName(SERVER_IP)
1.2: d("Log", "Client: Connecting...")
1.3:
socket : Socket
1.4: d("Log", "Connected..." + socket.toString())
1.5:
outString : PrintWriter
opt
[outString != null && !outString.checkError()]
1.6: println(stringToSend[0])
1.7: d("Log", String.valueOf(stringToSend[0].length()))
1.8: flush()
1.9:
in : BufferedReader
1.10: readLine()
opt
[serverMessage == null]
1.11: i("Log", "Server returned null")

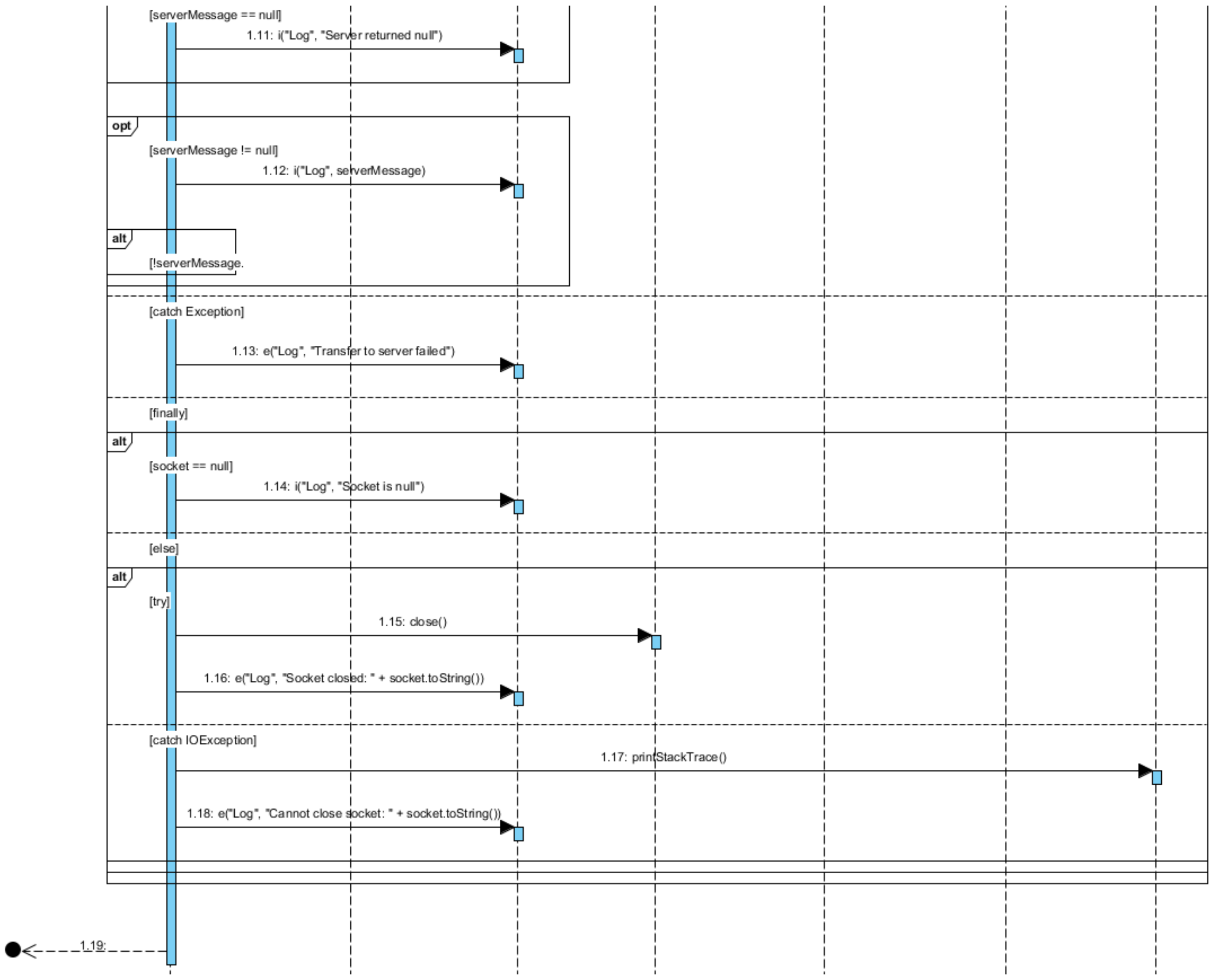

[serverMessage == null]
1.11: i("Log", "Server returned null")
opt
[serverMessage != null]
1.12: i("Log", serverMessage)
alt
[!serverMessage.
[catch Exception]
1.13: e("Log", "Transfer to server failed")
[finally]
alt
[socket == null]
1.14: i("Log", "Socket is null")
[else]
alt
[try]
1.15: close()
1.16: e("Log", "Socket closed: " + socket.toString())
[catch IOException]
1.17: printStackTrace()
1.18: e("Log", "Cannot close socket: " + socket.toString())
1.19:

Figure 13: Sequence Diagram of doInBackground() inside ConnectTask class

3.2.8 App GUI Part 1

In this section some of the graphical user interface features will be presented. Along with this, there will be an explanation of the options and functionalities that this application presents to the user.

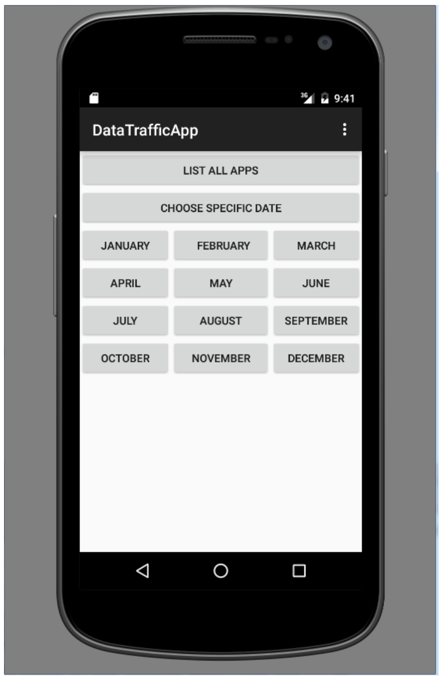


Figure 14: Close-up of MainActivity

Figure 14 shows a close up of the main activity.
As it can be observed, the user has the possibility to list all the apps installed on the device by clicking the **LIST ALL APPS** button. This will open a page with a list of all apps and by clicking an app item from the list, the user will be taken to another page which contains all the traffic information associated with that particular app.

By clicking any of the 12 buttons below the **CHOOSE SPECIFIC DATE** button, the user will open a page with information regarding his own data consumption for that particular month only.

By clicking **CHOOSE SPECIFIC DATE** button, a **Date Picker** dialog will be opened as the one seen in figure 15, where the user can pick a certain day for which he may want to view his data consumption.

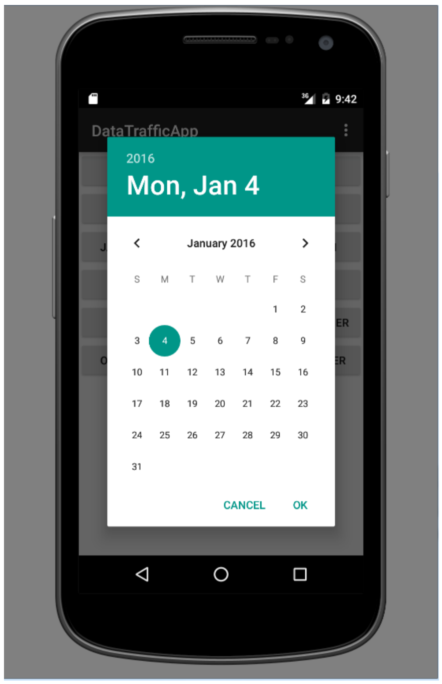


Figure 15: Close-up of Date Picker

It should be mentioned that the user is able to see his data consumption for 356 days and not more. Therefore, once an entire year passes, the data will start to be overwritten.

The main activity also has a menu button. Once clicked, the user is presented with a drop down list as the one seen in figure 16.

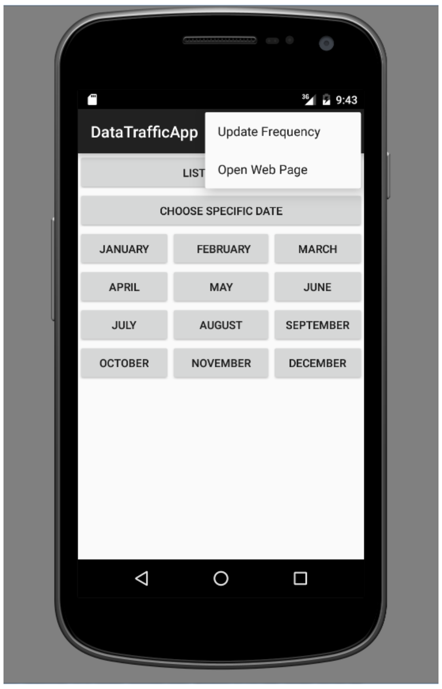


Figure 16: Close-up of main activity menu

Consequently, the user can choose the **Update Frequency** option or the **Open Web Page** option.

If **Update Frequency** is selected, a new page is opened, which can be seen in figure 17.

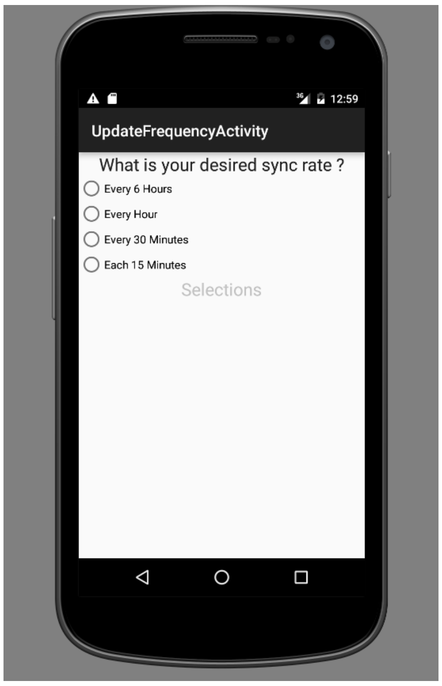
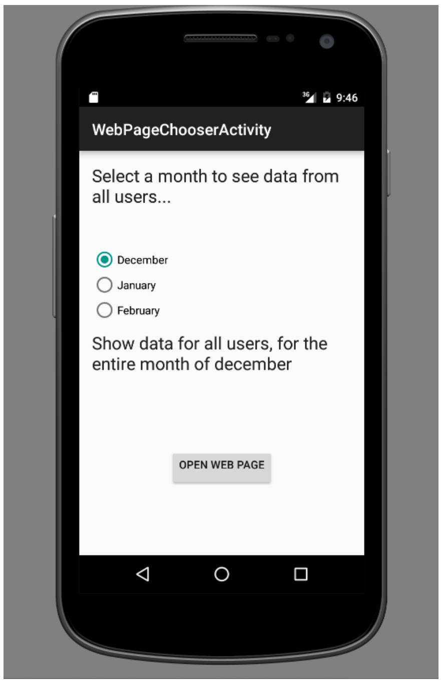


Figure 17: Close-up of Update Frequency Page

Here the user selects the time interval in which the application iterates its main background activities, namely: the data transmission to the server and the internal file system updates. Note that these intervals may be subject to change.

By selecting the **Open Web Page** option from the menu, the user lands on another page (see figure 18) where he must select the month for which he wants to view the cumulative data, aka traffic for all the users using this app on their devices.



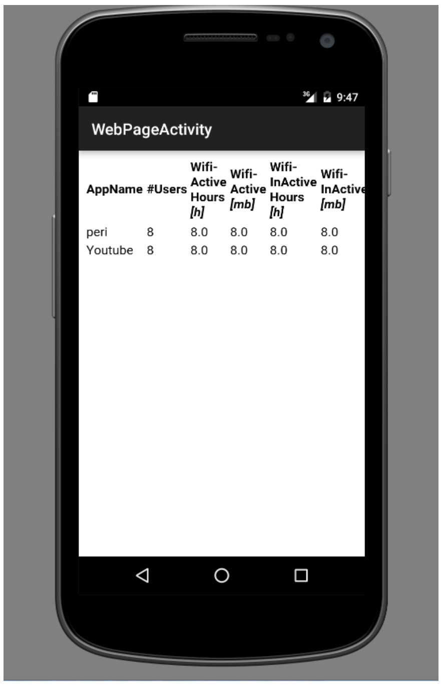
Select a month to see data from all users...

- [✓ December]
- January
- February

Show data for all users, for the entire month of december

Figure 18: Close-up of Web Page Chooser Page

After selecting the desired month, the user must click **OPEN WEB PAGE** in order to open the phone browser, which will then show the *http* page (see figure 19) if internet connection is available:



AppName	#Users	Wifi-Active Hours *[h]*	Wifi-Active *[mb]*	Wifi-InActive Hours *[h]*	Wifi-InActive *[mb]*
peri	8	8.0	8.0	8.0	8.0
Youtube	8	8.0	8.0	8.0	8.0

Figure 19: Close-up of Web Page Activity

This web page can be scrolled to the left and right. It can also be viewed in landscape mode if the phone is rotated 90 degrees.

Some important GUI related code considerations:

UpdateFrequency class makes use of service binding in order to gain access to the **controller** object which is instantiated only once inside the **onCreate()** method of **MyService**

This is done in the following way:

- Inside **onCreate()** of **UpdateFrequencyActivity**, an intent is used along with **bindService()** method:

```
Intent intent = new Intent(this, MyService.class);\\
bindService(intent, serviceConnection, Context.
    BIND_AUTO_CREATE);
```

- Inside **onDestroy()** of **UpdateFrequencyActivity**, **unbindService(serviceConnection)** is called:

```
@Override
    protected void onDestroy() {
        unbindService(serviceConnection);
        super.onDestroy();
    }
```

- A **ServiceConnection** object is instantiated inside **UpdateFrequencyActivity**

```
private ServiceConnection serviceConnection = new
    ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name,
            IBinder service) {
            MyService.LocalBinder binder = (MyService.
                LocalBinder) service;
            myService = binder.getService();
            isBound = true;
        }

        @Override
        public void onServiceDisconnected(ComponentName
            name) {
            isBound = false;
        }
    };
```

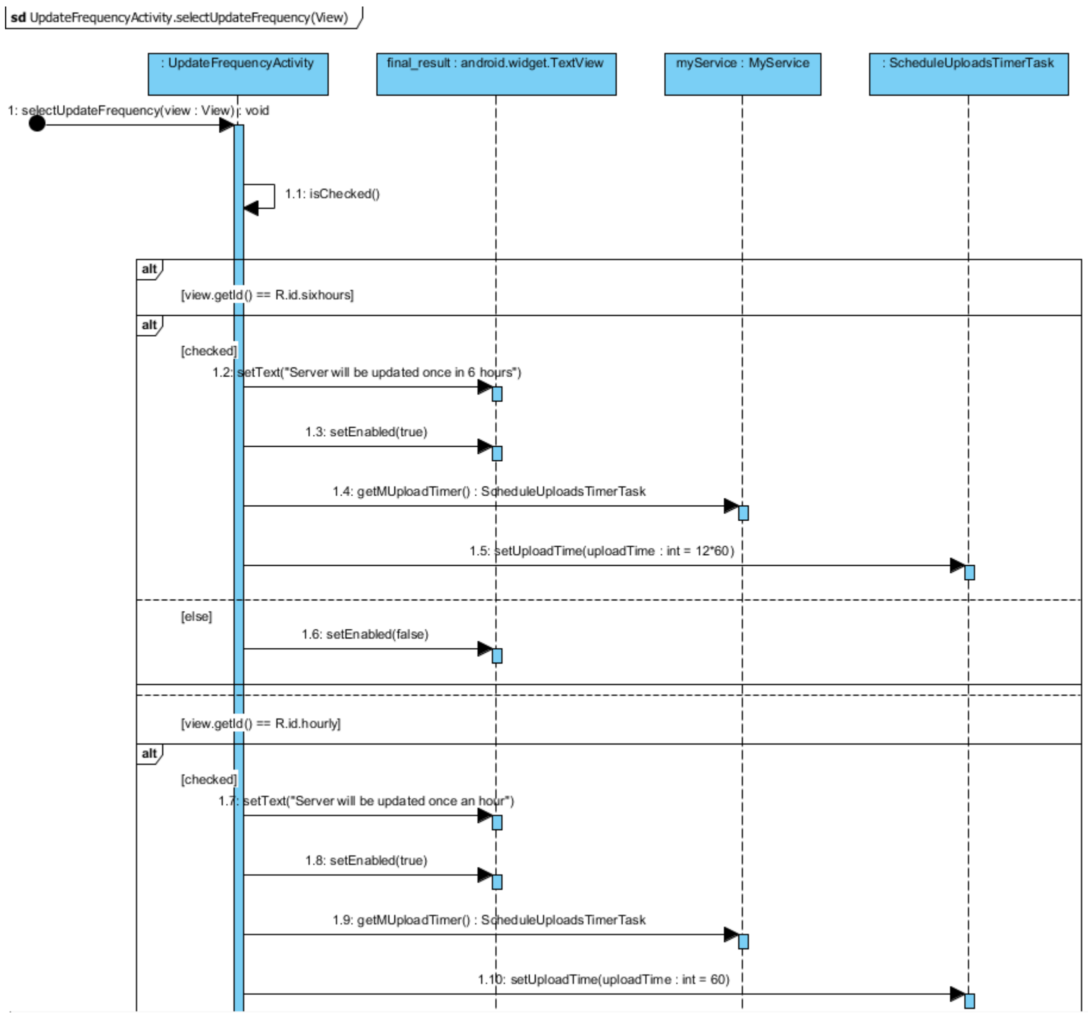
- Inside **MyService**, an inner **LocalBinder** class is implemented:

```
 public class LocalBinder extends Binder {
        MyService getService(){
            return MyService.this;
        }
    }
```

- The **LocalBinder** object is instantiated and returned in **onBind()** method inside **MyService**:

```
private final IBinder iBinder = new LocalBinder();


    public class LocalBinder extends Binder {
        MyService getService(){
            return MyService.this;
        }
    }
```

Having said all this, here is a sequence diagram of the **selectUpdateFrequency(View v)** method, which illustrates how exactly is the GUI connected to the controller part of the application:


sd UpdateFrequencyActivity.selectUpdateFrequency(View)
: UpdateFrequencyActivity
final_result : android.widget.TextView
myService : MyService
: ScheduleUploadsTimerTask
1: selectUpdateFrequency(view : View) : void
1.1: isChecked()
alt
[view.getId() == R.id.sixhours]
alt
[checked]
1.2: setText("Server will be updated once in 6 hours")
1.3: setEnabled(true)
1.4: getMUploadTimer() : ScheduleUploadsTimerTask
1.5: setUploadTime(uploadTime : int = 12*60)
[else]
1.6: setEnabled(false)
[view.getId() == R.id.hourly]
alt
[checked]
1.7: setText("Server will be updated once an hour")
1.8: setEnabled(true)
1.9: getMUploadTimer() : ScheduleUploadsTimerTask
1.10: setUploadTime(uploadTime : int = 60)

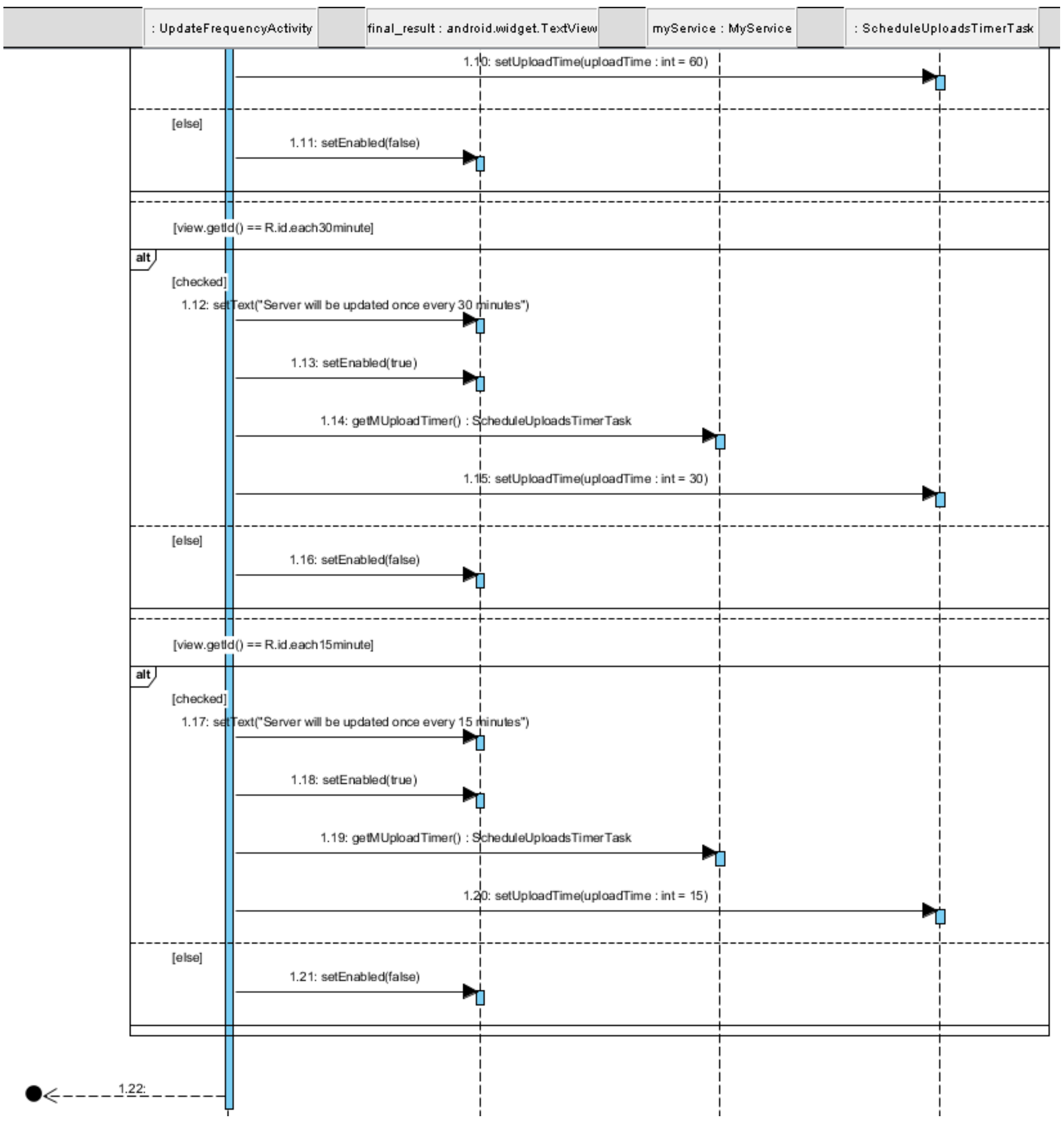

: UpdateFrequencyActivity
final_result : android.widget.TextView
myService : MyService
: ScheduleUploadsTimerTask
1.10: setUploadTime(uploadTime : int = 60)
[else]
1.11: setEnabled(false)
[view.getId() == R.id.each30minute]
alt
[checked]
1.12: setText("Server will be updated once every 30 minutes")
1.13: setEnabled(true)
1.14: getMUploadTimer() : ScheduleUploadsTimerTask
1.15: setUploadTime(uploadTime : int = 30)
[else]
1.16: setEnabled(false)
[view.getId() == R.id.each15minute]
alt
[checked]
1.17: setText("Server will be updated once every 15 minutes")
1.18: setEnabled(true)
1.19: getMUploadTimer() : ScheduleUploadsTimerTask
1.20: setUploadTime(uploadTime : int = 15)
[else]
1.21: setEnabled(false)
1.22:

Figure 20: selectUpdateFrequency(View v) Sequence Diagram

Another useful thing to mention is the way in which one can pass data between 2 activities. For instance, in order to open a web page on the phone, the user must select first the month he would like to view. Once this is selected, a certain url string is generated. This string needs to be passed to the next activity in order to open the web page. Here is the way this is done inside the app:

- The url string is set depending on the option selected inside **WebPageChooser-Activity**
- Inside **WebPageChooserActivity** an intent is instantiated and the url string is passed using the **putExtra()** method:

```java
Intent intent = new Intent(getBaseContext(),
    WebPageActivity.class);
        intent.putExtra("url_data", url);
        startActivity(intent);
```

- The url is retreived inside the next opened activity, namely: **WebPageActivity** using **getExtras()** method:

```java
String url=null;

        Bundle extras = getIntent().getExtras();
        if (extras != null) {
            url = extras.getString("url_data");
        }
```

3.2.9 App GUI Part 2

The traffic measurement application is designed to inform the user about the data statistics of his/her phone, not only to gather statistics on the server side to compare different users. In order to create the friendly environment for the user, a GUI was developed.

The GUI of the application has several priorities for the following project:
1. To display the data traffic of the device per month.
2. To inform the user regarding the traffic of the device where the application is installed.
3. The possibility to open the web page of the server, where the data can be compared with other users.
4. The detailed information per application, regarding the: active/inactive time and data traffic and also wifi/cellular networks.

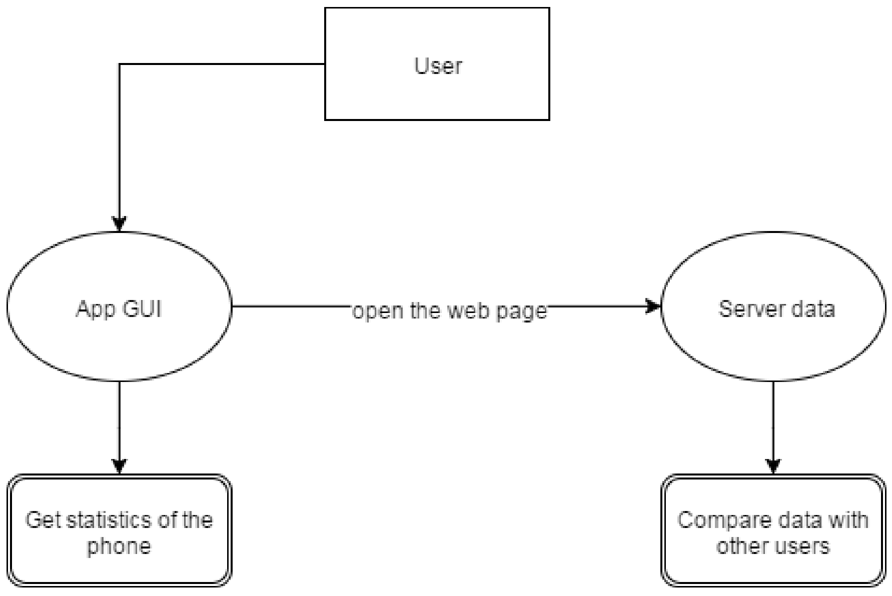


Figure 21: GUI vs server

As it can be seen in figure 21 the user has two possibilities to use the GUI to check the phone statistics or to compare the other users statistics in order to get a clear picture about the usage of certain programs by others.

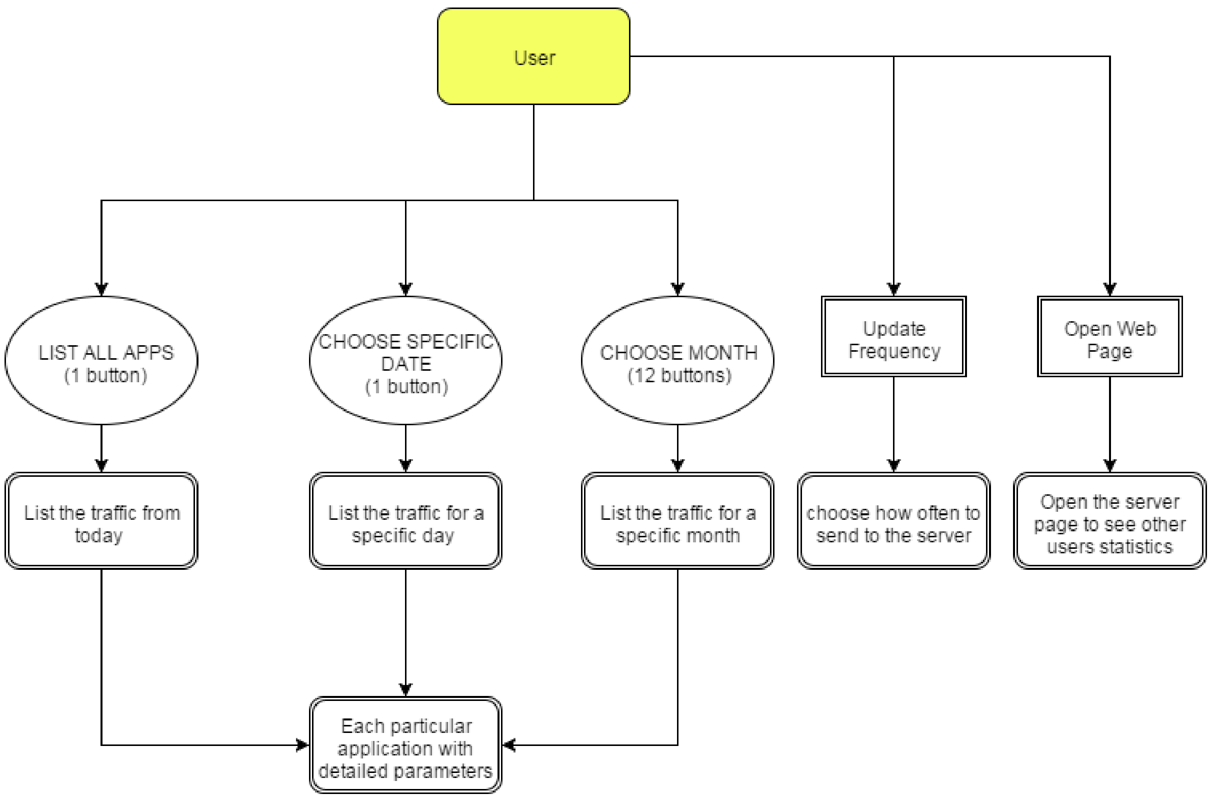

User
LIST ALL APPS (1 button)
CHOOSE SPECIFIC DATE (1 button)
CHOOSE MONTH (12 buttons)
Update Frequency
Open Web Page
List the traffic from today
List the traffic for a specific day
List the traffic for a specific month
choose how often to send to the server
Open the server page to see other users statistics
Each particular application with detailed parameters

Figure 22: GUI flow chart

The specific functions of the GUI can be seen in figure 22, it shows the 5 main functions of the GUI(from which 2 specific buttons and 12 months buttons and 2 features).

Buttons:
1. List All Apps Button.
2. Choose Specific Date Button.
3. Choose Month Button (12 buttons).

Features:
1. Update Frequency select.
2. Open Web Page select.

Every button refers to a specific action which is described in figure 22 .

As it can be seen in figure 23, the main functions can be selected from the first page of the application.

The two features can be selected in the upper right corner by pressing the three dots.

On the other side the three buttons always open one list from figure 24, the difference is the information that it actually shows, the "List All Apps" shows the current day traffic , the "Choose Specific Date" displays the specific day traffic and the third choice is to select the month and to display the traffic per month.

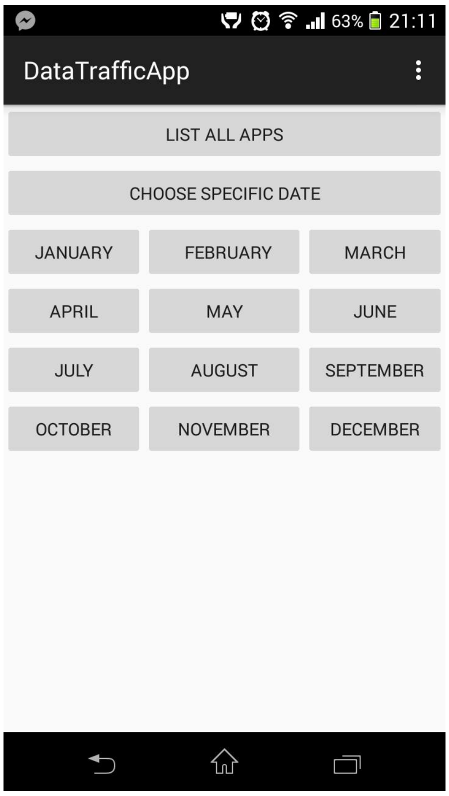


Figure 23: First Initial Page

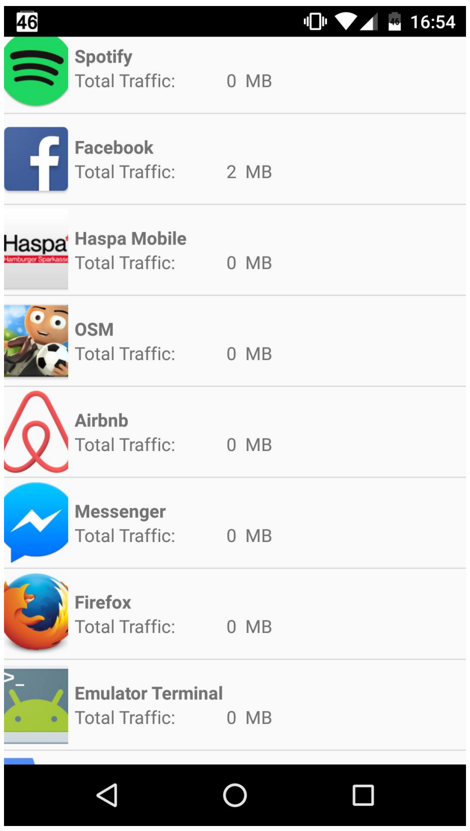
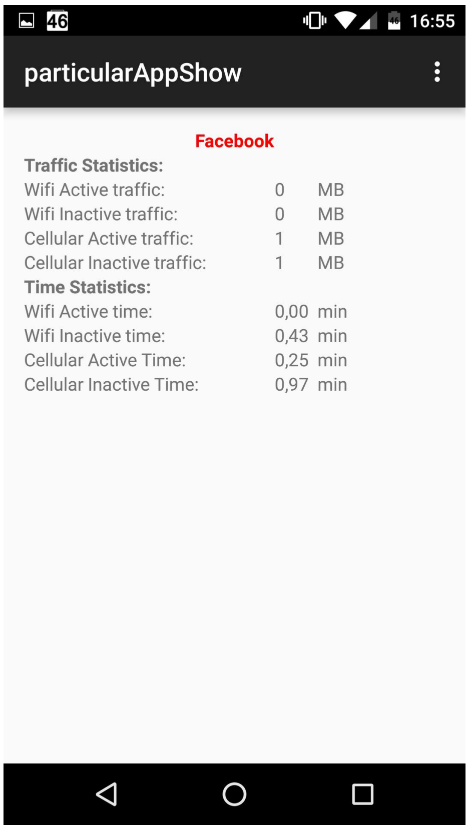

Spotify
Total Traffic: 0 MB
Facebook
Total Traffic: 2 MB
Haspa Mobile
Total Traffic: 0 MB
OSM
Total Traffic: 0 MB
Airbnb
Total Traffic: 0 MB
Messenger
Total Traffic: 0 MB
Firefox
Total Traffic: 0 MB
Emulator Terminal
Total Traffic: 0 MB

Figure 24: List of Application

The list of applications from figure 24 does not include the specific details about statistics for every individual application. Therefore by clicking on any particular application, a more detailed view will open which can be seen in figure 25 .

In figure 25 every application, in this case Facebook has 8 attributes which describe the traffic statistics and time statistics.



Facebook

Traffic Statistics:		
Wifi Active traffic:	0	MB
Wifi Inactive traffic:	0	MB
Cellular Active traffic:	1	MB
Cellular Inactive traffic:	1	MB
Time Statistics:		
Wifi Active time:	0,00	min
Wifi Inactive time:	0,43	min
Cellular Active Time:	0,25	min
Cellular Inactive Time:	0,97	min

Figure 25: Particular application statistics

Depending on the list which is opened, per month, a specific day or the actual day, the attributes of every particular application are read from the corresponding files in the phone, therefore displaying the wanted information.
The attributes per application are:
Traffic Statistics(in MB):
1. Wifi Active traffic - the wifi used by the application which is on foreground (opened).
2. Wifi Inacite traffic - traffic per application in the background.
3. Cellular Active traffic
4. Cellular Inactive traffic
Time Statistics(in minutes):
5. Wifi Active time
6. Wifi Inactive time
7. Cellular Active time
8. Cellular Inactive time

This is basically the last layer of the GUI and the last information that is provided to the user.

The actual design of the GUI in java can be seen in figure 26, which shows the class diagram of the code but does not include the xml files too.

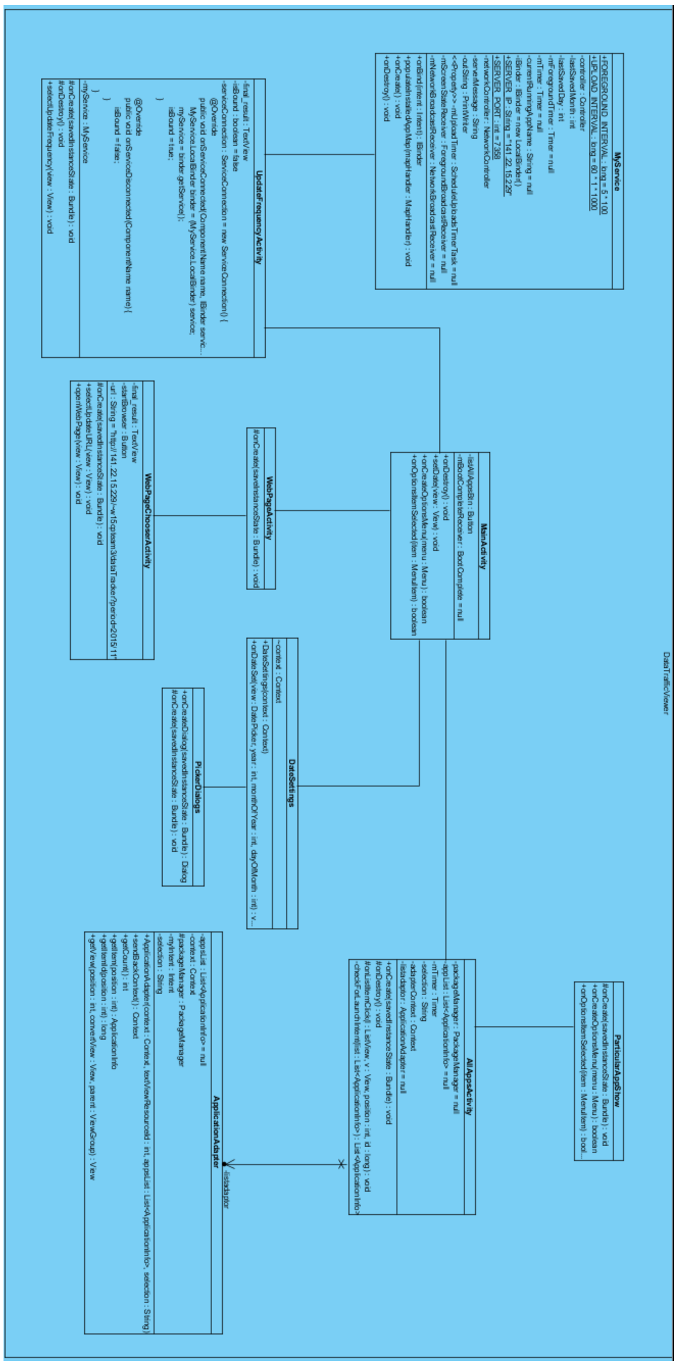


Figure 26: GUI class diagram

The class diagram has 10 classes and also 6 xml files which are not displayed in the class diagram due to very big size. The xml files are used to design the gui interface and the actual calculations and processes are done in the java classes.

The class "MainActivity" is the main class of the GUI where most of the buttons are initialized or at least have a connection with the other parts/pages of the GUI.

Another important class is "AllAppsActivity" which is responsible mainly for creating the list view of all installed applications and with help of the class "Application-Adapter" displays the total traffic per application in the main view, which can be seen in figure 24 . See listing 60 for the code for total traffic calculation in "Application-Adapter" class, reading the values from files.

```
float calculateTotalTraffic = ((mapHandler.get(data.
    packageName).getWiFiActiveB() + mapHandler.get(data.
    packageName).getWiFiInactiveB() +
mapHandler.get(data.packageName).getCellularActiveB() +
    mapHandler.get(data.packageName).getCellularInactiveB())
    /1024.0f)/1024.0f;

totalTraffic.setText(String.valueOf(String.format("%2f",
    calculateTotalTraffic)));
```

Listing 60: Calculate TotalTraffic in class ApplicationAdapter

After getting the list of application, the class "ParticularAppShow" is responsible for displaying the information for all attributes of every application when any application is clicked from the list, can be seen in figure 25 . This class gets the information from the class "AllAppsActivity" regarding the attributes, which gets the information from the mapHandler, the class that is responsible for the file storage. In the end the information displayed in the GUI is read from the corresponding files stored in the phone.

In order to move from one activity to another activity, from "AllAppsActivity" class to "ParticularAppShow" class in order to display the attributes of each particular application, the methods "putExtra()" and "getExtra()" were used, which can put and get the intent data from two activities.

See listing 61 for the code in class "AllAppsActivity", which reads the data from the files, in this case the example is for variable wifiActiveB (wifi for the application running in the foreground) and see listing 62 for getting the value in class "ParticularAppShow" and displaying it eventually in the GUI, using Xml.[1]

```java
Intent intent = new Intent(AllAppsActivity.this,
    ParticularAppShow.class);

intent.putExtra("wifiActiveB", String.valueOf(String.format("
    %.2f", wifiActiveB)));
```

Listing 61: putExtra() example with wifiActiveB

```java
String wifiActiveB = getIntent().getExtras().getString("
    wifiActiveB");
```

Listing 62: getExtra() example with wifiActiveB

The other class "UpdateFrequencyActivity" is responsible for choosing how often the user wants to transmit the information to the server, which has a association with "MyService" class, where the actual frequency is set.

The 2 classes "WebPageActivity" and "WebPageChooserActivity" implement the opening of the webpage where the statistics from all users can be seen and compared to the users's phone. Before opening the webpage, the class "WebPageChooserActivity" implements the selection of the month that the user wants to open.

The last feature of the GUI comes with possibility for the user to display specific month or day of the traffic of the phone, which is implemented in classes: "DataSettings" and "PickerDialogs".

In the end the XML flow chart can be seen in figure 27 . It shows the 6 Xml files which contribute to the actual design of the GUI. The main file "activity_main" creates the buttons for month picker, list all applications and also day picker. The "list_row" file adds every application to the list with total traffic statistics. The "activity_particulat_app_show" file corresponds to the displaying of all the attributes of every application. The "activity_update_frequency" creates the interface to choose how often to send to the server. The "activity_web_page" and "activity_web_page_chooser" files create the interface to pick the month and to open the webpage of the server side per month.

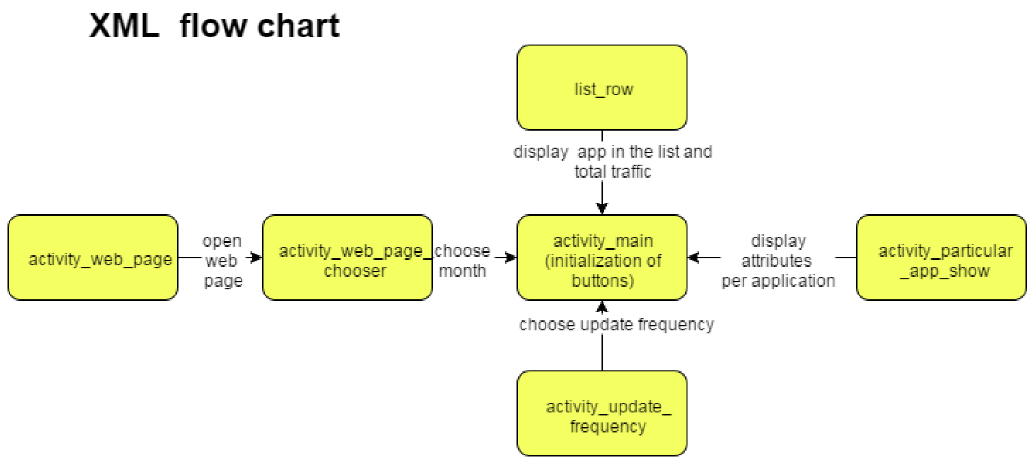
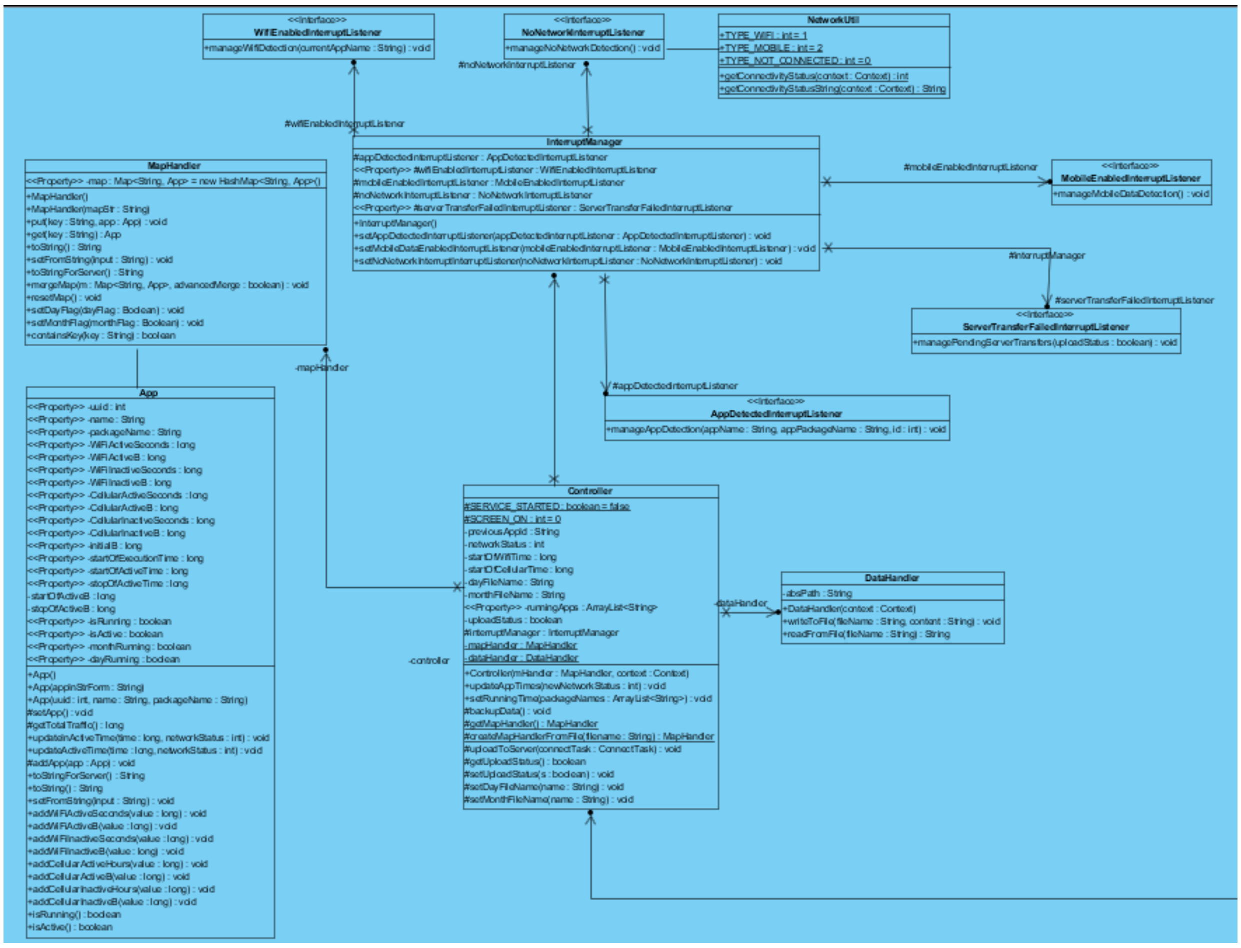

XML flow chart
list_row
display app in the list and total traffic
activity_web_page
open web page
activity_web_page_chooser
choose month
activity_main (initialization of buttons)
display attributes per application
activity_particular_app_show
choose update frequency
activity_update_frequency

Figure 27: XML Flow Chart

3.2.10 Final View - Class Diagram

The final class diagram of the application can be seen in a couple of figures: 28 and 29, due to the size of the class diagram it is split in a couple of parts.

The GUI classes were presented in sub-chapter 3.2.9, therefore for simplicity they were ignored in this chapter.



<<interface>>
WifiEnabledInterruptListener
+manageWifiDetection(currentAppName : String) : void
<<interface>>
NoNetworkInterruptListener
+manageNoNetworkDetection() : void
NetworkUtil
+TYPE_WIFI : int = 1
+TYPE_MOBILE : int = 2
+TYPE_NOT_CONNECTED : int = 0
+getConnectivityStatus(context : Context) : int
+getConnectivityStatusString(context : Context) : String
#noNetworkInterruptListener
#wifiEnabledInterruptListener
InterruptManager
#appDetectedInterruptListener : AppDetectedInterruptListener
<<Property>> #wifiEnabledInterruptListener : WifiEnabledInterruptListener
#mobileEnabledInterruptListener : MobileEnabledInterruptListener
#noNetworkInterruptListener : NoNetworkInterruptListener
<<Property>> #serverTransferFailedInterruptListener : ServerTransferFailedInterruptListener
+InterruptManager()
+setAppDetectedInterruptListener(appDetectedInterruptListener : AppDetectedInterruptListener) : void
+setMobileDataEnabledInterruptListener(mobileEnabledInterruptListener : MobileEnabledInterruptListener) : void
+setNoNetworkInterruptInterruptListener(noNetworkInterruptListener : NoNetworkInterruptListener) : void
#mobileEnabledInterruptListener
<<interface>>
MobileEnabledInterruptListener
+manageMobileDataDetection() : void
#interruptManager
#serverTransferFailedInterruptListener
<<interface>>
ServerTransferFailedInterruptListener
+managePendingServerTransfers(uploadStatus : boolean) : void
MapHandler
<<Property>> -map : Map<String, App> = new HashMap<String, App>()
+MapHandler()
+MapHandler(mapStr : String)
+put(key : String, app : App) : void
+get(key : String) : App
+toString() : String
+setFromString(input : String) : void
+toStringForServer() : String
+mergeMap(m : Map<String, App>, advancedMerge : boolean) : void
+resetMap() : void
+setDayFlag(dayFlag : Boolean) : void
+setMonthFlag(monthFlag : Boolean) : void
+containsKey(key : String) : boolean
-mapHandler
App
<<Property>> -uuid : int
<<Property>> -name : String
<<Property>> -packageName : String
<<Property>> -WiFiActiveSeconds : long
<<Property>> -WiFiActiveB : long
<<Property>> -WiFiInactiveSeconds : long
<<Property>> -WiFiInactiveB : long
<<Property>> -CellularActiveSeconds : long
<<Property>> -CellularActiveB : long
<<Property>> -CellularInactiveSeconds : long
<<Property>> -CellularInactiveB : long
<<Property>> -initialB : long
<<Property>> -startOfExecutionTime : long
<<Property>> -startOfActiveTime : long
<<Property>> -stopOfActiveTime : long
-startOfActiveB : long
-stopOfActiveB : long
<<Property>> -isRunning : boolean
<<Property>> -isActive : boolean
<<Property>> -monthRunning : boolean
<<Property>> -dayRunning : boolean
+App()
+App(appInStrForm : String)
+App(uuid : int, name : String, packageName : String)
#setApp() : void
#getTotalTraffic() : long
+updateInActiveTime(time : long, networkStatus : int) : void
+updateActiveTime(time : long, networkStatus : int) : void
#addApp(app : App) : void
+toStringForServer() : String
+toString() : String
+setFromString(input : String) : void
+addWiFiActiveSeconds(value : long) : void
+addWiFiActiveB(value : long) : void
+addWiFiInactiveSeconds(value : long) : void
+addWiFiInactiveB(value : long) : void
+addCellularActiveHours(value : long) : void
+addCellularActiveB(value : long) : void
+addCellularInactiveHours(value : long) : void
+addCellularInactiveB(value : long) : void
+isRunning() : boolean
+isActive() : boolean
#appDetectedInterruptListener
<<interface>>
AppDetectedInterruptListener
+manageAppDetection(appName : String, appPackageName : String, id : int) : void
Controller
#SERVICE_STARTED : boolean = false
#SCREEN_ON : int = 0
-previousAppId : String
-networkStatus : int
-startOfWifiTime : long
-startOfCellularTime : long
-dayFileName : String
-monthFileName : String
<<Property>> -runningApps : ArrayList<String>
-uploadStatus : boolean
#interruptManager : InterruptManager
-mapHandler : MapHandler
-dataHandler : DataHandler
+Controller(mHandler : MapHandler, context : Context)
+updateAppTimes(newNetworkStatus : int) : void
+setRunningTime(packageNames : ArrayList<String>) : void
#backupData() : void
#getMapHandler() : MapHandler
#createMapHandlerFromFile(filename : String) : MapHandler
#uploadToServer(connectTask : ConnectTask) : void
#getUploadStatus() : boolean
#setUploadStatus(s : boolean) : void
#setDayFileName(name : String) : void
#setMonthFileName(name : String) : void
-controller
-dataHandler
DataHandler
-absPath : String
+DataHandler(context : Context)
+writeToFile(fileName : String, content : String) : void
+readFromFile(fileName : String) : String

Figure 28: Final class diagram part 1

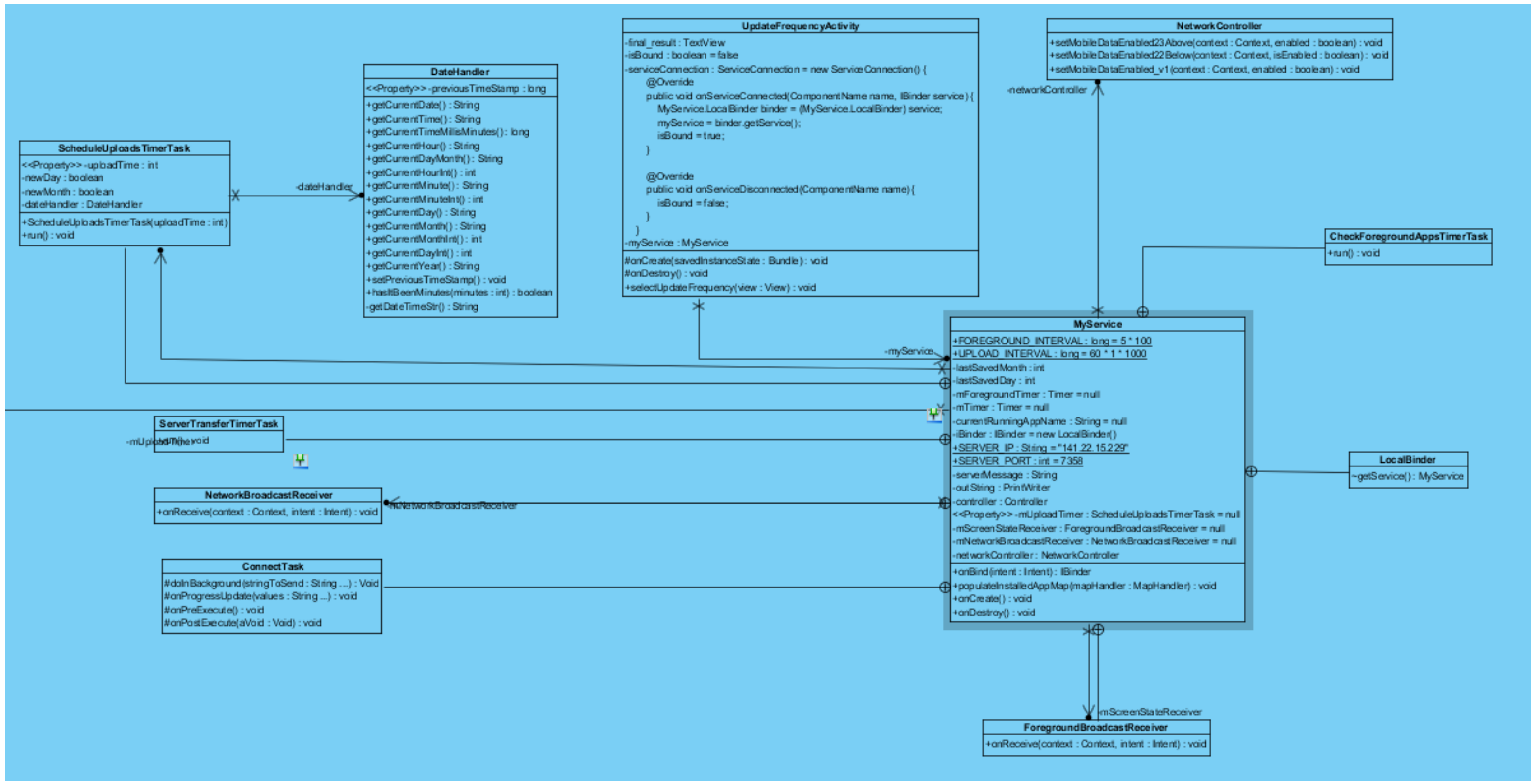

ScheduleUploadsTimerTask
<<Property>> -uploadTime : int
-newDay : boolean
-newMonth : boolean
-dateHandler : DateHandler
+ScheduleUploadsTimerTask(uploadTime : int)
+run() : void
-dateHandler
DateHandler
<<Property>> -previousTimeStamp : long
+getCurrentDate() : String
+getCurrentTime() : String
+getCurrentTimeMillisMinutes() : long
+getCurrentHour() : String
+getCurrentDayMonth() : String
+getCurrentHourInt() : int
+getCurrentMinute() : String
+getCurrentMinuteInt() : int
+getCurrentDay() : String
+getCurrentMonth() : String
+getCurrentMonthInt() : int
+getCurrentDayInt() : int
+getCurrentYear() : String
+setPreviousTimeStamp() : void
+hasItBeenMinutes(minutes : int) : boolean
-getDateTimeStr() : String
UpdateFrequencyActivity
-final_result : TextView
-isBound : boolean = false
-serviceConnection : ServiceConnection = new ServiceConnection() {
@Override
public void onServiceConnected(ComponentName name, IBinder service) {
MyService.LocalBinder binder = (MyService.LocalBinder) service;
myService = binder.getService();
isBound = true;
}
@Override
public void onServiceDisconnected(ComponentName name) {
isBound = false;
}
}
-myService : MyService
#onCreate(savedInstanceState : Bundle) : void
#onDestroy() : void
+selectUpdateFrequency(view : View) : void
NetworkController
+setMobileDataEnabled23Above(context : Context, enabled : boolean) : void
+setMobileDataEnabled22Below(context : Context, isEnabled : boolean) : void
+setMobileDataEnabled_v1(context : Context, enabled : boolean) : void
-networkController
CheckForegroundAppsTimerTask
+run() : void
-myService
MyService
+FOREGROUND_INTERVAL : long = 5 * 100
+UPLOAD_INTERVAL : long = 60 * 1 * 1000
-lastSavedMonth : int
-lastSavedDay : int
-mForegroundTimer : Timer = null
-mTimer : Timer = null
-currentRunningAppName : String = null
-iBinder : IBinder = new LocalBinder()
+SERVER_IP : String = "141.22.15.229"
+SERVER_PORT : int = 7358
-serverMessage : String
-outString : PrintWriter
-controller : Controller
<<Property>> -mUploadTimer : ScheduleUploadsTimerTask = null
-mScreenStateReceiver : ForegroundBroadcastReceiver = null
-mNetworkBroadcastReceiver : NetworkBroadcastReceiver = null
-networkController : NetworkController
+onBind(intent : Intent) : IBinder
+populateInstalledAppMap(mapHandler : MapHandler) : void
+onCreate() : void
+onDestroy() : void
ServerTransferTimerTask
-mUploadTimer
+run() : void
NetworkBroadcastReceiver
+onReceive(context : Context, intent : Intent) : void
-mNetworkBroadcastReceiver
ConnectTask
#doInBackground(stringToSend : String ...) : Void
#onProgressUpdate(values : String ...) : void
#onPreExecute() : void
#onPostExecute(aVoid : Void) : void
LocalBinder
~getService() : MyService
-mScreenStateReceiver
ForegroundBroadcastReceiver
+onReceive(context : Context, intent : Intent) : void

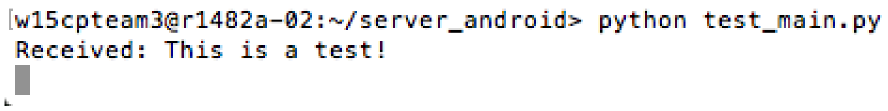
Figure 29: Final class diagram part 2

4 Test Cases

4.1 Server

4.1.1 Server Socket

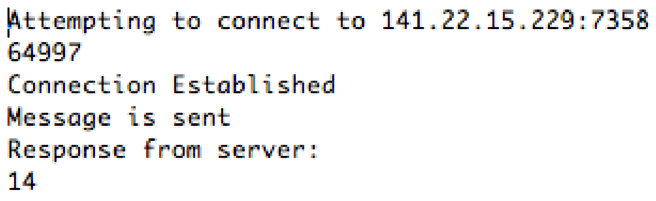
In order to test the server socket for incoming connections as well as the acknowledgement protocol, a temporary Java class is written. This class, having a client socket, is designed to connect to the server socket and send a fixed message and wait for an acknowledgment. As can be seen in Figure 30, server socket is able to receive incomming connections.



```
w15cpteam3@r1482a-02:~/server_android> python test_main.py
Received: This is a test!
```

Figure 30: Console output of the server socket - Incoming message

As mentioned before, the server socket issues a message back to the client as an acknowledgement including the length of the message received. In Figure 31, it can be seen that the server is able to send acknowledgement and it can also be received by the client.



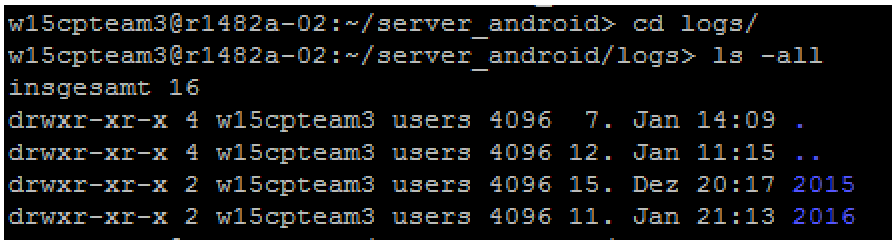
```
Attempting to connect to 141.22.15.229:7358
64997
Connection Established
Message is sent
Response from server:
14
```

Figure 31: Console output of the client socket - Acknowledgement

During these tests, it is discovered that, due to firewall settings of the wireless network *HAW.1X*, devices that are part of this network are not able to communicate with the server socket. Unfortunately a workaround has not been discovered to this issue.

4.1.2 File Handler

Besides the source files, on the server side there is a necessity to to store the app measurements, plus the error log files. all these are stored in the /logs/ folder.

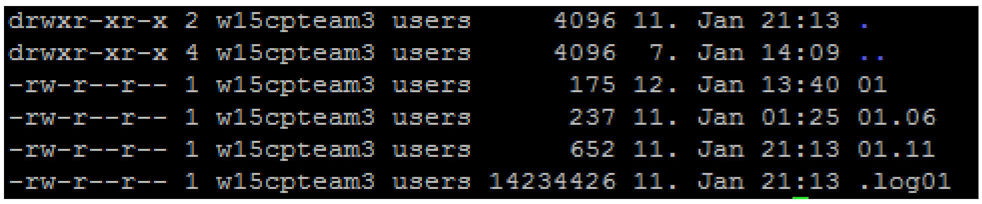


```
w15cpteam3@r1482a-02:~/server_android> cd logs/
w15cpteam3@r1482a-02:~/server_android/logs> ls -all
insgesamt 16
drwxr-xr-x 4 w15cpteam3 users 4096  7. Jan 14:09 .
drwxr-xr-x 4 w15cpteam3 users 4096 12. Jan 11:15 ..
drwxr-xr-x 2 w15cpteam3 users 4096 15. Dez 20:17 2015
drwxr-xr-x 2 w15cpteam3 users 4096 11. Jan 21:13 2016
```

Figure 32: The contents of the /logs folder

As it can be seen from (??, the Logger object, populate the /logs/folder with the year folders. Thy correspond to the year in which the data came from the client.

Inside each /year/ folder there are day , month files, as well as the error log files.



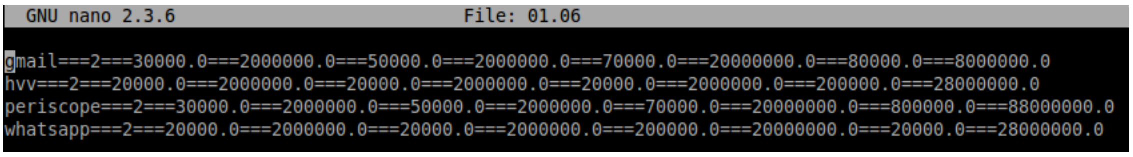
```
drwxr-xr-x 2 w15cpteam3 users     4096 11. Jan 21:13 .
drwxr-xr-x 4 w15cpteam3 users     4096  7. Jan 14:09 ..
-rw-r--r-- 1 w15cpteam3 users      175 12. Jan 13:40 01
-rw-r--r-- 1 w15cpteam3 users      237 11. Jan 01:25 01.06
-rw-r--r-- 1 w15cpteam3 users      652 11. Jan 21:13 01.11
-rw-r--r-- 1 w15cpteam3 users 14234426 11. Jan 21:13 .log01
```

Figure 33: The contents of the /logs/2016 folder on the server

The folder contains:

- day Files - "01.06", meaning January sixth (??
- month Files - "01", meaning January
- log Files - ".log01", meaning the error log for January (35)

An app data log file would contain inside a list of app data strings, each app written in a new line (34). The contents of the app are separated by "===" (34). It is suppose to reflect the raw app data for all clients using the Data Tracker app, for a certain period of time.



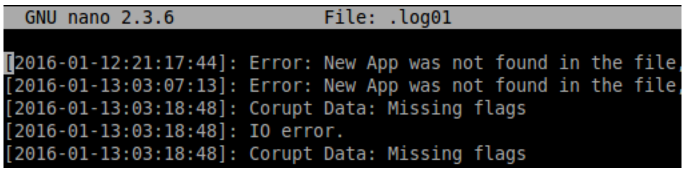
```
GNU nano 2.3.6                    File: 01.06

gmail===2===30000.0===2000000.0===50000.0===2000000.0===70000.0===20000000.0===80000.0===8000000.0
hvv===2===20000.0===2000000.0===20000.0===2000000.0===20000.0===2000000.0===200000.0===28000000.0
periscope===2===30000.0===2000000.0===50000.0===2000000.0===70000.0===20000000.0===800000.0===88000000.0
whatsapp===2===20000.0===2000000.0===20000.0===2000000.0===200000.0===20000000.0===20000.0===28000000.0
```

Figure 34: The contents of a Day App Data Log File

An error log file contains multiple types of error messages, regarding:

- missing flags
- incorrect data types
- corrupt data
- empty names

Depending on which type of error occurs, a specific message, with probably the place where it occurred is appended to these files. Each of these message is preceded by a real time time stamp (35).



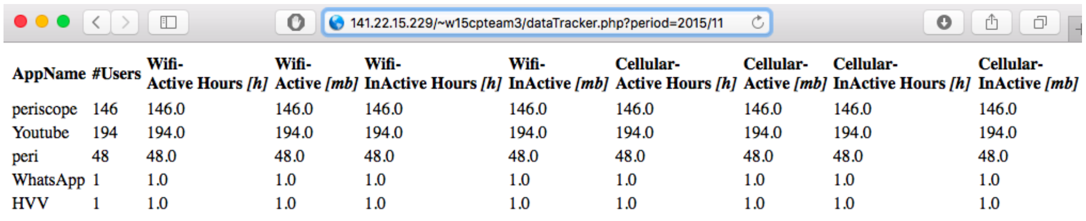
```
GNU nano 2.3.6                File: .log01

[2016-01-12:21:17:44]: Error: New App was not found in the file
[2016-01-13:03:07:13]: Error: New App was not found in the file
[2016-01-13:03:18:48]: Corupt Data: Missing flags
[2016-01-13:03:18:48]: IO error.
[2016-01-13:03:18:48]: Corupt Data: Missing flags
```

Figure 35: The contents of an Error Log File

Writing all these messages is necessary for debugging the behaviour of the server, and comparing it , to the one from the client.

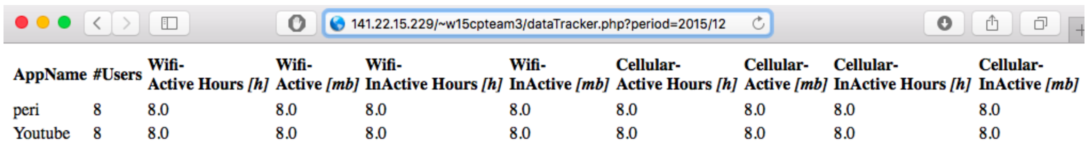
4.1.3 Data Table

First the table view is tested. In order to do so, two different time periods are set: 2015/11 and 2015/12. As can be seen in Figures 36 and 37, tables are successfully created.



AppName	#Users	Wifi-Active Hours *[h]*	Wifi-Active *[mb]*	Wifi-InActive Hours *[h]*	Wifi-InActive *[mb]*	Cellular-Active Hours *[h]*	Cellular-Active *[mb]*	Cellular-InActive Hours *[h]*	Cellular-InActive *[mb]*
periscope	146	146.0	146.0	146.0	146.0	146.0	146.0	146.0	146.0
Youtube	194	194.0	194.0	194.0	194.0	194.0	194.0	194.0	194.0
peri	48	48.0	48.0	48.0	48.0	48.0	48.0	48.0	48.0
WhatsApp	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
HVV	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

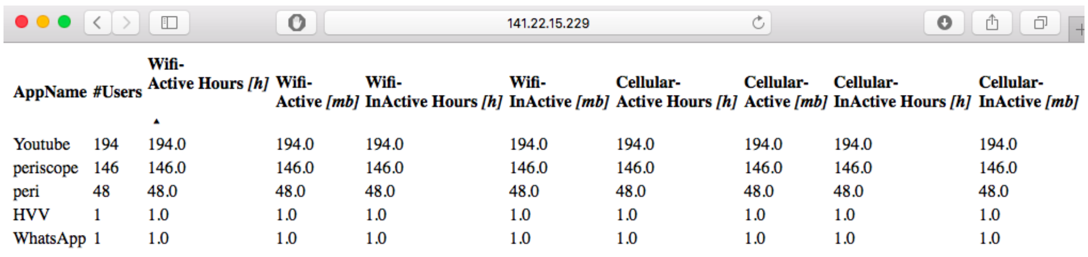
Figure 36: Table view - 2015/11



AppName	#Users	Wifi-Active Hours *[h]*	Wifi-Active *[mb]*	Wifi-InActive Hours *[h]*	Wifi-InActive *[mb]*	Cellular-Active Hours *[h]*	Cellular-Active *[mb]*	Cellular-InActive Hours *[h]*	Cellular-InActive *[mb]*
peri	8	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0
Youtube	8	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0

Figure 37: Table view - 2015/12

Afterwards sortability of the columns are tested by simply clicking on the columns and looking at how they are sorted. This functionality found out to be working without any issues as well. One of the screen shots of the sortable columns is demonstrated in Figure 38.



AppName	#Users	Wifi-Active Hours *[h]* ▲	Wifi-Active *[mb]*	Wifi-InActive Hours *[h]*	Wifi-InActive *[mb]*	Cellular-Active Hours *[h]*	Cellular-Active *[mb]*	Cellular-InActive Hours *[h]*	Cellular-InActive *[mb]*
Youtube	194	194.0	194.0	194.0	194.0	194.0	194.0	194.0	194.0
periscope	146	146.0	146.0	146.0	146.0	146.0	146.0	146.0	146.0
peri	48	48.0	48.0	48.0	48.0	48.0	48.0	48.0	48.0
HVV	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
WhatsApp	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Figure 38: Table view - Sorted column

4.1.4 Data Plotting

The user has the possibility to choose between a pie chart or a bar chart. A pie chart has the purpose of plotting the proportions of data and time usage between the cellular and wifi networks of an app. The user has to type in the period of time to which which one wants to refer. So if for example the data for January 2016 is required, "2016/01" needs to be typed inside the "Year/Month" text area. If the data for Sixth of January 2016 is required, "2016/01.06" is necessary to be typed. These dates correspond to specific files on the server. If they are not found, an error message is outputted to the browser console, and an alert will pop up (41). In the "App Name" text area the name of the app that it's data is to be plotted needs to be written by the user. Again if the app is not found an error message will be outputted to the browser console. Finally in order to plot the "Plot" button needs to be pressed, after which all the necessary searches, and data processing will be made.

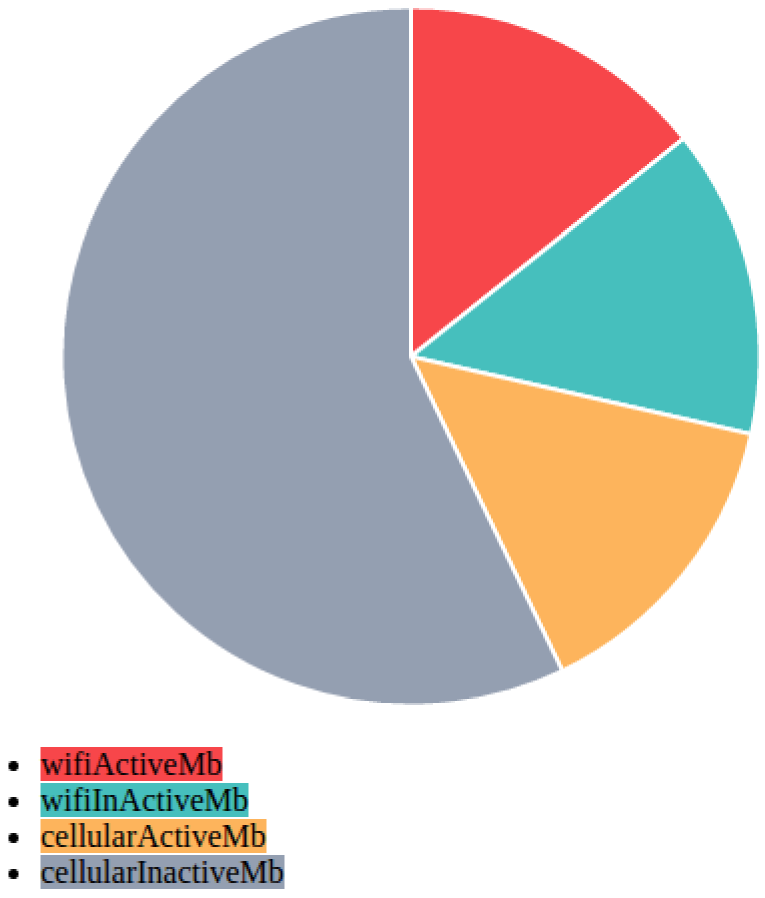


Figure 39: Image of a Pie Chart

As it can be seen in (39), the plot fills up the full page of the browser. This is a plot of the Mb data of the "HVV" app, for the sixth of January 2016. These plots show the proportions from the total amount of Mb (in this case) used by the "HVV" app, on January 6, 2016, by all users of the dataTracker app. Bellow the actual chart there is a list of labels, that make easy to differentiate between the pie chart parts, each having a different color. In order to see the actual value for each part it is only necessary to hover over the chart, and the value will show near to the cursor.

A Bar chart contains more information. It has the purpose of comparing the data of two apps side by side. An additional perk of using this type of chart is that it is possible to get either the total data, or the average per user data per app.

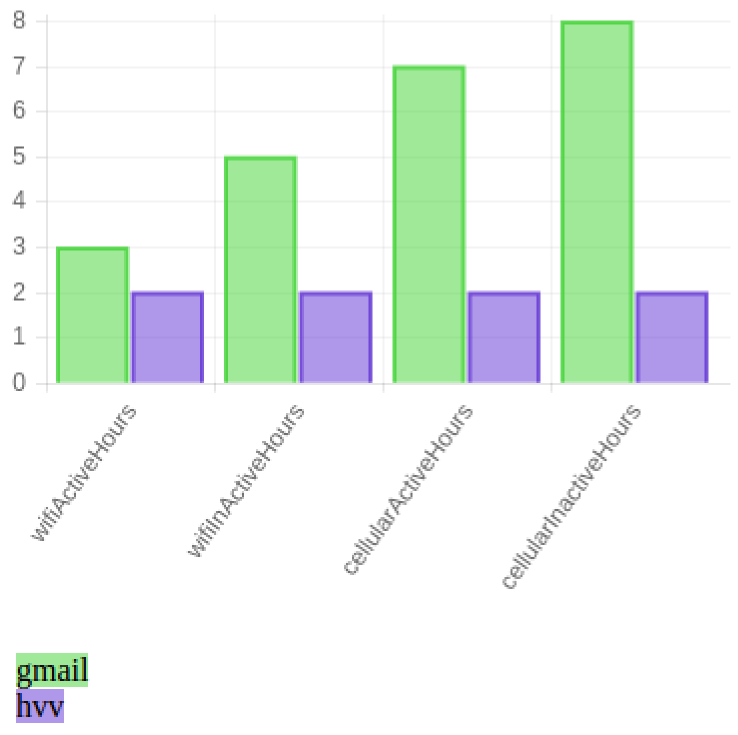
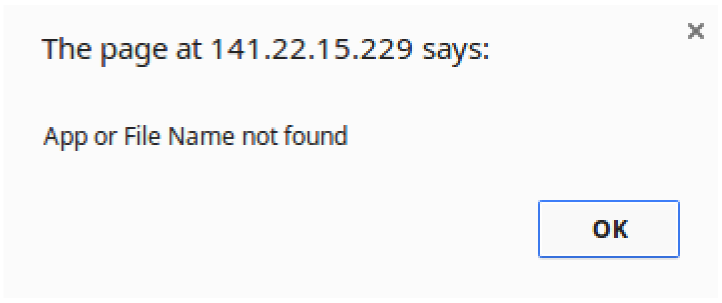


Figure 40: Image of a Bar Chart

The image above depicts a bar chart of the time usage per person for "Gmail" (green) and "HVV" (blue). As it can be seen above (40), it is relatively straight forward to compare the usage of two apps when they are plotted in a graphical way, side by side.


The page at 141.22.15.229 says:

App or File Name not found

Figure 41: Image of an alert when the file or app name entered was not found

When (41) occurs,it is only necessary to check weather the file name, which corresponds to the period of time entered in the "Year/Month" text area. Or it is possible that the app name entered in the "App Name"/ "Second App Name" was not found in the file written in the "Year/Month" texts area. In order to check this, it is easier to check the dataTracker web page (Listing (63))

```
http://141.22.15.229/~w15cpteam3/dataTracker
```

Listing 63: Data tracker web page link

4.2 App

4.2.1 Interrupts

In order to test the Screen ON, Screen OFF interrupts, the methodology is very basic: a simple Log message in the console will suffice when the phone power button is pressed. When the power button is pressed and the phone screen goes ON/OFF, the **Log.i()** is executed in the **onReceive()** method and the message is printed in the console (either "SCREEN ON" or "SCREEN OFF").

The same **Log.i()** method can be used to test the Network interrupts. To simulate a network disconnection, the phone is set to use Wifi data. Then, the router is switched off. The log statement : "Network unavailable" is immediately seen in the console. Then the router is switched on again and the console shows: "Wifi connection available".

There is one small observation here though. In case the phone is not sleeping, the behaviour is exactly the one described above. However, if the phone is sleeping, it is a bit different. It should be noted that the **NetworkBroadcastReceiver** does not detect a change in the status of the network if the phone is in sleep mode and no application needs network resources. The interrupt therefore is only broadcast if there is a specific network related request from an app. For example: say that the phone is in sleeping mode and all of a sudden the wifi network becomes unavailable. Even if the **NetworkBroadcastReceiver** is set, the traffic measurement application will not receive any information about the status of wifi being changed, if no other application needs access to wifi at that specific moment in time. However, if an application suddenly needs to gain wifi access and requests it, the android operating system will acknowledge that there is no connection any more and will broadcast this information to the entire system. Then, the traffic measurement application will be able to catch this broadcasted information since it has a registered **NetworkBroadcastReceiver** listening for it.

This kind of behaviour does not affect in any way the correctness of measurements taken by the traffic measurement application. If the **onReceive()** method is not triggered, it means that no application on the phone is consuming data.

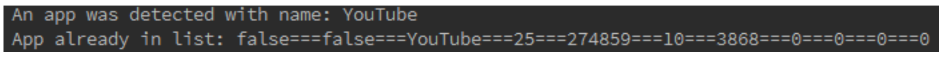
An important source of bugs regarding interrupt handling was the fact that an interrupt might occur before the controller has the time to set and implement the 5 interfaces mentioned in figure 8. If this happens, the application throws an error. To fix this, before calling the interrupt handling routine, it must be conditioned that the interrupt is set inside the **InterruptManager** class (aka interrupt is different from null)
Example:

```
if((controller.interruptManager.wifiEnabledInterruptListener
    != null)){
    controller.interruptManager.wifiEnabledInterruptListener.
        manageWifiDetection(currentRunningAppName);
}
```

Therefore,
if(controller.interruptManager.wifiEnabledInterruptListener != null)
is an important **if** and similar **if**s should be used (for each type of ISR) each time an interrupt handling routine is called.

4.2.2 Measurement Techniques

In order to test whether the measurements are correct Log outputs were set in the source code in order to print the measurements for the app that just went into the foreground.



```
An app was detected with name: YouTube
App already in list: false===false===YouTube===25===274859===10===3868===0===0===0===0
```

Figure 42: Test that shows the measurements of the app which is in the foreground

The test 42 shows what the "Youtube" app measurements data are. Since it is not zero, it means that the app was used sometime before.

4.2.3 Api Dependency Workarounds

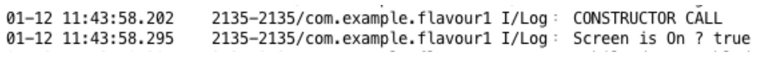
The API dependencies that are mention in Section 3.2.6, are tested individually with two Android phones one running the latest version available and another one running a version older than Lollipop.

First thing tested is the manual detection of screen status. For this purpose a log statement is added to the if block responsible of detecting the screen status at the initial start up of the program. (Listing 64)

```
Log.i("Log", "Screen is On ? " + Boolean.toString(screenOn))
```

Listing 64: Log statement for screen detection

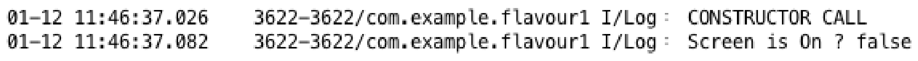
As shown in Listing 53, the boolean *screenOn* contains the screen status; true for screen being on and false for screen being off. First the phone with the new API is tested by installing the app while the screen being on. As can be seen in Figure 43, test is successfully executed and the correct screen status is retrieved.



```
01-12 11:43:58.202    2135-2135/com.example.flavour1 I/Log: CONSTRUCTOR CALL
01-12 11:43:58.295    2135-2135/com.example.flavour1 I/Log: Screen is On ? true
```

Figure 43: Console output of the new API - Screen on

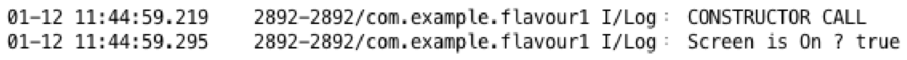
Next, the same phone is tested by installing the app while the screen being off. The corresponding console output can be seen in Figure 44. As can be seen, this test is successfully executed with the correct screen status being retrieved.



```
01-12 11:46:37.026    3622-3622/com.example.flavour1 I/Log: CONSTRUCTOR CALL
01-12 11:46:37.082    3622-3622/com.example.flavour1 I/Log: Screen is On ? false
```

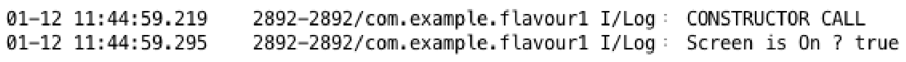
Figure 44: Console output of the new API - Screen off

Same test are then performed on a phone with an older API. Both tests are also successfully executed with the correct screen statuses as can be seen in Figures 45 and 46.



```
01-12 11:44:59.219    2892-2892/com.example.flavour1 I/Log: CONSTRUCTOR CALL
01-12 11:44:59.295    2892-2892/com.example.flavour1 I/Log: Screen is On ? true
```

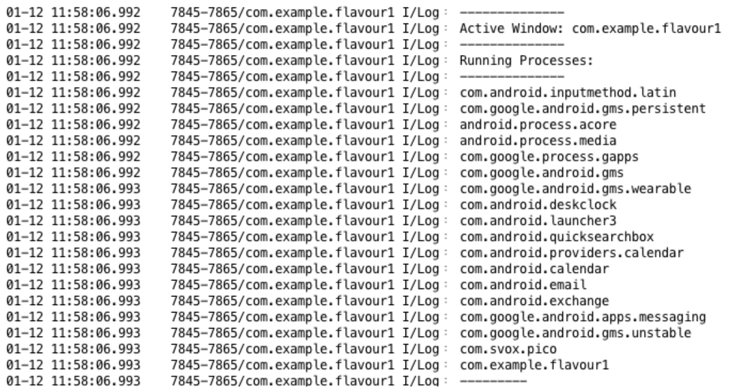
Figure 45: Console output of the old API - Screen on



```
01-12 11:44:59.219    2892-2892/com.example.flavour1 I/Log :  CONSTRUCTOR CALL
01-12 11:44:59.295    2892-2892/com.example.flavour1 I/Log :  Screen is On ? true
```

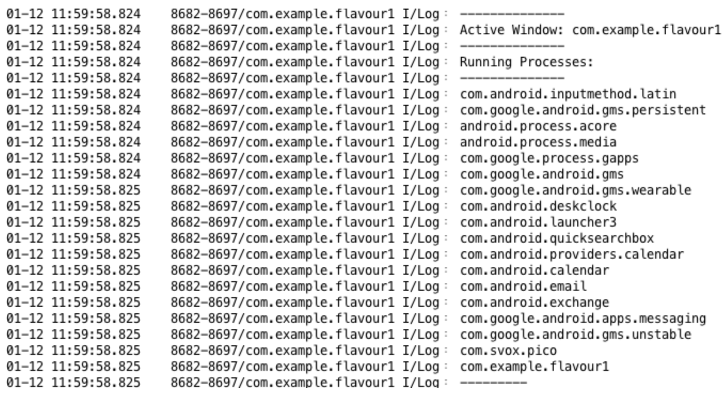
Figure 46: Console output of the old API - Screen off

As the second part of the API dependency test cases, detection of running apps and the active app window is tested on both phones. For this purpose the active app name together with the list of running apps are logged into the console and the corresponding outputs can be seen in Figures 47, 48.



```
01-12 11:58:06.992    7845-7865/com.example.flavour1 I/Log :  --------------
01-12 11:58:06.992    7845-7865/com.example.flavour1 I/Log :  Active Window: com.example.flavour1
01-12 11:58:06.992    7845-7865/com.example.flavour1 I/Log :  --------------
01-12 11:58:06.992    7845-7865/com.example.flavour1 I/Log :  Running Processes:
01-12 11:58:06.992    7845-7865/com.example.flavour1 I/Log :  --------------
01-12 11:58:06.992    7845-7865/com.example.flavour1 I/Log :  com.android.inputmethod.latin
01-12 11:58:06.992    7845-7865/com.example.flavour1 I/Log :  com.google.android.gms.persistent
01-12 11:58:06.992    7845-7865/com.example.flavour1 I/Log :  android.process.acore
01-12 11:58:06.992    7845-7865/com.example.flavour1 I/Log :  android.process.media
01-12 11:58:06.992    7845-7865/com.example.flavour1 I/Log :  com.google.process.gapps
01-12 11:58:06.992    7845-7865/com.example.flavour1 I/Log :  com.google.android.gms
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.google.android.gms.wearable
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.android.deskclock
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.android.launcher3
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.android.quicksearchbox
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.android.providers.calendar
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.android.calendar
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.android.email
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.android.exchange
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.google.android.apps.messaging
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.google.android.gms.unstable
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.svox.pico
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  com.example.flavour1
01-12 11:58:06.993    7845-7865/com.example.flavour1 I/Log :  ---------
```

Figure 47: Console output of the new API - Active window and running apps

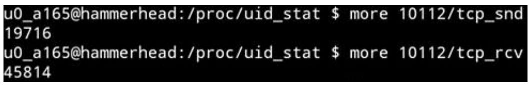


```
01-12 11:59:58.824    8682-8697/com.example.flavour1 I/Log :  --------------
01-12 11:59:58.824    8682-8697/com.example.flavour1 I/Log :  Active Window: com.example.flavour1
01-12 11:59:58.824    8682-8697/com.example.flavour1 I/Log :  --------------
01-12 11:59:58.824    8682-8697/com.example.flavour1 I/Log :  Running Processes:
01-12 11:59:58.824    8682-8697/com.example.flavour1 I/Log :  --------------
01-12 11:59:58.824    8682-8697/com.example.flavour1 I/Log :  com.android.inputmethod.latin
01-12 11:59:58.824    8682-8697/com.example.flavour1 I/Log :  com.google.android.gms.persistent
01-12 11:59:58.824    8682-8697/com.example.flavour1 I/Log :  android.process.acore
01-12 11:59:58.824    8682-8697/com.example.flavour1 I/Log :  android.process.media
01-12 11:59:58.824    8682-8697/com.example.flavour1 I/Log :  com.google.process.gapps
01-12 11:59:58.824    8682-8697/com.example.flavour1 I/Log :  com.google.android.gms
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.google.android.gms.wearable
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.android.deskclock
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.android.launcher3
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.android.quicksearchbox
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.android.providers.calendar
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.android.calendar
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.android.email
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.android.exchange
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.google.android.apps.messaging
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.google.android.gms.unstable
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.svox.pico
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  com.example.flavour1
01-12 11:59:58.825    8682-8697/com.example.flavour1 I/Log :  ---------
```

Figure 48: Console output of the old API - Active window and running apps

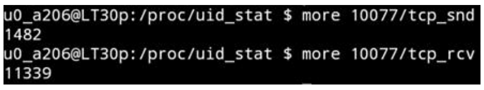
As can be seen, these tests are successfully executed with the correct active window detection including the list of running apps.

Final part of the API dependency test cases, is about the measurement of app specific data usage. First of all the uid of the app Facebook is determined on each phone. Afterwards, by using a terminal app, contents of the files *tcp_ rcv* and *tcp_ snd* are displayed as can be seen in Figures 49 and 50.



```
u0_a165@hammerhead:/proc/uid_stat $ more 10112/tcp_snd
19716
u0_a165@hammerhead:/proc/uid_stat $ more 10112/tcp_rcv
45814
```

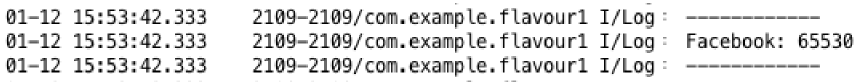
Figure 49: Contents of textittcp_rcv and *tcp_ snd* - New API



```
u0_a206@LT30p:/proc/uid_stat $ more 10077/tcp_snd
1482
u0_a206@LT30p:/proc/uid_stat $ more 10077/tcp_rcv
11339
```

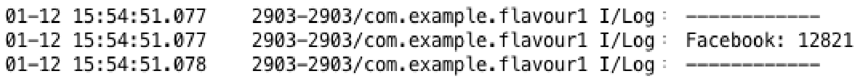
Figure 50: Contents of textittcp_rcv and *tcp_ snd* - Old API

Afterwards, the contents of these files are read by the Android app and logged into the console as the total traffic usage. The corresponding outputs can be seen in Figures 51 and 52.



```
01-12 15:53:42.333    2109-2109/com.example.flavour1 I/Log:  ------------
01-12 15:53:42.333    2109-2109/com.example.flavour1 I/Log:  Facebook: 65530
01-12 15:53:42.333    2109-2109/com.example.flavour1 I/Log:  ------------
```

Figure 51: Total traffic - New API



```
01-12 15:54:51.077    2903-2903/com.example.flavour1 I/Log:  ------------
01-12 15:54:51.077    2903-2903/com.example.flavour1 I/Log:  Facebook: 12821
01-12 15:54:51.078    2903-2903/com.example.flavour1 I/Log:  ------------
```

Figure 52: Total traffic - Old API

As can be seen, these tests are also successfully executed with the correct total traffic calculation by reading the contents of files textittcp_rcv and *tcp_ snd*.

4.2.4 Client Socket

In order to test if the application transmits data to the server and receives data back, different methods were used:

1. Basic use of **Log** statements to show if:

    a) A socket was established with the correct port number and IP address

    b) A valid string was sent to the server using a **PrintWriter** object

    c) The same string was received on the server side, with the same length.

    d) The app receives null **serverMessage** string in case network connection is broken.

    e) An Error/Exception occurred

    f) Socket was successfully closed after a successful transmission and also after an unsuccessful transmission.

2. Use of a testing class **NetworkController** in order to disable the network connection programatically at specific points to simulate a sudden break of connection at the worst moment(aka simulate worst case scenario)

3. Program the Server to not send anything back in order to verify if **BufferedReader's .readLine()** method is blocking or not(in other words, will the app wait forever to receive a string back from the server, or is there a time-out built in the **.readLine()** method)

4. Catch the following exceptions to see if it solves crash problems:

    a) **UnknownHostException**

    b) **IOException**

    c) generic **Exception**

All subitems (1.a)-(1.f) were successfully verified. (1.e) was used to detect what kind of errors can appear while the transmission was in process. There are 2 main errors that can occur as mentioned in 4.a and 4.b: **UnknownHostException** and **IOException**. **UnknownHostException** is thrown when a connection to a server with a specified ip address cannot be established because the address is not valid or because there are network related issues. **IOException** is thrown when the **Socket** object cannot be created/opened. An important bug that crashed the application many times was the fact that socket was closed in a **'finally'** section of a **try-catch** block without checking whether the socket object is null or not. Once these two exceptions were caught and the socket was verified for not being null before closing, the number of crashing instances decreased significantly(by approximately 80 %)

The problem that remained was when network connection failed *after* the socket connection was established and the transmission was taking place. This can happen for multiple reasons:

1. Server loses connection to the network
2. Phone is carried into a new location where there is no network.
3. Interference (especially with wifi) that decreses the SNR to the point of effectively disrupting communication.

In order to recreate such events, simply disabling the wifi or mobile data manually does not work, because it has to be done exactly when the transmission is taking place (in the middle of it).Therefore, a special class was created: **NetworkController**. This class contains multiple methods(for different API versions of the phone) that allow to enable and disable mobile data and wifi connection *programatically.* Consequently, the method: **setMobileDataEnabled(boolean enabled)** was used with the **enabled** parameter set to false in order to disconnect after the socket was created. The method was used in multiple places:

1. After successfully creating the socket and before sending the string to the server.
2. Exactly after sending the string to the server and waiting for the acknowledgement.
3. Exactly after the acknowledgement was received and before closing properly the socket.

All of the above scenarios resulted in the application crashing. Therefore, the final solution was to remove all the previous catch blocks and catch a single generic error/exception of type **Exception**. Once it is caught, the application logs the error message and generates an interrupt to the controller informing it that it will have to attempt a retransmission at a later point in time:

```
controller.interruptManager.
    serverTransferFailedInterruptListener.
    managePendingServerTransfers(false);
```

As it has been mentioned earlier, the application has to receive an acknowledge string from the server which specifies the number of letters that were sent. The application checks if the number is correct and logs a success or a fail depending on the situation.

One important thing is to make sure that in case the server does not reply back, the application does not wait indefinitely for an acknowledgement. The code was tested initially without any custom timer to kill the socket after a certain amount of waiting time. As it turns out, there is a time-out mechanism built in the **readLine()** function that attempts to read the incoming server string. Consequently, there is no need for a custom timer to kill the socket. In case the server does not reply, the **serverMessage** will have the value **null**, so the application will register a fail flag and attempt retransmission later.

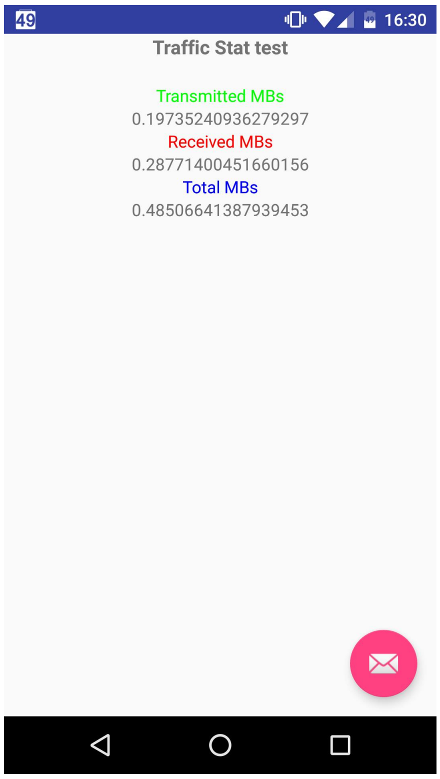
One interesting situation took place during the initial testing phase. The server was returning all the time an acknowledgement number, which was one unit higher than the correct string length. As it turned out, there was a new line character inserted by **Python** on the server side. In the end, a **split()** on newline was used in order to get rid of the error.

4.2.5 App Gui

The test cases for the GUI of the application are connected to the aim of the project, which is to display active/inactive time of the application and wifi/cellular network.

The GUI of the application was different depending on the technique used to measure data and time and also depending on different test case purposes. Therefore different designs were considered for test cases.

1. First measurements using TrafficStats library:



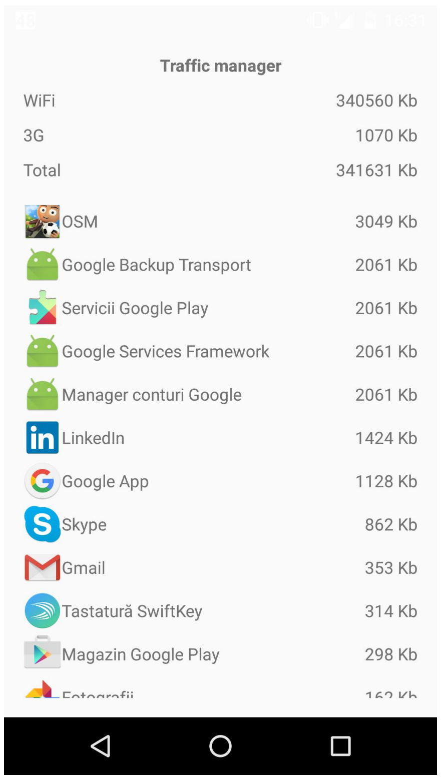
Traffic Stat test

Transmitted MBs
0.19735240936279297
Received MBs
0.28771400451660156
Total MBs
0.48506641387939453

Figure 53: FirstVersionApp

In figure 53 can be seen the first GUI of the project where the TrafficStats library was tested with functions getTotalTxBytes() and getTotalRxBytes(), due to some bugs of these functions in some specific situations, the project measurement techniques were changed, therefore the information about data traffic was read from the files stored in Android System, without calling the functions from TrafficStats.

2. Real time measurements per application:



Traffic manager

WiFi	340560 Kb
3G	1070 Kb
Total	341631 Kb
OSM	3049 Kb
Google Backup Transport	2061 Kb
Servicii Google Play	2061 Kb
Google Services Framework	2061 Kb
Manager conturi Google	2061 Kb
LinkedIn	1424 Kb
Google App	1128 Kb
Skype	862 Kb
Gmail	353 Kb
Tastatură SwiftKey	314 Kb
Magazin Google Play	298 Kb
Fotografii	162 Kb

Figure 54: SecondVersionApp

In figure 54 can be seen a better developed GUI, where each particular application has the total consumption of data. It also has displayed the total amount of data used in wifi and cellular networks. The problem with this design is the fact that the user does not have the possibility to check the particular details of active/inactive time and wifi/cellular data usage per application, which is basically the most important part of this project. In this design the TrafficStats library was still used to calculate the data traffic, but starting from the next version of the GUI, the data was read from the file stored in Android System.

3. Final Application List and individual details per application:

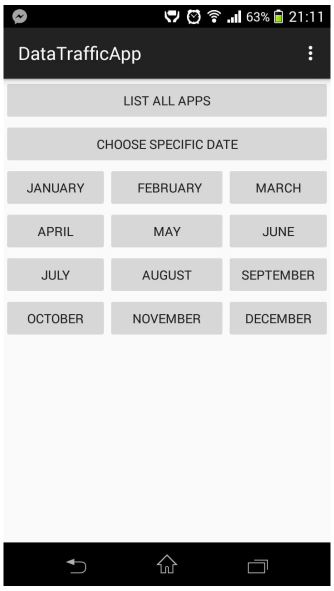
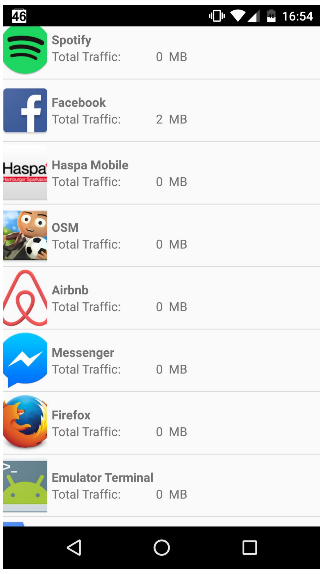
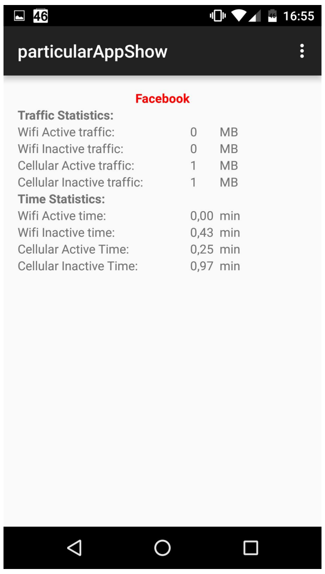


Figure 55: First Initial Page


16:54

Spotify	Total Traffic: 0 MB
Facebook	Total Traffic: 2 MB
Haspa Mobile	Total Traffic: 0 MB
OSM	Total Traffic: 0 MB
Airbnb	Total Traffic: 0 MB
Messenger	Total Traffic: 0 MB
Firefox	Total Traffic: 0 MB
Emulator Terminal	Total Traffic: 0 MB

Figure 56: List of Application


16:55

particularAppShow

Facebook

Traffic Statistics:

Wifi Active traffic:	0	MB
Wifi Inactive traffic:	0	MB
Cellular Active traffic:	1	MB
Cellular Inactive traffic:	1	MB

Time Statistics:

Wifi Active time:	0,00	min
Wifi Inactive time:	0,43	min
Cellular Active Time:	0,25	min
Cellular Inactive Time:	0,97	min

Figure 57: ParticularApp

The final design of the GUI can be seen in figures: 55, 56 and 57 .

The detailed view of each app:

The list of parameters with traffic statistics and time statistics per application is shown in figure 57 . The measurements are read from the files stored in the phone. The test cases to test the actual application were to measure the traffic in both wifi/-cellular networks and active/inactive modes.
The first test case was to use the functions that measure the specific information and store them in files.
The second test case was to read the information from files and to display it on the screen.
After first two test cases the information was successfully displayed in all cases, depending on the button pressed: "choose month", "list all apps" or "choose specific date".

5 Conclusion

In conclusion, the final application fulfilled most of the requirements, coming up with a quite user-friendly environment which can provide data for the user. The GUI displays the information regarding every application in terms of wifi/cellular traffic and also active/inactive time.
It also gives the possibility for the user to access the server webpage and to see the other users statistics, and also to select how often to send information to the server. Moreover, it provides the possibility for the user to choose the view of the statistics: per month, per day or the actual view in the whole year review.

Things that can be upgraded/updated in order to get a better/user-friendly application.

- A sorted list of applications. Even better would be to display the applications that only used traffic, once a application used some traffic it should be displayed in the GUI. It would also be useful to sort the applications in the list using different parameters, eg: decreasing/increasing total traffic, decreasing/increasing wifi traffic, decreasing/increasing mobile data traffic, decreasing/increasing active time/inactive time, etc. All these options could be selected by the user using the *GUI*.

- Update of the GUI every 10 seconds or so, not only when the button is pressed. This feature can be added in case it does not consume too much energy, otherwise can stick with the current implementation.

- A better design of the interface, not only with buttons, which will offer the user the possibility to see data in graphs or other eye-catching features.

- Extend the mapHandler class in order to store the data also for years, this add-on will give the user the possibility to analyze the data in a bigger range therefore having more reliable statistics.

- A better experience comparing the statistics of the other users or average with the own data. (right now implemented in a way that the user can analyze the statistics of the phone then open the server webpage and see the other users statistics which is not so convenient).

- Also, instead of the file system, one can develop an *SQLite* database. This would make **find**, **sort** and other queries much easier (in case the user needs some more advanced functionalities). It also makes the app more 'professional' and scalable. However, the file based approach is also not bad. Linux and Unix make use of such a system to store and manage the information about all the system processes. And after all, as the quote says: "In Unix, Everything is a file"...

References

[1] http://stackoverflow.com/questions/4233873/how-do-i-get-extra-data-from-intent-on-android

[2] Open source Javascript script for sortable table columns by Stuart Langridge: http://www.kryogenix.org/code/browser/sorttable/

[3] Open source GitHub project that checks proc directory to determine running apps as well as the active app at that given time: https://github.com/jaredrummler/AndroidProcesses

[4] http://www.chartjs.org/docs/

[5] http://api.jquery.com/