Computer Networks III
Project Report

Patrik Ehrencrona Kjellin, paeh9336@student.uu.se
Sergej Maleev, sergej.maleev.6750@student.uu.se
Sofia Backman, sofia.backman.5092@student.uu.se
Zijian Qi, Zijian.Qi.7526@student.uu.se

Project Description

In this project we have made an application that collects data about biker in, in this particular case, Uppsala city. The application is meant to collect data to help the city understand its' bikers. The data can be used to understand what changes needs to be made to routes or what roads to put more or less effort into improving for them to be as safe and of as good quality as possible for the bikers.

As a first step into making this applications we have made an application that simply collects time and locational data about the bikers as to be able to put together statistical summaries and analyze those summaries. The application has three major parts, the application in the phone of the user and the database and the server on the administrative end and the website where the information collected can be viewed in a more intuitive way for a human. The user applications simply records data about the user while biking and sends that data to the server where it is saved in the database for future use. In the website conditions can be specified in form of time and locations to enable the user to view the data graphically.

Components

Here we describe the four main components involved in this project; a mobile application, a server coupled with a database, and a website for visualization of tracking data.

Mobile Application

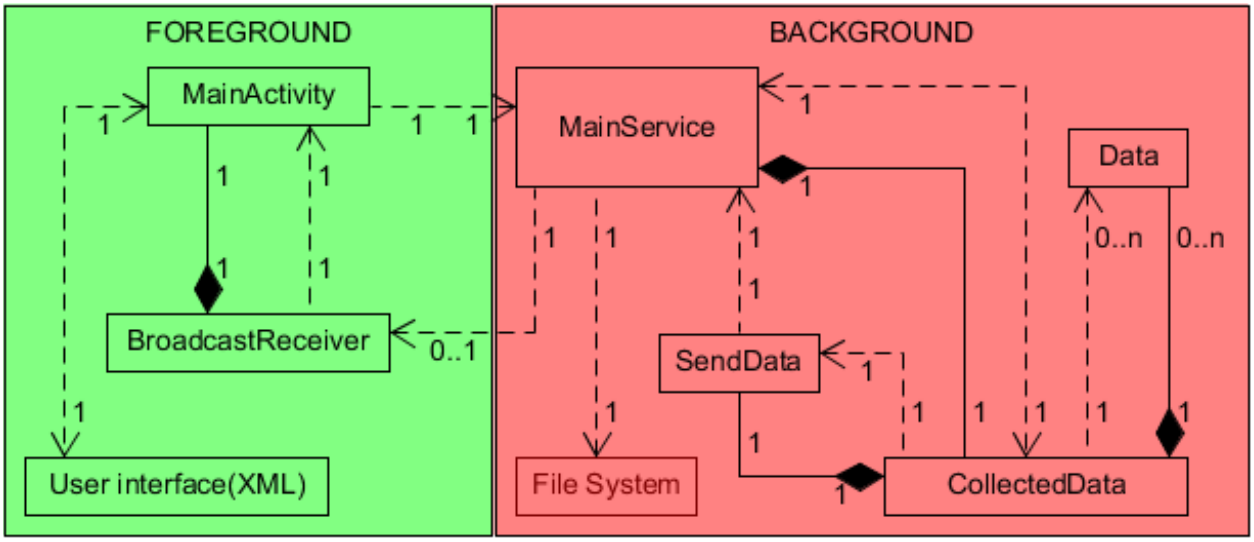

Fig 1. Class diagram, without including functions

Here a simple class overview over our application, there we have the MainActivity class that visualizes devices activity on the google map, the MainService class which we mostly focused on, the Data class which holds the information about single data point, the CollectedData, and the SendData class which is an instance of a Runnable class. The SendData is used to create a new Thread each time we want to send the Data to our server. Notice that the CollectredData

and Data classes are not the background threads and that User Interface and the File System are not classes actually, but they are included here to understand which part cooperates with wich.

About the composition of our classes, the MainService creates a CollectedData variable, which in turn hold a SendData and the list of Data variables. And the MainActivity class have a BroadcastReceiver that receives broadcastmessages from our background service thread.

Dependencies of our implementation might be a bit complicated. We have clearly understandable dependencies like that the CollectedData uses Data class to store all the data and uses SendData class to obviously send the data to our server or that MainActivity uses the BroadcastReceiver to receive the messages, but there is also a bit tricky dependencies like that CollectedData is responsible for starting a new Thread which will send the data over the Internet, but before we start a new Thread we want to check if the internetconnection is available in order to not waste the battery power and to avoid unnecessary allocation of the memory space, BUT the process of checking the internetconnection has to be done in a backgound thread and the CollectedData is not a background thread, this is why we have added a static function to our MainService thread, which is used by the CollectedData class, to check the status of the internetconnectivity. In order to get the result if data was sended successfully or not, the SendData class use the appropriate static function from the MainService class.
A general overview over most important classes in our application is described below.

Data

A single Data point have a session identification number which is unique for each unique biketrip, and each biketrip usually will include more than one datapoint, it have a coordinate point which is the latitude and longitude of the map position, the time is in milliseconds (known as Unix time or Epoch time), the speed represented in meters per second and the accuracy represented in meters. The Data class have toString() function that returns a JSON string of the data.

CollectedData

The CollectedData class have such responsibilities as add the datapoint, get the datapoint, remove the datapoint(s), import the data from a file and prepare the data to be sent over the internet. In order to prepare the datapoints it also have toString() function, which returns the JSON string there all collected data is represented in form of JSON list.

MainService

The MainService class is responsible for tracking and handling the device’s activities and in order to get devices activities we use the imported GoogleApiClient library, it is also responsible for saving the data in a filesystem when the person is biking, for sending activity information to the MainActivity class when it is up, and to clear the recorded data when the data is successfully uploaded to the server.

Further information about other classes and used functions is described in the appendix.

Server

We set up a RESTful (Representational State Transfer) web service using standard tools from the J2EE environment, and launched it within an Apache Tomcat container. Namely, a servlet implementing the HttpServlet abstract class was used, with methods corresponding to different types of HTTP methods.

The server consumes JSON-formatted data through HTTP POST requests sent from the clients. It then parses the JSON data and connects to an SQL database using a JDBC driver. The database connections are pooled and managed entirely by the container, and are injected into the servlet class at request. The server then populates the designated tables with bicycle tracking information via prepared statements. The prepared statements serve to provide some basic protection against SQL injection, while simultaneously increasing efficiency.

No information pertaining to any specific user is stored on the server. Instead, the biking sessions are labeled with non-unique IDs generated on the client-side, which in turn are used by the server to assign unique IDs identifying related data points within the database. The server is meant to scale with an arbitrary number of users. Scaling for a larger amount of users should only be a matter of configuring the container, since the container handles HTTP requests as well as database connections, and invokes the web service upon receiving a request.

Future work on the server should include increasing security by authorization and authentication. Most likely, container based authentication would suffice for this type of application. Such a solution defines different user types, and restricts their access to certain URL patterns or HTTP methods. However, for any type of authentication to be implemented, special care must be taken to ensure that any necessary user information is sufficiently protected.

In a continuation of this project, it will also be necessary to serve data to clients at request. In order to provide the clients with some form of incentive to use our application, the server would need to provide clients with information such as heat-maps describing the concentration of bicyclists or the average speed at some location, or perhaps the fastest path between two points as derived from the data of other bicyclists. Such an extension could easily be implemented using the current structure of the server, by simply adding the necessary GET method handlers, and exposing them on different URL paths.

Database

The vision for the database was in the beginning a bit bigger than the final result. We wished at first to collect the information gathered into tables that reflected the citys' streets to make future features like pathfinding easier to implement. But this vision proved to be a bit too big for the applications we in the end produced.

The database is made out of one table seeing that all data of interest can be derived from the attributes that make up that table.

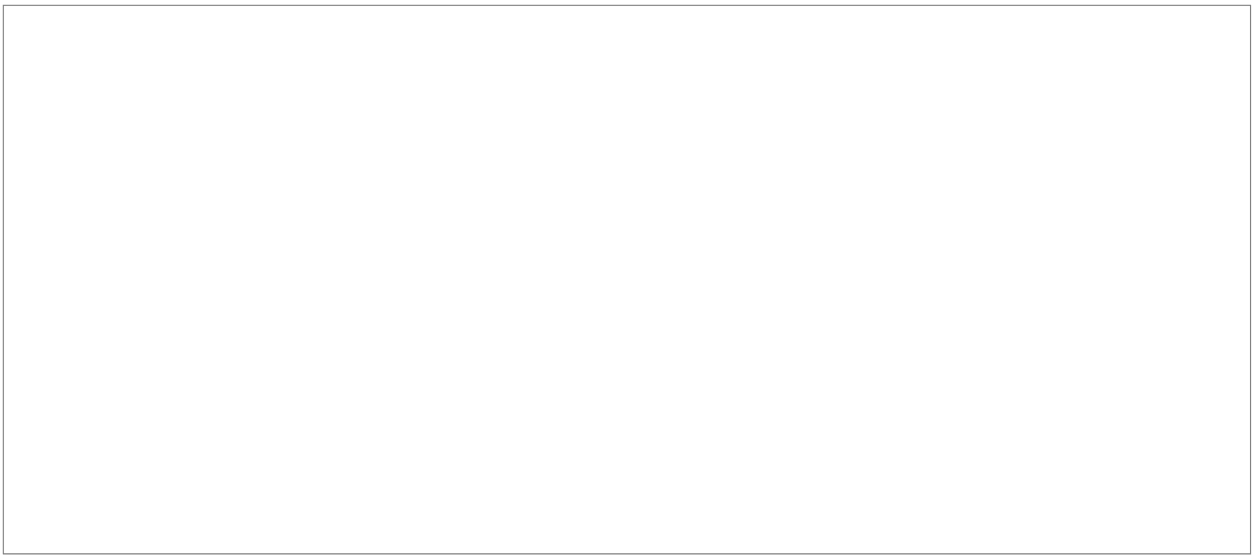

Fig 2. A snapshot of the database.

The nodeID-attribute in Fig 2. is only used as the primary key of the table. sessionID represents the user recorded. The number 30, in this case, shows that all the collected data belongs to one single user. Should that user start a new session, start biking again after not having been biking for at least 10 minutes, he or she will obtain a new number. Getting a new sessionID for every new session enables user privacy and anonymity. time, longitude and latitude of course are the time the user is recorded and the location where the user is recorded. speed is the speed at which the user is traveling and acc corresponds to the accuracy of the recorded data in meters. Should the connection of the phone not allow the service to determine the location of the user very precisely the accuracy-attribute will tell us the inaccuracy of the recording with a radius in meters.

With the attributes in the database we can derive what we thought to be most important to this application, where all bikers of Uppsala are and at what time. This way we can map them on the map on the website.

Website

In our project, we displayed the data, which fetched from mobile phone users, on a website. The website demonstrates data from a specified time range on a heat map. Besides, there is a map of bicycling route, which is showing how friendly the road for a cyclist is, is used for comparing with the heat map.

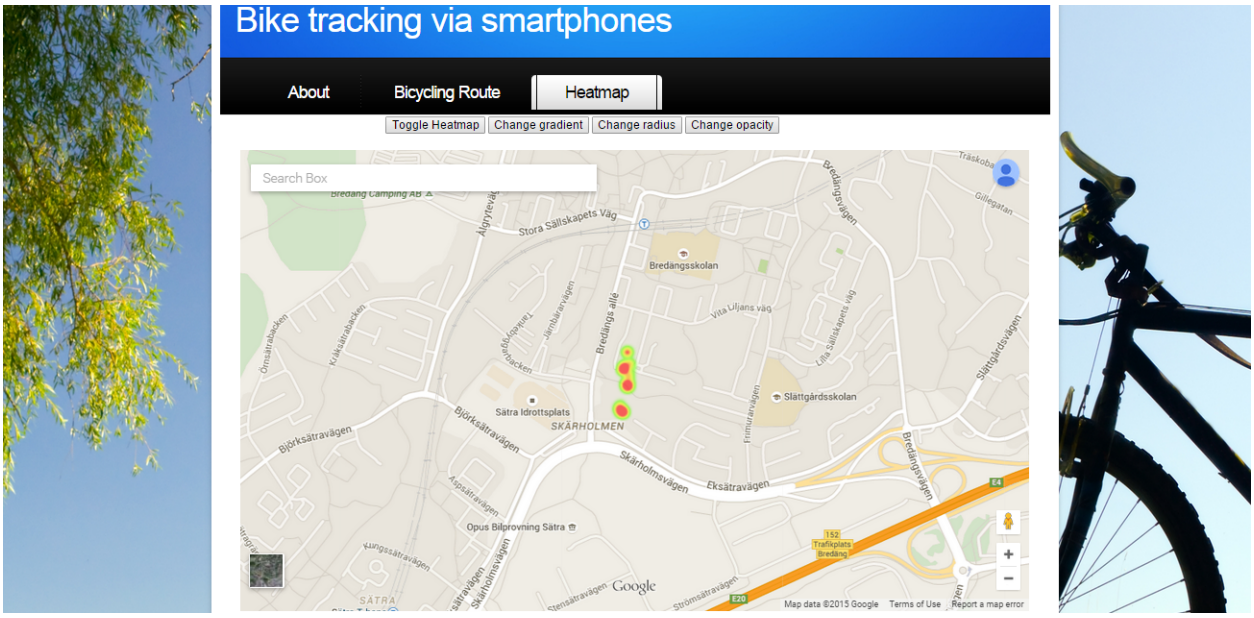

Fig 3 A snapshot of heat map

Both the heat map and the map of bicycle route are built by Google API, so in the map there are all the basic functions the Google map has. The user can search anywhere and view the street. Besides, there are some additional functions in the heat map which are shown on the Fig 3. Users could toggle the heat map or change gradient, intensity and opacity of the marker.

In the heat map, by default, the areas of higher intensity will be colored red, and the areas of lower intensity will appear green. The user can also specify that show the data in which time range. For example, the user can select any date that contains data or select any hour in the day.

Therefore, by the heatmap the user can clearly see that there are more cyclist or few cyclist at which time and which area. If it needs, the user could also compare the heatmap with the map of bicycling route. Then the user can obtain much useful information.

There are still many places we can improve about this website, for example, we should make the time range more flexible. Now we can only select one day or one hour, but we really should make it available to allow users selecting many days or many hours. Besides, There are also much more functions we can add to this website, we could add a function to show the biking path individually by a dotted line.

Appendix: Mobile application classes

```
class MainService:

    public static void successfullyUploaded()
    // Sends a broadcast Intent (with action 'DATA_UPLOADED') to the
    // MainActivity, clears the file with the collected data and resets current
    // CollectedData object.
    // **This function is used by SendData thread, to verify that the data was
    // uploaded successfully.

    public static void failedToUpload()
    // Sends a broadcast Intent (with action 'UPLOAD_ERROR') to the MainActivity
    // **This function is used by SendData thread, to verify that it was unable to
    // upload the data.

    private boolean clearCollectedFile()
    // Writes an empty string in a file, and the path to the file target is stored in a
    // private string named filePath. By default 'sdcard/collected_data.txt'.

    public static boolean isOnline(Context context)
    // Determines if the device is connected to an external router or if the cellular
    // internet is enabled.
    // **This function is used by the CollectedData object. In order to start new
    // SendData thread, the returned value if this function has to be 'true'.

    private boolean addUpdate(String text)
    // Append the string content to the file content, referred by the filePath string.
    // The returned boolean determines if the appending process was successful or
    // not.

    protected synchronized void buildGoogleApiClient()
    // Build an appropriate GoogleApiClient and start the connection process.
    // Uses only in onCreate().

    @Override
    public void onCreate()
    // Execute the function named buildGoogleApiClient.

    private CollectedData data()
    // Return the pointer to the CollectedData object, that contains all the collected
    // data.

    @Override
```

```
public void onStart(Intent intent, int startId)
// Defines static variables (boolean named active and ) on service start.

@Override
public void onConnected(Bundle p1)
// This function is called after GoogleApiClient has successifully connected to
// google services. It initialize all variables, that is needed for future
// update/remove function calls. Default input ‘p1’ is never used.

private void requestRecognitionUpdates(int sec)
// Start the Activity Recognition updates by using ActivityRecognitionApi and
// current GoogleApiClient. Integer sec is the intermediate time between each
// activity recognition Intent.

private void requestLocationUpdates()
// Start the Location updates by using FusedLocationApi and current
// GoogleApiClient.

private void removeRecognitionUpdates()
// Stop the Activity Recognition updates by using ActivityRecognitionApi
// and current GoogleApiClient.

private void removeLocationUpdates()
// Stop the Location updates by using FusedLocationApi and current
// GoogleApiClient.

private long longDiff(long l1, long l2)
// Return the difference between l1 and l2.

@Override
public void onLocationChanged(Location p1)
// If the boolean named record is true, then it uses the information from input p1
// in order to calculate the speed, current session id, and save the results and
// the additional information from p1 to the file (referred by filePath) and to the
// CollectedData object.

private String getActivityName(int type)
// Integer type is compared with static types in DetectedActivity class, and the
// appropriate string is assigned to the return value.

@Override
public int onStartCommand(Intent intent, int flags, int startId)
// If the incoming intent has the Acivity Recognition Result, it send a broadcast
// intent that include necessary information from the incoming intent’s extracted
```

```
        // data. Further actions depends on the collected type of activity and a boolean
        // named 'record'. In general it manages the activity of location and recognition
        // updates.
        // If the action of incoming intent is 'SERVICE_STOP', then onDestroy function
        // is used.



class MainActivity:

        @Override
        protected void onCreate(Bundle savedInstanceState)
        // When activity starts it send a start service Intent in order to start MainService
        // (if it is not 'active'), define a BroadcastReceiver in order to get broadcast
        // intents, define other needed variables.

        @Override
        protected void onResume()
        // Register the BroadcastReceiver and prepare the application's map interface
        // before showing it to the user by using a function named setUpMapIfNeeded.

        @Override
        protected void onPause()
        // Unregister the BroadcastReceiver.

        public void updateActivity(String name, int confidence)
        // Updates the title of the marker on the application's map. Is used by the
        // BroadcastReceiver and function named changePosition.

        private void setUpMapIfNeeded()
        // Check if map is already defined, if not create a map with help of a function
        // named setUpMap.

        private void setUpMap()
        // Applies needed settings to the GoogleMap object.

        private void updateMarkerTitle(String title)
        // Updates the title of marker for user's position with the input string named title.
        // This function is used in function named updateActivity.

        public void changePosition(double latitude, double longitude)
        // Update marker's position and title on the map and update the coordinate
        // values in the titlebar of application's interface.
        // This function is used by the location change listener, defined in function
```

```
        // named setUpMap.



class CollectedData:

        public boolean uploading()
        // Return a boolean value which tell about if the SendData background three is
        // currently running or not.

        public boolean dataIsEmpty()
        // Will return true if the size of ArrayList, used to hold data, is zero, false
        // otherwise.

        public void add
        (int id, double latitude, double longitude, long time, float speed, float accuracy)
        // Create new Data point with function's input as values and add it to the main
        // ArrayList of data.

        public boolean isOnline()
        // Return the value of the static function in MainService class, named isOnline.

        public void resetData()
        // Remove all data from the ArrayList, by using the function named
        // removeLastData.

        public int getLastSID()
        // Return the session ID of last element in the ArrayList.

        public long getLastTime()
        // Return the time value of last element in the ArrayList.

        public boolean uploadData()
        // Manages the start of SendData thread and returns a boolean representing if
        // thread was started or not.

        private void removeLastData(int last)
        // Integer last defines the number of last elements in the ArrayList, those
        // elements are removed at the end of function's execution.

        public boolean importFile(String path)
        // Read the file referred by input string path and fill in the ArrayList if the
        // structure of context does match requirements.
```

```
    public String toString()
    // Return a current class definition, including the ArrayList, in form of JSON
    // string.

    public String dataToString()
    // Return ArrayList in form of JSON string.

    public void finalization()
    // Reset data and call default finalize function.


class Data:

    public int getSID()
    // Return the stored session ID.

    public void setSID(int sid)
    // Change value for session id with the input integer sid.

    public double getLatitude()
    // Return value of latitude.

    public double getLongitude()
    // Return value of longitude.

    public long getTime()
    // Return value of time.

    public float getSpeed()
    // Return value of stored speed number.

    public void setSpeed(float speed)
    // Assign the value of input float called speed, to the current speed variable.

    public float getAccuracy()
    // Return the value of accuracy.

    @Override
    public String toString()
    // Return all stored information in form of JSON string.

    public boolean finalization()
```

```
        // Run finalization process.


class SendData:

        public void setInput(String i)
        // Assign the variable string that will be sended, with the value of input string
        // named i.

        private void setUploading(boolean uploading)
        // Assigns the boolean that indicates if the thread is running by the value of the
        // input boolean named uploading.

        public boolean uploading()
        // Return the boolean which indicate if the thread is running or not.

        public void run()
        // Setup the connection with the server and upload the defined string.
        // This function is being executed when thread is running.
```