Visualizing Mobile Phone Sensor Data in an R Environment

Riccardo Miccini
Technical University of Denmark - DTU

January 30, 2017

Abstract

This document will cover the development of an infrastructure for collecting and visualizing geolocalization data from mobile devices.

The project has been carried out under the supervision of profs. John Aasted Sørensen and Ian Bridgwood, as part of a multidisciplinary project.

The content hereby presented follows this scheme: an introductory section containing the problem formulation and an overview of the resulting design, an analysis of the necessary requirements and tools, and comprehensive description of the implementation details. Conclusions will be drawn, assessing the overall results and the newly gained experience and skills.

Contents

Chapter 1

Introduction

This introductory chapter will provide a description of the project in the form of assignment formulation, and a brief overview of the implementation, which will be elaborated further in the following chapters.

1.1 Problem formulation

The aim of the project is the application of methods for integrating mobile phone (Android, iPhone) sensor measurements and an R data visualization environment using the Google Cloud as buffer.

1.2 System description

The implemented system is composed of three main elements: a series of end users' mobile devices, a remote host, and a *data analyst* station.

The former are equipped with a custom-made application capable of transmitting geolocalization data to a remote destination.

The remote destination is represented by a *Google Sheet* document where such data is stored. This spreadsheet acts as a database for the collected data, and is accessible through the cloud.

The user can then visualize the collected data in real-time, using the provided R scripts and a web browser.

Chapter 2

Analysis

The following sections aim to elaborate on the Problem formulation and create a more solid base to use as a guideline and template during the actual development. This analysis does not aspire to be exhaustive, but rather satisfactory in the level of detail required to establish the project.

2.1 Milestone plan

Here is the project milestone plan as formulated during the initial phases of the project. This version of the plan lists a series of achievements deemed necessary to accomplish the major functionalities. It is arranged in a progressive way, with a time axis roughly flowing vertically. Each task depends from its sub-tasks and — more indirectly — on the previous ones within its hierarchal level.

- Extract requirements and use cases
- Evaluate and choose remote hosting service
- Familiarize with necessary tools and languages
    - R and RStudio
    - Java and Android Studio
        - Acquire device location
        - Read and append content to the remote platform
- Software development
    - Configure remote service authorizations and keys

  - Implement Android application
  - Implement R script for visualizing data

- Integration and validation tests

- Compiling necessary documentation

2.2 Requirements

The *software requirements specification* is the formal description of the system to be developed, thoroughly pointing out what the product is — and is not — expected to do. It also helps assessing the extent of the endeavor in terms of workload, costs, and other resources. The requirements are commonly laid out in agreement with the client, and provide the basis upon which the product or project is evaluated. Following the *Unified Software Development Process* — upon which this analysis is loosely based — the requirements can be classified in *functional* and *non-functional*.

As far as this project is concerned, the supervisors have not set any particular requirements apart from those implied in the project name and problem formulation. The list of requirements is therefore mostly based on former knowledge, common sense and best practices within the industry.

2.2.1 Functional requirements

This section defines specific behaviors or functionalities, and is the basis for the tests. Requirements introduced by the modal verb *shall* indicate mandatory conditions, whilst those introduced by the modal verb *may* indicate optional features.

ID	Description
SRS.F.1	The mobile apps shall submit its GPS and identification data at a given interval
SRS.F.2	The GPS data shall comprise location, and altitude and speed when available
SRS.F.3	The location shall be expressed in geographical coordinates (latitude and longitude)
SRS.F.4	The identification data shall comprise of a device ID and timestamp
SRS.F.5	The device IDs shall be unique
SRS.F.6	The data shall be submitted and stored in a *Google Sheets* document
SRS.F.7	The GPS data acquisition shall be explicitly enabled by the user
SRS.F.8	The front-end application shall update with new data at a given interval
SRS.F.9	The front-end application shall allow the user to select a date range and a set of devices
SRS.F.10	The front-end application shall visualize the path and current position of the selected devices on a map
SRS.F.11	The front-end application shall visualize the altitude and speed of a single device on a plot
SRS.F.12	The front-end application shall allow the user to export the collected data

Table 2.1: Software requirements specification: functional requirements

2.2.2 Non functional requirements

This section specifies desired characteristics and qualities of the system.

ID	Description
SRS.NF.1	The location provided by the mobile apps shall be the most precise available
SRS.NF.2	The Android app code shall be written in Java
SRS.NF.3	The Android app code shall employ *Android Studio* build system
SRS.NF.4	The iOS app code shall be written in Swift
SRS.NF.5	The front-end application shall be written in R
SRS.NF.6	The front-end application should limit its data requests to the bare necessary
SRS.NF.7	All source code shall follow a consistent format style
SRS.NF.8	All source code shall be properly documented
SRS.NF.9	All source code should be trivially buildable and executable on other platforms
SRS.NF.10	The released documentation shall be written in English

Table 2.2: Software requirements specification: non-functional requirements

2.3 Tools

This section will cover various software tools that have been employed during the course of the project. These include development tool and software components or libraries used by the implemented code.

2.3.1 Android Studio

Android Studio is the official IDE (integrated development environment) for native Android development. It is distributed freely by Google for Windows, Mac OS X and Linux platforms, and it is based on emphJetBrains' IntelliJ IDEA, a proprietary IDE for Java. It replaced emphEclipse Android Development Tools as Google's primary IDE for Android application development.

The IDE provides a series of Android-specific tools and features. The most notable ones are:

- Code editor with intelligent completion, linter, and refactoring system
- Integrated debugger and emulator based on virtual machines
- Graphical layout editor and wizards for code generation of UIs and other common software components

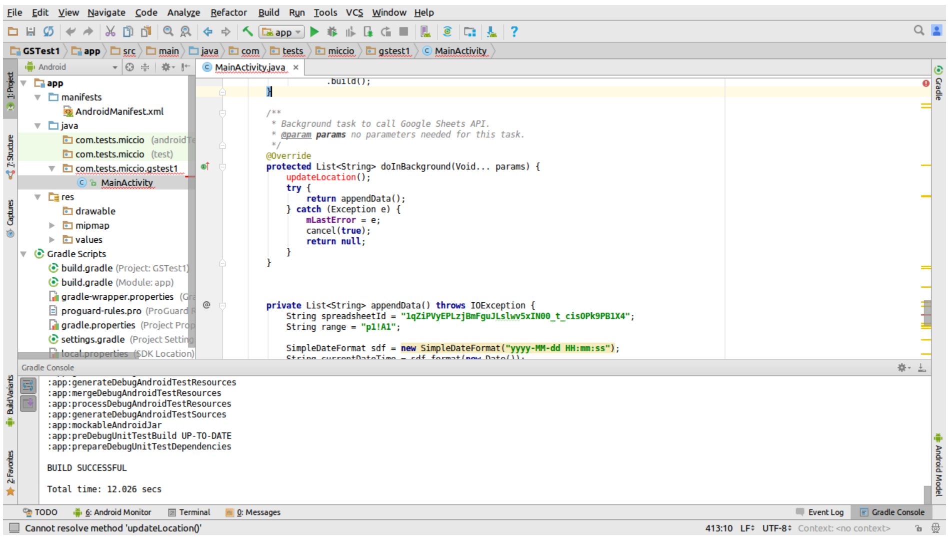

Figure 2.1: Android Studio: example view with text editor, project navigator, and build console

2.3.2 RStudio

RStudio is a free and open-source IDE for R, a programming language for statistical computing. Several editions of the software exist: a emphDesktop one, available on Windows, OS X, and Linux, and a emphServer and emphServerPro one, with allows access through web browser from several terminals.

The software comprises a text editor with code completion and syntax highlighting, an interactive command interpreter with built-in debug, command history, and data viewer, as well as a package manager and a documentation browser.

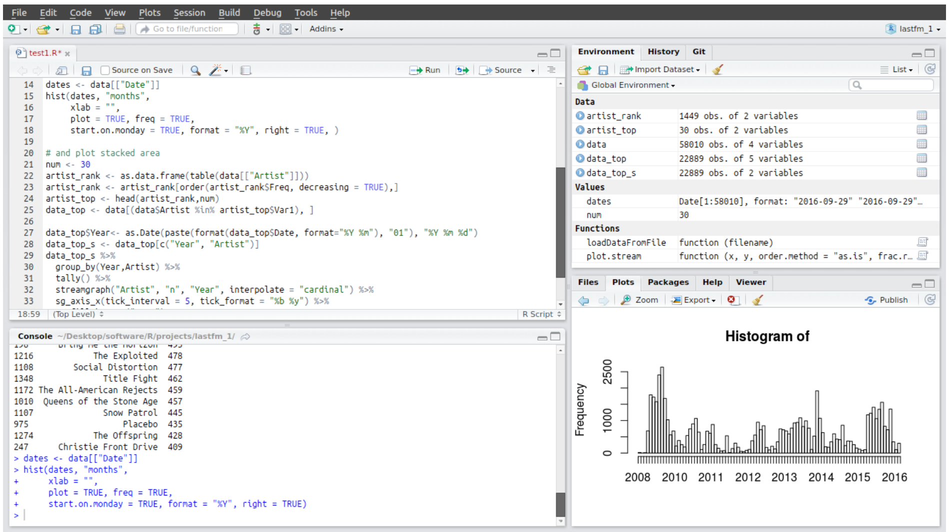

Figure 2.2: RStudio: example view with text editor, console, list of currently loaded data, and plots

2.3.3 R packages

The capabilities of R can be extended through a system of packages. Such packages may provide alternative graphics tools, additional statistical and data handling functions, or APIs and bindings for other services and software. R packages can be developed in R, C, C++, and Fortran, and are normally distributed through the *CRAN* (Comprehensive R Archive Network).

Packages that are not part of the core implementation have to be downloaded and loaded into the environment prior to their use. This can be done through the R console with the `install.packages()` and `library()` commands, or in RStudio's own package manager. Unofficial resources can also be obtained through GitHub or other hosting platforms.

Here follows a list and description of the most important R packages that will be employed for the fulfillment of the project:

`shiny` A framework for building live web applications in R. It comprises control widgets and graphic outputs, and uses a reactive model for determining which parts of the pages needs updating.

`leaflet` Integration with *leaflet*, an open-source javascript library for interactive maps. Maps can be enhanced with custom graphic elements such as polygons, lines, and markers.

`plotly` A powerful, open-source plotting engine, with support for numerous types of charts, and fully interactive and customizable

`googlesheets` A wrapper for the *Google Sheets* APIs. It can be used to perform common file managing operations, as well as for accessing and editing data in a worksheet.

`dplyr` This package provides a set of useful and functions for manipulating data, such as `select`, `filter`, and `arrange`, with a particular focus on performances. It also provides the pipe operator `%>%`, which can be used to concatenate operations on datasets of various nature.

2.3.4 Other tools

In order to be able to contribute from any given location (work PC, personal laptop…), the project source code has been hosted on a web-based `git` repository hosting service called *GitHub*.

`git` is a version control system for collaborative development of projects. It tracks changes in local files and allows them to be synchronized across multiple machines and remote hosts. Within the purpose of this project, `git` has been used to provide a means of accessing and safely storing the code base. Ordinary maintenance of the repository is performed from command-line, with the aid of graphical `diff`-tool *Meld* for code revision.

Whilst the powerful code completion of the aforementioned IDEs is useful for general development tasks, external text editors such as *Atom* have been used as well, especially when large amount of code had to be refactored or restructured. In particular, the project documentation has been authored using *Atom* and several plugins for LaTeX integration.

Chapter 3

Design

This chapter focuses on the design and implementation of the proposed solution into a viable product. It will attempt to thoroughly describe the overall design of the system, as well as the implementation details of its building blocks. Adequate considerations about the development process will be drawn in the conclusion section.

For the sake of convenience, the proposed system can be divided into two conceptual categories: the part that shall be deployed at the end users' premises (the apps running on mobile phones) and the one that shall be under direct control of the data analyst. The latter further comprises a cloud-based infrastructure, and an R application, which can be executed on any host machine, or hosted on a server.

3.1 Infrastructure

The devised infrastructure relies on a cloud-based storage solution, provided by *Google Sheets*. This well-known platform can be easily accessed through its web interface, which mimics the appearance and functionalities of most modern spreadsheet software such as *Microsoft Excel*, and through its extensive API (application programming interface), which allows third-party applications to read and update the content of a given spreadsheet.

The sensors data, consisting in GPS positioning information (as described in the Software Requirements Specifications), are collected collected by the tracking apps in the end users' mobile phones, and submitted to a specific spreadsheet document, which is otherwise private. New data is continuously added to this file, with new entries being appended below existing ones.

The recorded data can then be accessed through a web application written in R. The application periodically queries the document in order to update the information shown. Is can be executed from the data analyst computer, or hosted on a cloud platform.

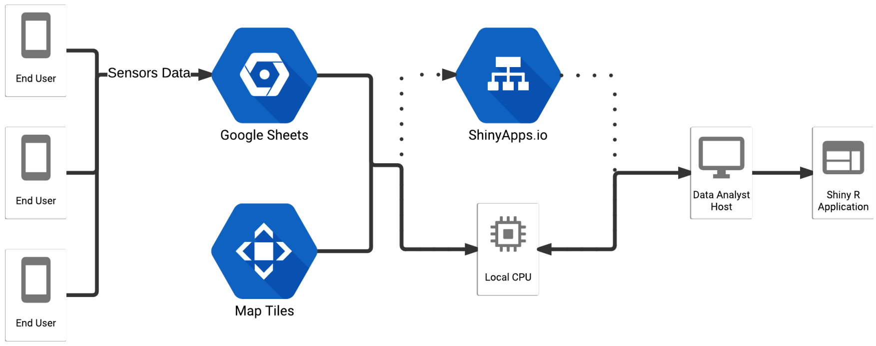

Figure 3.1: Overall system infrastructure: network diagram

The figure above expresses the relations between the aforementioned elements of the infrastructure. The blue hexagons represents cloud-based services. The dotted line suggests a possible hosting solution, based on *ShinyApps.io*.

3.2 Mobile phone app

The phone app is the part of the system that runs on the end users' mobile device, and is responsible for collecting the GPS data and submitting them to the remote storage location. The Android version of the app — which is the main focus of this report — has been developed in JAVA using the *Android Studio* development environment and the *Android SDK* libraries, which are freely distributed by Google. The IDE built-in device emulator has been extensively used for testing the app during its initial stages, while the data collection was performed on a device. A corresponding *iOS* application has also been developed, performing the same tasks as the Android version.

3.2.1 Project structure

The Android project stub has been generated using the built-in wizard, which takes care of creating all the necessary build scripts, configuration, and source files. The conventional folder structure has been adopted. In particular, these are the most important components of the app:

Directory or file	Description
`.`	Project root folder
`./gradlew`	Build script for gradle (Linux)
`./build.gradle`	Project-level build file
`./app/`	Phone and tablet app module folder
`./app/build.gradle`	App-level build file
`./app/proguard-rules.pro`	Configuration for ProGuard (code shrinker)
`./app/src/main/AndroidManifest.xml`	App manifest document
`./app/src/main/java/`	App Java source code folder
`./app/src/main/res/`	App resources folder
`./app/src/main/res/layout/`	Layouts for user interfaces
`./app/src/main/res/mipmap/`	Launcer icons
`./app/src/main/res/values/colors.xml`	List of colors
`./app/src/main/res/values/dimens.xml`	Dimension values (margins, font sizes...)
`./app/src/main/res/values/strings.xml`	Text strings, for ease of internationalization

Table 3.1: Android app: folder structure

Android Studio projects use the *gradle* build automation system for compiling the code, managing dependencies, and generating the necessary `.apk` archive. Including external libraries and other dependencies can be done by adding an entry to the app-level `build.gradle` file.

For the fulfillment of this project, the following libraries have been included:

`com.android.support` Support library implementing *Material Design* recommendations for widgets and user interfaces

`com.google.android.gms` Google *Play Services* APIs providing a variety of functionalities; used for logging into a Google account and obtaining location data

`com.google.api-client` Common functionalities for accessing Google APIs (authentication, JSON and XML parsing, error handling...)

`com.google.apis:google-api-services-sheets` Google Sheets-specific definitions and methods

Along with the previously mentioned `build.gradle` file, another important document is `AndroidManifest.xml`, which provides essential information about the app to the Android system. This may include:

- Java package name for the application
- Any component included in the app, such as activities, services...
- Permissions required to access protected parts of the APIs; for the developed app, they are the following: `ACCESS_FINE_LOCATION`, `INTERNET`, `ACCESS_NETWORK_STATE` and `GET_ACCOUNTS`

3.2.2 Interface

The UI of the app is implemented within a single `Activity` class, and is composed of three graphical elements: a text input box for entering a custom time interval in seconds, and start and stop tracking buttons. The widgets are arranged using a `LinearLayout`, and are declared in the `res/layout/activity_main.xml` file. Other aspects of the user interface include *toats*, which are small, temporary pop-up messages that appear at each submitted location. The figure below shows a screen capture of the app, with one such message.

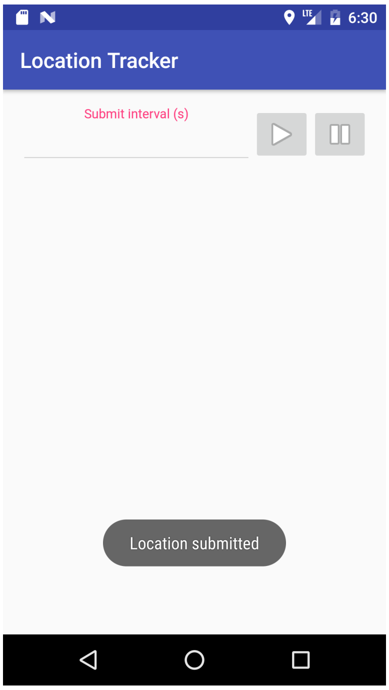

Figure 3.2: Android App: user interface layout

A previous version of the app — not backed by the `Service` class — featured a text box showing information about the latest location acquired.

3.2.3 Preliminary tasks

Before starting to collect location data, the app has to correctly set up several objects and handlers, which will provide access to the necessary functionalities and APIs. These objects are:

`GoogleApiClient` Entry point for Google Play Services integration. This object is constructed using the *Builder* design pattern, which lets the developer include optional arguments, such as the requested API (location, in this case) and connection callbacks. Once built, the object has to connect to Play Services to be functional. Depending on the result of the connection attempt, the `onConnected()` method or the `onConnectionFailed()` will be invoked.

`LocationRequest` Data object storing the desired characteristics of the location update request. The parameters that have been set for this project include: nominal update interval, minimum update interval, accuracy level, and minimum distance between points.

`GoogleAccountCredential` Manages authorization and account selection for Google accounts. This object is constructed using a *OAuth 2.0*-based *Factory* method, which requires a list of scopes (i.e. requested functionalities) as argument. The object has no associated account by default, so a new dialog will prompt the user to choose one.

`Sheets` Abstraction of Google Sheets APIs. It is used to execute operations on documents, and it is constructed using a Builder too. It requires an HTTP transport abstraction object, a JSON factory object — which are both generated on-the-fly, and the previously described credential object.

3.2.4 Location tracking

Once the preliminary setup has been performed and the Google API client object is connected to the Play Services, it is possible to request location updates. This method of acquiring data is more effective and efficient than polling the current known location, as it provides finer control o the accuracy and frequency of the updates (through the previously mentioned `LocationRequest` object). Location updates can be requested by calling the `requestLocationUpdates()` method, which takes three arguments: a `GoogleApiClient` object, a `LocationRequest` object, and an object of a class that implements the `LocationListener` interface — namely, the main activity class. This interface lets the developer override the `onLocationChanged()` callback method, which is invoked whenever a new point is available.

It is important to note how particular work flow is best suited for a foreground use case. During the writing of this document, efforts were being made to convert the application to a more universal solution based on a `PendingIntent` object. This class is constructed using the `getService` method, which generates a `PendingIntent` that will start a service, with the start arguments given from the extras of the `Intent` argument. The implemented service sub-class overrides the `onStartCommand()` method, from where the new location is extracted and processed.

3.2.5 Data submission

The location data is contained in a `Location` data object that stores latitude, longitude, heading, speed, and many other GPS-related measures. Uploading these values — along with the time-stamp and device ID — is outsourced to an asynchronous task. The task is implemented by inheriting from the `AsyncTask` class. The main functionalities are coded into the overrideable `doInBackground()` method, and merely consist in packing the data into a `ValueRange` object and submitting it. This is done by calling the `append()` function on the `Sheets` object created earlier. The call internally executes an `HTTP POST` request to the Google API servers, with the data encoded in JSON format. Possible errors returned in the response are thrown as instances of `IOException`, and taken care of in the task `onCancelled()` method by requesting further missing authorizations or showing the error content in a message.

3.3 R

The R application is the part of the solution that is supposed to run on-premises. It takes care of gathering the data from the spreadsheet document and generating a web interface with a map and chart for visualizing said data. The software is built around the *Shiny* framework, which provides functionalities for developing web applications in R.

3.3.1 R *Shiny* architecture

The features that allows Shiny to provide responsive feedback lies in the update policy of certain code block and interface elements, which can be invalidated by other expressions or user interaction. Expressions that are invalidated will be immediately reevaluated or redrawn.

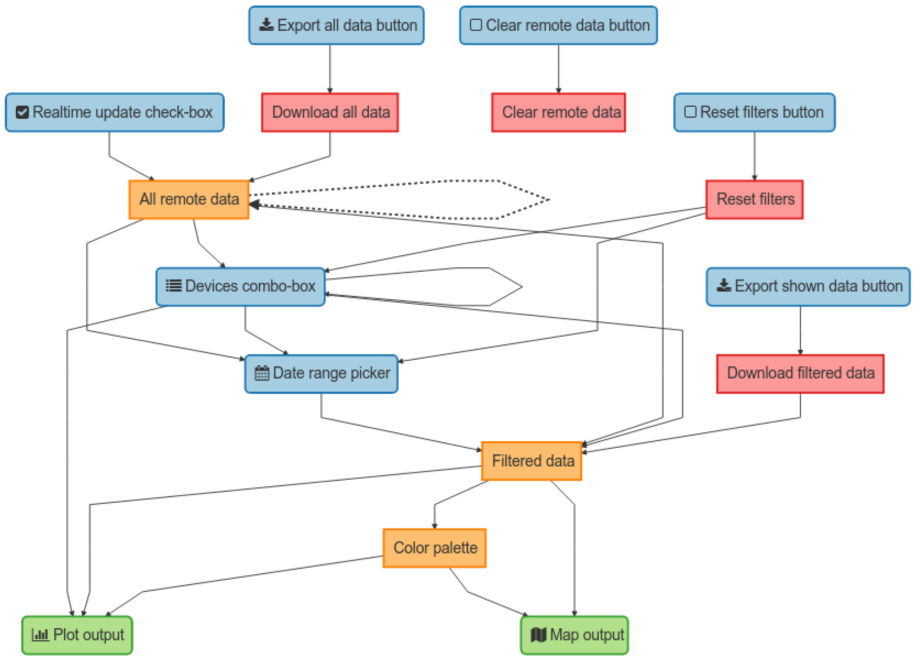

Figure 3.3: Shiny Application: reactivity diagram

The picture above shows the relations between the different blocks of code. If two elements are connected, whenever the origin changes changes, the destination is notified that it needs to re-execute. The dotted line represents a periodic operation. The system inputs, outputs, and reactive expressions are denoted by the cyan, green, and orange nodes respectively. The red nodes represents observer and reactive events that are triggered by the invalidation of the affected elements.

3.3.2 Interface

The UI of the web application is written in R, using a declarative paradigm. Each graphic element is defined within its hierarchical parent, along with a set of options. The R code returns a block of `html` elements that implement the UI. Shiny uses the `bootstrap` front-end framework to allow for responsive pages, and comes with a series of built-in widgets for user interaction.

The page is based on the the `sidebarLayout`, which consists in a main area, used to visualize a map with the end users' positions and a plot of recorded speed and altitude, and a side bar, with filtering and data export options.

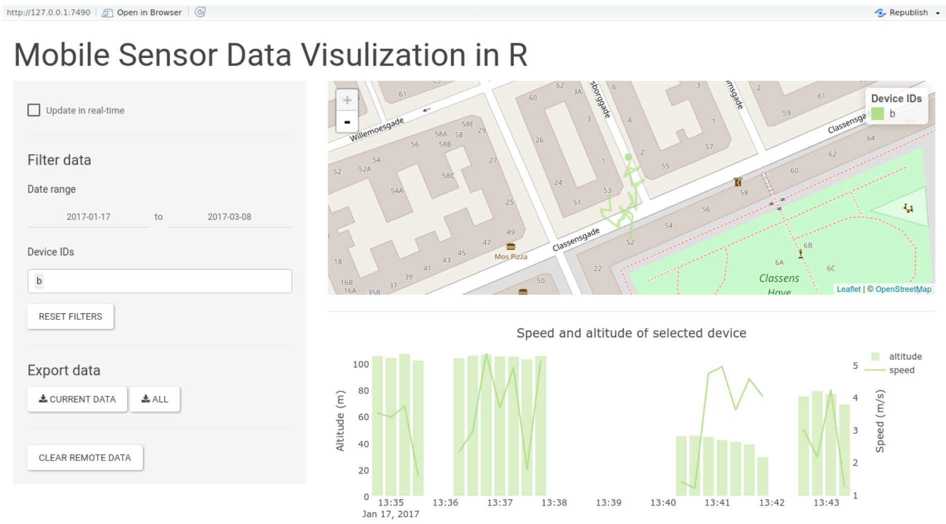

Figure 3.4: Shiny Application: user interface layout

The figure above shows an example session, with all the previously described controls. The following UI widgets have been employed:

`checkboxInput()` check-box with optional default state

`dateRangeInput()` dates range selector, with integrated calendar picker; maximum as well as default start and end dates can be set

`selectInput()` combo-box with optional multiple selection support

`actionButton()` common press-button

`downloadButton()` download button

Each widgets takes an ID and a label as mandatory parameters. Commonly used formatting tags such as `hr`, `p`, and `h` headings are also available through Shiny.

3.3.3 Data collection

The sensors data from the users' mobile phones is queried within a reactive expression, which invalidate itself every five seconds. This triggers a continuous background update of the data, which is then propagated through all the reactive elements.

The data is accessed using the Google Sheet APIs and the `googlesheets` package: a file object is created from the document key, which is then used to read a range of columns. The result is stored in a data-frame, where a new column is added, containing the date and time in POSIX format.

3.3.4 Data filtering

Once the row data have been collected, the Shiny back-end proceeds to reevaluate the filtering, which is based on the device IDs and date range chosen in the interface. The resulting subset of entries is then sorted by device ID and time, in order to simplify its visualization.

These operations are performed usings the `dplyr` package, which provides useful tools for data manipulation. In particular, the `filter()` and `arrange()` functions have been employed.

3.3.5 Map visualization

The map integration package `leaflef` allows the developer to include custom elements, including markers, polygons, and lines. Though a simple data frame with latitude and longitude values in separate columns is enough to describe most geometrical entities, more complex data formats are available as inputs, provided by the package `sp`, which contains classes and methods for spatial data.

The *Spatial Lines Data Frame* class has been used to plot several polylines and color them accordingly. The data has been encapsulated into the correct format — comprising a list of coordinates and an ID field — using the R function `lapply()`, which performs a user-defined function on the elements of an array. It this case, it has been performed on the list of unique device IDs, that have then been used to filter the relevant coordinates and generate *Spatial Line* objects.

In order to avoid reloading the whole map at every update cycle, a reactive *observer* is employed, where only the overlaying elements are redrawn. These include the lines, markers, and legend, which follow a consistent color scheme. The colors are picked by a reactive function with internally uses a discrete palette and performs interpolation when the number of devices to show differs from the number of available colors. Further formatting options, including stroke width and opacity, is also applied.

3.3.6 Plot visualization

Although R comes with several valid built-in plotting libraries and methods, they output static images, which has deemed too limiting in terms of user interaction. It has therefore been chosen to adopt *Plotly*, an online data visualization framework written in JavaScript. Whilst several similarly-aimed tools are available, Plotly provides seamless integration with R and Shiny through its official package, and has recently been released as open source.

The chart featured in the final product has been generated using the following three functions:

`plot_ly()` instantiate a new plot device, with optional dataset parameter

`add_trace()` adds a new set of data to the current plot. It is possible to select between various types of plots and visualizations; in this case, two scatter traces (altitude and speed) have been added and visualized as bars and lines respectively

`layout()` sets up the plot canvas with title, axes names, labels, and grid

Chapter 4

Testing

This chapter will attempt to assess the correct functioning and performance of the developed system, by tracing the current status of accomplishment of each requirement. Moreover, the process of evaluating the time performances of the Google Sheet solution will be described, and its outcomes discussed. Information about the overall performances is available in the product assessment section.

4.1 Acceptance tests

In software and system engineering, acceptance testing is conducted to determine if the SRS (software requirement specifications) are met. For the tests, the following tools have been used:

- A mobile phone with GPS sensor, running Android 5.0.1
- A host machine
- A web browser, i.e. *Google Chrome*
- The developed Android and R Shiny applications
- The Google Sheets document

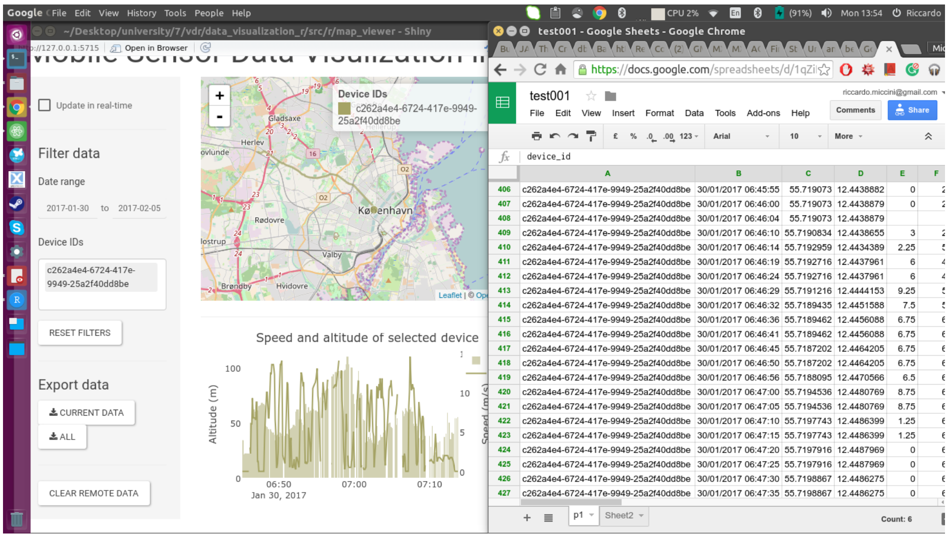

Figure 4.1: Testing: example setup with R application and Google Sheets document loaded in the browser

ID	Description	Related SRS	Espected result	Outcome
T.1	Press button ▶	SRS.F.7-6	The Google Sheets document shows a new row every second	Passed
T.2	Insert number, press button ▶	SRS.F.7-6-1	The Google Sheets document shows a new row every given interval	Passed
T.3	Press button ❚❚	SRS.F.7	The Google Sheets document keeps the same number of rows	Passed
T.4	Verify content of the Google Sheets document	SRS.F.6-4-3-2	Entries in the file should contain latitude, longitude, speed, and altitude	Passed
T.5	Load R application and perform T.1	SRS.F.10-8	The map view periodically shows new path segments	Passed
T.6	Use UI controls to select a single device	SRS.F.11-10-9	The map view only shows selected device; the plot view populates with bars and a line	Passed
T.7	Press button ⭳ *all*, inspect the file	SRS.F.12	The file contains all the GPS data stored in the Google Sheets document	Passed
T.8	Perform T6, press button ⭳ *current data*, inspect the file	SRS.F.12	File contains the GPS data relative to the selected device	Passed

Table 4.1: Testing: functional requirements

4.2 Timing tests

Chapter 5

Conclusions

The project has reached a satisfactory level of completion. As it could be seen in the testing chapter, all of the functional requirements have been met.

Nevertheless, the system is far from optimal and could still benefit from further development. The following sections will elaborate on the flaws of the solution and attempt to provide viable ways of improving them.

As a whole, the project has been a valid and compelling opportunity for exploring the Google ecosystem as well as the Android and R development environments. Although the tools used have not been fully mastered, the principles acquired over the course of the project provides a solid foundation upon which future knowledge will rest.

5.1 Product assessment

The final prototype works as intended and meets all of the functional requirements. The major usage limitation is set by the Google Sheets as storage back-end, whose API provides its content in JSON format. The amount of data that needs to be transmitted at each update cycle grows considerably, and reaches the order of magnitude of few megabytes for a thousand entries. Moreover, in order to be accessed by mobile devices and the R application, the Sheets document has to be publicly available through its URL, which may raise privacy and security concerns.

The Android app requires to be running in the foreground for location data to be collected and submitted, which may not be the intended use case. The version of the app used for testing doesn't store its device ID permanently, meaning that it would change whenever the app is closed and launched again. This limitation has been addressed and fixed in one of the repository branches, using Android data storage guidelines.

As far as the R Shiny application is concerned, its responsiveness drops during the first update cycle, causing the web page to freeze. Although more investigation might be necessary, this is believed to be caused by the allocation of the memory resources. Nevertheless, the application has been tested and proved functional on several hosting platforms, including *Shinyapps.io*, local hosting, and an Ubuntu-based R server.

5.2 Future improvements

As mentioned above and throughout the design chapter, the Android app should be capable of running in the background. This issue has already been addressed, though its implementation has only reached a preliminary stage.

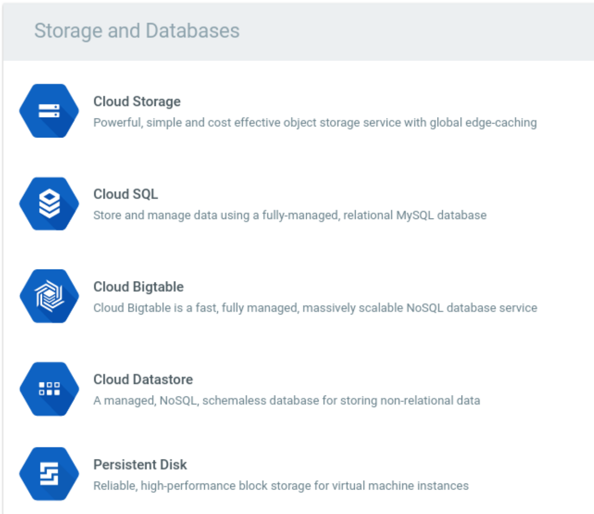

Figure 5.1: Data storage: solutions offered by Google Cloud Platform

Regarding the demanding data transfers, optimization may be achieved by adopting a different storage back-end. Google Could Platform provides several solutions for storing data in a database-like fashion. In particular, the *BigQuery* and *Cloud SQL* products may be suitable for the purpose. Other cloud services providers like Amazon or Microsoft have their own counterparts, namely *AWS Redshift* and *Stream Analytics*. Another interesting option may be one offered by *Google Firebase Platform*, which integrates a cloud-hosted, real-time database.

Bibliography

[1] *Aapplication layout guide.* URL: https://shiny.rstudio.com/articles/layout-guide.html.

[2] *Android Activity.* URL: https://developer.android.com/reference/android/app/Activity.html.

[3] *Android AsyncTask.* URL: https://developer.android.com/reference/android/os/AsyncTask.html.

[4] *Android Quickstart.* URL: https://developers.google.com/sheets/api/quickstart/android.

[5] *Android Service.* URL: https://developer.android.com/reference/android/app/Service.html.

[6] *Git - everyday documentation.* 2016. URL: https://git-scm.com/docs/everyday.

[7] *Google API Client.* URL: https://developers.google.com/android/reference/com/google/android/gms/common/api/GoogleApiClient.

[8] *Google Sheets API.* URL: https://developers.google.com/sheets/api/.

[9] *GoogleAccountCredential.* URL: https://developers.google.com/api-client-library/java/google-api-java-client/reference/1.20.0/com/google/api/client/googleapis/extensions/android/gms/auth/GoogleAccountCredential.

[10] *googlesheets Basic Usage.* 2016. URL: https://rawgit.com/jennybc/googlesheets/master/vignettes/basic-usage.html.

[11] *Introduction to dpyr.* URL: https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html.

[12] *Leaflet for R.* 2014. URL: https://rstudio.github.io/leaflet/.

[13] *plotly R chart attribute reference.* 2014. URL: https://plot.ly/r/reference/.

[14] *plotly R Library*. 2014. URL: https://plot.ly/r/.

[15] *Reactivity: an overview*. URL: https://shiny.rstudio.com/articles/reactivity-overview.html.

[16] *Receiving Location Updates*. URL: https://developer.android.com/training/location/receive-location-updates.html.

[17] *Reference manual for sp - Classes and Methods for Spatial Data*. 2016. URL: https://cran.r-project.org/web/packages/sp/sp.pdf.

[18] *Shiny, a web application framework for R*. 2012. URL: https://shiny.rstudio.com/.

[19] *Storage Options*. URL: https://developer.android.com/guide/topics/data/data-storage.html.

[20] *THE LATEXTOOLS MANUAL*. 2016. URL: https://github.com/msiniscalchi/atom-latextools/blob/master/MANUAL.md.

Appendix A

Attachments description

The following table contains a description of the material found in the hand-in support. The `./src/android/LocationTracker/` folder is a git repository with two branches: the one called `android_bg` contains a version of the app with the location tracking runnig only in foreground. The `master` brach contain a stub of the app which uses a background service, currently under development.

Directory or file	Description
`.`	Attached archive
`./src/r/map_viewer/`	R Shiny application
`./src/r/time_tests.r`	Data timing R script
`./src/android/LocationTracker/`	Android project
`./poster/`	Poster LaTeXcode
`./report/`	Report LaTeXcode
`./report/figures/rd_src.html`	Reactive diagram code

Table A.1: Attachments: folder structure