Smartphone App Profiling

Bachelor Thesis

Nicolas Forster

forstern@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Tobias Langner
Prof. Dr. Roger Wattenhofer

September 8, 2014

Abstract

With the development of different operating systems for mobile devices and with the launch of different mobile application stores, the development of mobile applications gained a huge increase in its popularity, leading to that there are masses of mobile applications. But with this variety of mobile applications comes a big problem for the user: how to find an application the user likes and needs? Of course the user could take the ratings offered by the mobile application stores to decide if he wants to download an application, but these ratings suffer from different problems, making them to vague to be helpful.

To overcome this problem do we want to develop ratings and recommendations based on the usage data of the users, without any explicit input by the user. The first problem we have to solve is to get the usage data of the user. Therefore we developed *AppMin*, an application which monitors the usage behaviour of the user and sends the collected data to a server. We present in this thesis the design and implementation approaches taken to implement the whole system. Additionally, we implemented a first approach of implicit ratings and we designed ways to visualise the usage data of the user.

Contents

Abstract **ii**

1 Introduction **1**

1.1 Background . . . 1

2 Related Work **3**

2.1 AppAware . . . 3

2.2 AppJoy . . . 3

3 Project Overview **5**

3.1 System Overview . . . 5

3.2 Client Overview . . . 6

3.3 Server Overview . . . 6

4 Implementation **8**

4.1 The Android Platform . . . 8

4.2 The Client – AppMin, The App Administrator . . . 9

4.2.1 Monitoring the Usage Data . . . 9

4.2.2 Sending and Retrieving Data from the Server . . . 11

4.2.3 Data Storage . . . 12

4.2.4 Propose Applications for Removal – The Removal Service 14

4.3 The Server . . . 15

4.3.1 The Server Logic . . . 15

4.3.2 Security . . . 16

4.3.3 Data Access and Storage . . . 17

4.3.4 Collecting Data from the Google Play Store . . . 17

4.4 The Rating Algorithm . . . 19

5 Visualisation **20**
5.1 Overview 20
5.2 The Pie Chart 21
5.3 The Stream Chart 22
5.4 Weekly Usage Graph 24

6 Discussion **25**

7 Future Work **26**

Bibliography **27**

A Appendix A **A-1**

Chapter 1

Introduction

1.1 Background

With the release of the iPhone in the year 2007 and the launch of the *AppStore* in the year 2008, Apple laid the foundation for the distribution of mobile applications. In the following years, Google and Microsoft successfully developed their own mobile operating system and launched the *Google Play Store* and the *Windows Phone Store* respectively. These two stores boosted the popularity and the usage of mobile applications even more. Currently there are around 1.3 million applications[1] [1] in the Google Play Store and 1.2 million applications[2] [2] in the AppStore available.

The stores mentioned above play an important role in the distribution of applications by offering possibilities for the user to browse applications in different categories, search applications by keyword and install applications directly on their phone. As well do these store offer a possibility to rate applications.

Considering the huge number of available applications, it is still challenging to find applications that serve the purpose the user is looking for. The next problem for the user is, that he needs to choose one application out of all the obtained applications. Of course, the ratings offered by the stores could be used to choose an application, but often these ratings do not really tell what the user wants to know, because they are subjective, meaning the personal opinion of the users matters a lot. Another problem is that user quite often do not rate the applications at all, because they are not willing to rate them or they simply forget to rate the applications. These properties make the ratings too vague to be representative.

An idea to deal with the problems that arise with the usage of explicit ratings is to derive implicit ratings, that do not require any explicit user input. One

[1] August 2014
[2] June 2014

approach to generate such implicit ratings for applications, is to monitor the currently running applications on a device in order to derive which applications are actually used an which ones remain dormant. Based on these informations, one can then generate implicit relatives ratings for these applications. After deducing that two applications A and B serve the same purpose, not an easy task at all, one can use the gathered usage information to generate ratings of the form "A is better then B, because A was used more often than B by different users".

The first problem one has to solve, to get these ratings is to collect the above mentioned usage data, which is the starting point of this thesis. I developed a system to collect usage data from mobile phones, with an Android application called *AppMin – The App Administrator* as the core of the system. The application monitors the usage of all installed applications and sends the collected data periodically to a server. This data then will be used in future studies, in particular to generate the implicit ratings mentioned above. As an additional feature, the aggregated personal usage data is visualized in different graphs. From these graphs the user could learn a lot about his usage behaviour of a certain application and his mobile phone itself. In addition *AppMin* repeatedly proposes to uninstall applications that were not used during a certain time period. This helps the user to keep his phone clean. A first approach of implicit ratings is taken, based on the studies of Giradello et al. [3]. These ratings are accessible through the Android application *AppMin*.

Chapter 2

Related Work

In this section, we briefly introduce two systems that collect data form the user and recommend applications based on this data. We show as well how our system differs from the two introduced systems.

2.1 AppAware

AppAware [3] tries to exploit the serendipitous effect in application discovery. This means it tries to help the user to find a new application by happenstance. Therefore AppAware displays a real time stream of application installs, updates and removals to the user from other users in the same area. In this way the user is aware of the applications used by other users.

As an additional feature AppAware calculates the *"acceptance rate"* of an application based on the actions related to the application (install, update and removal). In contrast, our system does not only take installs, updates and removals into account to calculate new ratings, but also how long and how often a user actually used the application.

2.2 AppJoy

AppJoy [4] makes personalized application recommendation based on how the user uses an application. Based on these usage records, AppJoy uses an collaborative filtering algorithm to calculate the individual recommendations for an particular user.

The AppJoy system sends their usage data packed up in hourly intervals, meaning before sending the usage record they put together the single usage intervals occurred during a hour in to a single sum. *AppMin* in contrast sends each usage interval without summarizing over a timer interval.

We believe by doing so we are able to generate more accurate ratings. Additionally, we will not use any kind of collaborative filtering to calculate new ratings. Instead we develop in further studies own ratings and recommendations based on the collected usage data.

Chapter 3

Project Overview

In this chapter we explain the concepts of the system design. Therefore, we start of with an brief overview over the system design and continue after with the two main parts the system consists of: the client *AppMin* and the server. A detail explanation of the implementation is later given in chapter 4.

3.1 System Overview

This section gives an brief overview about the system design chosen to implement the given task. Our system consists of two parts, the Android application and the server. We implemented the client-server architecture as shown in Figure 3.1.

On the client side, the application *AppMin* monitors the usage data of each application that runs in the foreground. The collected data is then periodically uploaded to the server. In addition, the client is responsible for requesting rated applications for certain categories and to display the usage data and its visualisation to the user. The application proposes to remove not used applications on a daily basis and notifies the user through a notification.

On the server side a web service is deployed to offer access to the server through HTTP. The server handles all incoming requests sent by *AppMin* and reacts depending on the request. The two main tasks of the server are to store the sent usage data to the database and to fetch rated applications from the database. The server updates these self generated ratings for the applications on a daily basis. Another task of the server, is to collect all available Android applications. To do this task, the server maintains a crawler, which crawls the Google Play Store.

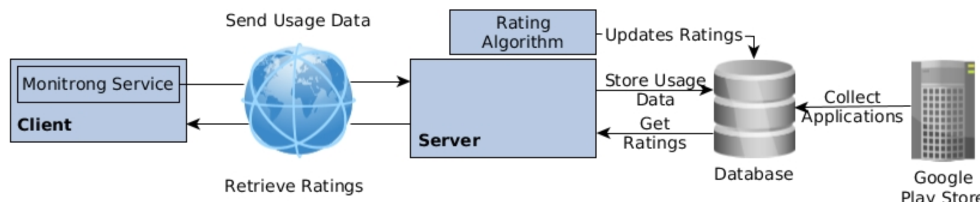


Figure 3.1: Diagram that represents the client/server design approach.

3.2 Client Overview

To show how the application works and how it looks like, the usual work flow and the Graphical User Interface are presented in this section.

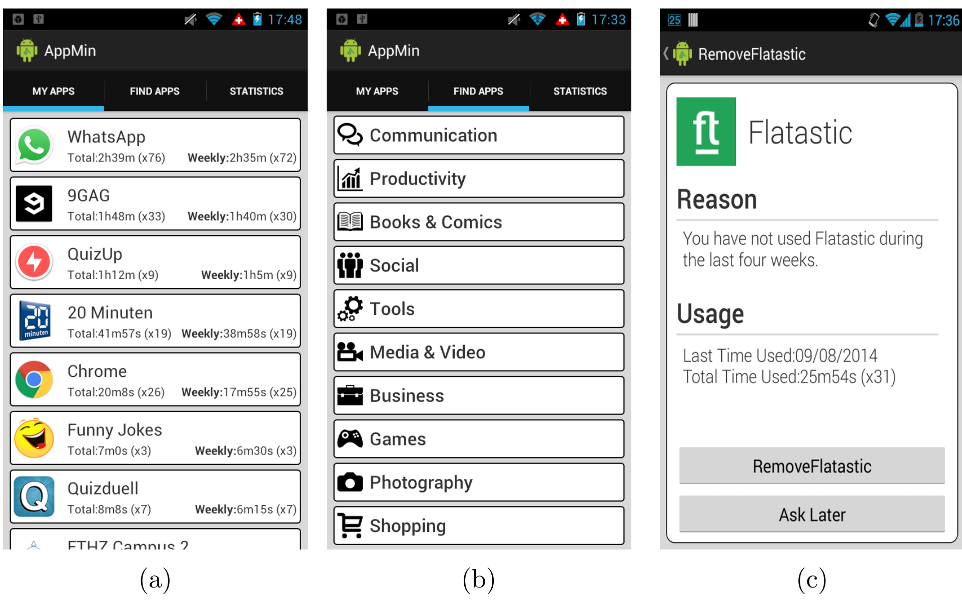
As soon as the user starts the application, the home screen is showing the *"My Apps"* tab seen in Figure 3.2a. This tab displays all currently installed applications and their usage time. If the user clicks such an application a list of actions regarding the application pops up. The user has the choice to either open the application, to show the weekly usage graph of the application or to remove the selected application from the phone. To the right of the *"My Apps"* tab the user can find the *"Find Apps"* tab shown in figure 3.2b. There a list of categories is displayed and if the user selects a certain category, rated applications of this category are shown. An example can be found in Figure 4.3. By clicking on a rated application, the Google Play Store is opened, so that the user is able to install the application directly from *AppMin*. The third and last tab is the *"Statistics"* tab. This tab includes the Pie Chart and the Stream Chart. These charts are explain further in chapter 5.

If *AppMin* proposes an application for removal, it first displays a notification in the status bar of the mobile phone. After taping the notification the user is led to a list of all applications that are proposed for removal. If the user wants to have more informations, why the application is proposed for removal he can call up a more detailed view for an application, as shown as in Figure 3.2c.

3.3 Server Overview

To show how the server works, the usual work flow of the server is presented in the current section.

The most important task of the server is the handling of incoming request. As soon as a request arrives the web service, it is parsed and split up into the different parameters. According to the parameters sent by the client, actions are



(a) (b) (c)

Figure 3.2: Here (a) shows the “My Apps” tab, (b) shows the “Find Apps” tab and (c) shows the detailed removal screen.

taken. These action can be split up in to two main categories: reading data from the database or writing data do the database. All actions are described in detail in 4.3.1.

In parallel to the web service runs a crawler which crawls the Google Play Store, to add missing applications to the database. The informations that are crawled are needed to have an initial value for the new ratings (see section 4.4) and later to adjust the newly generated recommendations. The response time of the server is faster, if the information sent to the client could be read out of a local database instead of the Google Play Store. The crawler is described in detail in section 4.3.4.

The last task that the server has to handle, is to calculate the current implemented ratings on a daily basis. How these ratings are calculated is explained in section 4.4.

Chapter 4

Implementation

4.1 The Android Platform

To implement the client side of the system, the *Android* platform was chosen [5]. Android is an open source operating system based on a Linux system, developed by Google to run on mobile devices, such as tablets or phones. Once installed each Android application lives in its own sandbox, meaning that each application runs in its own processes. The Android system implements the *principle of least privilege*. That means, each application has the permission to only use the system parts, it really needs to do its work. If the application needs access to other components one has to define these additional permission in the *manifest file*, a central file, where all components of the application are defined.

An Android application consists of the following four major components: *activities*, *services*, *broadcast receiver* and *content providers*. An activity provides a part of the Graphical User Interface of the application. It handles also the interaction between the user and the mobile phone by reacting to touch gestures, voice commands or similar actions. As soon as the application is started, the main activity is created. After the initialisation each activity is able to create new activities, making the move from one activity to another really simple. As soon as a new activity is displayed, the current activity goes to the background, waiting for its reactivation or gets killed by the systems memory manager. The full life cycle of an activity is shown in Figure 4.1. A service does not have a Graphical User Interface and runs in background, even if the parental application is not in the foreground. Services are normally used to perform long-running operations, such as large calculations or retrieving data from the web, and to prevent the blocking of an activity during such an operation. Other components such as activities or broadcast receivers are able to start a service. A broadcast receiver responds and reacts to system-wide system events, such as the installation of an application. To react to a certain event broadcast receivers normally start a service, as they are intended to do a minimal amount of work. Content providers are not used during the thesis and thus not explained further.

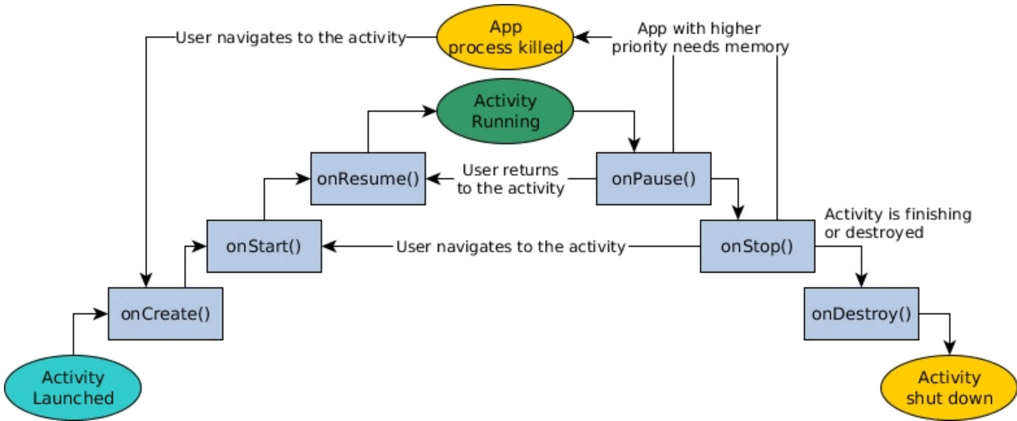


Figure 4.1: The life cycle of an activity and the most important state transitions are shown here.

4.2 The Client – AppMin, The App Administrator

This section presents how *AppMin,* works and how the client side of the whole system is designed. In the first subsection, the data collection service is presented. In the following subsections the removal service and the networking part of the client are described in detail. A complete overview over all components of the client is given in Figure 4.2.

The application is designed to support Android devices, that at least run Android version 4.0, also called *Ice Cream Sandwich*, with the Android API 14. This is necessary as the application uses features introduced in this API such as tabs and fragments. As more than 85% of all Android mobile devices run Android version 4.0 or higher[1] [6], we decided that supporting API 14 and higher is sufficient for our purposes.

4.2.1 Monitoring the Usage Data

To collect the required data that is needed to generate new ratings, *AppMin* has to monitor when an application A is started and how long A remains active. In addition, *AppMin* needs to keep track of which applications are installed on the mobile phone, to decide which applications are displayed in the application usage list.

Applications, flagged as system applications are not monitored, as they are either not downloadable in the Google Play Store, such as custom applications from the manufacturer, or not relevant for the usage statistics, for example the

[1] August 2014

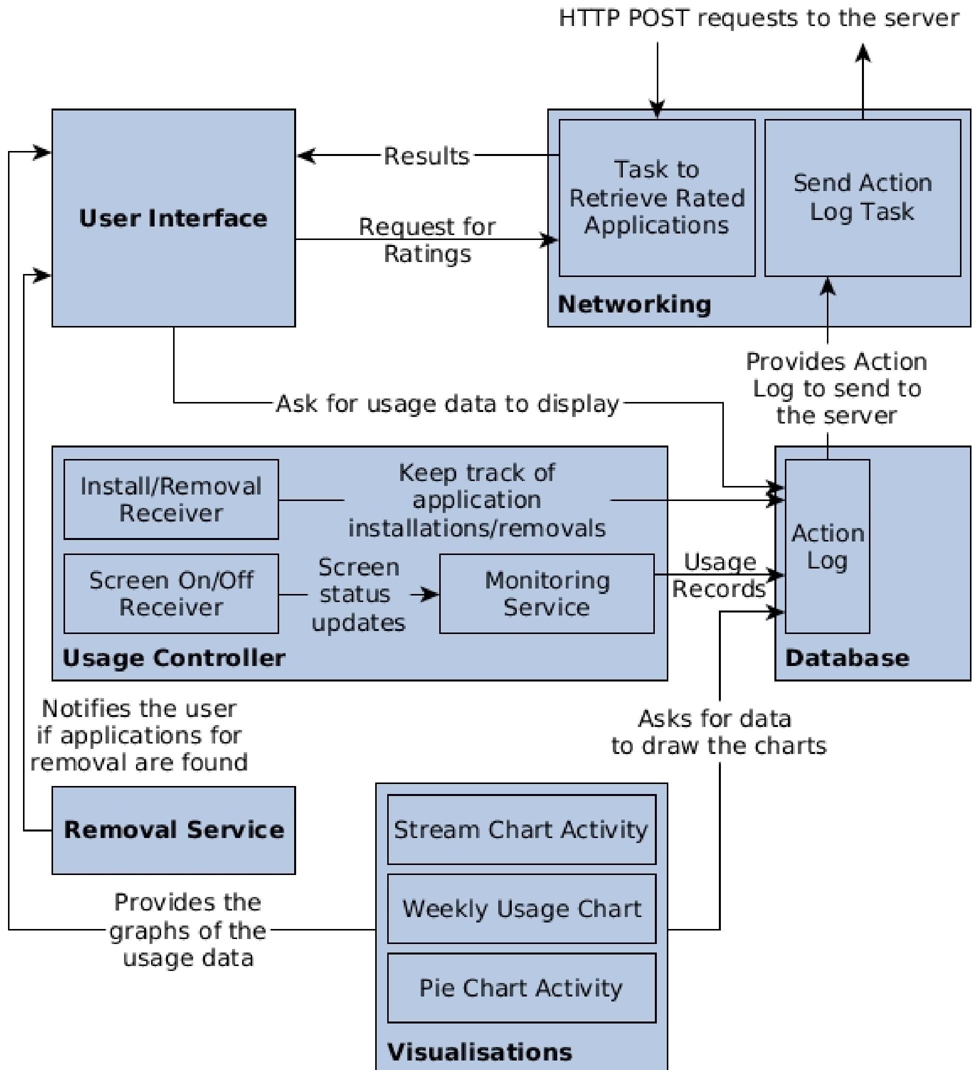


Figure 4.2: The six main parts of the client and how they interact are shown in this figure.

GPS service or the Graphical User Interface. Unfortunately, also a few normal user-applications by Google are flagged as system applications, such as *Chrome* or *GMail*, while obviously still being relevant for monitoring. Thus, such applications need to be added manually to the list of monitored applications.

Android does not provide any mechanics or events that notify an application if the foreground application changes. Hence, *AppMin* deploys a background service that checks periodically if the foreground application did change. As soon as it detects that a new application got active, *AppMin* writes a start event with the current unix timestamp to the *action log*. An corresponding end event is written to the log for the application that was active before. The periodicity can be adjusted to a five, ten or twenty seconds interval by the user in the application settings. Of course, such a periodical checking of the active applications leads to an inaccuracy, as it is possible that multiple applications are started during such an interval without getting noticed. However, the intervals are quite short and should capture most of the usage duration of an application.

Because a default background service gets paused, while the phone is inactive (the screen is turned off), a broadcast receiver is deployed, that captures the *"Screen On/Off"* event, sent by the Android operating system. As soon as the screen gets turned off, *AppMin* handles this as an end event for the current active application. We are aware of the issue that applications that keep running in the foreground, even if the screen is turned off, such as a music player, are monitored as inactive, even though they are active. Currently, we did not find a solution to overcome this problem.

To keep track of the currently installed applications, a broadcast receiver captures if a *"Package Installed"* or *"Package Removed"* event is send out by the operating system. As soon as such an event is detected a corresponding event for installation or removal of an application is written to the action log.

4.2.2 Sending and Retrieving Data from the Server

To send and retrieve data from the server, *AppMin* has to be able to communicate with the server over the internet. *AppMin* contains three different main parts that need to interact with the server. These parts are to retrieve a list with rated applications, to retrieve the according application logos and sending the action log to the server. The structure and the actions taken on the server side is explained in section 4.3 and thus omitted in the following explanations.

In general, the client uses HTTP POST requests, which are sent to the servers web service. For most of the communication the already existing HTTP libraries of Android are used.

To retrieve rated applications *AppMin* sends an according request, containing the category for the requested applications, the user name and the password. These information are wrapped in a *JSON object*, that is serialized to a string using Gson [7]. The *JavaScript Object notation*, short JSON, is used to serialize objects, to be able to exchange data between different applications. The servers response contains a JSON list, containing the fifty top rated applications for a certain category, which is transformed to a Java list, again using Gson. The list contains *Application* objects. An *Application* object is a wrapper containing the application name, the application ID, the relative path of the application icon on the server, the developer name and the rating. All these informations are then displayed to the user as shown in Figure 4.3. In a next step, the application icons are requested asynchronously from the server, using the third party library *"Universal Image Loader for Android"* [8]. We decided to download the application icons asynchronously to avoid long waiting times. The list of applications is relatively small compared to the application icons and can be transferred relatively fast. In this way it is possible to show the user the important results after only a short waiting time.

The most important part in the communication with the server, regarding this thesis, is to send the action log to the server. To send the action log periodically to the server, an alarm is scheduled which invokes a service that checks if all conditions are met to send the new parts of the action log, that were monitored since the last time the action log was sent. The service first checks if the mobile phone has a connection to the internet. Second the service tests if the user name and password were received properly. As a last point the server checks a user setting indicating whether the action log can be send at any time or only if a Wifi-connection exists. To increase the probability that the action log is sent regularly even with a Wifi-connection only deployed rarely, a broad cast receiver is registered to the *"Wifi Stated Changed"* event. This broadcast receiver then starts the service described above manually. To create the message that is sent to the server, first a list containing *ActionLog* objects is prepared. An *ActionLog* object is a wrapper that contains one row of the action log. This list is then serialized to a JSON string using Gson. This string is sent to the server.

4.2.3 Data Storage

In this subsection we briefly explain, how the data used by *AppMin* is stored on the mobile phone. *AppMin* needs to manage two kinds of data. The first kind of data is all the data regarding the applications installed on the mobile phone. The second kind are the application icons that were downloaded for the recommendation section.

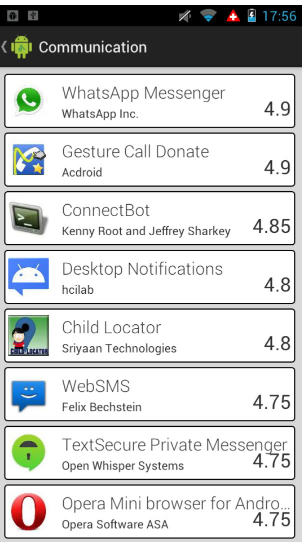


Figure 4.3: Here the rating screen with all retrieved data is shown.

The application icons are either stored on the internal storage or on the external SD-card, if the mobile phone has one. The whole storage management is done by third party Android library *"Universal Image Loader for Android"* [8].

The storage of the data regarding the usage monitoring of the applications is a little bit more complex. All the data is stored in a relational database, using the *SQLite* database management system that comes with the Android operating system. The database itself consists of the following six tables containing different informations:

- **Application Information Table:** This table contains all applications currently installed on the mobile phone as well a flag which indicates if the application is currently active.
- **System Application Table:** This table contains all applications that are flagged as system applications and that are not monitored by *AppMin*. Applications that are manually added to the *Application Information Table* are removed from this table.
- **Action Log:** This table contains the whole action log, since the installation of *AppMin*. It contains tuples of the form *(appId, action, timeOccurrence, timezone)*. The *action* field indicates what happened with the application, the *timeOccurrence* field contains the according Unix time stamp and the *timezone* field indicates the time zone, in which the action took place.
- **Total Time Used Table:** This table contains for each application, which was installed on the mobile phone, how long and how often it was used in

total. In addition it contains the date when the application was removed the last time. This is important to detect if an application only was updated or reinstalled after a long time. An update causes the application to be removed and installed in a very short time (usually less then a minute). If the total usage time would be stored in the *Application Information Table* it would be lost after each update. Now the total usage time only gets reset, if the new install date and the last removed date are more than a day apart of each other. We assumed that if an user reinstalls an application, he does not want to know how long he already has used the application the last time it was installed.

- **The Judgement Table:** This table contains if a certain application should be generally taken in to account for proposals, as described in section 4.2.4, or not.
- **The Flag Table:** This table contains additional information that did not fit in another table.

4.2.4 Propose Applications for Removal – The Removal Service

To keep the users mobile phone clean from rarely used applications, *AppMin* offers a functionality that periodically checks if there are applications that have not been used during some period of time. *AppMin* solves this task by having a service periodically running in the background. The periodicity of the checks can be manually adjusted by the user to be either daily or weekly.

Internally the service uses a self implemented framework that consists of so called *Judge* object. A *Judge* object decides for an application if it should be proposed for removal or not. By implementing a *Judge* object, we can adjust under which condition a application gets proposed. Currently only *Judge* objects are implemented, which check if the application was used during the last week or the last two, four or eight weeks. An other idea for an *Judge* object is that if two applications that serve the similar purpose are installed, the *Judge* object proposes the less used application.

As soon as *AppMin* found an application that satisfies the conditions for a removal, *AppMin* notifies the user through a notification in the status bar of the mobile phone, leading to a list of all application proposed for removal. In a detailed view, shown in Figure 3.2c, the user has the choice to either remove the application, or to adjust the behaviour of the removal service. By clicking on the *"Ask later"* button, the user delays the next proposal for some weeks. The new delay is twice as much as it already was e.g. if the application was not used during the last two weeks, the next time it gets proposed is in four weeks. If the user hits the *"Never ask again"* button, the application is excluded from future proposals. This is useful to avoid that widgets, which have no foreground time,

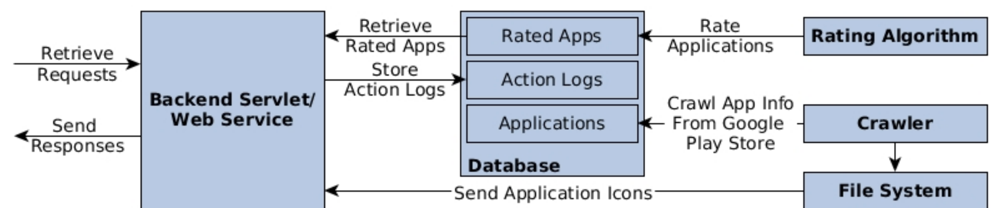


Figure 4.4: The main parts of the server and how they interact are shown in this figure.

or applications that are rarely used but still essential, such as phone books, are not periodically propose for removal.

4.3 The Server

This section presents how the server side of the whole system works and describes the most important features. In the first subsection, a detailed explanation of the server logic is given. Further subsections describe how the data is stored and accessed, how requests are authenticated and how the Google Play Store is crawled. An overview of all server components and how they are connected is given in Figure 4.4.

The front end of the server is implemented using *Java Servlets*. Java Servlets are Java objects that handle HTTP requests. They run inside *Servlet container* on a *Java web server* and they are part of a *Java Web Application*. The system uses the *Apache Tomcat Server* as the *Java Web Server* [9]. Servlets implement methods that respond to an HTTP request and prepare the HTTP response, which is then sent back to the client. The HTTP request is sent to the web server and gets redirected to the responsible servlet [10].

The web service running on the server is implemented in a stateless manner. A web service is stateless, if it does not hold any information from request to request. This simplifies the implementation of the web service a lot. As a consequence, a request must contain all information necessary to serve the request.

4.3.1 The Server Logic

After the request got interpreted and any data got fetched out of the request the appropriate actions according to the parameters are executed. Currently, the server is implemented to be able to serve the following five different request: send a new user name and password, receive a crash report, send a list of rated

applications according to a category, send the application icon to an application, receive an action log. These five actions should be now describe in more detail.

The server has to generate a new user name and a new password every time a user installs *AppMin* on his phone. After a the first time registration the request have to contain this pair for authentication. More thoughts to the authentication mechanism are discussed in section 4.3.2. The user name and password are then wrapped up to a JSON object which is serialized to a string using Gson. This string, is then sent back to the client.

To be able to debug the *AppMin* after the release, *ACRA, Advanced Crash Reports for Android* [11], a third party library, has been used to generate more precise crash reports. This is necessary, as Google does not provide very helpful and precise crash reports. The crash report is sent as JSON string to the server and is then processed and stored to the file system. Additionally a mail is sent to the developer, to notify him, that a crash happened.

If a list for rated applications is requested, fifty rated applications are fetched from the database and sent back to the client. Again JSON is used to prepare the list to be sent to the client.

If the client requests a application logo for a certain application, the server first builds up the path to the application. After fetching the icon from the file system, it is scaled down to the requested size to match screen settings of the client. By scaling down the icon on the server side, one has to send less data over the network, increasing the transmission rate. Additionally less work on the client side has to be done, as the icon already matches the screen settings of the client.

If the client sends an action log the JSON string is translated back to the actual Java list using Gson. Then all list entries, containing *ActionLog* objects (see section 4.2.2) are stored to the database.

4.3.2 Security

An authentication mechanism has been deployed to prevent a malicious user to spam faked action logs to the server, so that he could boost the usage time of his application. Therefore, as soon as the user starts up *AppMin* the first time, a random 32-bit user name and password are generated and send back to the client. Additionally, both are stored in the database to be able to authenticate a request. The password is hashed using SHA1 provided by MySQL, so that if the database is hacked not all passwords are revealed. After the registration, each request has to contain the user name and the password. If the pair does

not match or if the user is not found in the database the request is just ignored.

If during the registration something goes wrong, like a crash or a loss of the internet connection, the user name and password are requested again, as soon as the user wants to use a feature that needs communication with the server, such as requesting new rated applications or sending the action log to the server.

4.3.3 Data Access and Storage

All the data that is collected either from the Google Play Store or from the action logs is fully structured meaning that one can use a relational database to store the data. Hence we decided to us MySQL as a relational database management system and its underlying database. The *Entity/Relationship-Model* on which the database is build can be found in Appendix A.

The most important tables are quickly described below:

- **actionLogs table:** The table contains all the collected action logs sent by the user. Additionally the user ID is stored with the action logs.
- **applications table:** This table contains all crawled applications from the Google Play Store.
- **similar table:** This table contains all applications that are stated as similar by Google. Maybe this data could be used in terms with cluster analysis to get a measure about the similarity of two applications.
- **user table:** This table contains all users that downloaded *AppMin*. As it is not possible to detect, if *AppMin* is removed form a mobile phone, even users that have not installed the application any more are listed in the table.

To access the data, a *Data Access Object*, short *DOA*, design pattern is used. The *DOA* provides a interface to the used database and contains all important methods to interact with the database. This pattern separates the application layer from the persistence layer. Additionally it is possible to easily exchange the underlying database in case an other database system should be used.

4.3.4 Collecting Data from the Google Play Store

To collect the application data from the Google Play Store, a crawler and a scraper have been written to fulfil this task.
In general a crawler is a Internet bot that scans the World Wide Web and follows the links it encounters during the scans. Given a set of starting URLs, the crawler works as long as it encounters new links to other web pages.

In our case, we restricted the crawler to the Google Play Store and to only follow links that either lead to another application or to a developer page, where all applications of a developer are listed. A scraper is a bot which is able to extract information from a website. It extracts all desired information form an applications Google Play Store site. In our set up the crawler and scraper are working together, in the way that the crawler passes the crawled web pages to scraper, so that the scraper can read out the relevant information.

To implement the crawler and the scraper, we used *Scrapy* [12], an application framework written in Python that offers all the functionalities to implement both. In the following paragraph we describe how Scrapy works and introduce the most important concepts.

The first step is to define the data we want to scrap, using a Scrapy item. This is a wrapper that contains all the fields one wants to scrap. The second step is to set up the crawler and the scraper. Scrapy combines these two concepts as the crawled web page gets immediately harvested by the scraper. The scraper then stores the found links for the crawler in to a list, so that the crawler can continue its work with a new page. To extract the desired data, *XQueries* can be used, as an HTML document is just a special case of an XML document. The extracted data is then stored in to the items, described above. After an item has been scraped, it is sent to the *Item Pipeline* which processes the item through several stages. Each stage of the Item Pipeline is a Python class, implementing a method that preforms an action with the current item. In the project we used two stages which the item has to pass, the *ImagePipeline* and the *StoragePipeline*. The ImagePipeline, offered by the Scrapy framework, downloads the application icon of the scraped application and stores it to the file system. The StoragePipeline stores the scraped item into the database table *"applications"*.

During the scraping phase as much information as possible was scraped out of the description page of an application. The most important and not self explaining fields that were scraped are listed below:

- **appID:** The appID is the unique identifier of an application.
- **avgScore:** This field holds the average score that the application received in the Google Play Store. This could be used to compare the new ratings and recommendations to the ones of the play store.
- **description:** This field contains the Google Play store description of the application. Could be used to say something about the similarity of two applications.
- **price:** This filed represents the price the application has in the Google Play Store. The currency of the price is CHF.

4.4 The Rating Algorithm

To be able to justify the collection of the usage data on a central server, a first approach of application ratings has been implemented. This first approach of the ratings is based on the studies of Girardello et al.

First we weighted the *last* event for an application *app* for each user u and came up with the formula shown in equation 4.1.

$$last(app, u) = \begin{cases} 0 & \text{if last event of user for app = removed} \\ 100 & \text{if last event of user for app = installed} \end{cases} \tag{4.1}$$

Then we tried to rate the applications using the equation 4.2, introduced by Girardello et al. [3], where U is the set of users that have at least one action taken with the application *app*.

$$rating(app) = \frac{\sum_{user \in U} last(app, user)}{|U|} \tag{4.2}$$

However, this formula lacks a jump start feature, meaning it returns zero if no users have installed the application *app*. To prevent zero valued ratings for an application, we modified the formula in 4.2 to return a non zero value if no users have installed the application *app*. Therefore the rating we obtained from the Google Play Store, scaled on a range form zero to one hundred, was added to the formula as shown in equation 4.3. Adding the Google Play Store rating (gpsr) to the rating formula in the way as it is done, returns the Google Play Store rating if no users have installed the application. As more users install the application *app* the rating is pulled away from the Google Play Store rating to the rating in equation 4.2.

$$rating(app) = \frac{gpsr + \sum_{user \in U} last(app, user)}{|U| + 1} \tag{4.3}$$

Chapter 5

Visualisation

5.1 Overview

In this section we give a brief overview of the visualisations realised in the Android application *AppMin*. The visualisations take the collected usage data as base and illustrate them in different ways over different periods of time. Currently, the following three different charts are implemented: A Pie Chart, a Stream Chart and a chart that visualized the total usage time of an application versus the number of times the application was opened. We present and explain all three charts further in the following subsections.

The goal of these charts is to increase the value the application has for the user. As the applications main goal is to collect data from the user he needs another motivation to keep *AppMin* installed on his mobile phone. Therefore the charts should serve as an eye catcher to attract the user and to provide additional information to him. With the visualisation the user can have a look at his usage statistics of the mobile phone, as well on individual applications. With this additional knowledge he might be able to learn about his usage behaviour.

To simplify the graphic part the Android chart library *MPAndroidChart* [13] has been used. But it was necessary to change some parts of the library as they did not match all requirements needed to realise the visualisations. Especially they drawing of the Stream Chart was implemented newly, taking a simple line chart as base.

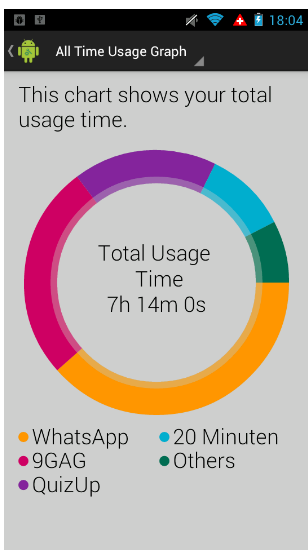


Figure 5.1: Here the current implementation of the Pie Chart is shown, displaying the total usage time of a device.

5.2 The Pie Chart

This section, we describe the main concepts behind the *Pie Chart* shown in Figure 5.1.

A pie chart is a circular chart split in different sections. The size of each section is determined by the data underlying the chart and each section is proportional to the quantity it represents. In our case the data is the usage data of the user over a certain time period. Currently an all-time-usage-Pie Chart or a weekly-usage-Pie Chart are available.

In *AppMin*, each Pie Chart is split up in at most ten different slices, representing the nine most used applications and a “Others” slice, to represent the rest of the applications that are not below the most used applications. Additionally only applications account for more than five percent of the total usage time are shown as a separate slice. Applications with less than five percent are also put in the “Other” slice.

In the middle of the Pie Chart a hole is left out to show additional information. If no slice of the chart is selected the total usage time over the currently selected time period is shown. As soon as a slice is selected the usage time for the particular application is shown.

5.3 The Stream Chart

A *Stream Chart* as shown in Figure 5.2a, is a special case of a stacked chart, introduced by Byron et al. [14]. An example of an simple stacked chart is shown in Figure 5.2b. According to Byron et al., the two main goals a stack chart should fulfil, are to show many different data sets in one graph, while covering their sum as well. In addition the chart should show some aesthetic quality. They discussed the influence of different factors on the aesthetic quality. The factors are the ordering of the data sets used, the colouring of each layer and the choice of the baseline. In our opinion the choice of the baseline influences the appearance the most, therefore we put the most effort into adjusting the baseline. The baseline is the lowest slope of the graph, i.e. for a x value, the smallest y value.

In the end we used the formula of equation 5.1 to calculate the new baseline of the chart for a certain x value.

$$g_0 = -\frac{1}{2} \cdot \sum_{i=1}^{n} f_i \tag{5.1}$$

Here, g_0 is the newly calculated value of the baseline at the according x value, n is the total number of data sets, and f_i is the y value of the i-th data set at the specific x value. This formula leads to that the newly generated graph is symmetrical around the x-axis.

The Stream Graph shows the usage data of the last week. We took hourly intervals to calculate the data points to fill the graph. This leads to 168 data points in total. As the usage data tends to spike heavily we took for a certain data point a moving average over a seven hour time window to be displayed in the actual graph. This leads to a flattening of the graph. A graph that takes only the current data point in to account is shown in Figure 5.2c. There the heavy spikes of the graph could be seen.

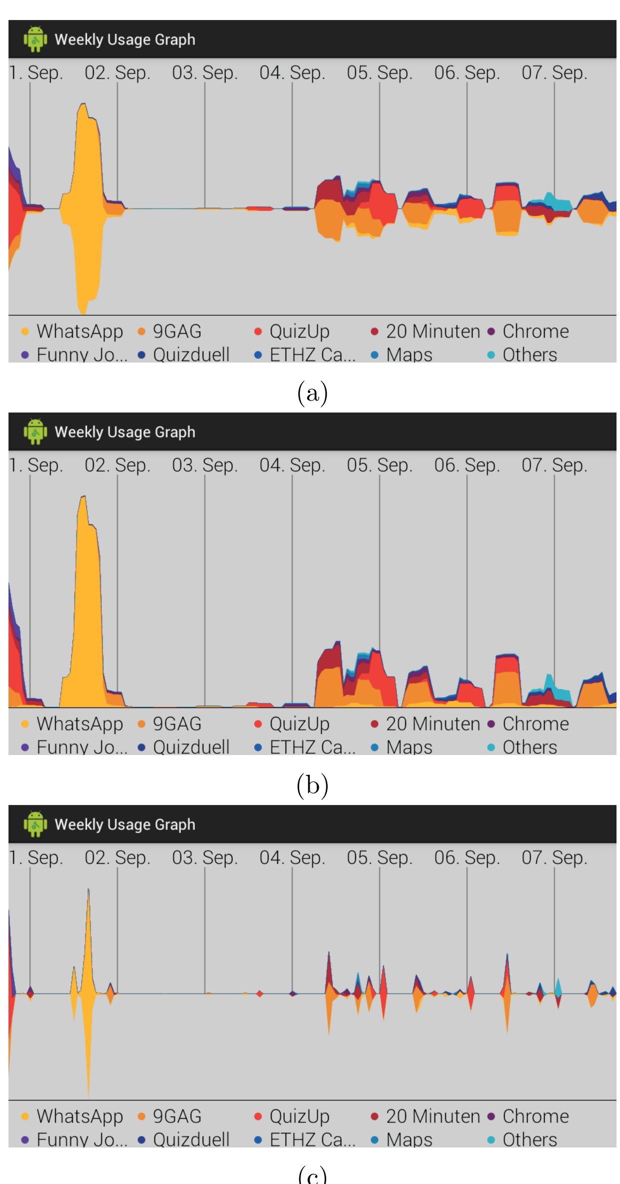


Figure 5.2: (a) shows the stream chart as it is currently implemented. (b) shows a classical stack chart. (c) shows problem of the heavy spikes that could occur in a stream chart. All three charts display the same data set.

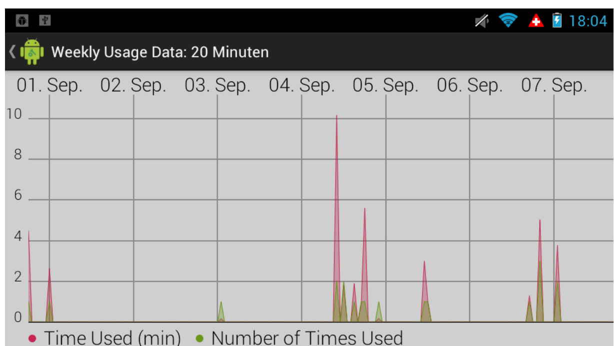


Figure 5.3: Here the weekly usage graph for the application “20 Minuten” is shown.

5.4 Weekly Usage Graph

The *Weekly Usage Graph* displays both the total usage time and the total number of usages of a certain application in a simple line chart. Both values are calculated again in hourly intervals and are displayed accordingly. It is possible to display both values in the same graph with the same y scale as the total usage time in minutes lays between zero and sixty and it is hard to use an application more than sixty times in one hour. So none of the values dominates to much and they could be displayed in parallel. An example of the Weekly Usage Graph is shown in Figure 5.3.

The Weekly Usage Graph is the only graph that is not accessible through the *“Statistics”* tab. To see graph one has to select an application in the *“My App”* tab and use the according option *“Show Graph”*.

Chapter 6

Discussion

The Android application *AppMin – The App Administrator* was freely released on the Google Play Store in the early June 2014. Since the release *AppMin* counts 234 unique installations. 85 of them have the application still installed on their mobile phone[1]. During the period of the thesis it was possible to collect around 400,000 action log events, containing 4,700 different applications.

To increase the probability, that *AppMin* gets downloaded, a small advertising campaign was started and the application was presented on different forums specialised for Android. But unfortunately the advertising did not work as one could wish for. We think that *AppMin* lacked some attractiveness, as the visualisations were at this point of time not implemented and the ratings implemented seemed mostly random, as only a few users have installed it. As well were some users suspicious of the collected data, as it is not possible to suppress the collection. They were afraid that *AppMin* tries to harm their privacy. To counteract this, we updated the description of the application, to be more precise about the usage of data and what data in detail is collected. We hope to be able to increase the popularity of *AppMin* whit the release of the visualisation and with the improvement of the the ratings and recommendations.

Currently *AppMin* is only monitoring applications that are active in the foreground. This penalizes applications that that are designed to run mostly in the background even though the user uses them, such as music players or widgets. These background applications are currently not monitored, because there exist applications that have always a background service running, such as *WhatsApp* or *AppMin* itself. It was not possible for us to automatically distinct applications that run a service which is important for the user, such as a music player, from applications that run a service that does not bother the user, such as WhatsApp. This fact leads to that if ratings are given based on the usage time, applications that only run in the background are rated lower than applications that run in the foreground.

[1] 5. September 2014

Chapter 7

Future Work

As a future work we plan to collect more data. Either through increasing the popularity of *AppMin* or through adding the collection algorithm of *AppMin* to already popular applications.

As soon as we collected enough data we plan to start the development of the often mentioned recommendations and ratings for mobile applications. As a first step we want to find out what applications are similar. Similarity thereby not only means that two applications do the same, also if the context in which the applications are used is the same. Considering the data that is collected, one could group applications in to groups like "used in the morning", "always used longer then 5 minutes", "used 10 seconds, multiple times in a row" or even "Application X used often after application Y" whereas Y is static and X can change.

If such a similarity measure is deployed, we want to compare applications with each other. For example if we have two applications A and B that are considered to be similar we can compare them like "A is better than B because A is used more often than B". Even stronger statements would be of the form "User U has installed Application A and B and stopped using B after installing A. This leads to that he likes A more than B". After we are able to come up with such statements it is possible to rate A better than B and recommend rather A than B. If such statements are made over different time intervals even new coming applications have a chance to get recommended.

Bibliography

[1] http://www.appbrain.com/stats/number-of-android-apps
Accessed: 2014-08-26.

[2] Cook, T.: Key note. Apples World Wide Developer Conference (2014)

[3] Girardello, A., Michahelles, F.: Appaware: Which mobile applications are hot? In: Proceedings of the 12th international conference on Human computer interaction with mobile devices and services, ACM (2010) 431–434

[4] Yan, B., Chen, G.: Appjoy: personalized mobile application discovery. In: Proceedings of the 9th international conference on Mobile systems, applications, and services, ACM (2011) 113–126

[5] http://developer.android.com/index.html Accessed: 2014-09-07.

[6] https://developer.android.com/about/dashboards/index.html
Accessed: 2014-08-28.

[7] https://code.google.com/p/google-gson/ Accessed: 2014-04-16.

[8] https://github.com/nostra13/Android-Universal-Image-Loader
Accessed: 2014-05-07.

[9] http://tomcat.apache.org/ Accessed: 2014-04-03.

[10] http://docs.oracle.com/javaee/6/tutorial/doc/bnafd.html
Accessed: 2014-09-02.

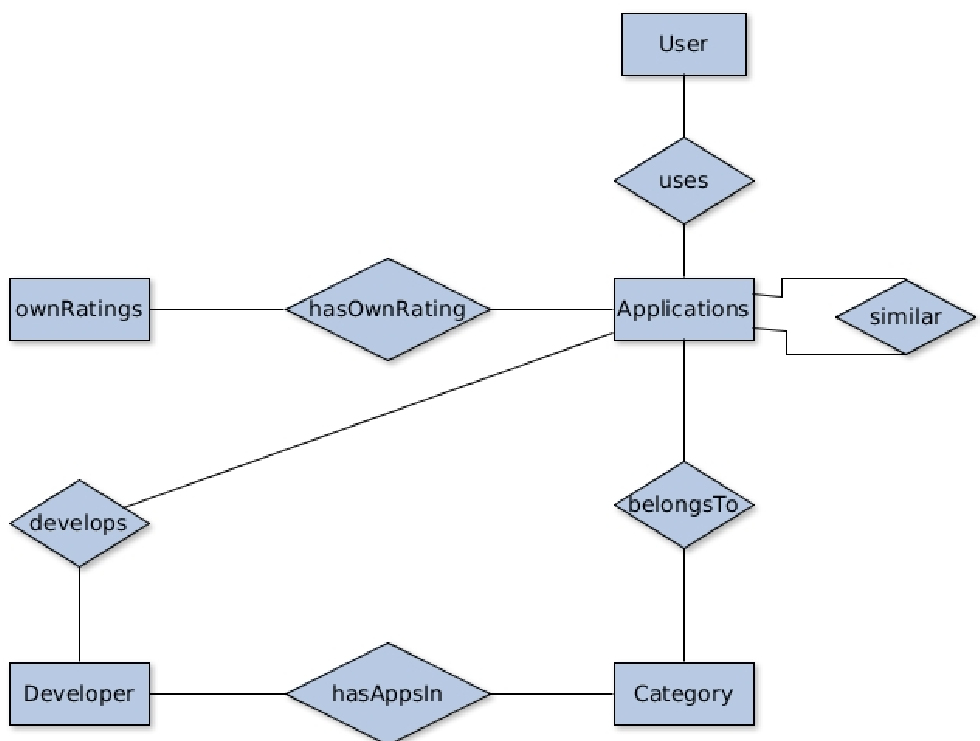
[11] http://acra.ch/ Accessed: 2014-05-17.

[12] http://doc.scrapy.org/en/latest/ Accessed: 2014-02-20.

[13] https://github.com/PhilJay/MPAndroidChart Accessed: 2014-06-25.

[14] Byron, L., Wattenberg, M.: Stacked graphs-geometry & aesthetics. IEEE Trans. Vis. Comput. Graph. **14**(6) (2008) 1245–1252

Appendix A



The Entity/Relationship diagram on which the database on the server is build on.