Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Opportunistic Trip Planner

Semester Thesis

Petar Jokic

jokicp@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zurich

Supervisors:
Pascal Bissig
Philipp Brandes
Prof. Dr. Roger Wattenhofer

February 28, 2016

Abstract

The process of planning a trip for a group of people can be complex and time-intensive. Checking all date-combinations and flight-offers on a daily basis would lead to the best-fitting dates for the lowest cost but is infeasible due to the large number of possibilities.

During this semester thesis a system that automates the hardest parts of such a search process was developed, simplifying the planning phase. For a number of friends, who want to go on a trip toghether, a group can be created on this system. The group members only have to install the developed Android application "TripPlanner", set their location and enter preferences on flight times, vacation duration and maximum flight costs. Calendar events of each member are automatically extracted and compared on a server. In order to allow for everyone to attend, the server searches for flight offers which satisfy the entered preferences and fit between all scheduled calendar-events. Once the server has found all offers within the next month, the individual trip suggestions are sent to the user application.

If everyone in the group found an acceptable flight on a certain date, the application displays this common trip date to all members, which means that the planning phase ended and flights can be booked. With this tool, the user effort is limited to a minimum, as only suggested flight offers have to be rated.

Contents

Chapter 1

Introduction

1.1 Motivation

Planning a vacation for a group of friends can be a time-consuming project. Especially when your friends are spread all over the world and have busy schedules. The process of finding flight offers that fit the preferences of all group members implies a lot of work. Comparing calendars to find gaps between scheduled events while looking for cheap flights on these discussed dates are not the sole challenges. Also, different persons have different preferences concerning their flight times, the number of stopovers and costs. Combining all these parameters during the search for the optimal vacation can therefore be complex.

The goal of this project is to simplify these time-intensive steps by making use of automation techniques. The first step is to find dates on which all friends are available. This can be achieved by analyzing their calendars in order to find a certain number of consecutive free days, which is a simple search problem. Several semi-automatic tools for this purpose already exist on the market as mentioned in Section 1.2.2. In the second step, once a set of possible dates is known, cheap flights can be searched for each of the date-combinations. Doing this manually might be possible for a few date combinations but seems infeasible for tens to hundreds of combinations. Gathering flight information is more complex, as offers of many different airlines have to be compared in order to get the cheapest offer. Well known web-services, such as Skyscanner, exactly perform this task. For an input set of location, destination, outbound and inbound date, Skyscanner searches for flight offers by comparing many airlines and by this finds the cheapest flights for a certain route (see Section 1.2.1).

Going one step further by avoiding the usage of trip boundary dates, the time period within which a trip is being searched for, does not have to be specified directly but will be continuously adapted to the search space of one month ahead. By assuming that the majority of people have their most important calendar events set about one month in advance, their calendar will usually be up-to-date for the upcoming 30 days. If the trip planner system repeats the whole

search process on a daily basis by extracting each calendar once a day and recalculating its suggestions, no constraints on dates have to be set manually. This increases the ease of planning a trip, as the group of friends have to simply specify a destination, enter their preferences once and join the group. From this moment on, they will receive daily trip suggestions for possible trips within the following 30 days. The suggestions they receive already agree with their calendar and the preset individual preferences, which makes them much more relevant to the trip. By rating each trip either positively or negatively, the set of liked date-combinations can be compared with those of all other group members. If everyone in the group liked a certain trip, it is displayed to all members, meaning that everyone is able to attend on these dates and everyone has found flights meeting their individual preferences. In that sense, the trip planner is an opportunistic tool (and therefore called Opportunistic Trip Planner), that continuously tries to find matching trip offers and suggesting them to the group members.

In order to keep things simple, this thesis focuses on finding flight offers and refers to the fact that most concepts are potentially transferable and expendable to other offers, such as hotels (or combined deals).

1.2 Related Work

There are numerous useful tools on the market which help users during the steps of planning a trip. Existing services can find the cheapest deals for flights, hotels, rental cars and many other offers. Some of them provide APIs (Application Programming Interfaces) which simplify the access to their data set for automated queries. A problem is that some APIs either require a minimum search volume or are otherwise limited by a low maximum request rate.

1.2.1 Flight Search Tools

Many well-known web-services are specialised on finding the cheapest flights for a certain route- and date-combination. Most of them are based on a principle called data-scraping (see Section 2.1), which means that they collect flight offers from websites of a large number of flight providers. This process is hard to be set up, which makes it lucrative to use their provided API. The following (non-exhaustive) list of flight search tools shows services which provide APIs for automated flight searches as they are required for this project. A more complete list of holiday planning APIs can be found in [1].

(i) Skyscanner.net is one of the most used flight search tool available and features an extensive dataset. It requires a minimum monthly request rate of 200'000 unique requests to get a private API key. A test key is available but has a limited request rate only [2].

(ii) Expedia.com provides a free API for flights, hotels and rental cars, but requires a certain request-to-booking ratio [3].

(iii) Cleartrip.com has separate APIs for flights and hotels. In order to use this API, a certain request-to-booking ratio is expected [4].

1.2.2 Calendar Tools

Apart from calendar tools integrated in email services, such a Google Calendar or Microsoft Outlook, there are various other tools available, which help to easily schedule events for a group of people. One of the most famous ones is Doodle [5]. Its principle is to find dates on which everyone (or the majority) is available. In order to do that, a user creates an event and distributes the link to it. Everyone who wants to join, enters its dates of availability (or extracts them from another calendar), which then are processed to find the best matching date.

Chapter 2

Data

In this thesis many different kinds of data, such as calendar entries and flight information, are being used. None of them are directly accessible in a form which would make them automatically processable. For example, the flight offers on a travel website are structured in a way that makes them easily readable by a person but hard to directly understand for a computer. To analyze data programmably, lists of numbers are much more important as they can be directly processed and compared. Therefore data scraping is used to extract datasets from their sources, such as websites or calendars.

2.1 Data Scraping

Scraping is the process of detaching and collecting data from a source. The extracted numbers and text-strings are subsequently stored in a format that is easier to process. In that way, the scraper needs to know how the website is structured (e.g. knowing the HTML model) so that it can browse through it and read the specified data. Furthermore the correct requests have to be sent to make the website provide the desired information. If the website provides a well-documented API, the whole scraping process is much easier. An API forms the interface to access website data in a direct way. The request contains all control information (like flight destination and dates), whereas the response is mainly a set of requested data (such as flight offers). The Skyscanner flight search tool used in this thesis (see Section 1.2.1) works in this exact manner. It scrapes the airline websites and collects their data in order to be able to compare offers from many providers [6]. Depending on the chosen routes and the dates, a single flight-query can return around 200 offers from different providers and on different times. Due to this tool, the trip planner system has access to condensed flight information and can therefore focus on its other tasks. More details on how flight data are being received can be found in Appendix B.2.

2.2 Calendar Analysis

The second set of data used in this thesis is calendar data. Even though this data-type is easily accessible via calendar applications or online-tools, the extraction of events, to automatically find out the free dates of a person, is more complex.

This project needs to deal with calendars on Android phones, where the events of the user are assumed to be listed. Each calendar consists of a database which contains all events with their specific attributes. Starting with the very first calendar-entry, which can date back to several years ago, the number of events can be very large. If one wants to find all events occurring during a specific period of time, like it is necessary for planning a trip, the simplest approach would be to go through all events and copy the ones which are within the specified interval. Unfortunately, this is not sufficient as recurring events are only listed on the day when they start and do not have single events for every recurrence. So only one entry exists, which contains attributes specifying the type of recurrence. In some cases theses entries are not within the observed dates, so all past periodic entries have to be checked for occurrences withing the planning time-span. By generating a copy for every such event, these cases can be extracted too.

Knowing when events are listed in a calendar is usually not enough to actually determine the availability of a person. Deciding on how important a scheduled event is, can be the more crucial factor. To exemplify, events named "laundry day" could be rescheduled for the sake of a trip, whereas "my weeding day" most certainly is more important. This judgment is a complex decision for a machine. To simplify, and because analyzing the importance of an event goes far beyond the scope of this thesis, it is assumed that as soon as a day has a scheduled event, it is unavailable for trips.

Apart from extracting calendar information, the gathered dates have to be compared between the different members of a group, aiming to find dates on which all members are available. On one hand this can be done quite easily by taking the overlapping dates of all users. But on the other hand, a single user might block all dates by having a full schedule. One approach of solving this problem would be to rate the importance of blocking dates and ask blocking users, whether they are willing to cancel them. This could possibly solve the problem if a group does not have long enough free periods. However, the opposite could happen if a group has too many possible date-combinations, which would ask for more restrictions. As it can be seen, the task of analysing and comparing calendars of several group members is arbitrarily complex and has therefore been simplified for this prototype version of a trip planner. Only the dates when all group members have empty schedules are interpreted as possible dates for a trip.

More sophisticated solutions could probably be provided by algorithms, which know or learn when a user is probably available. This could be achieved by means of habit recognition or analysis of their position.

Chapter 3

Implementation of Opportunistic Trip Planner

3.1 Overview

3.1.1 Hardware Selection

The hardware of the trip planning system has to be carefully chosen in order to meet the diverse requirement of the different parts, such as high computational power and fast Internet-connection for data scraping but also the possibility of simple user-interactions and access to individual calendar data. Higher computational power asks for server-hardware, whereas the interface to the user is probably best implemented in an application on a mobile phone.

Accessing calendar data on the smart phone is more independent from the type of calendar-service used (as it is synchronisable with different calendar-accounts). This is a major advantage, as for example the extraction of calendar data from a web service, such as Google Calendar of Microsoft Outlook, would require the transmission of user-names and passwords and would additionally need to be implemented for all of these different services individually.

All tasks which do not have to be processed close to the user are more efficiently implementable on a server. The only disadvantage is the data exchange, which is then needed between the phone and the server. However, data exchange is necessary either way and the volumes of transmitted data are small compared to the ones that would be needed for data scraping on the phone.

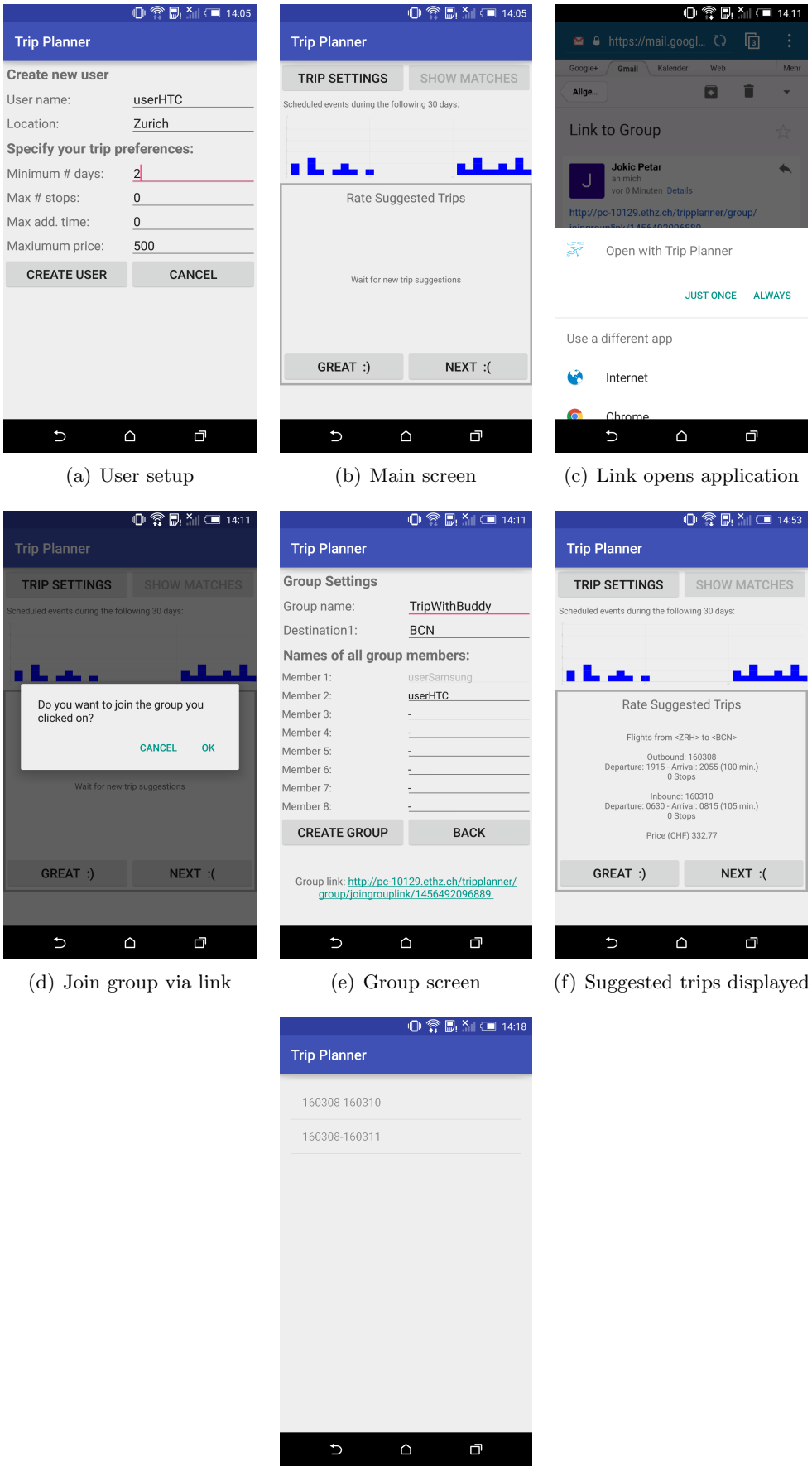
3.1.2 Prototype Implementation

As stated in the section above, the Opportunistic Trip Planner system consists of of two subsystems: a smartphone application and a server program. The idea of the Android application is to give the users an easy access to the sys-

tem. Each user application can communicate with the server, which performs all calculations and most other analysis tasks.

The first time a user starts the application, he or she has to set up a user-profile. All calendar entries within the next 30 days are automatically extracted. Then the user can create a new group, which gets stored on the server, and invite friends by sending them a group link. This link, previously generated by the server, automatically opens the application when clicking on it. By accepting to join the group, the user sends its calendar entries to the server and updates the group file in the database. The group screen gets periodically updated, so that every group member can see who has already joined the group. As soon as all users have joined the group, the server starts to search for flight offers. This is done for each group member individually, as their locations and preferences might differ. During this time, the user application keeps polling for offers until the server has finished its task. Subsequently, the suggested offers are sent to the user in response to the polling requests. On the Android side, these results are stored and then sequentially displayed to the user, who can rate them by clicking "GREAT" or "NEXT". Every rating is sent back to the server, where the ratings of all group members get stored and compared. If all members liked a specific date-combination, it gets sent to the user, who can see the periodically updated list of matching dates via the appearing "SHOW MATCHES" button.

Figure 3.1 presents theses steps in a chronological order from the view of a fictional user named "userHTC", who joins a group by clicking on a group link received by e-mail.



(a) User setup (b) Main screen (c) Link opens application

(d) Join group via link (e) Group screen (f) Suggested trips displayed

(g) Matching dates

Figure 3.1: Screenshots from Android application

Figure 3.2 shows the interactions between the two subsystems graphically, pointing out the involved Java-classes for an enhanced understanding of the system structure. Documentations of the implemented code can be found in Appendix A and B.

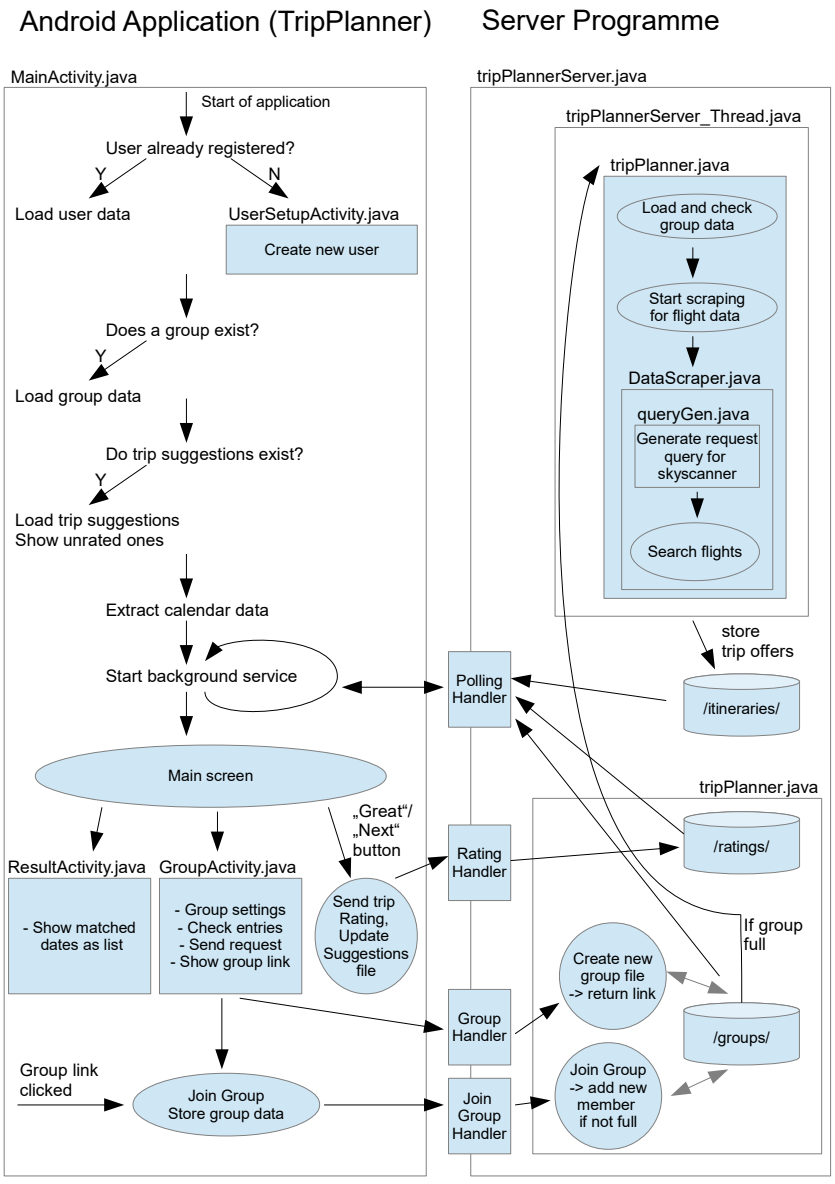


Figure 3.2: System overview

3.2 Android Application

The developed mobile phone application is called "TripPlanner" and serves mainly as a user-interface to access the Opportunistic Trip Planner system. A pseudo flow-diagram of the application can be found on the left-hand side of Figure 3.2.

All user-specific data are stored in the TripPlanner application. As soon as it gets started for the first time, a new user profile is asked to be set up. In order to plan a trip with friends, a group needs to be created. One of the users has therefore to write down all friends in the "TRIP SETTINGS". After clicking the "CREATE GROUP" button, a group link is shown in the trip settings and the user automatically joins the group. Each joined user is always able to see a periodically updated list of the (user-) names of all joined members or "not joined yet" in case not everyone has joined the group yet. By copying the generated link, it can be sent to the other group members, allowing them to join the group.

After clicking on the link, Android asks whether the TripPlanner application shall be started or the browser. If the browser is selected, a simple HTML message appears, showing "Welcome to TripPlanner! Get the App from Play Store to join the trip group!". If TripPlanner is chosen instead, the application asks whether the user wants to join the group or not. Clicking "CANCEL" aborts the process and returns to the main screen, whereas the button "OK" makes the user join the group (which is confirmed or denied by showing the appropriate toast message, depending on whether the user already joined the group or not). As soon as another user joins the group, all joined group members receive an updated version of the group entries, showing all names, ordered by the time they joined it.

Once the group is complete, each user receives suggested trips from its location to the common group destination. The trip suggestions are shown on the main screen as soon as the server has finished planning trips for the specific user. By clicking either the "GREAT" button or the "NEXT" button, the user rates the shown suggestion and iterates through all received ones. The state of the ratings is stored and reloaded when resuming the application. Finally, if all group members liked a certain flight-date, the "SHOW MATCHES" button gets enabled. By clicking on this button, the matches get displayed on a list, so that the group members know on which dates they could book a flight.

Appendix A provides more details on the Android code implementation.

3.3 Server Program

On the server-side of the trip planning system, a Java program is running a request-handler, which forms the interface between the flight search tools and the "TripPlanner" application. The right-hand side of Figure 3.2 shows its structure and the connections to the network.

The functionality of the server is essentially described in Section 3.1.2. It is responsible for all group-related actions as well as for the data scraping process. Therefore it communicates with the application via Internet. As there is no steady connection between the two subsystems, the server needs to store all relevant data in its database.

As soon as a group is complete, the server starts to search for flights by analysing the group data and scraping offers from the Skyscanner website. The resulting offers are filtered in order to meet all user-preferences, and then stored as lists of suggestions. By polling data from the server, these files can be accessed by the application. If a member positively rates a trip, the appropriate date-combination gets stored on the server, so that it can later be compared with the ratings of all other group members.

A detailed documentation of the server code can be found in Appendix B.

Chapter 4

Tests and Validation

In order to extensively test a system like the Opportunistic Trip Planner, a user study with many participants would have to be conducted. Due to the limited time for this thesis, such a survey was not feasible. As a proof of concept, fictional users were created instead, so that the system could be tested on a smaller scale. The functional tests (Section 4.1) and performance tests (Section 4.2) were conducted on the final version of the developed system.

4.1 Functional Tests

The idea of the functional test was to create a group of two members in order to verify that the flights get suggested correctly and that the matching functionality works as expected. Therefore, two mobile phones with arbitrarily modified calendars were used to easily verify the test results. To make sure that flight offers exist, the well-connected cities Berlin and Zurich were chosen as user-locations and Barcelona as their destination. The users were given the test parameters listed in Table 4.1. The numbers in brackets are used to indicate multiple events on this day and the column "Days" shows the preferred minimum number of days for a trip.

User	**Location**	**Days**	**Scheduled Events in Calendar**
userOne	Berlin	15	12.02.2016, 13.02.2016 (x3), 04.03.2016 (x2), 05.03.2016
userTwo	Zurich	14	15.02.2016, 16.02.2016

Table 4.1: Calendar entries of test users

The observed time period lasts from the day when the test was performed, the 10.02.2016, until the day 30 days ahead, which was the 10.03.2016. For the first user this means, that the only possible gap for a trip lasting at least 15 days, lies between 14.02.2016 and 03.03.2016, which makes 19 days . "UserTwo" asks

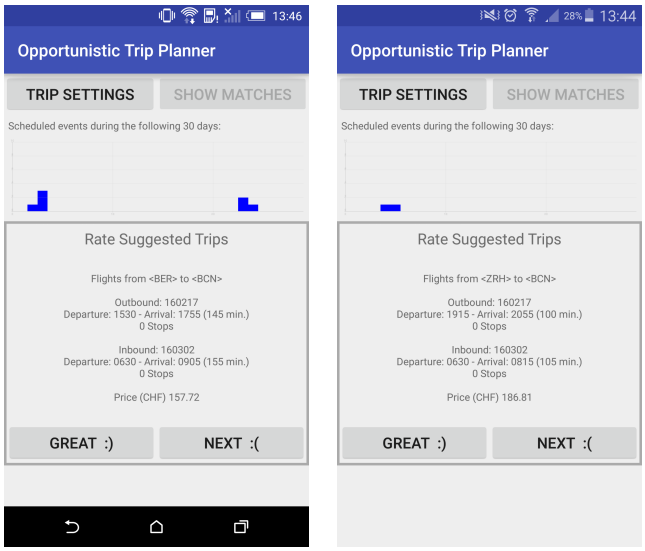
for at least 14 days, which can be realised between 17.02.2016 and 10.03.2016, a period of 23 possible days. As the planner software, running on the server, has to meet all the the requirements and preferences given by the users, the minimum number of days for the group has to be 15 (constrained by the first user) and the search for a common period of possible days should result in the period 17.02.2016 - 03.03.2016. Combinatorially this should result in only 3 different possible date-combinations, namely: 17.02.2016-02.03.2016 (15 days), 17.02.2016-03.03.2016 (16 days), 18.02.2016-03.03.2016 (15 days). Figure 4.1 summarizes these dates graphically.

Only offers with these date-combinations should therefore be suggested by the system. In order to test this, the users have to positively rate one offer for each such combination. By observing that only these date-combinations are then shown in the list of matches, the functionality of the system can be verified to a good portion.

	10.02.16	11.02.16	12.02.16	13.02.16	14.02.16	15.02.16	16.02.16	17.02.16	18.02.16	19.02.16	20.02.16	21.02.16	22.02.16	23.02.16	24.02.16	25.02.16	26.02.16	27.02.16	28.02.16	29.02.16	01.03.16	02.03.16	03.03.16	04.03.16	05.03.16	06.03.16	07.03.16	08.03.16	09.03.16	10.03.16
userOne			1	3																			2	1					
userTwo						1	1																						
Group			1	3		1	1																2	1					
Trip 1																													
Trip 2																													
Trip 3																													

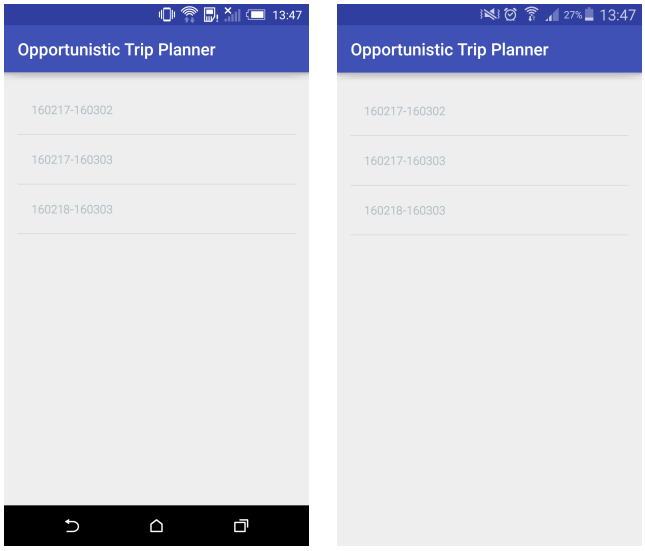
Figure 4.1: Calendar of tested group

After the expected outcome has been determined above, the two users were set up on the 2 mobile phones. Their calendars and preferences were previously adapted according to Table 4.1. Approximately two minutes after both users have joined the formed group, the suggested trips could be validated. It could be confirmed, that only flights on the expected date-combinations were suggested by the server. Examples of these suggestions can be seen in Figure 4.2. By positively rating the first occurrence of each date-combination and discarding all other offers, the main screen appeared empty again. A few seconds later, the "SHOW MATCHES" button became enabled and all expected date-combinations were listed (see Figure 4.3).



(a) Suggestion for "userOne" (b) Suggestion for "userTwo"

Figure 4.2: Offered trip suggestions



(a) Matches of "userOne" (b) Matches of "userTwo"

Figure 4.3: Matching date-combinations

4.2 Performance Tests

During the previously performed tests, one bigger limitation was found: The waiting time before getting results. This drawback of the prototype version is probably due to restrictions of the maximum number of requests given by the Skyscanner API (see Appendix B.2). In order to quantify the restriction, a test trip from Zurich to Barcelona was planned for a single user with a calendar, that allowed for 91 possible date-combinations. The goal was to determine the maximum request-rate of the Skyscanner API.

After 22 minutes and 18 seconds, the 91 requests were processed. This results in an average processing time of 14.7 seconds per request, or in other words, around 4 requests per minute. The estimated number of daily requests, which can be processed by the Opportunist Trip Planner, can therefore be calculated:

$$N_{est} = 24\,\frac{h}{day} * 60\,\frac{min}{h} * 4\,\frac{requests}{min} = 5760\,\frac{requests}{day}$$

The maximum number of date-combinations results, if a user has an empty calendar and set the minimum number of days for his trip to 1. This simple combinatorial calculation yields 450 combinations.

$$N_{max} = \frac{1}{2} * 30\,days * 30\,days = 450\,combinations$$

If this number gets compared with the maximum number of requests per day, a total number of 13 such users could be constantly served by this system:

$$N_{served} = N_{est}/N_{max} = 12.8\,users$$

Although this setup might not be very probable, it still shows that the request rate is a limiting factor of the Opportunistic Trip Planner. As soon as several groups want to be served at the same time, measures like limiting the maximum number of combinations per group would have to be introduced.

Chapter 5

Conclusion and Future Work

5.1 Powerful Organisation Tool

The main achievement of this thesis was to show how powerful the concept of opportunistic trip planning can be. By using calendar analysis and data scraping, the developed prototype is able to automatically suggest trip offers to a group member without any necessary arrangement with the other users. Opportunistic organization tools could therefore help their users so that they can focus on more important tasks. Although more extensive user-studies would have to proof the usability, this project indicates how promising the integration of this tool into a future project could be.

5.2 Extendable Prototype

During the development process, many possible extensions and improvements were found, but could not be implemented due to the time restrictions of this project. Though, a selection of the most important ideas is given below.

The change with the least conceptual influence would be a re-design, which would make the application more appealing to the customer and also more user-friendly. For example, a calendar display that shows the suggested trip highlighted between the scheduled events would make a rating decision much easier. Other improvements in usability could include an automatic localisation of each user, so that the departure airport does not have to be entered manually. This would be possible by using GPS data and a functionality that lets the user pick from a list of recommended departure airports. Another feature that might save trip costs would be a more flexible selection of traveler categories (such as children/ adult) or also an aggregation-tool, which detects if several members depart from the same airport (which would save booking fees). An option to book the trip directly via the Android application, as indicated in Appendix A.4, would be one more marketable feature.

Also more sophisticated suggestion algorithms could improve the planning tool. Rather than defining a single destination airport, a region could be selected instead. This would enlarge the search space for flights but at the same time simplify the planning phase and allow for more spontaneous trips. Suggestions would then be more diverse and could additionally include other holiday offers such as hotels and rental cars.

5.3 Future Work

Based on the prototype developed in this thesis, numerous applications could be derived by either extending the concept as mentioned in the previous Section 5.2 or by applying the system model to other fields. For example, the whole system could be applied to business trips, where meeting rooms, hotels and flights have to be booked by comparing calendar entries.

Finally, the concept could be used in any other field, where activities can be planned and the user prefers to get surprised by suggestions, instead of manually searching for opportunities.

Bibliography

[1] ProgrammableWeb:
"http://www.programmableweb.com/category/travel/apis?category=19965"
Listing of travel APIs, accessed 24.11.2015

[2] Skyscanner Business:
"http://business.skyscanner.net/portal/en-GB/Documentation/ApiOverview"
Documentation of Skyscanner API, accessed 24.11.2015

[3] Expedia Affiliate Network:
"http://developer.ean.com/"
Documentation of Expedia API, accessed 24.11.2015

[4] Cleartrip
"http://www.cleartrip.com/faq/api/air/"
Documentation and FAQ of Cleartrip API, accessed 24.11.2015

[5] Doodle
"http://doodle.com/de/"
Online calendar tool, accessed 26.02.2016

[6] Skyscanner
"http://www.skyscanner.net/whywearefree.aspx"
Description of concept, accessed 26.02.2016

[7] AChartEngine:
"www.achartengine.org/content/download.html"
Charting software library for Android applications, update from 15.05.2013

[8] Vadim Maslov: Travel Info
"http://www.kovrik.com/sib/travel/iata-airport-codes.txt"
Simple listing of IATA codes, accessed 26.11.2015

[9] Skyscanner Business:
"http://business.skyscanner.net/portal/en-GB/Documentation/
FlightsLivePricingQuickStart"
API Test Harness, accessed 26.11.2015

[10] Wireshark
"https://www.wireshark.org/"
Network protocol analyzer, version 2.0.0, 2015

Appendices

Appendix A

Documentation of Android Code

A.1 UserSetupActivity

When "TripPlanner" gets started for the very first time, a screen with user settings will appear. The user name, its location and a set of trip preferences can be entered there. After that, the main screen will start and the user data will be stored in the "UserData.txt" file for all subsequent application-starts. In a future version, the location setting could be automatically gathered from localisation services, such as GPS, which would make the user setup more flexible. The list of preferences is limited at the moment. It includes the minimum number of days of the trip, the maximum number of stops per route, the maximum extra time (compared to a direct flight) if stopovers exist and finally the maximum price for the whole return trip.

A.2 MainActivity

The state from the last application usage gets restored during the start, which includes a series of text file loading-processes. After the extraction of all necessary calendar entries, the graph, showing the sum of scheduled events during the observed period of time, gets generated and shown on the main screen. A graphing library called "AChartEngine" [7] is being used for this purpose. During the same initialisation process, a background service is being started. Its major task is to periodically ask the server for updates such as the current state of the group, the availability of new trip suggestions and information on matching dates. In order not to block the the rest of the application, it runs independently of the application and only sends broadcast messages, which can be received by the running application as a sort of an interrupt. The receiver routine then updates the files and variables so that, for example, new trip offers can be displayed to the user.

The initial main screen shows the scheduled events of the extracted calendar and suggested trip offers (in case there are any available). A button at the top left corner leads to the group settings screen, where all group-related data can be accessed. Right beside this button, another one in the right corner can be used to display the list of all matching dates for the joined group. This button is only enabled when matches exist, as otherwise only an empty screen would appear. Below the calendar section, a quadratic display with two buttons shows suggested trips, which is intended to rate these offers. Therefore, trip offers are being read from the "SuggestedTrips.txt" file while making sure that only those trips are shown which have not been rated yet. One line in the "RatingStatus.txt" file stores the index of the next unrated offer to control this feature. By using either of the two rating buttons, the user can browse through the unrated offers. Whenever a suggestion is positively rated, a message with this decision is being sent to the server.

A.3 GroupActivity

Creating a group on this screen is kept simple and consists of choosing a name, typing a destination and listing a number of friends. In order to simplify the search process for flights, the entered destination must be specified in the list containing all IATA (International Air Transport Association) airports. Such a file has been downloaded and linked to the application [8]. This is necessary as official IATA-codes have to be used for the Skyscanner API [2]. In order to make this file part of the application (which means that it gets automatically downloaded with the application package), it was stored in an .mp3 format, as .txt files do not get included in the downloaded package. Improved versions of this activity could include searching tools, which find the closest airport to a certain destination or even to a region consisting of several destinations.

While the creation process only has to be performed by a single user of a group, everyone has to join it somehow. For the creator this automatically happens once the group was successfully created. In this case the server generates a link to this group, which can be copied from the group screen to share it with friends. By clicking on this link, the application automatically opens, or if it has not been installed yet, a simple HTML message appears that suggests to do so. This so-called intent-link contains the group-ID, which allows the joining member to request all group information from the server. If a user accepts to join the group, he clicked on, the application gathers all group data and stores it locally. All other group members will then see this newly joined user on their group screen.

A.4 ResultActivity

This screen has a very simple structure, as it only displays a list of date-combinations received from the server. A click-listener is pre-implemented, which could be used in a future implementations for booking a selected trip offer.

A.5 Database

The application locally stores the entered user settings, group data and current trip suggestions, so that the current state can always be reloaded whenever it gets started. A set of four .txt files, listed in the local storage under "/TripPlanner/", is used for this purpose and can be seen in Table A.1 below. The unrestricted access on one hand allows simple validation for debugging but also makes the text files vulnerable to being corrupted, which would have to be prevented in a public-release version.

Text File	**File Content**
UserData.txt	All entries from the user setup screen
GroupData.txt	A periodically synchronised version of the group file on the server, which includes all entries for the group screen plus additionally the group ID, the total number of group members and the number of already joined members
SuggestedTrips.txt	List of all suggested trips meeting the preferences
RatingStatus.txt	Group ID of rated trips, date of last synchronisation with server, number of suggestions, number of already rated suggestions

Table A.1: Text files containing application state

Appendix B

Documentation of Server Code

B.1 Server

The server program is running on an server with URL "http://pc-10129.ethz.ch/" under "tripplanner/" and listens to port 3309. While the standard port number for HTTP transmissions would be port 80, the random port 3309 was chosen as other services on the server might use the standard one. But for security reasons, the firewall blocks such non-standard ports and therefore makes a port-mapping necessary. From an application point of view, this means that all HTTP requests to URL "http://pc-10129.ethz.ch/tripplanner/" can be sent to port 80, which is then mapped inside the server to a request on port 3309 (where the server program is listening).

After starting the server program, the interface to the network essentially consists of four handlers, which listen to different URL-path endings. The following table lists the URLs with their associated handlers.

URL-path	**Associated Handler**
/tripplanner/group/newgroup/	Starts the "GroupHandler", which initiates the creation of a new group based on the transmitted data.
/tripplanner/group/joingroup/	Starts the "JoinGroupHandler", which extracts the user data from the request and adds the user to the specified group (if it is not full yet).
/tripplanner/group/polling/	Starts the "PollingHandler", which returns the status of the group, flight offers (if any are available) and the list of matching date-combinations for the group.
/tripplanner/group/rating/	Starts the "RatingHandler", which adds the transmitted rating to the database

Table B.1: List of handlers on the server-side

In general, the "GroupHandler" creates a new group file and the "JoinGroupHandler" adds members to this file as long as the number of joined mem-

bers is smaller than the number of total members. Once this limit has been reached, which means that the last member has joined the group, the server program starts planning the trip. For this purpose a new thread gets created, so that the flight offers for each group member can be searched without blocking the rest of the program. This is necessary to be able to handle multiple groups simultaneously.

The search process itself starts with the extraction of all member data from the group file. With this information, the calendar entries can be compared in order to determine possible trip dates, when all members are available. Then it tries to fit the largest minimum trip duration among all members into the found gaps of free days. If there are any combinations that meet this requirement, the function iterates through all group members to find individual trips on these dates. Each iteration step is processed sequentially and contains the exact same steps for every member. Summarized, for each possible date-combination, a scraper is set up with all member-specific preferences and locations. After completion, the resulting flight offers are stored in a file, from where the associated "PollingHandler" can access them. This closes the circle of data flow from the user-application via the data scraper, running on the server, back to the Android application, where offers are displayed to the user.

Last but not least, each positive trip rating sent by the TripPlanner application is being received by the "RatingHandler". There it gets added to a member-specific rating file, where all previous ratings are listed. Each polling request compares the rating files of all group members and returns the matching date-combinations in its rating section.

B.2 Scraper

The source for this system is the website skyscanner.net, which provides an API for searching flights from a large number of different airlines (see Section 2.1). Although Skyscanner provides an API documentation [2], most parts of it only give a rough idea of how flight requests can be sent and received. In order to understand the structure of the strings in both the requests and responses, the provided testing tool [9] (in combination with a Wireshark network protocol analyser [10]) served as a good reference. Skyscanner uses an API-key for user-authentication, which means that without such a key, the API does not work. For the sake of testing an implementation, the website provides an example key, which is limited to a certain number of session requests per minute. Higher rates are available with a private key, but to get a such, a minimum number of 200'000 unique monthly requests have to be guaranteed. Despite this restriction, Skyscanner is still a promising data source for future versions, hence the rate-restrictions were accepted for this prototype.

The process of getting flight offers via Skyscanner API was found to be divided into two subsequent steps: first creating a session and then polling for results from this session. All important travel parameters (API key, dates, locations, number of travelers and many more) have to be sent as a HTTP POST request to get a session key. A developed query generator, forms the request string, which is then sent to the specified URL "http://partners.api.skyscanner.net/apiservices/pricing/v1.0". Once a session has been successfully established, the server can start polling for results by using the obtained session key as an identification. Polling has to be repeated until all trip offers are available, which can last several seconds up to more than a minute. Upon completion, the HTTP GET request, which is used for polling, returns all matching flight offers.

B.3 Database

The flight offers, as they are being received from the Skyscanner API, are encoded in a string format, which has to be parsed in order to extract the relevant information. Firstly, all flights of a certain request (date-combination) are copied into an array and then filtered with the user-specific preferences (which are loaded from the group file). This is repeated, so that for all possible date-combinations an array is being generated. Secondly, all arrays, which now only contain flights meeting the user requirements, get merged into one overall array. This contains the trip suggestions for one group member. It then gets stored as a comma-separated text file in the itineraries folder, from where the group member can access it through a polling request. Flights which are available through different websites are only included once by only selecting the cheapest offer.

The files listed in Table B.2 are used for each group which has been created. The name specifies the group-ID (here GROUPID), and for itineraries and ratings also the user-ID (here USERID). The latter one is the index of the user-position within the group file and therefore also known by each user.

File Name	**File Content**
/groups/GROUPID.csv	All group-specific values (group name, number of joined members, total number of members, destination) and all member-specific entries (member-name, location, preferences, unavailabilities)
/itineraries/ GROUPID_USERID.csv	Comma-separated list of all possible flights including trip details (and booking link) for the specified member
/ratings/ GROUPID_USERID.txt	List of positively rated date-combinations of the specified member

Table B.2: Files containing trip data for a group