FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Data Prefetching in Smart Systems

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software and Information Engineering

eingereicht von

Dr. Sabine Weninger
Matrikelnummer 0301520

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Dr.-Ing., B.Sc., Dipl.-Oec. Stefan Schulte

Wien, 13. April 2018

Sabine Weninger

Stefan Schulte

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Data Prefetching in Smart Systems

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Dr. Sabine Weninger
Registration Number 0301520

to the Faculty of Informatics

at the TU Wien

Advisor: Dr.-Ing., B.Sc., Dipl.-Oec. Stefan Schulte

Vienna, 13th April, 2018

Sabine Weninger	Stefan Schulte

Erklärung zur Verfassung der Arbeit

Dr. Sabine Weninger
Herndlgasse 5/48, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 13. April 2018

Sabine Weninger

Danksagung

Zuallererst bedanken möchte ich mich bei meinem Betreuer Stefan Schulte. Er hat es mir ermöglicht meine Bachelorarbeit zu diesem spannenden Thema zu schreiben und ist mir immer mit Rat und Tat beiseite gestanden.

Des Weiteren möchte ich mich bei Michael Borkowski für die Unterstützung bedanken. Michael hat mir geholfen das Spring Framework aufzusetzen und ist mir immer für Fragen zur Verfügung gestanden. Von ihm stammt das SpiceJ Tool, welches ich in meiner Arbeit verwende.

Für die großartige Unterstützung bei den Testläufen möchte ich mich bei Alexander Knoll bedanken. Er hat IP Adressen und Netzwerkbuchsen vorbereitet, mir geholfen die Rasperry Pis aufzustellen und so das Testen der Software möglich gemacht.

Schlussendlich möchte ich mich noch bei meiner Familie bedanken, welche mich immer unterstützt hat.

Kurzfassung

Geräte des Internet of Things (IoT), welche als Sensoren oder Akteure eingesetzt werden, können in Alltagsgegenstände eingebaut und so fast unbemerkt in die Umgebung integriert werden. Mobiltechnologien ermöglichen zudem das Zusammenführen vieler IoT-Geräte zu einem Netzwerk, einem sogenannten „smarten" System, welches autonom auf Umwelteinflüsse reagieren kann. Ein Beispiel für ein „smartes" System ist eine „smarte" Stadt, in der IoT-Geräte Daten, wie z. B. Verfügbarkeit freier Parkplätze, Verkehrsaufkommen usw., sammeln und einem Benutzer über dessen Mobilgerät zur Verfügung stellen.

Ein Netzwerk von IoT-Geräten kann eine sehr große Menge an Daten generieren, welche verarbeitet und gespeichert werden müssen. Alternativ zur Cloud können Daten auch direkt am Gerät gespeichert werden. Dies macht Sinn für sicherheitsrelevante Daten oder für Geräte in einem lokalen Netzwerk ohne Internetzugang. Ein großer Nachteil eines solchen dezentralen Ansatzes ist eine schlechtere Verfügbarkeit und eine höhere Wartezeit bei der Beschaffung der Daten. Ein Grund dafür ist, dass Daten möglicherweise von vielen verschiedenen Geräten und über mobile Verbindungen mit schwankender Qualität geholt werden müssen. Ein Prefetchen von Daten, also das Abrufen von Daten bevor diese überhaupt benötigt werden, kann dieses Problem lösen.

In dieser Arbeit evaluieren wir wichtige Faktoren welche beim Prefetchen von Daten auf IoT-Geräten eine Rolle spielen, beispielsweise wann welche Daten benötigt werden. Des Weiteren implementieren wir eine funktionsfähige Prefetching-Lösung für IoT-Geräte in einer dezentralen Umgebung, in der jedes Gerät nur mit seinen direkten Nachbarn verbunden ist. Um unsere Lösung zu evaluieren, testen wir Antwortzeiten mit bzw. ohne Prefetching in einer Testumgebung bestehend aus Rasperry Pi Einplatinencomputern, welche IoT-Geräte repräsentieren. Diese Geräte senden Daten an den Benutzer über eine mobile Android-Applikation. Unsere Arbeit zeigt, dass das Prefetchen von Daten die Antwortzeiten signifikant reduzieren kann.

Abstract

Internet of Things (IoT) devices which act as sensors or actuators, can be embedded into everyday objects and seamlessly integrate into our environment. Advancements in mobile computing allow to connect all these IoT devices into one network, creating a smart system which can autonomously adapt to changes in the environment. An example of a smart system is a smart city, where sensors collect information such as parking space availability, live traffic data etc. and provide this information to users via their mobile phones.

All those devices generate an enormous amount of data, which needs to be processed and stored. An alternative to storing the data in the cloud is to keep them locally on the device. This might be useful if the data contain sensitive information or the devices do not have access to the Internet. However, a big drawback of this solution is reduced accessibility of data, because the data might need to be fetched from multiple locations over a mobile connection with fluctuating quality, leading to delays. Prefetching of data, that is fetching data before they are requested, is a way to overcome this problem.

In this work, we evaluate all the factors that need to be considered for device-side data prefetching. These factors include the prediction of what data are needed at what time point. Furthermore, we implement a prefetching solution for IoT devices in a fully distributed environment, meaning that each device is connected only to its direct neighbours. Our implementation focuses on providing a template for prefetching which is meant to be extended using specific prediction algorithms. To evaluate our solution, we compare response times with prefetching versus no prefetching in a test setup of Raspberry Pis, representing IoT devices. These devices provide data to a client-side application running on an Android mobile phone. We show that prefetching significantly reduces response times compared to no prefetching.

Contents

CHAPTER 1

Introduction

1.1 Smart Systems

Smart systems are systems that display "smart behaviour", by autonomously adapting to environmental changes. They consist of sensors, which detect changes in the environment, and actuators, which adapt the system to these environmental changes [Akh00, GBMP13].

Examples of smart systems are smart cities and smart factories. A smart city would bring benefits in a number of public services, like transport and parking, lighting, maintenance of public areas, garbage collection and so on [ZBC+14]. For example, there could be a number of sensors dotted throughout the city, collecting data such as parking space availability or traffic information. A user travelling through the city will be able to gather this live information on his or her mobile phone, pick the ideal route and reduce the time spent searching for parking, which in turn means less CO_2 emissions from the car and less traffic congestion [ZBC+14]. Furthermore, a lot of information could be made available to the public, increasing transparency and quality of services offered to the citizens and stimulate these to actively participate in public administration and gather ideas for further improvement [ZBC+14]. Smart factories might have sensors detecting idle machines and automatically assign work pieces to machines to optimize workflow and maximize throughput.

An important factor in the advancement of smart systems is mobile computing. Smartphones are omnipresent in our lives and mobile traffic has grown 18-fold over the past 5 years. This trend will continue. Global mobile traffic is estimated to increase 7-fold and mobile network connection speeds will increase threefold, between 2016 and 2021 [cis17]. Mobile computing provides the technology necessary for connecting devices over large distances, creating global communication networks [GBMP13].

However, smartphones and portables alone cannot fulfil the vision of a smart system. In order to reach that goal, technology needs to be embedded in everyday objects and

seamlessly integrate into our environment [GBMP13]. Internet of Things (IoT) devices, which act as sensors, collecting data or as actuators which control technological objects, can achieve that. Everyday things such as household appliances, medical devices, even clothes can become IoT devices. By connecting these objects to the Internet, they are able to communicate with each other and achieve common goals, thereby creating a smart environment [AIM10, GBMP13].

1.2 Motivation

Having smart technology incorporated into everyday objects and connecting them through a continuous network, poses several problems. All devices need to be uniquely identifiable to allow communication with and between devices. Therefore, a unique addressing scheme, which is reliable, persistent and scalable is required [GBMP13]. Furthermore, billions of devices will create an enormous amount of data which needs to be processed and stored. Computing and storing resources need to fulfil certain criteria. They need to be secure, efficient, market oriented and scalable [GBMP13].

A convenient and reliable way to store, process and distribute data collected by IoT devices is the cloud [GBMP13]. In the context of smart cities, data collected by parking space sensors might be sent to a cloud storage from which the user can obtain these data.

However, sometimes such a cloud solution might not be possible or desired, because the data contain sensitive information or IoT devices are only connected locally and do not have access to the internet, e.g. because of security issues. Such a scenario makes sense in the context of a smart factory, where all sensors and machines are connected in a local network but are protected from the outside.

An alternative to cloud computing is to store the data directly on the device, collecting the data. However, this approach has some serious drawbacks. IoT devices usually have limited memory and processing power [MDT+16]. Therefore, this approach is only feasible, if the amount of collected data is limited or old data can be discarded and no complex calculations need to be performed. Another drawback of this approach is that there is no central place to access these data. On each request, the correct data will need to be fetched from one or possibly multiple other devices. In addition, these devices often use wireless links with fluctuating quality for communication [MDT+16]. Therefore, it might take some time to collect the necessary data and send them to a user requesting information. This leads to user-perceived latency and reduces Quality of Experience (QoE).

One solution to this problem is to prefetch the data that the user will need. Prefetching is the process of transferring/fetching data ahead of them being required, to have them already available and ready to send, when they are requested by a user [HSHD14].

1.3 Aim of this Work

The aim of this thesis is to evaluate all the factors that need to be considered for IoT device-side data prefetching. Additionally, we will implement a testbed using Raspberri Pi's as IoT devices, for data prefetching in a distributed environment, as depicted in Figure 1.1.

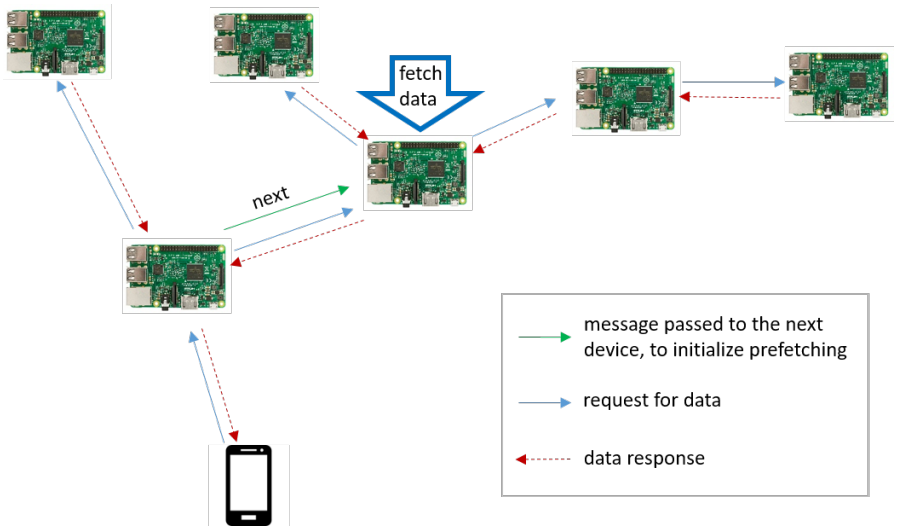


Figure 1.1: Data prefetching in a distributed environment. The user's smartphone sends a request for data to the closest in range IoT device. This device and each following device in turn fetches data from neighbouring devices. A second message is passed to the next IoT device that the user will encounter, telling this device to start prefetching data.

1.3.1 Issues with Data Prefetching

In particular, the following issues with prefetching in a fully distributed environment, will be considered and evaluated:

- **User mobility prediction**: Gauge the movement speed and direction of a user, to predict which IoT device the user will come into range next and when they will arrive there.
- **Data prediction**: Predict which data will be requested by the user.
- **Scheduling/Timing**: Determine the exact time point to start prefetching to get the most current data for the user.
- **Data allocation**: It might be necessary to split information passed to the user between several devices. Therefore, the devices need to communicate with each other to make sure that the user receives each data unit exactly once.

The first two points can only be estimated, certainly when the user is human. Humans do not make their decision based on an algorithm, they often act in a spontaneous and unpredictable way. However, user context information can be used and has been used in the literature to predict user mobility and data usage [HSHD14, SK13, KKLS12].

Modern smartphones can measure the movement speed, direction and location of the user. With this information, the next IoT device the user will come into range, can be assessed to a certain degree. This prediction can even be more accurate if the user follows the directions of a navigation service. Indeed, many mobile apps exist for Android mobile phones, offering navigation services, including Google's own Maps app.

Assumptions as to what data a user might need, can be made based on user context data such as the destination of a travel. In the context of a smart city a user travelling to a certain location, might be interested in parking space availability in that region, traffic problems on the way etc.

Some sensor data like for example parking space availability change quickly. Therefore, it is important that the most current information is passed to the user. In order to have the most current information, the time point to start prefetching needs to be calculated from the amount of data that need to be prefetched and the time the user will arrive.

The reach of one IoT device (the reach of the devices Wi-Fi signal) is limited. Therefore, it will be necessary to split information passed to the user between several devices. In order to do so, a message needs to be passed from the current to the next device. This message will tell the next device to start prefetching and also needs to include which data have already been sent to the user. This is important so that the user will not receive the same data from multiple IoT devices.

1.3.2 Methodology and Approach

Our aim is not to provide specific algorithms that attempt to solve each of these issues. Algorithms addressing these issues already exist in the literature to some point [BSSD16, HSHD14, SK13]. The main goal of this work is to provide a template for data prefetching on IoT devices and to address all the possible issues that could arise. The template will be written in such a way that algorithms addressing specific issues can easily be added to the code.

However, we will implement a working data prefetching solution in a simple environment of one building. We will prepare a testbed, consisting of five Raspberry Pis which represent IoT devices. We chose to use a completely distributed approach. Therefore, there will be no central data storage or central coordination unit. Each device will be connected only to its direct neighbours. These devices provide data to a user via a client-side mobile application running an Android operating system. In detail, the user device will send metadata, such as the current location, movement direction, destination etc. to the nearest IoT device. Each device the user comes into range, will in turn provide the user with information based on this context information.

1.4 Structure

In Chapter 2, we will discuss some preliminaries and related work necessary to understand the content of this thesis. In Chapter 3, the design of the data prefetching solution will be presented. This chapter will give a general description of all the required classes and services. Implementation details will be discussed in Chapter 4. An evaluation of the work is provided in Chapter 5, where we will compare the outcome of no prefetching versus prefetching. Chapter 6 will conclude the thesis with a focus on future work that needs to be done.

CHAPTER 2

Background

This chapter will give an overview over the concepts and technologies related to this work. The aim is to provide the reader with the information required to fully comprehend the following chapters.

2.1 The Internet of Things (IoT)

Several definitions exist for the IoT. The definition by the RFID group goes as follows: "The IoT is a worldwide network of interconnected objects uniquely addressable based on standard communication protocols" [GBMP13]. Based on the focus, three orientations were identified by Atzori et al. (2010). These are: things-oriented (sensors and actuators), Internet-oriented (middleware) and semantic-oriented (knowledge) [GBMP13].

Some of the technologies used for connecting and identifying IoT devices are:

Radio Frequency Identification (RFID): RFID tags are microelectronic transponders that are used for uniquely identifying objects, they are attached to. They act like a more powerful electronic barcode, but with a larger storage capacity and the ability to be reprogrammed [GBMP13, Fin10]. Their name stems from the procedure to transfer data and power, namely radio frequency identification. RFID tags do not require a power cell to function, the power needed is provided by the reading device. This has led to their use in many applications, a prominent example being bank cards [GBMP13, Fin10]. Their small size, absent energy consumption and low price makes RFID tags attractive for identifying sensors and other IoT devices.

Wireless Sensor Networks (WSN): WSN were originally developed in the 1980s to be used in military operations, especially surveillance [YLL+14]. As the name implies, a WSN is a network of nodes that sense physical and environmental conditions such as

temperature, sound etc. and may also control the environment. They consist of a large quantity of sensor nodes, which transmit their collected data along other sensor nodes by hopping, to finally reach a management node through the internet [YLL+14]. The small, inexpensive and low powered sensors of WSN are an important factor driving the IoT as they can be installed in virtually every IoT device [YLL+14].

IoT devices are technically heterogeneous and can be a wide variety of devices such as, household devices, surveillance cameras, vehicles, medical devices and so on [MDT+16]. All of these devices, easily accessible over the Internet, may generate an enormous amount of data, which need to be processed and stored. An Internet-centric architecture, with the cloud at the centre offers a solution to these challenges by providing scalable storage, computation time as well as other tools [GBMP13].

2.2 Data Storage

As mentioned before, using cloud storage is a simple and convenient way of storing, processing and distributing data from IoT devices. However, in some cases this might not be the desired solution. In that case, alternatives need to be considered.

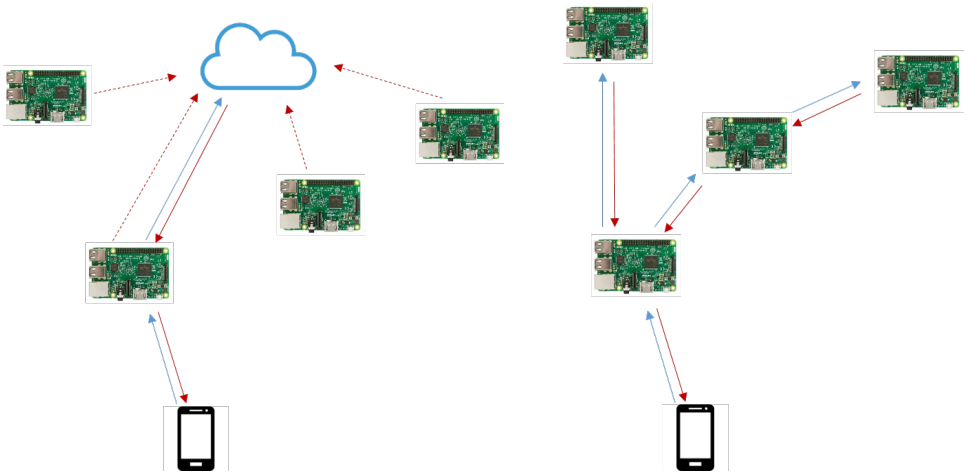
Ways of storing data in smart systems, without using Internet cloud storage:

- Collect and store the data from all IoT devices centrally in a local "cloud". See Figure 2.1a.
- Each IoT device stores the data collected by its sensors locally. See Figure 2.1b.

The first method is very similar to using Internet cloud storage. The difference is that data are stored on a local server acting as a cloud. This server as well as all the IoT devices do not need to be connected to the Internet. This method has the advantage that all data are easily accessible from that central location. A user needing data can get the data from one of the IoT devices, which fetches the data from this local central storage or alternatively the user can directly interact with the central storage and get the data from there.

The second method has the advantage that data stay where they were collected until needed elsewhere. However, storing data locally on the IoT device, which collects them, brings certain problems with accessing these data. A user connecting to one such device, might request data stored on another device, or even distributed over multiple devices. Before being able to send these data to the user, they need to be fetched from other devices, possibly via connections with fluctuating quality. This process can take some time and will lead to some user-perceived latency and reduce QoE.

One solution to this problem is to prefetch data that will be needed in the future, ahead of time.



(a) Centralized storage in a local "cloud". (b) Distributed storage in every IoT device.

Figure 2.1: Data storage places in smart systems. *The dashed red arrows represent data being uploaded from the IoT devices to the local "cloud". Blue arrows indicate requests for data and red arrows indicate the data response.*

2.3 Data Prefetching

The idea to prefetch data is not new. Already in 1998, Tuah et al. investigated a model for data prefetching in mobile networks with low bandwidth [TKV98]. Indeed, a lot of work has been done in the field of client-side data prefetching for mobile devices. Here data prefetching helps to overcome times of bad or no connectivity and thereby increases QoE [BSSD16, SK13, HSHD14, IWF+12]. Borkowski et al. (2016) found that data prefetching significantly decreases response times and outperforms approaches with no prefetching, even when prediction of mobile connectivity rates is inaccurate.

Several studies exist, proposing algorithms for mobile data offloading to WiFi networks [SK13, DHX+13, BMV10, LLY+13, MDT+16].

As mentioned in Section 1.3.1, several issues need to be considered with prefetching: user mobility prediction, data prediction, scheduling/timing and data allocation.

User mobility can be predicted to a certain extent based on user context information, such as the current location, routes suggested by a navigation system etc. Hummer et al. (2014), use user location data to predict mobile network speed and provide personalized prefetching solutions. They further show that this approach is superior to periodic prefetching and decreases user-perceived latency [HSHD14]. In their work on mobile data offloading to Wi-Fi networks, Siris et al. (2013) use their own mobile app "OptiPath", which calculates the optimal route between two locations in urban environments, to predict user mobility [SK13, KKLS12].

Several algorithms have been described in the literature addressing the problem of when

to start prefetching [BSSD16, SK13]. Borkowski et al. (2016), studying data prefetching in mobile service applications, propose an algorithm which minimizes both data age and data volume of prefetched data. This leads to the start of prefetching at the last possible time point [BSSD16].

CHAPTER 3

Design

The goal of this thesis is the design and implementation of a prefetching solution for IoT devices. In the context of "Smart Cities", the data to be prefetched and sent to the user may, e.g., contain information about parking space availability, live traffic information or public transportation information. This information is sent to the user directly by the sensors or by computing devices connecting several sensor units. One data unit may include the bus timetables at one bus stop, traffic delays for a particular road or area and so on. On each IoT device, an application listens for and responds to messages from a user device. On the user device, a mobile application is running which requests data of interest from the nearest IoT device and displays the response to the user.

3.1 Application Running on IoT devices

This application listens for messages from a user device and from other IoT devices and responds accordingly. When a device receives a message from a user, it in turn passes on some information to the device which the user should come into range next. This message tells the next device which data items the user will need so that they can be prefetched.

3.1.1 Data Prefetching

The actions taken by the next device after receiving such a FETCH message are depicted in Figure 3.1. First, the device stores all the received information. Before prefetching, the device needs to predict which data items should be sent to the user by this device, as only these data items need to be prefetched. Remaining data items will be passed on to the next device later (see Subsection 3.1.2). The device predicts how long the user will be in range and based on this, calculates the number of data items that can be sent to the user and therefore need to be fetched. Prefetching is done by generating a new

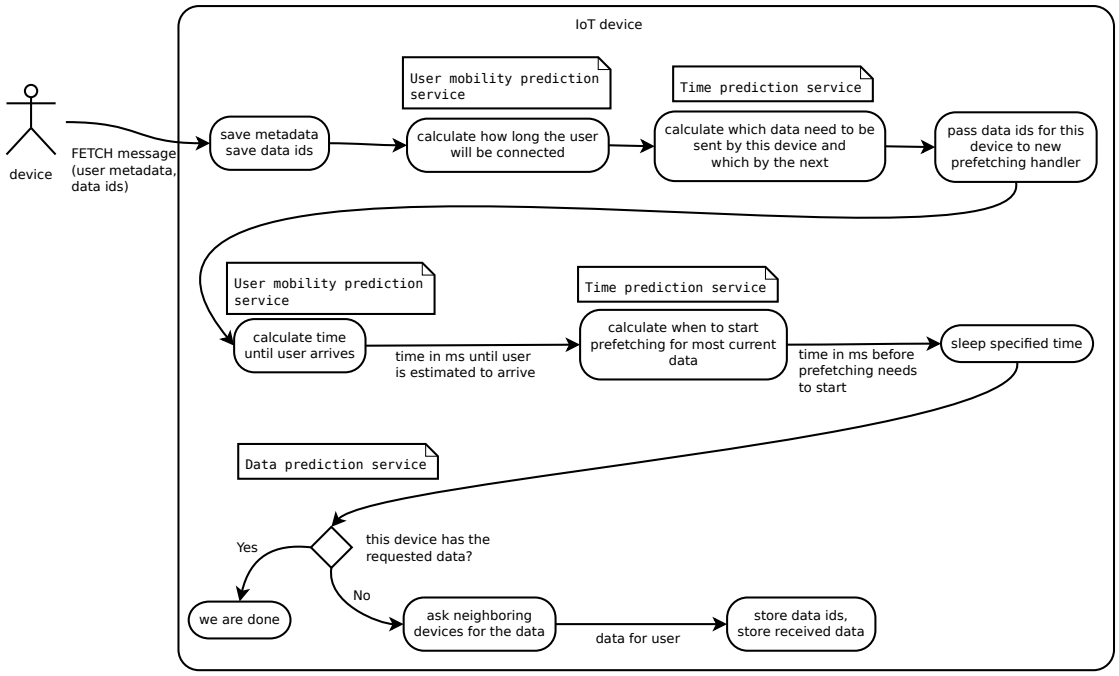


Figure 3.1: Data Prefetching by IoT Device

prefetching handler object. This handler is saved as well, so that it can be accessed in case the user arrives early and fetching of data has not finished yet. The next actions all take place in the prefetching handler, which is executed in a new thread. Several service classes are called which predict when the user will arrive and in turn when to start prefetching in order to retrieve data at the last possible time point. When it is time to start prefetching the device checks if the requested data items are available locally. If they are, no further actions need to be taken. Otherwise, the data items need to be collected from another device. In order to do so, a request is sent to all neighbouring devices. We will explain this process in detail later (see Section 4.1.2). The collected data are then stored and are available to be sent to the user.

3.1.2 Response to User Messages

In case the prefetching mechanism was successful, the IoT device will already have all data available locally, when a user comes into range. The actions taken by an IoT device upon receiving a message from a user are shown in Figure 3.2.

First, the device checks if the user metadata have been received previously (e.g., from a FETCH message) and whether the data have already been prefetched. If they have been prefetched, the data items can directly be obtained from the local storage and the following steps can be skipped. Otherwise, the device checks whether a prefetching handler for this user is available. If such a handler is available, data are currently in the

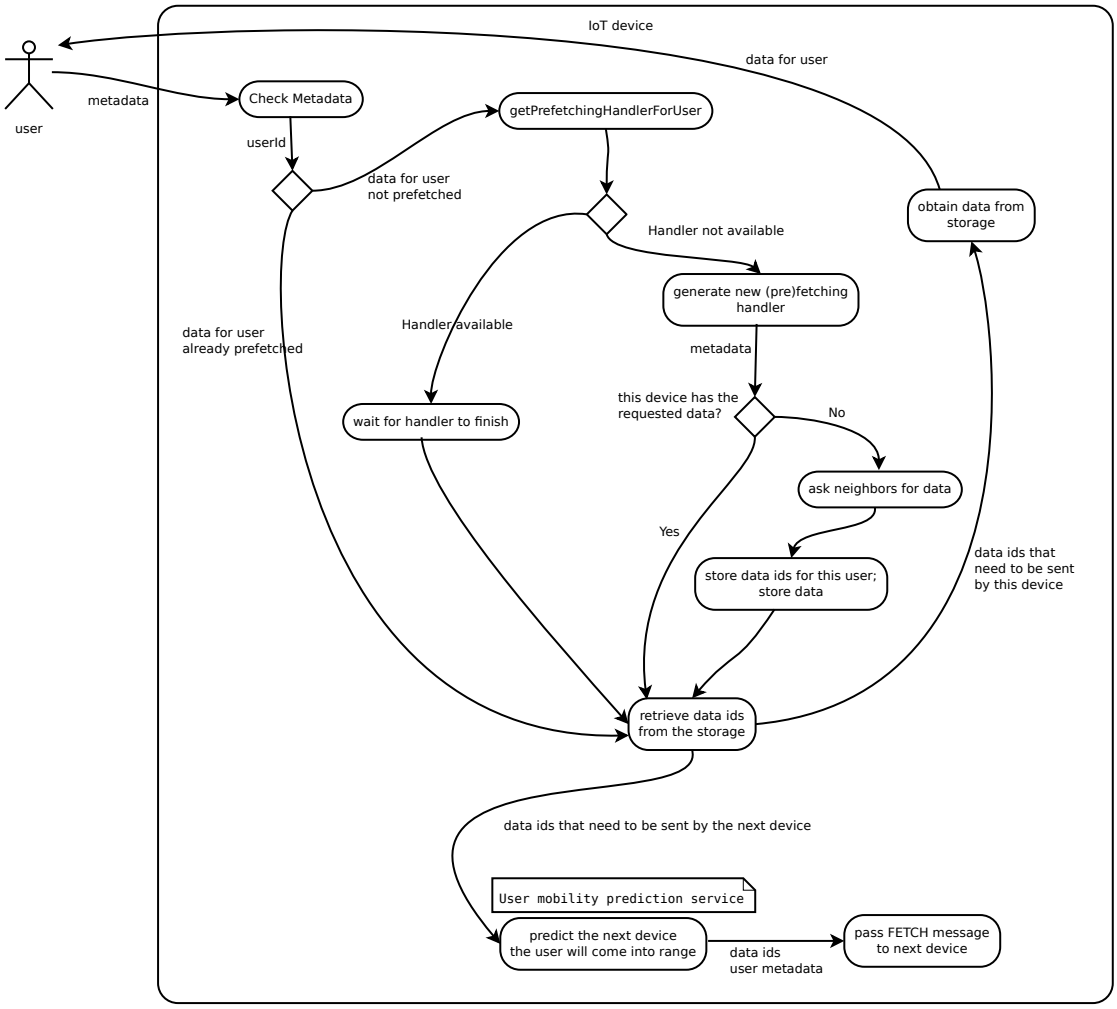


Figure 3.2: Actions taken in Response to a Message from a User

process of being fetched. In that case, the device needs to wait for the fetching process to finish before accessing the data. In case prefetching has not been initiated before, the device starts fetching data immediately. It does so by generating a new prefetching handler object, which checks what data need to be fetched and then sends corresponding messages to neighbouring devices, requesting the data. When the data have been fetched and saved, they can be retrieved from local storage. Before returning them back to the user however, the device prepares and sends a FETCH message to the next device. Typically the data ids that need to be passed on to the next device have already been calculated right before prefetching (see Subsection 3.1.1). However, in case no prefetching took place this still needs to be done. In order to do so, the device predicts how long the user will still be in range and based on this, calculates which data it can return to the user and which data units should be sent to the user by the next device. If not all data

can be transferred to the user by this device, the next device that the user will reach is predicted and a FETCH message is sent to that device. All the predictions are done by several helper classes which will be described in detail later (see Section 3.1.3). After the FETCH message has been passed on, the data items are returned to the user.

3.1.3 Service Classes

The following section describes the services used for various calculations relating user mobility, the data required etc. This work focuses on the interface and interaction between services, not on the actual implementation. In order to achieve optimal prefetching, these classes need implementations with better algorithms in order to better predict user behaviour and user needs. This is not the scope of this work, however.

Data Prediction Service

This service provides methods to predict which data will be needed by the user.

As input, it takes the metadata provided by the user in JSON format (see Listing 3.1).

Listing 3.1: User Metadata.

```
{
  "userId": "<id>",
  "currentPos":
    {
      "lat": "<latitude>",
      "lon": "<longitude>"
    },
  "destinationPos":
    {
      "lat": "<latitude>",
      "lon": "<longitude>"
    },
  "directionInfo": "<int>",
  "movementSpeed": "<m/sec>"
}
```

The service contains a method isResponsible() with the following signature:

```
boolean isResponsible(UserMetadata metadata);
```

Based on the user metadata, this method returns whether the required data items are available on the current device.

Furthermore, the interface contains the method getData() with the following signature:

```
List<ResponseData> getData(UserMetadata metadata,
  List<Integer> dataIds);
```

As mentioned above, the user metadata are needed as input. Optionally, the service also takes a list of data ids. If the data ids are specified (not null), these take precedence over the user metadata. If no data ids are given, a prediction algorithm calculates the data required by the user based on the user metadata. The method returns the data items as a list of ResponseData. ResponseData is a model class for the data items and contains fields for the ids as well as the content of a data item. To make sure the required data items are available on the current device, the method isResponsible() of the same class is always called first.

This thesis only implements a very simple Data Prediction service, but provides the possibility to integrate any Data Prediction service.

User Mobility Prediction Service

This service provides methods aiming to predict where the user will go and at what speed. It needs the user metadata, especially the GPS coordinates of the user, as described above.

The service provides a method to calculate when the user will come into range of a device's Wi-Fi signal:

```
1 long getTimeUntilArrival(UserMetadata metadata);
```

This method takes the user metadata as input and based on information in the metadata predicts when this user will arrive. It returns the result as time in milliseconds (ms) until the user will come into range.

This class also provides the method:

```
1 long getConnectionDuration(UserMetadata metadata);
```

This method estimates for how long a user will stay within this device's Wi-Fi signal range and returns this value as time in ms. In order to do so, it needs a way to predict speed of movement of the user. This information will typically be given in the user metadata.

Finally, this service provides the method:

```
1 NeighbourData getNextDevice(UserMetadata metadata);
```

Based on the user metadata, this method calculates the next device the user will come into range. It returns the necessary information about the next device as a NeighbourData object, a model class containing the id, the IP address as well as the port of a neighbouring device. This information is required to pass on FETCH messages.

Time Prediction Service

This service provides information about when to start prefetching and what data units should be prefetched by this device and which should be passed on to the next device. In order to do so, it needs the information provided by the Data Prediction and User Mobility Prediction services. It furthermore needs information about the size of a typical data unit.
It contains a method calcFetchData(), with the following signature:

```
Map<String, List<Integer>> calcFetchData(String userId,
  List<Integer> dataIds, long connectionDuration,
  int transferSpeed, int sizeOfDataItem, long arrivalTime);
```

By taking information from the User Mobility Prediction service on how long the user will be in range (parameter connectionDuration) as well as the transfer speed and the size of a typical data item, it calculates the data volume that can be sent to this user by the current device. The arrival time of the user (in ms from now) is also needed in the algorithm as an estimation on how much time there is for prefetching. Based on the data volume and time for prefetching it divides the data ids in data that should be returned to the user by this device and data ids that need to be passed on to the next device. The ids of the data for the current device are placed in a map with the key “this”, while the ids for next device can be obtained via the key “next”.
Furthermore, the service provides a method calculating when prefetching should be started, in order to get the most current data:

```
long calcFetchTime(List<Integer> dataIds, long arrivalTime,
  int transferSpeed, int sizeOfDataItem);
```

This method takes the ids of the data that need to be fetched, the arrival time of the user (provided by the User Mobility Prediction service), the back-link connection speed (transferSpeed) and the typical size of a data item. Based on this information, it returns the time span in ms until prefetching needs to start.

CHAPTER 4

Implementation

All actions on IoT devices occur in reaction to messages received from the user or from other devices. Therefore, the following section covers all message types before explaining in detail the actions taken in response to these messages.

4.1 Message Passing

There are three types of messages. Two message types coordinate communication between IoT devices (namely GET and FETCH messages) and one message type represents messages from the user. All messages are parsed to JSON format before sending. We use the gson library from Google [gso] for converting Java objects into JSON and vice versa.

4.1.1 User Message

A message from the user (sent from the Android App developed in this project) contains the following information:

- A unique user id: This id would normally be given to users upon registering for the service. In our case it is typed in by the user.
- The current GPS position of the user as provided by the GPS unit of the mobile device.
- The GPS position of the destination: The user can provide this information in the app. This information could also come from other sources, e.g. from a navigation app.
- Direction information: The direction the user is headed towards. This information is necessary to determine which IoT device the user will reach next.

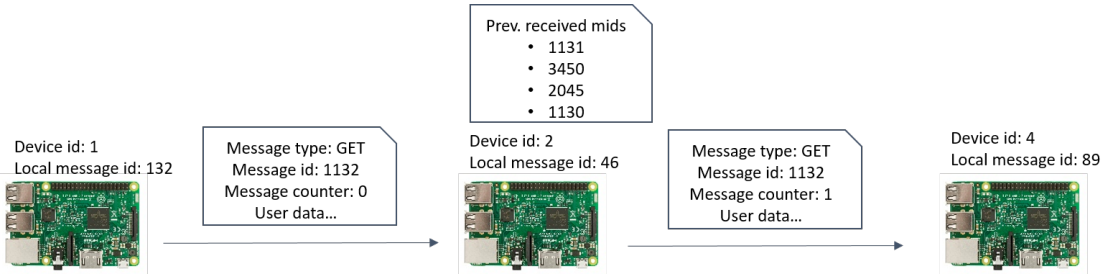
- Movement speed: The current movement speed of the user in m/s. This information is required to determine how long the user will be in range of the Wi-Fi signal of an IoT device and how long it will take to reach the next IoT device.

4.1.2 GET message

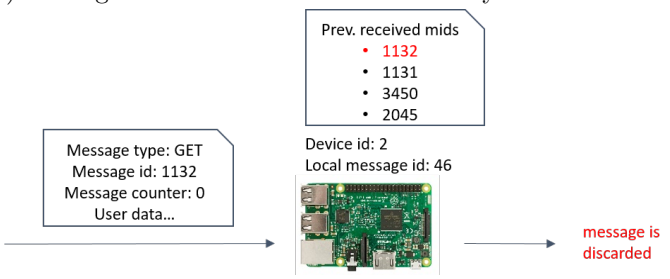
This message is passed from one IoT device to neighbouring IoT devices to request data for a user. Messages are passed in a completely distributed manner, meaning that each IoT device only is aware of devices which are direct neighbours to it. In order to reach all nodes in the system messages are passed in a recursive manner. If the device has the required data units stored locally it will return this information. Otherwise, the device will pass the message along to its neighbours (excluding the sender) and return the "best" answer from its neighbours. That way, every device in the network will pass on the message to neighbouring devices or return the required information. Eventually all devices in the network will receive the message and the required information will be returned. Each GET message contains the following information:

- Message id: A unique id for the message.
- Message counter: A number which starts at zero and is incremented at each device passing on the message.
- Device id: The id of the device which originally sent the message.
- User metadata: The metadata of the user, which contain information on which data are required by this user.
- Data ids: The ids of the data units that need to be collected. If this information is available it will take precedence over the user metadata and the specified data units will be returned, regardless of the user metadata.

To make sure that GET messages are not passed along between devices endlessly, some precautions were taken. The message id contains the id of the sender device plus an id for the message on this device. The device id prevents that two devices can send messages with the same id. Upon receiving a message each device will store the message id in a queue. If it already saw this id, it will not pass on the message and therefore prevent loops in message passing. The message counter will be incremented by each device passing on the message. If this counter reaches a certain threshold (devices hops), the message will also be discarded, preventing an endless passing on of messages. This process is depicted in Figure 4.1. In part a) of the figure the message has not been received before and is passed on, while in part b) an error case is depicted where the receiving device has already seen the message.



(a) Message has not been received before by device with id 2



(b) Message was received before by device with id 2

Figure 4.1: GET message format and distribution

4.1.3 FETCH Message

Messages of this type are sent when an IoT device receives a message from a user. This message informs the receiving device that a user is coming and which data need to be prefetched for this user. FETCH messages are structured as follows:

- Device id: The id of the device which sent the message.
- User metadata: The metadata of the user which contain information on which data are required by this user.
- Data ids: The ids of the data units that need to be collected. In contrast to GET messages, this information needs to be available for FETCH messages.

FETCH messages are designed to ensure that the user receives the correct information in a timely manner and that no information is sent multiple times from multiple devices. The first device that a user will contact, will have no information about this user yet. Therefore, it will always have to fetch data reactively. First, the device will determine which data ids are needed. The data ids will then be divided into units which can be sent to the user while he/she is in range of the current device and units which need to be sent to the user by the next device. Only the ids of the latter are passed on in the FETCH message. That way the user receives the first part of information from the current device and only the remaining data units will be sent by the next device the user comes into range.

4.2 Networking and Request Handling on the IoT Device Application

This section will focus on the classes which are involved directly in handling as well as sending messages.

4.2.1 Main Class

This class provides the program entry point. It sets up two request listeners, which listen on two different ports, for TCP messages from the user and other devices, respectively.

4.2.2 User Request Handler

The main purpose of this class is the communication with the user. It waits for user requests and returns the data items, required by the user. In order to provide the correct data, it calls several helper services with specialized functionality. The order of events upon receiving metadata from a user is illustrated in Figure 4.2. This sequence diagram shows the methods being called in different services as well as the objects being generated during the whole process.

Validate and save user metadata: In the first step, we check whether the user metadata contain all necessary information. If metadata for this user were stored previously (e.g. when prefetching data for this user), they are retrieved and compared with the currently received metadata. In case both data sets are the same, the device has already received a FETCH message for this user from the previous device. If no metadata were previously stored for this user or the metadata changed, new data need to be fetched for this user.

Fetch required data/data ids: If the correct data for this user have already been prefetched, they can simply be retrieved from storage and the following code is skipped.

Listing 4.1 shows the code which manages fetching of data. First a boolean "separated" is initialized, which will store whether the data ids have already been separted into data ids that need to be returned by this device and data that will be passed on to the next device in a FETCH message. Usually the device separates the data ids directly upon receiving a FETCH message (see Subsection 4.2.3). So by default this boolean is set to true. The method then checks whether data are currently in the process of being prefetched (line 5). If this is the case, the FutureTask currently fetching the data can be retrieved from storage via the user id. In case no prefetching took place for this user (handlerFuture == null), the data need to be acutely fetched now. Also the data ids were not separated yet and the boolean "separated" is set to false (line 8). To fetch the data, a new Prefetching Handler object is generated and wrapped in a new FutureTask (lines 9 and 10). The PrefetchingHandler is passed the user metadata and the separated data ids. The last parameter is null in this case as no data ids were prefetched for this user. Detailed information on the code executed in the Prefetching Handler is presented

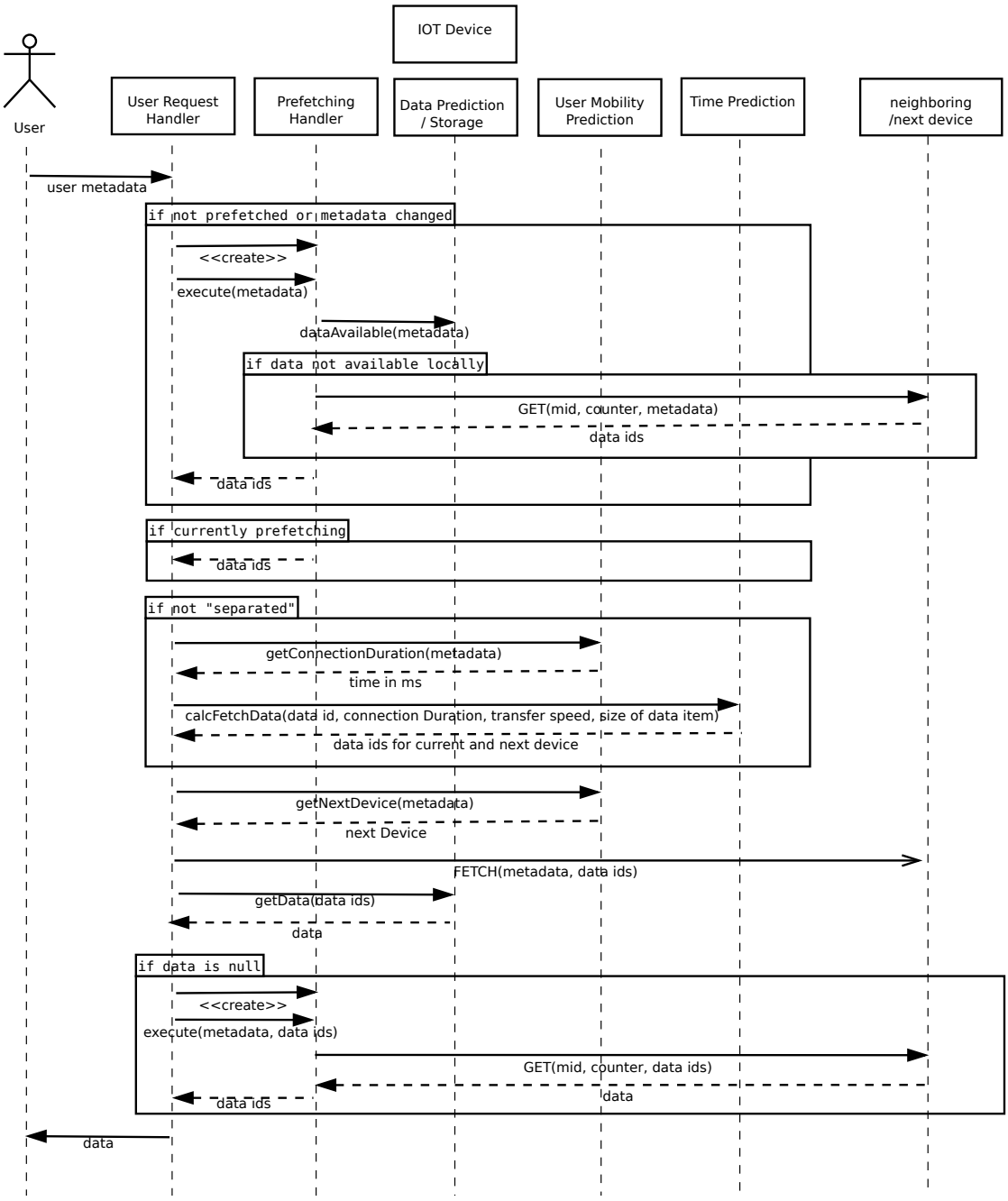


Figure 4.2: Sequence of events in response to a message from a user

in Subsection 4.2.4. In both cases, the active process then waits for the FutureTask to finish by calling the get() method on the task (line 14).

Listing 4.1: Manage Prefetching Handler

```
1  import java.util.concurrent.FutureTask;
2
3  boolean separated = true;
4  //Are data currently being fetched?
5  FutureTask<Map<ForDevice, List<Integer>>> handlerFuture =
       dataStorage.getPrefetchingHandler(metadata.getUserId());
6  if(handlerFuture == null){
7    //No they are not. So generate a new Handler to fetch them.
8    separated = false;
9    PrefetchingHandler handler = generateHandler(metadata,
         separatedDataIds);
10    handlerFuture = new FutureTask<>(handler);
11   new Thread(handlerFuture).start();
12 }
13 //Now wait for the Prefeching Handler to finish.
14 separatedDataIds = handlerFuture.get();
```

Separate data ids: If the data ids, returned by the FutureTask, were not separated already (boolean "separated" is false), this needs to be done now. In order to do so two service methods need to be called (see Figure 4.2). The getConnectionDuration() method in the User Mobility Prediction service is called first to predict how long the user will stay in range of this device. This information in then passed to the calcFetchData() method of the Time Prediction service which in turn separates the data retrieved for the user into the two sets described above.
Send FETCH message to next device: After the previous step, the ids of the data units that are required by the user are available. If the set of data ids that need to be sent to the next device is not empty, the User Mobility Prediction service is called to predict which device the user will reach next. A FETCH message containing the data ids from the set and some other required information (see Section 4.1.3) is then prepared and sent to the next device.
Return the data items to the user: Data items scheduled to be returned by this device can now be obtained from storage. In case no prefetching took place, the Prefetching Handler set up above only returned the data ids and not the full data, in order to speed up the retrieving process and not fetch data items which might not be needed by the current device. In that case, the full data items need to be obtained now. Therefore, a new Prefetching Handler is created, which receives the data ids needed by the device. After the Prefetching Handler process finished executing the data units are available in storage. They are now converted into JSON format and returned to the user.

4.2.3 Device Request Handler

This class is responsible for handling the communication between IoT devices. In particular, it responds to two types of messages received from other devices.

Response to GET messages

GET messages request data from other devices. Therefore, upon receipt of a GET message the device checks whether it has the requested data available and returns those data or passes on the message. In detail the process involves the following steps:

Message validation: Each device keeps a queue, where the most recent message ids are saved. If the id of this message is in the queue, the message has already been seen by this device previously and is therefore discarded. Otherwise the message id of the current message is added to the queue. Next the counter of the message is checked. If the counter is larger than the maximal number of allowed hops, the message is also discarded. This has been explained in detail in Section 4.1.2.

Data retrieval: GET messages have the user metadata and optionally also the ids of the required data items specified. If the ids are given, the device checks whether the requested data items are available locally. In case they are available, the requested items are returned. If no data ids were included in the message, only the data ids, not the actual data items, for the given user metadata are returned.

Request data from neighbours: In case the requested data are not available locally, the message is passed on. The device increments the message counter and passes on the rest of the message unchanged. The modified message is sent to all neighbours with exception of the sender of the message. The sending and receiving of data over a TCP connection is done in the DataFetchingService, described in Section 4.2.5.

Response to FETCH messages

FETCH messages inform the receiving device, that a user will be arriving shortly and that the device should start prefetching data. Figure **??** depicts the actions taken in response to a FETCH message. A FETCH message must have the user metadata as well as the ids of the data that should be prefetched specified. If any of these are missing, an error message is returned. First the receiving device validates the message and saves the user metadata. Then the received data ids are separated into data that need to be prefetched by the current device and data items that are kept for the next device. To do so, two service methods are called. The result of the getConnetionDuration() method of the User Mobility Prediction service is passed to the calcFetchData() method of the Time Prediction service, which in turn separates the given data ids (see Figure 4.3). Afterwards a new Prefetching Handler is generated, which handles the prefetching of the specified data items. See Section 4.2.4 for detailed information about data prefetching.

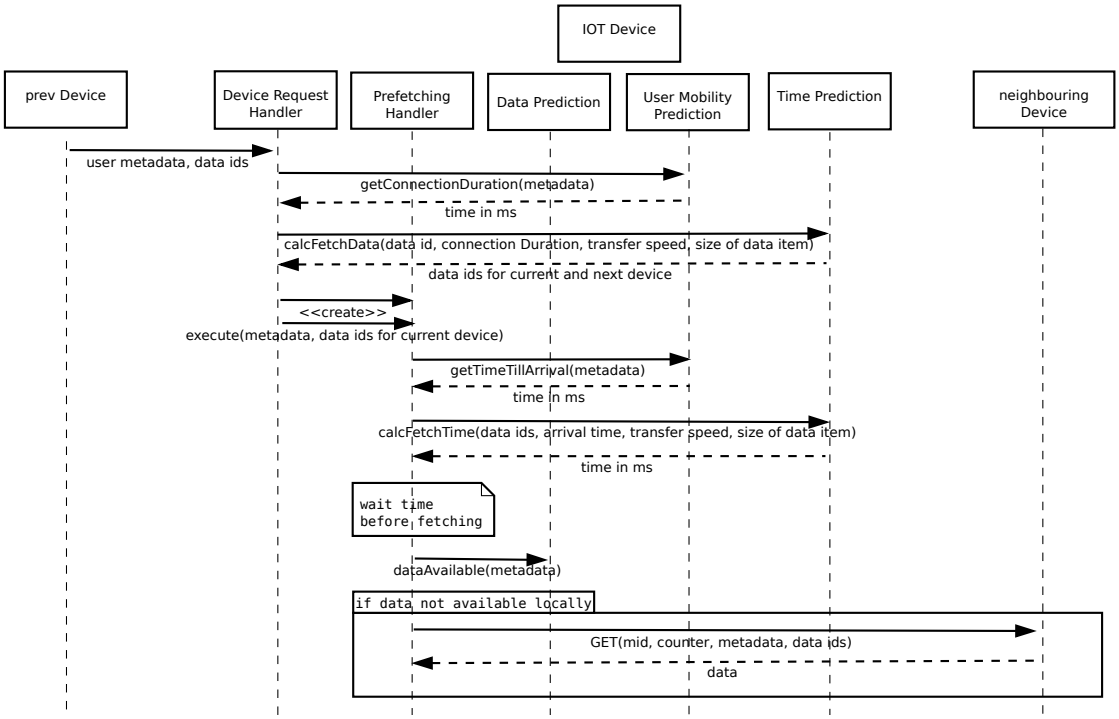


Figure 4.3: Sequence of events in response to a FETCH message from another device

4.2.4 Prefetching Handler

This class is responsible for (pre)fetching data. It is called by the User Request Handler if data for a user were not prefetched and need to be fetched urgently and by the Device Request Handler upon receipt of a FETCH message. This class implements the interface Callable and is executed as a Java FutureTask in a new thread. It requires that the user metadata and/or the ids of the data that need to be fetched are set. In order to fetch data at the last possible time point and therefore get the most current data, the thread may sleep a calculated amount of time before fetching the data. This step can be skipped, when a user is already in range, by setting the flag urgent to true. The process of data prefetching is also illustrated in the sequence diagram depicted in Figure 4.3. If the urgent flag is not set, first the User Mobility Prediction service is called to calculate the time of arrival of the user. The result is passed on to the calcFetchTime() method provided by the Time Prediction service, which then calculates when the data fetching needs to start, in order to obtain the results right before the user arrives. See Section 4.4 for a detailed description of these service methods. The thread then sleeps for the calculated amount of time, before fetching the data via the fetchData() method of the Data Fetching service. In case the data are already available on this device, this step is skipped. The obtained results are then validated and saved in storage. From there, the data can be accessed by the Request Handler classes.

4.2.5 Data Fetching Service

This class provides methods to communicate with other IoT devices. Communication between IoT devices is implemented in a distributed manner, meaning that there is no central unit coordinating traffic between the devices. Each device can only communicate with neighbouring devices. So data stored on a device further away may need to pass several links before reaching the destination device.

```
1 String fetchData(String message, NeighborData sender);
```

This method sends the provided GET message to all neighbouring devices, excluding the original sender in case this message is just passed on. In order to send the message to all the neighbours in parallel, we use the Java ExecutorService with a fixed sized thread pool. For every neighbour a new Data Fetching Handler object is generated, which implements the Java Callable interface. The Data Fetching Handler opens a client socket to the neighbour and passes this socket to a new TCP Channel object. The TCP Channel class implements the Channel interface which provides two methods, one for sending and one for receiving data over the provided connection. Via this TCP channel the Data Fetching Handler sends the request and then waits for a response. Execution of all handler threads is started with invokeAll() on the thread pool which returns a list of Future objects. We loop through this list, calling get() on each Future object. This method blocks until the underlying Data Fetching Handler finished executing and returns the answer from the neighbour. The first answer from a neighbour containing a valid data id is returned. If an error occurred or data were not sent by a neighbour for some other reason (e.g., if the neighbouring device has already seen the message), the neighbour will return a data id of -1, signifying a problem.

This class furthermore provides a method to send a message to a specific device, used for sending FETCH messages. As this method only sends the message to one neighbour and does not wait for a response, it does not require separate threads for execution. Therefore, this method uses the TCP Channel class directly without creating Data Fetching Handler objects.

4.3 Test Setup using Live Timetable Information from the Wiener Linien

To evaluate our application, we use live traffic information data from Wiener Linien. Of course every device in our setup can collect this information via the Wiener Linien API. However, the IoT devices in our application are not meant to have all data available on every device, as this would make prefetching obsolete. Therefore, this section will describe how data are collected from the Wiener Linien API and how we determine which device has which data units available.

The Wiener Linien API uses platform ids, so called RBL numbers. Upon sending

a request to `http://www.wienerlinien.at/ogd_realtime/monitor` with the RBL number as well as an API key as parameters, the server returns live timetable information for that particular platform. In order to get all stations and available platforms for a particular GPS position, the information from a total of three CSV files needs to be combined. We use an online tool to combine all required information into one single JSON file [hac]. The tool takes the three CSV files [wlC], containing station, line and platform data, respectively, as input and generates an output that looks as follows:

Listing 4.2: JSON file containing Wiener Linien station, line and platform information (generated via [hac])

```
{
  "stationID":"214460771",
  "relatedLines":"U1|14A",
  "latitude":"48.178132111491",
  "longitude":"16.3765391577784",
  "name":"Keplerplatz",
  "platforms":[
    {
      "line":"14A",
      "rbl":"762",
      "latitude":"48.1780782003958",
      "longitude":"16.3761708485119"},
    {
      "line":"14A",
      "rbl":"748",
      "latitude":"48.1781141411322",
      "longitude":"16.3761798316648"},
    {
      "line":"U1",
      "rbl":"4103",
      "latitude":"48.1792941813162",
      "longitude":"16.3762696631932"},
    {
      "line":"U1",
      "rbl":"4126",
      "latitude":"48.179258241407",
      "longitude":"16.3761169495949"}
  ]
}
```

To divide the responsibilities between the IoT devices, we partitioned the city area of Vienna into six squares based on the GPS coordinates. The first square includes all GPS coordinates within the latitude range from 48.20 until 48.25 and the longitude range from 16.29 until 16.35. Each Raspberry Pi is assigned such an area and will only provide information about this area. If a user requests information about an area the current device is not assigned to, the device will ask the neighbouring devices for this information.

By doing this, we emulate that one IoT device only has access to certain data units and needs to fetch information from other devices, if the user requests data not stored on the current device.

4.4 Services on the IoT Device Application

This section will focus on services running on the IoT device application, predicting which data a user might need, user mobility, timing etc. In this work, we define the service interfaces which are meant to be implemented according to the needs of individual projects. Therefore, our implementation of these services is kept simple as the focus of this work is the message handling backbone of the application.

4.4.1 DataPredictionService Implementation

This service reads the GPS range for each device (see Section 4.3) from a configuration file and saves this information in a list of GPSbounds objects. GPSbounds is a wrapper class containing fields for the minimum and maximum latitude and longitude of each area. One GPSbounds object is needed per device. To keep the configuration files simple, each device reads in all GPSbounds and then uses the correct one based on its assigned device id. The assigned GPS range is needed in the method:

```
1 boolean isResponsible(UserMetadata metadata);
```

In our implementation, this method takes the GPS position of the destination of the user (field destinationPos). If this GPS position lies within the assigned GPS range for this device, the method returns true, otherwise false.

This service also overrides the method:

```
1 List<ResponseData> getData(UserMetadata metadata,
2   List<Integer> dataIds);
```

This method first checks if data ids are provided (parameter dataIds). If dataIds is null or the list is empty, the data ids need to be calculated from the user metadata information. We do this by taking the GPS position of the destination from the user metadata and calculating all Wiener Linien stops within a specified radius. The following code fragment illustrates this process:

Listing 4.3: Get data ids according to user destination

```
1 for(WienerLinienStationInfo stop : wlData){
2   Position stationLocation = new Position(stop.getLatitude(),
        stop.getLongitude());
3   if(calcDistance(destinationLocation, stationLocation) <= radius){
4     Platform[] platforms = stop.getPlatforms();
5     for(Platform platform : platforms){
6       rblNumbers.add(platform.getRbl());
7     }
8   }
9 }
```

We loop through a list of WienerLinienStationInfo objects. This list represents the object form of the JSON file shown in Listing 4.2. The GPS position of the station is extracted from the object and stored in a Position object (line 2). Then the distance in meters between the destination of the user (destinationLocation) and the station is calculated, taking the radius of the earth into account. If this distance is within a specified radius, then the station is close to the destination of the user and live timetable information of that station will be collected. To do so, the platform information is extracted from the WienerLinienStationInfo object and the platform ids (rblNumbers) are returned (lines 4 to 6). If the data ids were not specified (parameter dataIds is null), then the method returns only those platform ids. Otherwise, the data for each platform id are then fetched by sending a request to the WienerLinien API. For each platform, a ResponseData object is generated containing the id and timetable information as received from the WienerLinien API. Finally, the method returns these ResponseData objects in a list.

4.4.2 UserMobilityPredictionService Implementation

This service overrides the following methods:

```
1   long getConnectionDuration(UserMetadata metadata);
```

We assume that this method is called as soon as a user enters the device's Wi-Fi range. From then on, the user will have to move the full distance between two devices, before switching to the Wi-Fi network of the next device. Therefore, in our simple implementation, this method takes the average distance between two devices in meters and divides this value by the movements speed of the user (in meters/second) as given in the user metadata. To return the time period that the user will be connected to this device in milliseconds, it multiplies the calculated value by 1000.

```
1   long getTimeUntilArrival(UserMetadata metadata);
```

In our setup, the Wi-Fi signals of neighbouring devices overlap at the edges. Therefore, a user will connect to a device directly after disconnecting from the previous device.

With this setup, the time until arrival of a user for the current device is the same as the remaining connection duration of the user to the previous device. The latter duration is already calculated by the getConnectionDuration() method, described above. However, the above method assumes that a user has just now arrived and the getTimeUntilArrival() method will be called when a device receives a FETCH message. At that time point, the user is already connected to the previous device for some time, depending on whether the previous device had to fetch data. Therefore, we multiply the result from the getConnectionDuration() method with the factor 0.8 and return the result.

```
1   NeighborData getNextDevice(UserMetadata metadata);
```

This method calculates the next device the user will come into range. It takes direction information from the user metadata. This can be a simple geographic direction or more advanced data based on navigation information. It then calculates which of the neighbouring devices the user will reach next. The returned NeighbourData is a wrapper class containing the device id, the IP address as well as the port of the device.

4.4.3 TimePredictionService Implementation

This class overrides the following methods:

```
1 Map<String, List<Integer>> calcFetchData(String userId,
2   List<Integer> dataIds, long connectionDuration,
3   int transferSpeed, int sizeOfDataItem, long arrivalTime);
```

The method first calculates the number of data items that can be retrieved and sent to the user by the current device. This is done as follows:

```
1   int sendableDataItems = (int)
       Math.floor(((double)connectionDuration)/1000 * transferSpeed /
       sizeOfDataItem);
```

It takes the connectionDuration in ms and divides by 1000 for seconds. By multiplying this value with the average transfer speed (in bytes/secs) we get the total data volume in bytes that can be sent to this user. To get the number of data items that can be sent, this value is divided by the average size of a data item. In our simple implementation we only check if the parameter arrivalTime is smaller than or equals zero. If this is the case, the user is already connected to this device and the number of sendable data items is divided by a factor 2. The method then creates a map containing two lists with keys “this” and “next”. The first list contains the ids of the items that can be sent to the user by the current device. While the data ids stored under “next” should be passed on the next device. To fill these lists, the method loops through all the data ids (parameter dataIds) and puts the first n (int sendableDataItems) ids into the list “this”, while the remaining data ids are put into the list “next”. This very simple implementation only

uses average transfer speeds as well as data item size. Also we do not distinguish between back-link transfer speed and speed of the Wi-Fi connection to the user. Typically, this method will be called upon receipt of a FETCH message. At this time point the user is not connected to this device yet. If there is enough time for prefetching before the user arrives, the back-link transfer speed will not be important, as the data items should be available locally when the user connects to this device. However, in a full implementation, both transfer speeds need to be taken into account, as the back-link speed will determine the data volume that can be prefetched before the user arrives.

```
1 long calcFetchTime(List<Integer> dataIds, long arrivalTime,
2   int transferSpeed, int sizeOfDataItem);
```

As the method before, this method will also be called upon receipt of a FETCH message. It returns the time in ms until prefetching needs to start in order to receive all the given data items before the user arrives. In this implementation, we first calculate how long the fetching of data will take. To do so, we divide the total fetch volume (number of data ids multiplied by the average size of a data item) by the average back-link transfer speed. Then we subtract this value from the time span until the user will arrive (arrivalTime) as returned by the User Mobility Prediction service.

4.5 Implementation of User Software (written for an Android Device)

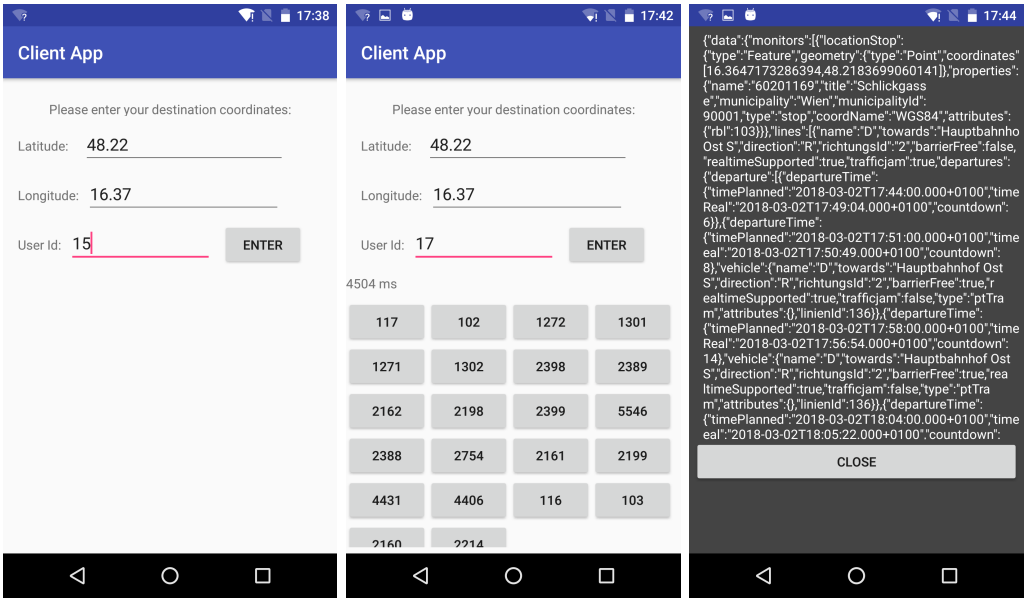
The user device (an Android mobile phone) sends user metadata to each IoT device it comes into range and displays the response to the user.

4.5.1 User Interface

In our prototypical implementation of the app, the user needs to manually type in the GPS coordinates of her destination as well as the user id. In a commercial implementation, the GPS information would typically come from another app, like a navigation app. The user id would typically be generated automatically upon registration of the user for a service. Figure 4.4a shows the input fields for the GPS coordinates and the user ID, as displayed in the app. Upon clicking the "send" button, the device will send the generated metadata, consisting of the user id and destination GPS coordinates given by the user. Other data fields (compare Section 4.1.1) were left empty for the prototypical implementation.

Each time the user device connects to another Wi-Fi network it resends the user metadata. Below the input fields, the response from the IoT device is displayed (see Figure 4.4b). The app displays a button for each data item with the data id as the label. Upon clicking the button, the content of the data item is displayed in an overlay window (see Figure 4.4c). In our implementation, we show the content as a JSON string, as it was received from the Wiener Linien API. For test purposes, the device also measures the time in ms

from sending the metadata upon receipt of the response and displays this information above the response data items (see Figure 4.4b).



(a) Input Fields (b) Overview of Results (c) Detailed View of Results

Figure 4.4: Mobile App: User Interface

4.5.2 TCP Client

This class opens the socket to the IoT device application, sends the metadata and returns the response. Within this class, we declare an interface called “OnMessageReceived” with one method having the same name. This interface is implemented in the Main Activity class and acts as a listener for received responses. The listener object is passed to the TCP client class in the constructor.

Each IoT device assigns itself the same local IP address within the generated Wi-Fi network. Therefore, the mobile device can always send the user metadata to the same local IP address and port. In the run() method, the TCP client opens a socket to this fixed local IP address, sends the metadata it received from the Main Activity class and waits for an answer from the server. The server answer is then passed back to the Main Activity class via the listener object (see Listing 4.4, line 5).

Listing 4.4: TCP Client - Returning the Server Answer to Main Activity

```
1 // wait for the answer from the server
2 serverAnswer = in.readLine();
3 if (serverAnswer != null && messageListener != null) {
4   //call the method messageReceived implemented in the Main Activiy
        class
5   messageListener.messageReceived(serverAnswer);
6 }
```

4.5.3 NetworkStateReceiver

This class triggers the sending of the user metadata to each new IoT device that the mobile device connects to. It extends the Android BroadcastReceiver class and is notified by the Android system when a change in network state occurs. Upon such a change, the onReceive() method is called and passed a Context and Intent object. From the Context object, we extract a NetworkInformation object, which contains information such as the name of the network (see Listing 4.5, lines 3 to 5). Upon connecting to a new network, the method checks whether this device has been previously (within the last 5 minutes) connected to this network. In that case, the user metadata have already been sent to this device previously and no action needs to be taken now (Listing 4.5, lines 7 to 9). Otherwise, the sending of a new message, containing the user metadata, to the connected device is triggered. This is done by calling the method resendMessage() in the Main Activity class (described in Section 4.5.4, see Listing 4.5, line 13).

Listing 4.5: NetworkStateReceiver Class

```
1  @Override
2  public void onReceive(Context context, Intent intent) {
3    ConnectivityManager connectivityManager =
         context.getSystemService(CONNECTIVITY_SERVICE);
4    NetworkInfo ni = connectivityManager.getActiveNetworkInfo();
5    String currentNetworkName = ni.getExtraInfo();
6
7    if(lastNetwork.equals(currentNetworkName) && timestamp >
         (System.currentTimeMillis()/1000 - COOLDOWN)){
8      //We already sent out info to this network
9      return;
10   }
11   lastNetwork = currentNetworkName;
12   timestamp = System.currentTimeMillis()/1000;
13   mainActivity.resendMessage();
```

4.5.4 Main Activity

This class is the program entry point. It is responsible for displaying all the input fields as well as the results. It also initializes the NetworkStateReceiver and TCPClient classes, described previously. Upon clicking the "send" button on the user interface, the sendMessage() handler method is executed. This method reads in the GPS coordinates and user id from the input fields, validates the given data and builds the corresponding user metadata JSON string. For simplicity, we only read in the destination position and user id. Other information, like the direction the user is heading, current position etc. are left blank, as they are not needed in our test setup. This method also clears previous results, if present. After generating the message, the sendMessage() handler generates a new ConnectTask, which extends the Android AsyncTask class and executes it. The ConnectTask overrides two methods from AsyncTask, namely doInBackground() and onProgressUpdate() as shown in Listing 4.6. The doInBackground() method generates a new TCPClient object, handing it a new implementation of the OnMessageReceived interface. This implementation overrides the messageReceived() method to trigger publishProgress() upon being called (Listing 4.6, line 6). The TCPClient object will call this method upon receiving an answer from the server (see Subsection 4.5.2). After generating the TCPClient, we pass the message to be sent and then call the method run() (Listing 4.6, lines 9 and 10). All of this happens in the background, with the user interface staying responsive. The onProgressUpdate() method is called, when the answer from the server has arrived. This method then displays the time it took to receive the response and the response on the screen (Listing 4.6, lines 17 to 19).

Listing 4.6: ConnectTask Class

```
1  @Override
2  protected void doInBackground(String... message) {
3    this.tcpClient = new TCPClient(new TCPClient.OnMessageReceived() {
4      @Override
5      public void messageReceived(String message) {
6        publishProgress(message);
7      }
8    });
9    this.tcpClient.setMessageToBeSent(message[0]);
10   this.tcpClient.run();
11 }
12
13 @Override
14 protected void onProgressUpdate(String... values) {
15   super.onProgressUpdate(values);
16   showTime(System.currentTimeMillis());
17   saveResponse(values[0]);
18   displayResponse();
19 }
```

The Main Activity class also has a method called resendMessage(), which is called by

the NetworkStateReceiver class upon connecting to a new Wi-Fi network (see Subsection 4.5.3). This method resends the previously generated metadata JSON to the newly connected IoT device. Just like the sendMessage() method, it generates a new ConnectTask and executes it. However, it does not clear the results view. Therefore, the results received from this IoT device are appended to the previously received results.

CHAPTER 5

Evaluation

To evaluate the application, we tested the response times with and without data prefetching. The response time is the time span from when the user application sent a request until it got the response from the IoT device.

5.1 Test Setup

As mentioned in Section 1.3.2, we used five Rasperry Pis, representing IoT devices, in our test setup. The Rasperry Pis were positioned on two separate floors of the institute building, approximately 10 – 30 meters apart. All the devices were connected via LAN in one network. However, each device only had means to communicate with its direct neighbours. Of course in a real IoT setting, devices might be much further apart from each other, being connected only by a slow mobile connection. To simulate a slow connection and limit backlink transfer speed, we used the SpiceJ library written by Michael Borkowski [bor]. This library takes the input stream from a TCP socket and reduces the bytes transferred per time interval to a specified amount. For our tests, we reduced the input stream of GET messages, as these carry the bulk of the transferred data.

This limitation is implemented in a RateLimited Channel class which implements the Channel interface and uses the TCP Channel class described in Section 4.2.5 for reading and writing messages over the TCP connection. On construction, the RateLimited Channel calls the method rate() on the InputStream and passes the returned rate-limited stream on to the underlying TCP Channel. The rate() method uses several classes from the SpiceJ library. It first generates a RateCalculator.Result object (see Listing 5.1, line 9), used for converting the rate from bytes per second to “ticks“. The Stream.limitRate() method then takes the input stream and the calculated values from the RateCalculator.Result object and returns a rate-limited stream (Listing 5.1, line 10).

Listing 5.1: Limit input stream transfer speed using SpiceJ library

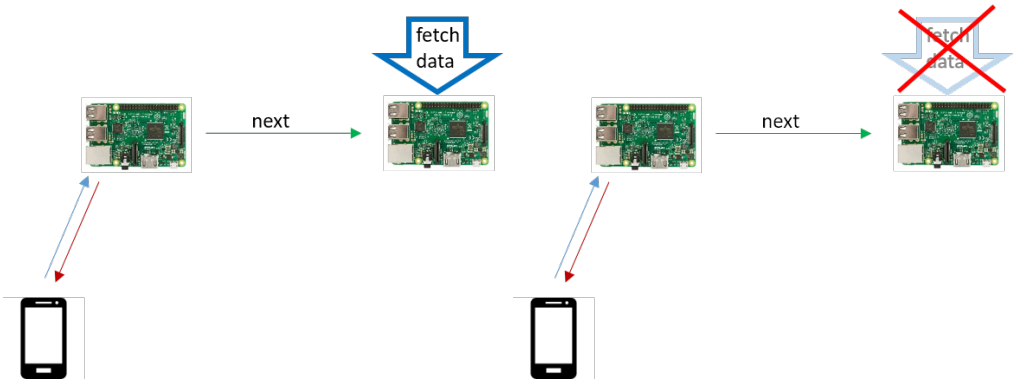
```
1  import at.borkowski.spicej.Streams;
2  import at.borkowski.spicej.rt.RateCalculator;
3  import at.borkowski.spicej.rt.RealTimeTickSource;
4
5  private static InputStream rate(InputStream inputStream, Float rate)
       {
6      if (rate == null)
7         return inputStream;
8
9      RateCalculator.Result calculation =
           RateCalculator.calculate(rate);
10     return Streams.limitRate(inputStream, new
           RealTimeTickSource(calculation.getTickNanosecondsInterval(),
           true), calculation.getBytesPerTick(),
           calculation.getPrescale());
11 }}
```

Each device generates a Wi-Fi network, which is used for interacting with the user device. The mobile device will connect to the device with the strongest Wi-Fi signal and send a user message, asking for specific data. These data are live traffic information from the Wiener Linien as described in detail in Section 4.3. In our test setup, the user will traverse all devices in order of their device id, either ascending or descending. The information whether a user is going in ascending or descending direction is passed from one device to the next in the FETCH message. Before sending a FETCH message, each device writes its own device id into the directionInfo field of the user metadata. The receiving device then compares the received device id with its own device id. If its own device id is higher, the user is assumed to be going in ascending direction and the method will return the neighbouring device with the next higher device id. Similarly, if its own device id is lower, it will return the neighbouring device with the next lower device id.

We compared the response times with and without prefetching. With prefetching, each device sends a FETCH message to the next device the user will come into range, informing that device which data items are needed and initializing the prefetching of these data (see Figure 5.1a). For comparison, we implemented a solution, where each device still sends a FETCH message. The receiving device keeps the information on which data items are needed by the user but does not prefetch these data items. Therefore, the next device will only fetch the required data items, upon receiving a message from the user. This scenario is depicted in Figure 5.1b.

5.1.1 Test Runs

We tested responses to four sets of user metadata, having differences in the GPS coordinates of the destination the user is heading towards. Therefore, each set of metadata requires different data items to be fetched. The coordinates were selected randomly



(a) Prefetching. (b) No prefetching.

Figure 5.1: Prefetching vs. no prefetching. *Green arrows visualize the message passed to the next device, to initialize prefetching. Large arrows with blue outline represent data being prefetched. Blue arrows indicate requests for data and red arrows indicate the data response.*

but making sure a decent number of public transport stops are located in that area. Each of these metadata was tested with two different backlink transfer speeds (1 kb/sec, 5 kb/sec) and with/without prefetching, making a total of 16 test runs. All test runs are listed in Table 5.1. This table also gives information on which device we started the test run (labelled Start Device), in what direction the user went (ascending or descending) and which device is responsible for fetching the data items from the Wiener Linien API (labelled Resp. Device).

5.2 Results

The response times from all 16 runs are listed in Table 5.2. We show the response times divided by the number of data items that were returned, giving the response time per data item. The number of data items returned are the same for all devices, as the underlying algorithm is the same. However, the last device sending data, often has less data items remaining to be sent, yielding a faster response. With a backlink transfer speed of 1 kb/sec each device returned two data items, while with a speed of 5 kb/sec 9 data items were sent per device. An exception was the first device of each run, which only sent half the number of data items. This reduction on the first device was implemented because this device did not receive a FETCH message from another device and therefore the first result in each run is the response time without prefetching (indicated with a $ sign). These times are excluded in further calculations. We also excluded response times from the device that is responsible for collecting the data directly from the Wiener Linien API (indicated by a # symbol). The call to the Wiener Linien API is executed at full connection speed, simulating data that is directly accessible on the device. Therefore, the responsible device in each run can generate the response much faster, as it does not

Table 5.1: Test Runs

Run	Start Device	Direction	GPS coordinates Lat	GPS coordinates Lng	Resp. Device	Speed	Prefetching
1	1	ascending	48.17	16.32	4	1 kb/s	yes
2	2	ascending	48.22	16.34	1	1 kb/s	yes
3	1	ascending	48.249	16.45	3	1 kb/s	yes
4	5	descending	48.22	16.37	2	1 kb/s	yes
5	1	ascending	48.17	16.32	4	1 kb/s	no
6	2	ascending	48.22	16.34	1	1 kb/s	no
7	1	ascending	48.249	16.45	3	1 kb/s	no
8	5	descending	48.22	16.37	2	1 kb/s	no
9	1	ascending	48.17	16.32	4	5 kb/s	yes
10	2	ascending	48.22	16.34	1	5 kb/s	yes
11	1	ascending	48.249	16.45	3	5 kb/s	yes
12	5	descending	48.22	16.37	2	5 kb/s	yes
13	1	ascending	48.17	16.32	4	5 kb/s	no
14	2	ascending	48.22	16.34	1	5 kb/s	no
15	1	ascending	48.249	16.45	3	5 kb/s	no
16	5	descending	48.22	16.37	2	5 kb/s	no

need to fetch data from other devices. The observed variability in response times can be explained partly by the differences in size between data items. For night bus lines these data items are rather short (∼120 characters) as no live information was available at the time of measuring while for some platform ids we received responses up to ∼3000 characters.

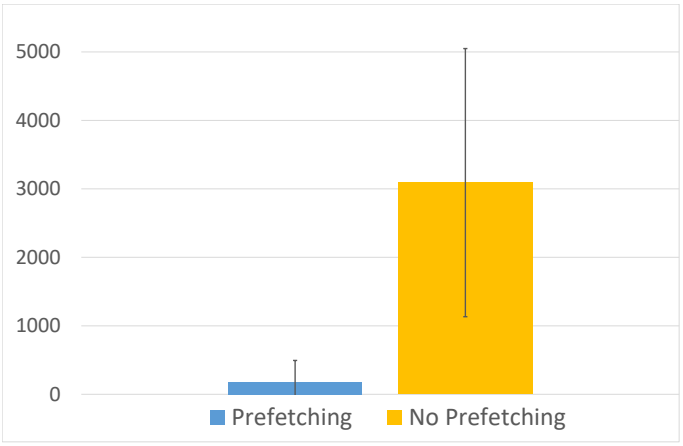
For a backlink connection speed of 1 kb/sec and no prefetching, it took an average of 3091 ms (± 1958) per data item to receive a response. With prefetching, the average response time per data item was only 180 ms (± 314), as depicted in Figure 5.2a. For a backlink connection speed of 5 kb/sec and without prefetching, the response time per data item was 689 ms (± 333) in average. While with prefetching it only took 26 ms (± 18) to receive a response per data item (Figure 5.2b). By prefetching data we saw a reduction in waiting time per data item by 2911 ms and 663 ms for a backlink transfer speed of 1 kb/sec and 5 kb/sec, respectively. A Student's t-test shows the significance of both reductions, giving P values of < 0.001 for both transfer speeds.

This work shows that data prefetching clearly reduces the response times and therefore can lead to a better user experience. The implemented prefetching solution still needs some work. One limitation of the current work is that while fetching data from neighbours, each device waits for the complete response before returning the data items. Speed of

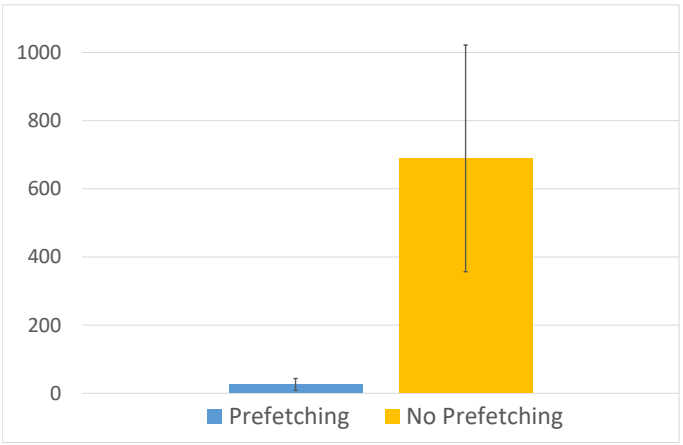
Table 5.2: Results of the Test Runs

	Time in ms until arrival of data / number of data items received				
Run	**Pi 1**	**Pi 2**	**Pi 3**	**Pi 4**	**Pi 5**
1	$ 7403	115	162	# 36	229
2	#	$ 5783	34	24	144
3	$ 1247	32	# 36	163	28
4	25	# 16	54	1154	$ 7587
5	$ 6685	7626	3124	# 183	3736
6	#	$ 4443	3546	708	4579
7	$ 1283	1534	# 77	1048	1555
8	2651	# 69	2150	4842	$ 7390
9	$ 1457	52	43	#	
10	#	$ 477	31	13	
11	$ 302	6	#13		
12	40	# 6	12	11	$ 1148
13	$ 1855	706	119	#	
14	#	$ 570	1044	1160	
15	$ 317	481	# 52		
16	666	# 46	500	839	$ 1926

fetching and prefetching could be increased by returning responses byte per byte as they are received. Better algorithms need to be implemented calculating the data that a user will need as well as other characteristics like where a user will go next and how long it will take him/her to arrive there etc. Currently, only very primitive solutions were implemented for these calculations. These solutions work in our controlled test setup, but will fail in a real-world IoT application. This was not the scope of this work, however, but needs to be addressed in future work.



(a) Transfer speed: 1kb/sec



(b) Transfer speed: 5kb/sec

Figure 5.2: Average Response times per data item and Standard Deviation with Prefetching vs. No Prefetching.

CHAPTER 6

Conclusion

In this work we developed a prefetching solution for IoT devices in a fully distributed environment. This means that there is no central node connecting the devices and data collected by each IoT device is stored locally.

In order to obtain data from another device, a request is sent to neighbouring devices. Each neighbour checks whether the requested data items are available locally. If they are not, the message is passed on to the neighbours of this device. Eventually the message will be received by the device, which has the requested data in storage and can return these data items. This means that several nodes might be passed before the data reach the device which sent the original request, resulting in a delay. A user requesting data from the closest device, might have to wait some time before the data items were fetched from a distant node and can be returned to her. This work provides a solution where the IoT device already expects which data might be needed by a user and prefetches them before the user comes into range. We also developed a simple mobile app for an Android mobile phone, which sends user metadata to a connected IoT device and displays the response.

In order to achieve prefetching of data, the IoT devices in the network communicate with each other via two different message types. One message type is for requesting data items, called GET message. The other message type, called FETCH message, is used to let the receiving device know that a user will arrive shortly and that it should start to prefetch certain data items. Upon receiving such a FETCH message, the device checks which data need to be prefetched, estimates how long this will take and also estimates when the user will arrive. This last estimation is based on the user metadata passed on in the FETCH message. Based on this information, the device then calculates when it needs to start prefetching in order to get the most current data versions to be sent to the user. The actual fetching of the data is done by sending GET messages to the device's neighbours.

The presented prefetching solution should be expanded. Especially better algorithms are needed estimating user behaviour, required data and fetching times. The software was designed in a way to be easily expandable. All methods used for calculation of user behaviour, timing etc. are specified in interfaces and can be easily overwritten. We implemented three service interfaces, namely a Data Prediction service, a Time Prediction service and a User Mobility Prediction service. The Data Prediction service provides methods aiming at predicting which data a user might need. The Time Prediction service provides methods estimating when prefetching needs to start in order to get the most current data and how many data items can be sent to a user by each device. Finally, the User Mobility Prediction service specifies methods to estimate when a user will arrive, the duration of the connection as well as where to user is heading towards. Good algorithms are needed for the prediction of all these variables in order to provide a correct and reliable prefetching solution.

In our test setup, we prefetched live traffic data from the Wiener Linien API. All the data items have unique ids and are provided in JSON format. In a business solution the data might not be as uniform. Sensors might collect data in different data formats. Mechanisms for converting data need to be implemented to provide a prefetching solution for a larger range of services and data types.

The current prefetching solution was tested using Raspberry Pi computers, running a Linux operating system. However, the hardware and also the software for IoT devices can vary a lot. Some sensors might only provide a very limited amount of computational power, requiring highly efficient algorithms to overcome this shortcoming in the hardware. The presented prefetching solution needs to be tested on other hardware and possibly be adapted to run smoothly on a variety of devices.

List of Figures

List of Tables

Bibliography

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[Akh00] Georges Akhras. Smart materials and smart systems for the future. *Canadian Military Journal*, 1(3):25–31, 2000.

[BMV10] Aruna Balasubramanian, Ratul Mahajan, and Arun Venkataramani. Augmenting mobile 3g using wifi. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 209–222. ACM, 2010.

[bor] Spice j. `https://github.com/michael-borkowski/spiceJ`. [Online; accessed 08-March-2018].

[BSSD16] Michael Borkowski, Olena Skarlat, Stefan Schulte, and Schahram Dustdar. Prediction-based prefetch scheduling in mobile service applications. In *Mobile Services (MS), 2016 IEEE International Conference on*, pages 41–48. IEEE, 2016.

[cis17] Cisco visual networking index: Global mobile data traffic forecast update, 2016–2021. 2017.

[DHX+13] Aaron Yi Ding, Bo Han, Yu Xiao, Pan Hui, Aravind Srinivasan, Markku Kojo, and Sasu Tarkoma. Enabling energy-aware collaborative mobile data offloading for smartphones. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2013 10th Annual IEEE Communications Society Conference on*, pages 487–495. IEEE, 2013.

[Fin10] Klaus Finkenzeller. *RFID handbook: fundamentals and applications in contactless smart cards, radio frequency identification and near-field communication*. John Wiley & Sons, 2010.

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[gso] Gson. `https://github.com/google/gson`. [Online; accessed 10-February-2018].

[hac] Wiener linien generator. https://github.com/hactar/WL-Generator. [Online; accessed 03-February-2018].

[HSHD14] Waldemar Hummer, Stefan Schulte, Philipp Hoenisch, and Schahram Dustdar. Context-aware data prefetching in mobile service environments. In *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on*, pages 214–221. IEEE, 2014.

[IWF+12] Selim Ickin, Katarzyna Wac, Markus Fiedler, Lucjan Janowski, Jin-Hyuk Hong, and Anind K Dey. Factors influencing quality of experience of commonly used mobile applications. *IEEE Communications Magazine*, 50(4), 2012.

[KKLS12] C Kalampokis, D Kalyvas, I Latifis, and VA Siris. Optipath: Optimal route selection based on location data collected from smartphones. In *Joint ERCIM eMobility and MobiSense Workshop*, page 4, 2012.

[LLY+13] Kyunghan Lee, Joohyun Lee, Yung Yi, Injong Rhee, and Song Chong. Mobile data offloading: How much can wifi deliver? *IEEE/ACM Transactions on Networking (TON)*, 21(2):536–550, 2013.

[MDT+16] Neal Master, Aditya Dua, Dimitrios Tsamis, Jatinder Pal Singh, and Nicholas Bambos. Adaptive prefetching in wireless computing. *IEEE Transactions on Wireless Communications*, 15(5):3296–3310, 2016.

[SK13] Vasilios A Siris and Dimitrios Kalyvas. Enhancing mobile data offloading with mobility prediction and prefetching. *ACM SIGMOBILE Mobile Computing and Communications Review*, 17(1):22–29, 2013.

[TKV98] Nor Jaidi Tuah, Mohan Kumar, and Svetha Venkatesh. Investigation of a prefetch model for low bandwidth networks. In *Proceedings of the 1st ACM international workshop on Wireless mobile multimedia*, pages 38–47. ACM, 1998.

[wlC] Csv station, line and platorm files. https://www.data.gv.at/katalog/dataset/add66f20-d033-4eee-b9a0-47019828e698. [Online; accessed 03-February-2018].

[YLL+14] Shu Yinbiao, Kang Lee, Peter Lanctot, F Jianbin, H Hao, B Chow, and JP Desbenoit. Internet of things: wireless sensor networks. *White Paper, International Electrotechnical Commission, http://www. iec. ch*, 2014.

[ZBC+14] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.