Institute of Distributed Systems, Ulm University

Project Report

Context Aware Application for the LoRaPark Ulm

Steffen Knoblauch steffen.knobluch@uni-ulm.de
Oliver Felkel oliver.felkel@uni-ulm.de

SoSe 2021

The goal of this project is to develop and implement a concept for a context-aware application using sensors from the LoRaPark. The LoRaPark is an Internet of Things (IoT) experiment field and show room in the city of Ulm for devices that use the LoRa radio protocol. The LoRa protocol is especially designed for long distances with low power consumption, which makes it ideal for sensor based applications. The sensor data can be used to describe a context, which enables an application to react with more appropriate reactions for this context.
Context is the information used to describe a situation like *where*, *when*, *who*, or *what*. There are several methods to model context, from which we chose a rule-based approach, that combines different sensor information to describe a context, and implemented it as an Android application. Finally, the result of this project is an app that showcases context awareness and the limitations of this concept.

Contents

List of Figures

1 Introduction

In recent years the Internet of Things (IoT) is undergoing a rapid growth with an increasing number of sensor deployments in all areas of our daily life [9]. Consequently, the amount of all kinds of sensors increases, which not only results in a substantial increase in the amount of data, but also creates new use cases for the data. Such use cases already made the way into various situations in our daily life, like the tracking of parcels, or the capturing of the current weather data.
In general, the amount and fine granularity of sensor data allows applications to access more and more knowledge about their environment and adapt their behavior based on it. For example, a mobile device may have a built-in GPS sensor that can determine its position. The device can then obtain weather data for that position from other sensors. The application accesses its current context and is able to act based on this information.

Context Awareness In order to understand context awareness, it is necessary to define what context is. Abowd et al. [1] define context as any information that is able to describe the situation of an entity. An entity can be anything from a person, place to a computational object. With the IoT in mind this can be any combination of sensor values or even the aggregation of the data of a set of sensors.
Context awareness/context-aware computing was already defined by Schilit and Theimer in 1994[10] as the ability of a mobile application to discover and react on environment changes. This definition focuses just on the reaction on environment changes and has therefore some limitations.
Abowd et al. later published a broader definition of context awareness: "*We define context-awareness or context-aware computing as the use of context to provide task-relevant information and/or services to a user*" [1]. This extends the previous definition with the dynamic selection of a service based on the context, which allows a way more general interpretation of context awareness.
The questions that arises from these definitions is why context awareness is useful. When humans talk to each other, they use implicit context information to increase the conversational bandwidth [1]. In Human–computer interaction (HCI) the traditional input methods like keyboard and mouse or on mobile devices touch are often bound to an explicit defined and limited amount of information a human can deliver to the computer. Enhancing the interaction from a human and a computer with sensor data/additional context allows the computer to become more context-aware and react in various different ways. For example if an application fetches the position of the device and the weather, it can reason about the context of the user and in case of rain, remind the user to take along an umbrella.

LoRa and LoRaWAN The growth and properties of the IoT (devices) creates new challenges for the communication between the sensor nodes and the user devices. Depending on the hardware and use case of the sensors there can be several constraints that restrict the communication. For example the power consumption of a radio transmission can be a important fact, because a sensor may only have a small power supply,

but it should also possible to transmit over a long-time and a long-range because sensors may deployed in a large, inaccessible area. In order to allow the communication in such a limited environment low-power, wide area networks (LPWAN) are used. They allow not only the communication over long-range data links, but also care about the power consumption of the nodes. For this use case Semtech[11] developed LoRa, a radio frequency modulation technology for LPWANs, that guarantees these properties by taking a trade-off between data-rate and power consumption.
LoRa defines the lower physical layer in the OSI-model. LoRaWAN sits on top of the physical layer as the Media Access Control (MAC) layer and it delivers, among other things, secure bi-directional communication standardized and maintained by the LoRa Alliance[1] [11]. A network that uses the LoRaWAN stack consists in general of these main components [11]:

- Sensors nodes
- Gateways
- Network server
- Application server
- (User device)

In such a network the sensor nodes communicate with the gateways via radio transmission using LoRa. A single gateway receives LoRa modulated radio frequency messages from any device in hearing distance. The gateway re-transmits the received data to a network server as IP traffic: Wi-Fi, hardwired Ethernet or cellular connection. In short the gateways just forward the received messages to the network servers.
The network server ensures the authenticity of every sensor and the integrity of every message (without access to the payload) and re-transmits the messages to the application server. The application server handles, manages and interprets the sensor data and can finally offer the processed data to a user device.

The Things Network The Things Network (TTN)[2] is a global and open LoRaWAN network to develop IoT applications. The goal is the provide access to a network with maximum security and low cost, but still maintain a good scalability.
The TTN is represented all over the globe with thousands of gateways, in order to provide access for the community. Community members are able to implement and add not only their own sensors that communicate with the already existing gateways, but can also setup additional gateways to provide an even better network coverage.
The Things Industries offers a complete software stack that consist of several components that implement the LoRaWAN standard [12] and therefore guarantees the compatibility with all LoRaWAN certified sensors. This facilitates the integration of new sensors from

[1] `https://lora-alliance.org/`
[2] `https://www.thethingsnetwork.org/`

community members, because they are able to use existing hardware, integrate it and use it. In general, the network consist, like the LoRaWAN specification defines, of gateways, network servers and application servers. Additional to this, the stack introduces several integration methods that allow the access to the data of your own sensors, for example via an HTTP, Message Queuing Telemetry Transport (MQTT) or a custom interface.
In other words, The Things Network provides the infrastructure for IoT applications, that allows the extension by your own hard- and software, like sensors or applications.

LoRaPark Ulm The LoRaPark is the first step for Ulm to become a pioneer for a smart city by deploying an IoT infrastructure. The main idea is the creation of an infrastructure in the Ulm area based on the TTN, and therefore the LoRaWAN components. In order to create this network several gateways and sensors were installed all around the town. Examples of such sensors are:

- Weather station
- Parking space monitoring
- Flood sensor
- Visitor flow measurement

The main idea behind this infrastructure is to show the citizens and the resident industry the possibilities of IoT devices and give them the chance to discover and explore them. In order to explore these possibilities, citizens and industry have the opportunity not only to access the infrastructure, but also to expand it with their own hardware. This project can be extended and used freely by the community, which means, that not only sensors and gateways can be added but it is also possible to develop applications that use the existing sensor data.
The park was officially opened on 22.07.2020 and already had a wide variety of sensors in use. There are already various interfaces that allow the access to the sensor data. On the one hand, there is the official website[3], which displays the current data, and on the other hand, there is a Grafana instance[4], which allows access to the data in form of graphs via a web interface. In addition, there is an API for the development of applications that allows direct access to the sensor values.
An overview of the architecture of the network is shown in figure 1. As already mentioned, the architecture is based on the TTN and therefore the sensors communicate with the gateways in the city of Ulm by sending the data via radio transmission using LoRa. The received messages will be securely re-transmitted by the gateways to the TTN backend.The backend makes the data available for a wide variety of purposes. On the one hand, this includes direct access to the data via an API or, on the other hand, the further processing of this data, such as the aggregation of several data sets. This is where the community can come up with new ideas on how to use the data and find new use-cases for the IoT data in their daily life.

[3] `https://lorapark.de/`
[4] `https://ttndata.cortex-media.de/grafana/d/wkcLA7VMk/`

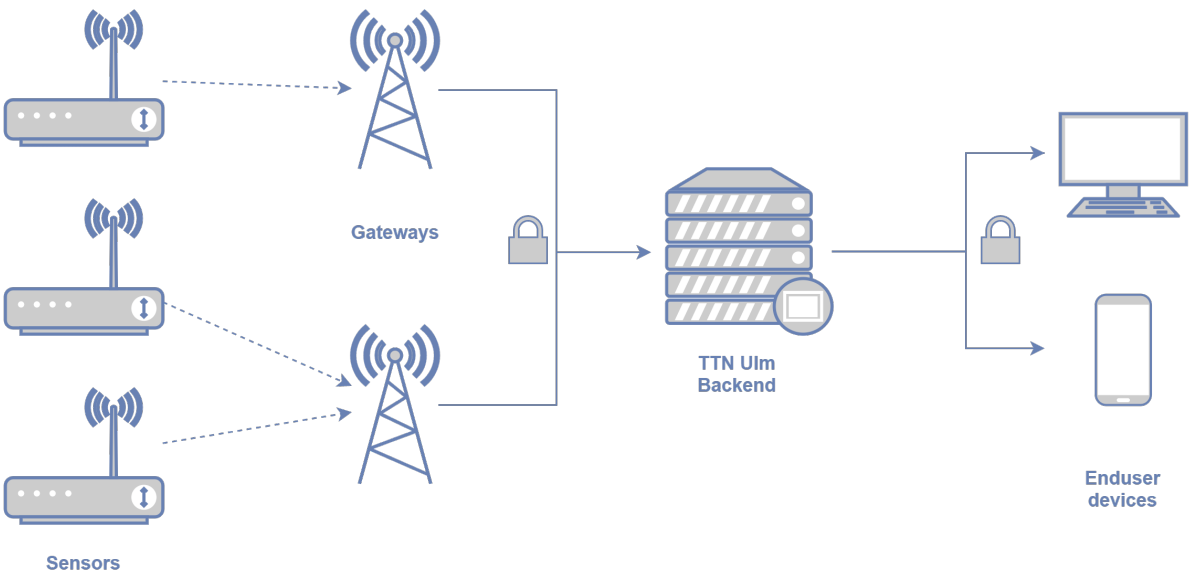

Figure 1: LoRaPark architecture overview [8, cf.]

2 Conception

The objective of this project is to develop a context-aware application. This application will use and process the data transmitted by the LoRaPark sensors. It will be used to introduce citizens to the potential of IoT sensors and applications by implementing real-life use cases.

2.1 LoRaPark Sensors

As mentioned in section 1, the LoRaPark already has a variety of sensors that provide a wide range of data. The application to be developed will access and process this data using an existing API.
LoRaPark sensors can capture a wide variety of environmental data, such as the weather or the number of people who have passed through a particular area. This data therefore reflects the environment of the sensor, based on the location where it is deployed. This environmental data can then be used by an application to evaluate the context of a device near this sensor. For example, the application can request weather data from a specific sensor and thus from a specific position. With this data, the application can draw various conclusions and, for example, show a user helpful notifications, such as taking along an umbrella.

2.2 Additional Context Data

In addition to the sensor values coming from the LoRaPark sensors, an application can also access the sensor data from other sources. In our case we developed a mobile application that has also access to the sensors of such an mobile device. A modern smartphone offers a number of different data sources, like internal sensors that measure different values, such as location or acceleration.
By combining the data from the LoRaPark with the data from the smartphone, it is possible for an application to evaluate the context of the device and thus of the user. Recognizing the context is the first step in developing a context-aware application and can be done based on the combination of these sensor values.
Probably one of the most important values of the smartphone for determining the context of a user is the determination of the location, as one can map the external values of the LoRaPark with this value. Because of this, the focus of this application will be mainly on the location of the device, but in theory there are other values that can be used for context determination. For example, the calendar can be accessed to match the data with possible appointments. With this information, the application could then calculate the route to this appointment and consider data from sensors that lie on the route.
After the context has been determined from the external LoRaPark data and the internal sensor data of the device, the application can then trigger any actions. Such actions can range from a simple notification based on the context, to calculating a route or sending a message.

2.3 Use Cases

This section introduces some exemplary use cases based on the data of the LoRaPark. The goal of this project is to design an architecture that showcases such use cases and make them easier to grasp for the public.

2.3.1 Direct Sensor Interaction

The most obvious use case for such an application is direct access to the LoRaPark sensor data. This can be used, for example, by a citizen to display the current weather data measured by a sensor.
As the LoRaPark can easily be extended with additional sensors, it would be possible with this functionality to provide the citizens of Ulm with a wide range of sensor values.

2.3.2 Flood Warning

Another use case in the context of the LoRaPark is the use of the already existing flood sensors. For this purpose, an ultrasonic sensor was installed that measures the distance to the ground. In the event of a flood, this distance decreases and can be registered by the sensor.
The flood measurement is a practical feature, because in case of an increased water level, the adjacent bicycle and pedestrian paths are also affected. In the event of a flood, a citizen who normally uses these routes can be notified so that they can choose an alternative route and avoid the inaccessible paths.

2.3.3 COVID-19 Pandemic Support

A very present topic is the COVID-19 pandemic and applications that help in the fight against it. With the background of the LoRaPark in mind, there are two examples that can support citizens in their daily life and help with the fight against the virus.
The first use case is about the masks that have to be worn in certain areas in Ulm, such as pedestrian areas. To solve this, the application could evaluate the current location of the device and display a notification when a citizen approaches one of these areas. This would help insofar as it would prevent people from forgetting to put on the mask because they would get an additional reminder.
Another meaningful use case is based on crowd measurements in the city center of Ulm. These measurements can tell something about the occupancy of nearby shops, e.g. that one of these shops is very crowded. For a citizen who wants to avoid these crowds and hence protect himself from COVID-19, these measurements can help in the selection of a shop. The application could facilitate this by combining the user's position with sensor data on possible crowds. It can evaluate whether a user is visiting a busy shop and suggest alternatives. For example, if a user wants to go to a drugstore, an alternative nearby could be suggested.

3 Implementation

This section begins with the description of the use cases that our implementation has to fulfill. These use cases are based on the example use cases already presented in section 2.3. In order to develop a final architecture for the application, several designs were made and described in detail in this section.
The architecture will then be used for the implementation of a demo application consisting of a server component and an Android application.

3.1 Use Case Generalization

The first step of the implementation was the concretisation of the use cases that the application should implement. The use cases presented in section 2.3 were used as a basis for this. However, the goal was not only the static implementation of these use cases, but an application that can be used for a wide variety of use cases and can also be expanded in the future.
The first step in the selection of the use cases for our application, was a rough selection of the scope. Since the application should be context-aware, the most obvious choice is to develop the application for mobile devices.
The second step was to decide how the context of the application is represented, as this affects the basic use cases for the application. As stated in the survey [9] of Perera et al., there are various ways to represent and evaluate a context:

- *Key-Value Modelling*: the simplest form of context modelling, by persisting and handling the context in form of key value pairs. This comes with several restrictions, as there are no relationships or complex objects possible.
- *Markup Scheme Modelling*: is an improvement of the key value approach by attaching tags to the data.
- *Graphical Modelling*: this approach is based on the idea to model the context with relationships (e.g. UML) and persists the data in a SQL or No-SQL database.
- *Object Based Modelling*: model the data using class hierarchies and relationships.
- *Logic Based Modelling*: model the context using facts, expressions and rules.
- *Ontology Based Modelling*: represent the context as ontologies.

Based on these techniques, we decided to design the context based on rules. The reason for this is that we wanted to have a flexible representation and also the possibility to easily define a new context that the application can process.
A rule consists of several building blocks: On the one hand, it has a condition, which is a logical expression. This expression can, among other things, consist of Boolean operators and variables. During the evaluation, these variables are replaced by concrete values, such as a sensor value. Another building block is the relevant sensors and geofences

that provide the values for the variables of the condition. The last component is the actions that are triggered as soon as the condition is met. A detailed description of the structure of the rules can be found in section 3.2.2.
In addition to this, rules make it possible to dynamically describe a new context. In general, there are static and dynamic context models [9]. A static context model is predefined and collects only the data needed for evaluation, also called facts. A dynamic context model can use different data sources to infer the current context. Rules may appear to be static only, but since it is possible to create rules dynamically, they can also be extended to dynamic use. An example of this would be the creation of a rule based on a route that has been calculated. A user can choose any route and a rule can be dynamically generated that uses the sensors on the route.
These first general decisions led to identification of the use cases shown in figure 2.

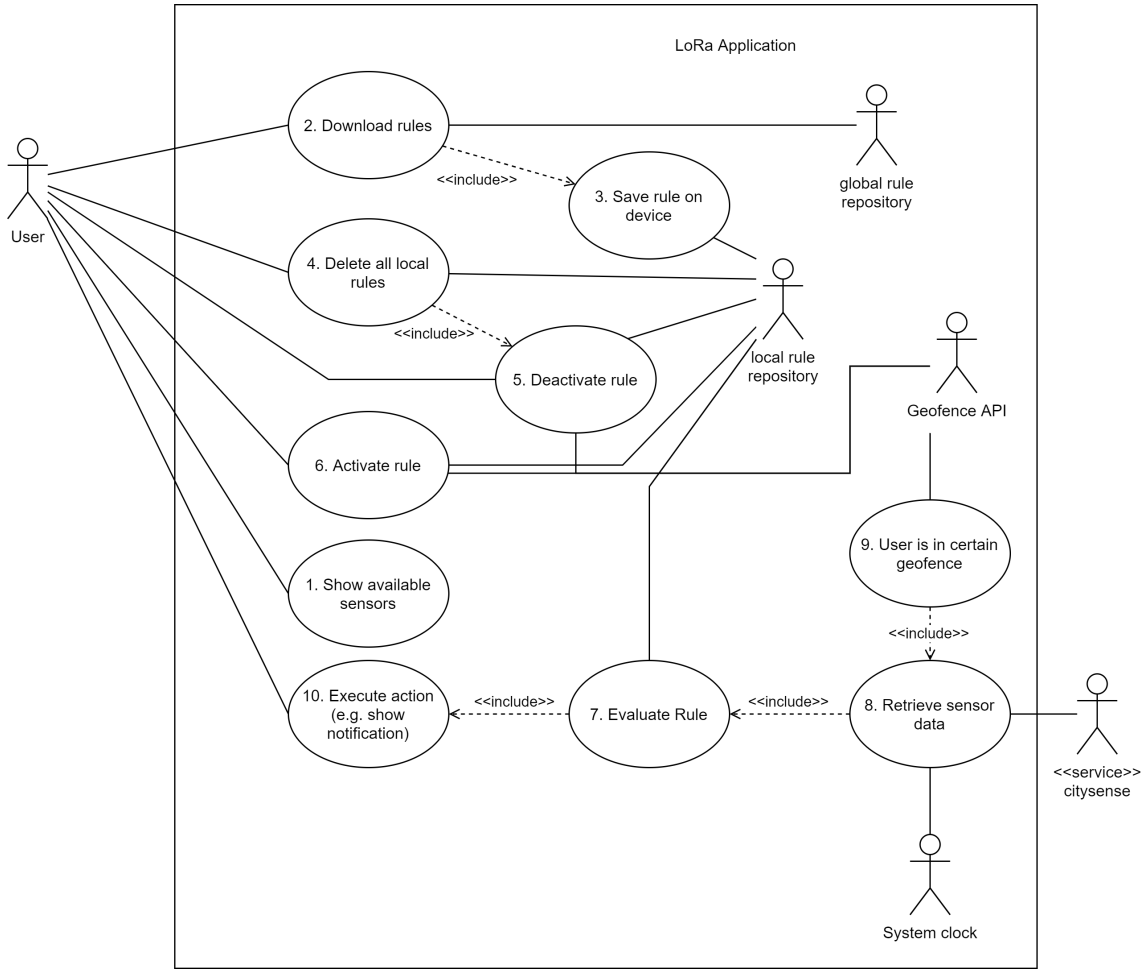

Figure 2: Use cases

The first identified use case, *1. Show available sensors*, allows the user to display all the existing sensors and also show the current sensor values. This gives the user not only an

overview of the LoRaPark, but also allows the user to interactively discover the sensors. The next use cases are about the handling of the rules and how they are transmitted to the smartphone. The *2.* use case deals with the *download of the rules*. As already described, a rule describes a context (including the relevant sensors) and the actions that are triggered when the condition is met. These rules are located in a global repository from which they can be downloaded. The decision to create a global repository was made because, on the one hand it is an easy way to add more rules later, and on the other hand you can also update existing rules on the smartphone through a controlled process.
A user can display the available rules with a short description and select and download the desired rules from this repository. The downloaded rules are saved afterwards in a local repository (*3. Save rule on device*). Use case *4. Delete all local rules* is required to give the user the ability to manage the rules and remove them from the local repository on the smartphone. At the beginning it is sufficient to delete all rules, but a future adjustment could be the deletion of just one or more selected rules.
To prevent a rule from always being evaluated when it is stored locally on a user's smartphone, you can activate (*6. Activate rule*) and deactivate (*5. Deactivate Rule*) it. If a rule is deactivated, it will be ignored in future evaluations and if a rule is based on geofences, these will be removed from the device. Conversely, if a rule is activated, it will be considered in the next evaluation and if the rule depends on a geofence, the geofence will be created.
The core of the application is the *evaluation of the rules(7.)* or in other words the context evaluation. The evaluation is done for all active rules. The sensor values from the LoRaPark sensors are used as input values, as well as the Geofence API data. The geofence data is used to determine whether the device is located in a geofence relevant for one of the active rules.
The evaluation of the rules can be initiated by two things, firstly by a regularly triggered job and secondly as soon as a user enters a geofence (*9. User is in a certain geofence*). As a result, the sensor data provided by the LoRaPark API must be loaded (*8. Retrieve sensor data*) and used to evaluate the active rules.
When a rule is evaluated as triggered, in other words the logical expression in the rule condition is met, the actions of the rules are executed (*10. Execute action*). For the first implementation, this action can be a simple notification to the user, but later other custom actions can be developed.

3.2 Design Decisions

This section deals with the decision-making process leading to the final architecture. Different aspects are considered, such as the basic architecture and the implementation of the rules.

3.2.1 General Architecture

The first decisions we made were related to the general architecture. The application is developed for the usage on a smartphone with the access to a global rule repository. As described in the use cases, the global rule repository can be used to facilitate the central rule management, like updating existing rules and adding new rules. Another factor that influenced the architecture is the ability to display the existing sensors. As there is currently no semantic description provided from the LoRaPark API (more details about this can be found in section 3.5.1), it is necessary to develop and provide one. This can be done like the global rule repository so that there is a convenient way to update the descriptions.
The architecture should also be flexible and easily expandable, for example in order to be able to use custom rules on the user's device in the future.
Based on the requirements described above, there are generally two variants for the architecture as shown in figure 3. The first variant is based on the idea of direct access to the sensor values of the LoRaPark API. This means that a user's device evaluates the rules and communicates directly with the API. In addition, another server component is needed that acts as the global rule repository and also provides the semantic description of the sensors.
The second variant is based on the concept of a single server component that provides the entire data. The user's device communicates only with this server and receives all information from it. The server, on the other hand, communicates with the LoRaPark API to receive the required sensor data. At the same time the server is providing the rules and the semantic sensor descriptions.

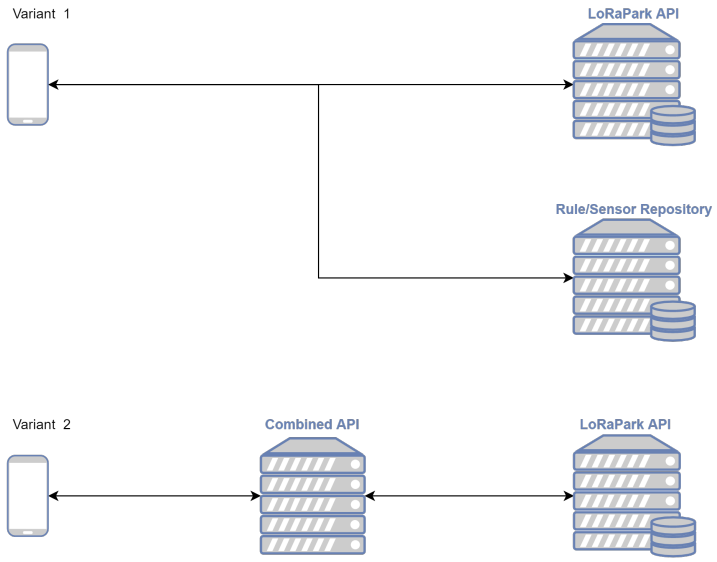

Figure 3: Architecture overview

For our purposes, the first variant is not applicable, as it entails some limitations. Since

the user devices access the API directly, the number of requests can increase significantly compared to the second variant, which allows the chaching of sensor values. Despite the direct communication with the LoRaPark API, this variant still requires a server component to provide the rules and sensor descriptions. In comparison to the second variant there is no advantage since you still require a server component, which led us to choose the second variant.
The second variant relies on a central middleware with which the user devices communicate. The second variant can be further refined which results in different subvariants. One solution would be to evaluate the rules on the server side and communicate with the smartphone if a rule is triggered. However, this solution has several disadvantages that eliminate it. To enable communication between the smartphone and the server, one can design a publisher-subscriber protocol where a user subscribes to the triggering of a rule. The server then sends an event as publisher as soon as a rule is triggered. This has some limitations due to Android (compare section 4.2) and also makes it difficult to extend the application later. For example, this would complicate an extension where the user can design their own custom rules, as user-designed code would be executed on the server side when evaluating the rules, which implies different security problems. In addition, this extension requires some kind of user authentication or management to upload and edit the rules.
Another problem of this solution is that the server evaluates the rules. The server has therefore unnecessary access to the user's data, such as their position, and consequently to their context. The goal is to protect privacy, especially since it is possible to offer the same functionality without this restriction.
Another solution, which was then also selected, is the evaluation of rules on the side of the smartphone. This has the advantage that the server only offers public data and does not process user data. A smartphone can simply query required data at certain intervals, for example every 15 minutes. Or, as in the special case of the use of a geofence, as soon as a user enters a geofence. This creates an advantage for future development, as you can easily scale up horizontally by using several servers to provide the data. In addition, it can reduce the number of queries to the LoRaPark API, because the server can be used as a cache, which means it will regularly retrieve the data from the API, cache it and then forward only the cached data to the clients.
As an alternative to regular polling, you can also use an event-based approach for this variant. In this case, a smartphone could subscribe to the values of a sensor and as soon as the server has a new value, it publishes it. The server could do this by a simple delta calculation where the delta between a cached value and the current value (provided by the LoRaPark API) is calculated and as soon as there is a change an event is sent to the subscribed clients. The LoRaPark API does not offer the option to react directly to sensor value changes, for example with a publisher-subscriber API, which makes the additional step of delta calculation necessary. Due to the limitations of Android (compare section 4.2) and the overhead for developing the delta calculation, we have decided to query the server regularly, but implement it in a way that it can be enhanced with events in the future.
In summary, the architecture consists of a server that caches the LoRaPark data and

makes it available to the smartphone. At the same time, the server also offers a repository with the global rules and the semantic descriptions of the sensors. The smartphone queries the sensor values at regular intervals and evaluates the active rules.

3.2.2 Rule Engine and Rule Design

As discussed already, it was decided to use rules to describe a context and how to react to the occurrence of such a context. For the evaluation of the rules, a rule engine is required, that evaluates the rules on a sensor value update or a context change. In order to select or develop a rule engine, different engines and rule formats were analyzed and examined for their applicability. A constraint for such a rule engine is that it should run on the client and the rules it processes should be interchangeable and craftable by users. This implies that the rules have to be stored somewhere and can be serialized.
RuleBook[5] is a library that uses Java code to specify a rule. The concept of RuleBook is *Given some Fact(s)* and *When a condition evaluates to true*, *Then an action is triggered*. Unfortunately, RuleBook offers no serialization and would require a custom implementation.
Another promising library was Easy Rules[6] that use the same Facts-When-Then pattern, but offers a serialization as YAML. However arbitrary Java commands could be inserted as action for rules. This creates some security implications and since Easy Rules is in maintenance mode, was not justifiable.
Based on these considerations, we decided to construct a custom rule engine, but rely on the same when-then pattern. As a serialization format JSON was chosen, since it is a common exchange format and easy to understand. Aside from that it was decided not to execute arbitrary source code through the rules. This meant that a format had to be found that allows high flexibility and still mets those considerations. For the condition of a rule JsonLogic was found to perfectly suitable. For the actions, a custom format was designed.
JsonLogic [7] is a format for modeling logical conditions as JSON. The conditions are parsed and interpreted with no side effects and deterministic computation time. Supported operations include logic and boolean operations, numeric operations, array operations, string operations, and custom ones. The operators in JsonLogic are represented as objects with the operator as the key and the arguments as an array:

```
{"operator" : ["arguments" ... ]}
```

By hierarchically applying the operators, complex structures can be built. The data types of the operators are not fixed. A NAND for instance could be expressed like this:

```
{"!": [
    {"and": [true, true]}
]}
```

[5] https://github.com/deliveredtechnologies/rulebook
[6] https://github.com/j-easy/easy-rules
[7] https://jsonlogic.com

An operator for requesting the sensor values and a operator for requesting geofences was added using a custom operator. Further operators could be added supporting more contexts.
For the actions we decided, that a rule supports multiple actions, which means when the rule condition is met all the specified actions are executed. Currently a text-to-speech action that reads out text, a notification action that creates a notification, and a navigation action are supported. Each action can have their own data that is required for the execution, for example a string that a notification action will show. Actions itself support a hierarchical structure, for instance the notification can show buttons, which trigger another action if the button is pressed. A complete rule could look like this:

```
{
  "id": "lorapark",
  "name": "LoRaPark Verschwörhaus",
  "description": "LoRaPark Verschwörhaus Rule",
  "sensors": ["elsysems-a81758fffe04887a"],
  "geofences":[{
      "id": "lorapark",
      "location": {
        "latitude": 48.39652,
        "longitude": 9.99076
      },
      "radius" : 300
  }],
  "condition": {"and": [
    {"geofence": "lorapark"},
    {"sensor": ["door", "elsysems-a81758fffe04887a", "extdigital"]}
  ]}
  ,
  "actions": [
    {
      "action": "notification",
      "data": {
        "title": "LoRaPark",
        "text": "Welcome to the LoRaPark Ulm. The door to the
        ↪  Verschwörhaus is open. Please come in if you are
        ↪  interested.",
        "buttons": []
      }
    }
  ]
}
```

The rules are stored on the device and evaluated when needed. Evaluation is either

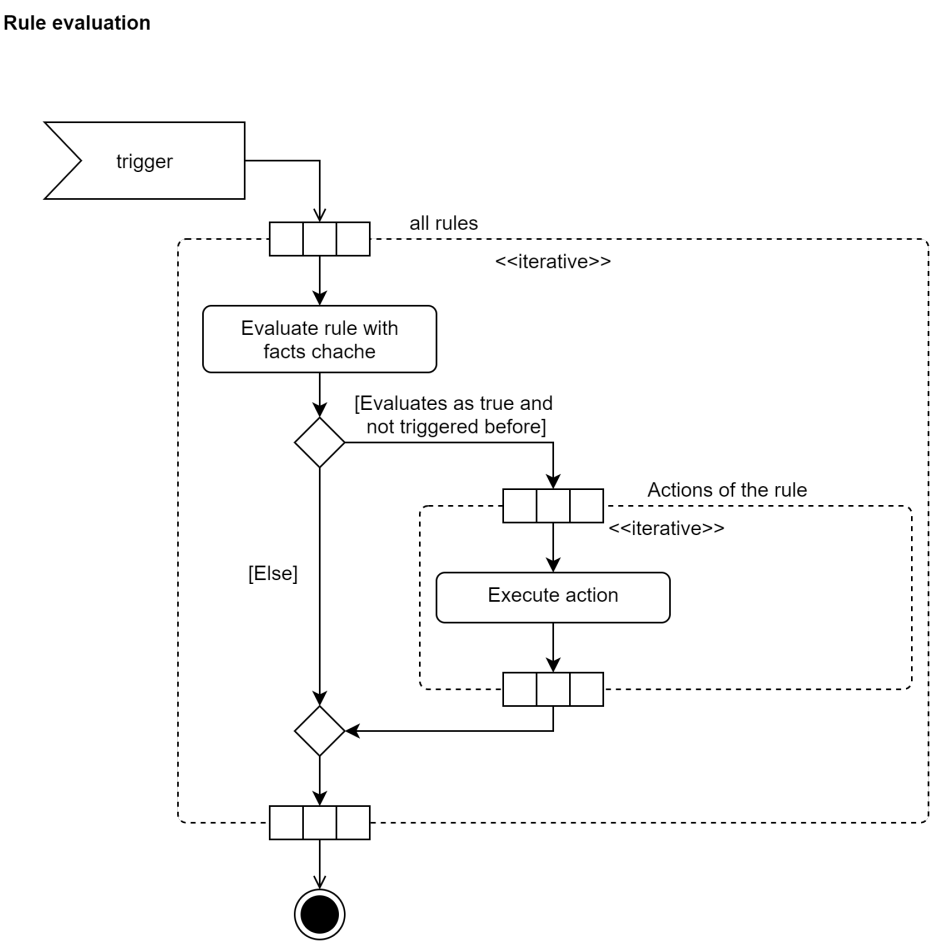

Figure 4: Rule Evaluation initiated by an external trigger (e.g. sensor update).

triggered periodically by the sensor update or when the context changes e.g. a geofence is entered or left (see Figure 4). Then for each rule the condition is parsed and evaluated by JsonLogic and if the condition is met, the actions are executed. To suppress unnecessary execution, the actions are only triggered on a rising edge (if the condition was not met before and is now met).

3.2.3 Geofence API

There are various ways to use geofences on Android devices. The most laborious solution is a custom implementation that regularly requests the position and compares it with registered geofences. This would require a background process that performs this regularly and thus has a correspondingly large impact on the battery consumption of the device. This solution requires also a high degree of fine-tuning, for example for the interval time and the granularity of the fences, which is why this solution has been ruled out.
Android, or more precisely Google (based on the Google Play Services), offers a Geofence

API[6] that allows up to 100 geofences to be registered per app. The API offers different types of geofences, for entering, leaving or staying in the geofence (for a certain period of time). As soon as one of the registered geofences triggers, an event is thrown for the corresponding geofence and custom code can be executed using a broadcast receiver.
Another API provided by Google is the Awareness API. "The Awareness API unifies 7 location and context signals in a single API, enabling you to create powerful context-based features with minimal impact on system resources" [2]. This API offers a wide range of different functions, including simple location fences, which are distinguished between entering, leaving and being in an area, as in the Geofence API. In addition to the location fences, there are other fences, such as a headphone fence that is triggered as soon as a headphone is plugged in. The fences of the API can be combined internally (and, or, negation) to create more complex constructs. An example of this would be: a user is in a certain area (location fence), using his bike (activity detection fence) and uses headphones (headphone fence).
Since we excluded other external providers (for example Pathsense[8]), we had the choice between these three alternatives. We chose the Awareness API because it allows us to manage the fences easily and also keeps other options open for the future. In the future, the rules, and therefore the context, could not only be based on the sensor values and geofences, but also use more complex fences that, for example, also include the current activity of the user.
Note: On devices running Android Oreo or newer, there are some restrictions on background location processing [5]. This causes some limitations that are discussed in more detail in section 4.3.

3.3 System Architecture

This section starts with an overview of the application components and afterwards goes into the details about the different aspects of the architecture.

3.4 Component Overview

As described in section 3.2 the application consists of a server and a client application. Based on the decisions we made, there are several components required in both subsystems, as shown in figure 5.
The server side consists of three independent components. There is the rule repository, which holds the global rules and offers them to the user for download. Another component is the sensor description repository, which is intended to hold a description of the sensors and make it also available to the user. The last component is the sensor data handler, which performs several tasks: polling the sensor values from the LoRaPark API on a periodical basis, caching the most recent values and making them available to the clients.
The Android subsystem consists also of several components. There is a component to

[8] `https://pathsense.com/`

retrieve the necessary sensor descriptions, the Sensor Description Handler. In addition, there is the Rule Management, which takes over all tasks regarding the rules. These tasks include downloading and persisting rules from the rule repository, as well as activating, deactivating and deleting rules. The central component of the client side is the rule engine component, which evaluates the active rules and triggers appropriate actions, such as displaying a notification. For this purpose, the relevant sensor data are requested from the server and used for the evaluation. In addition, the rule engine component also contains the geofence handling, which is relevant for the evaluation of rules that make use of them.

3.5 Server

The server provides data for the client and acts as a middleware between the client and the LoRaPark API. It is written in Python 3 and uses some additional libraries. The main components of the server is a REST API for the communication with the client, a cache for storing the sensor values, and an interface for pulling the sensor values from the LoRaPark API. Each component is implemented as an independent module that provides data for the other components. A configuration file is used for the setup of the components.

3.5.1 Semantic Sensor Description

The LoRaPark API does not offer a semantic description for the existing sensors, which is why we had to develop our own. The different endpoints that the LoRaPark API provides are bundled into *domains* that group types of sensors with common metrics (e.g. weather station, waste level, or flood data).
The different domains are defined in a model file. For each metric that could be present within the domain, a descriptive name and a unit is provided. The model file can be further extended to define aggregations for the metrics, for example to exchange numbers with labels based on a threshold or to convert between units. An example for this could be a sensor that sends the current temperature in °C and the client want access to it as °F. It is also intended to use the semantic description for internationalization by providing different model files for different languages with translated descriptions.

3.5.2 LoRaPark API

The LoRaPark API is a REST API with OAuth authentication. The credentials required to access the sensor values were provided by the API provider and are stored in the configuration file.
Initially when starting the application, an OAuth token is requested by sending the username and the password. The OAuth token can then be used in the following requests for authentication, but has to be refreshed in regular intervals, which is done by a background timer.

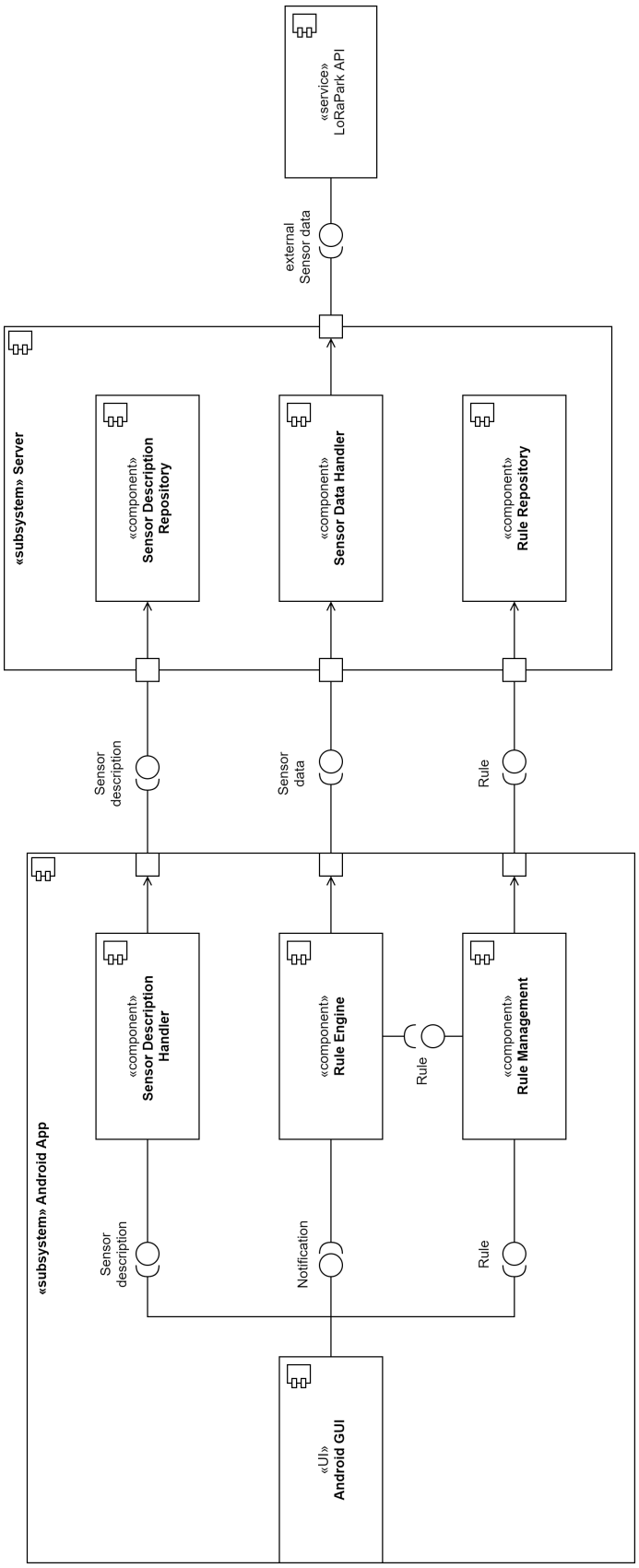

Figure 5: Components

The LoRaPark API provides different endpoints from which the sensor values for different domains can be requested. Each domain returns different metrics, for example a temperature or a distance. The sensor values are polled in regular intervals from the API via HTTP requests and stored in a cache. To specify which sensor data should be queried, another configuration file is used that holds the sensors and their corresponding domain. This allows to not only add new sensors, but also exclude existing sensors in the pulling process.
The result for a sensor, for example the flood sensor, returns a JSON object that is similar to the following:

```
[{
  "distance": 3249,
  "id": "ultrasonic1",
  "nodeid": 2772,
  "timestamp": "2021-03-16T09:36:42.679Z",
  "trials": 15,
  "version": 2,
  "voltage": 2795
}]
```

3.5.3 REST API

The REST API provides all information for the client, not only a list of sensors, but also the sensor values and the rules. For the implementation of the API the Python library flask[9] is used. All communication is done through HTTP/HTTPS requests with a JSON payload. The API offers following endpoints:

GET ***/sensors:*** returns a list of available sensors with description, position, and domains.

GET ***/sensors/raw:*** returns a list with metrics for all requested sensors.

GET ***/sensor/<id>/raw:*** returns the raw metrics for the sensor as pulled from the LoRaPark API.

GET ***/sensor/<id>/details:*** returns the metrics for a sensor with a description and a unit.

GET ***/sensor/<id>/details/<lang>:*** same as details but with the option to specify a language.

GET ***/rules:*** returns a list of the available rules with name and description.

GET ***/rule/<id>:*** returns the rule.

[9] `https://palletsprojects.com/p/flask/`

3.5.4 Server Startup and Update

The configuration of the server component is done with a configuration file. This allows for example to define the relevant sensors. The server will pull the sensor data for these sensors from the LoRaPark API. This configuration is loaded in the startup as show in figure 6.

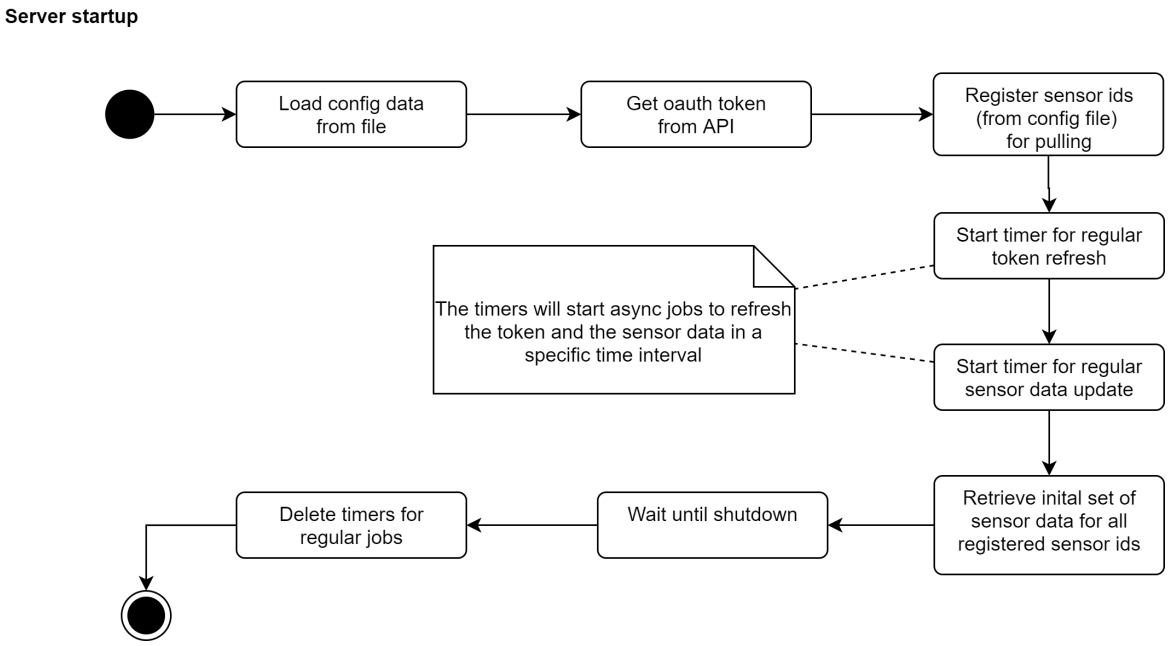

Figure 6: Server startup

The startup begins by loading the configuration file and the requesting of the initial OAuth token. In order to ensure the regular update of the sensor data and the OAuth token, two timers are started that will trigger the refresh of the token and the update of the cache. Additional a initial request for the first sensor values is made in order to have already data cached.
The timer for the server update triggers every minute and starts the update of the data. The update fetches the data for every sensor defined in the configuration file and updates the current data in the cache. When the Android application, or in other words a client, requests sensor values, the server takes the cache values and answers with a JSON object containing the relevant data.

Server update

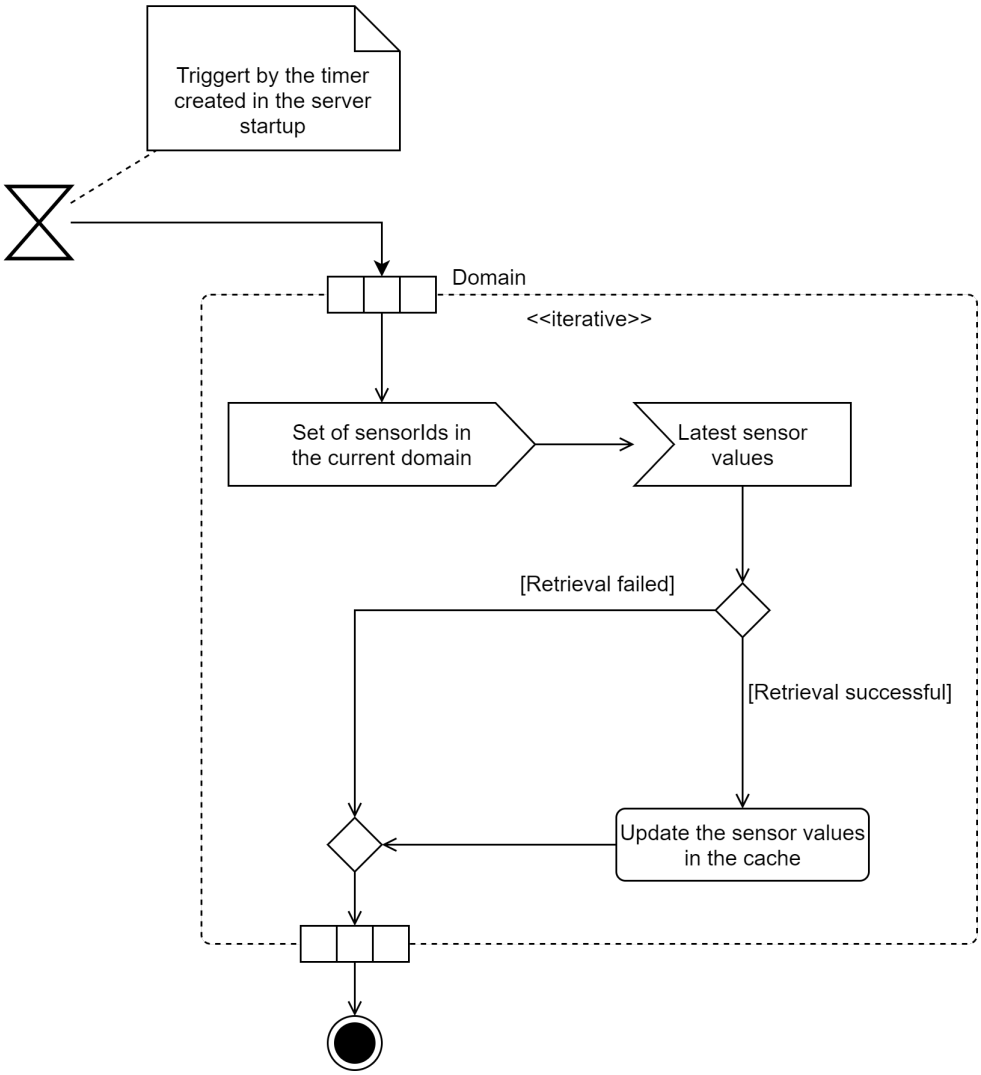

Figure 7: Server update

3.6 Client

This section covers the architecture of the Android application and other aspects relevant for the implementation.

3.6.1 Android Architecture

When writing an Android application, there are many aspects that need to be taken into account in order to achieve a good result. On the one hand, there are restrictions because the components of an Android application go through a specific lifecycle and, on the other hand, because Android has restrictions on the execution of an application in order to save the resources of a device, such as battery power.
In order to meet these requirements, the Android architecture components[3] were released, which are libraries that enable a developer to design a robust, testable, and maintainable architecture. These components save not only boilerplate code, but furthermore come with a recommendation for an architecture based on which most apps can be built.
The separation of concerns is one of the key principles that should be followed in the architecture of an Android application [7]. The idea is that components of an application should solve only one problem or concern. In the case of an Android application, the UI, the business logic and the data persistence are separated from each other, which means that these components can be integrated more easily into the Android lifecycle. An simplified example of this architecture is displayed in figure 8.

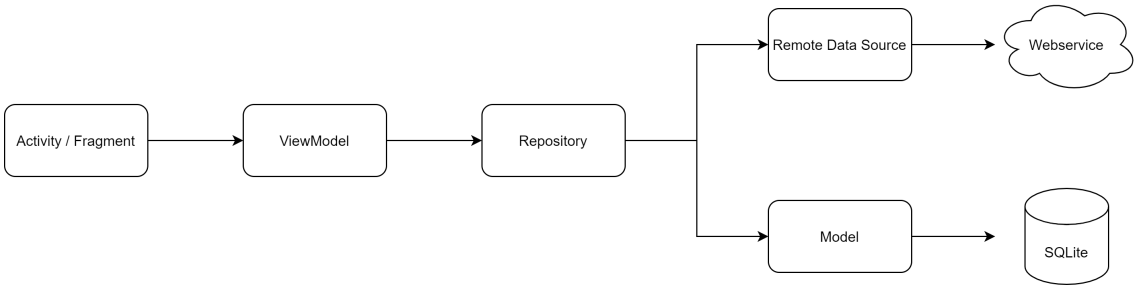

Figure 8: Android Architecture Overview [3, c.f.]

Activities or fragments are the UI of an Android application. These classes follow a specific lifecycle and manage the UI of the application. An example of this is that when the application is started, an activity loads the necessary data and renders the UI. Since an Android device can be locked, the activity must handle this accordingly. Simplified, in this case the UI is destroyed the moment the device is locked and a new instance is created when the smartphone is unlocked. This would mean that in the recreation the activity would have to load the data again in order to be render the UI again.
In order to circumvent such lifecycle issues ViewModels were introduced. A ViewModel survives some of the lifecycle changes without being destroyed to prevent unnecessary

data loading, for example when a, device is locked and unlocked or it is rotated. A rotation will destroy the activity and create a new instance to re-rendered the UI. The data wouldn't change in such cases, which means that an activity can receive on its creation the same ViewModel that the previous activity used and the data is already loaded.
The question arises as to why there is a further separation between the ViewModel and the repositories. ViewModels are intended for a specific UI component, which means that there is a separate ViewModel for each activity. If the data access is implemented in a ViewModel and several activities need access to the same data sources, the access code would be duplicated. To prevent this, the repositories were introduced, which provide a clean API for data access. A repository can perform various tasks in the background, from storing data in a SQLite database to loading data from a web service. A ViewModel may also need data from different sources and can therefore use several repositories and provide this data to the UI.
Separating the UI from the data processing, in form of the ViewModels and the repositories, also has an advantage, as the processing can be done in a background thread and the UI can therefore not freeze. This is enforced by Android and supported by this architecture, as the data is processed asynchronously and then passed to the UI using event listeners. Another advantage of this architecture is the resulting modularization. It is for example possible to replace one repository easily by another, as long as they implement the same interface. In our case, the implementation of a geofence repository based on the Awareness API could be replaced by an alternative based on the Geofence API.
This led us to design this project based on these architectural components. For this, a relatively simple mapping between the components from the component diagram (figure 5) and these components is used. In essence, each component can be split into a set of architecture components, which means in detail that each component usually requires at least one UI class, one ViewModel and access to one or more repositories.
Some of the components do not need a UI, such as the rule engine, but should only be executed in the background. For this purpose, Android offers the so-called JobScheduler, which makes it possible to plan and execute a background job in the future. This is required, for example, for the regular polling of sensor values for the rule evaluation. The JobScheduler takes into account Android's resource handling, which leads to some restrictions that are discussed in more detail in section 4.2.
Based on the components needed for the application, we decided on the following UI components:

- *Rule overview*: an activity that displays an overview of the locally stored rules and their state (active/inactive)
- *Rule details*: shows the details, like the description and name, of a saved rule and allows to toggle the state of the rule (active/inactive)
- *Sensor overview*: an activity that displays a map (using the Android implementation of OpenStreetMaps[10]) with an overview of the existing sensors

[10] `https://github.com/osmdroid/osmdroid`

- *Sensor details*: an activity that displays the details of a sensor and the latest values it provided

In addition to the UI, we developed several repositories, which on the one hand fetch the data from the server and on the other hand manage the persistence and access of the data. The most relevant repositories for this application are:

- *Sensor data repository*: handles the retrieval of sensor data from the server
- *Sensor description repository*: handles the retrieval of sensor descriptions from the server
- *Rule data repository*: handles the tasks related to the local rules, like persistence and access
- *Geofence repository*: handles the tasks related to geofences

For the implementation of the presented components, additional libraries were used that facilitate the data persistence (Room[11]), the asynchronous work (the Android implementation of RxJava[12]), the sending of http requests (OkHttp[13]) and the parsing of the JSON result from the server(Gson[14]).
As already described in section 3.2.2, the rule is transmitted from the server in the form of a JSON object and the evaluation in the rules engine also takes place based on a JSON object. Since the relational database SQLite is typically used in Android, there is either the option of keeping the complete rule as string and parsing it again each time it is used, or alternatively building a database schema that keeps parts of the rules as common database types.
Since some data is needed in the form of JSON objects, for example by the rule engine to evaluate the condition with JsonLogic, we have decided to parse only parts of the rule and store for example the condition directly as a string, containing the JSON object. This avoids unnecessary parsing for the saving and calling process and also facilitates the access via Room.
Based on this, the ER diagram in figure 9 was created. A rule consists of certain properties, such as the name or the condition. Furthermore, a rule can have multiple actions that will be triggered when the condition is met in the evaluation. An action can consist of the type, like a notification action, and a payload in the form of a string (JSON object). A rule can also have multiple geofences and multiple sensor. Both have additional to the id that is used as primary key a second identifier. This additional identifier is used as reference in the condition of the rule. The reason for this is that there might be multiple instances with the same identifier.

[11] `https://developer.android.com/training/data-storage/room`
[12] `https://github.com/ReactiveX/RxAndroid`
[13] `https://square.github.io/okhttp/`
[14] `https://github.com/google/gson`

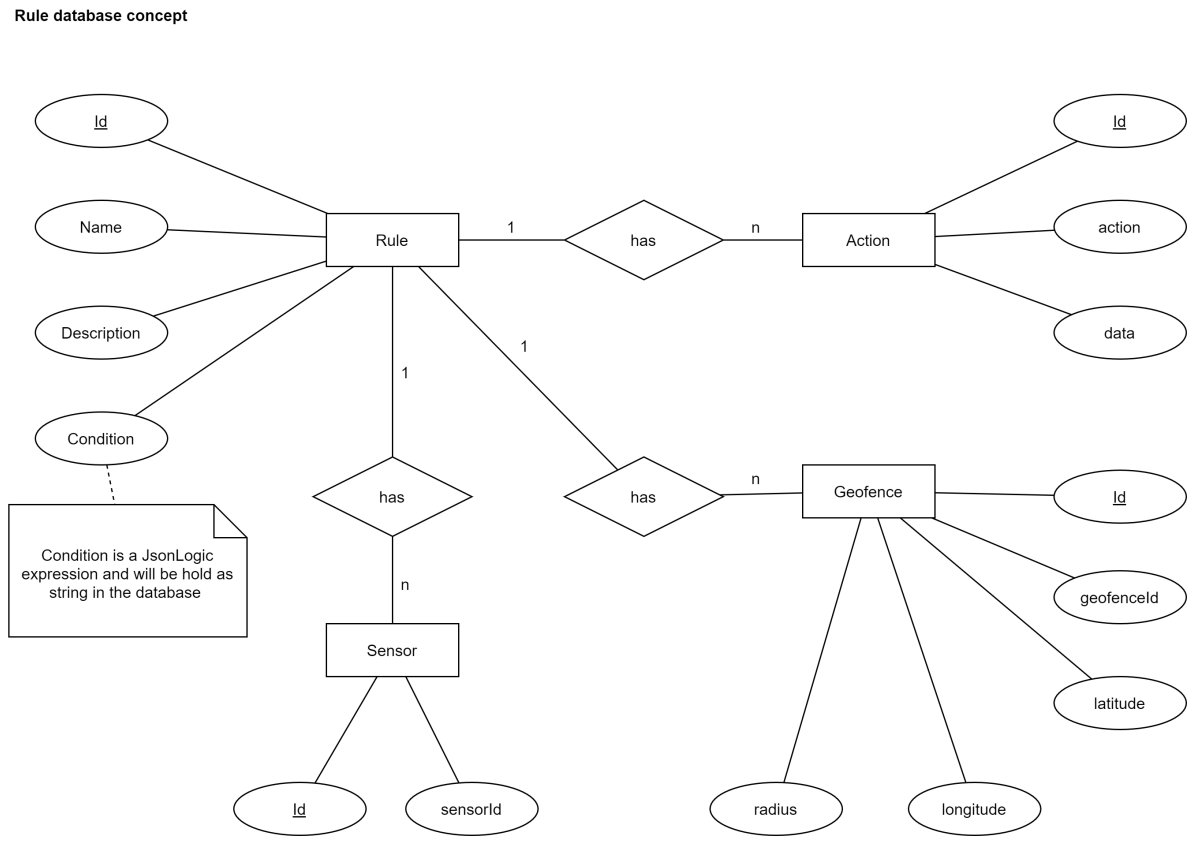

Figure 9: Rule ER-Diagram

A topic that has not yet been dealt with is how to react to a triggered geofence, which means that a user is in the radius around the GPS position of the geofence. For this purpose Android offers so-called broadcast receivers which are executed as soon as a special event is thrown. In this case, the Awareness API will throw such an event as soon as a user enters a geofence. The receiver will react on this event and then schedule a rule engine job, that evaluates the rules.
In summary, the architecture consists of several layers, which are used to comply with the Android lifecycle restrictions on the one hand and offer modularization on the other. This facilitates a later extension of the app as well as an exchange of the components, for example to support a different geofence solution.

3.6.2 Rule Engine

The rule engine is a central part of the architecture that is responsible for the evaluation of rules. When the rule evaluation is started, the active rules are loaded and the relevant sensors and geofences computed. The sensor values are pulled from the server and the state of the geofences is checked. The conditions of the rules are parsed by JsonLogic and checked if the conditions are met. If the condition is met and was not met before,

the actions of this rule are executed. This ensures that the rule is only triggered on changes. The complete rule evaluation is shown in figure 10.

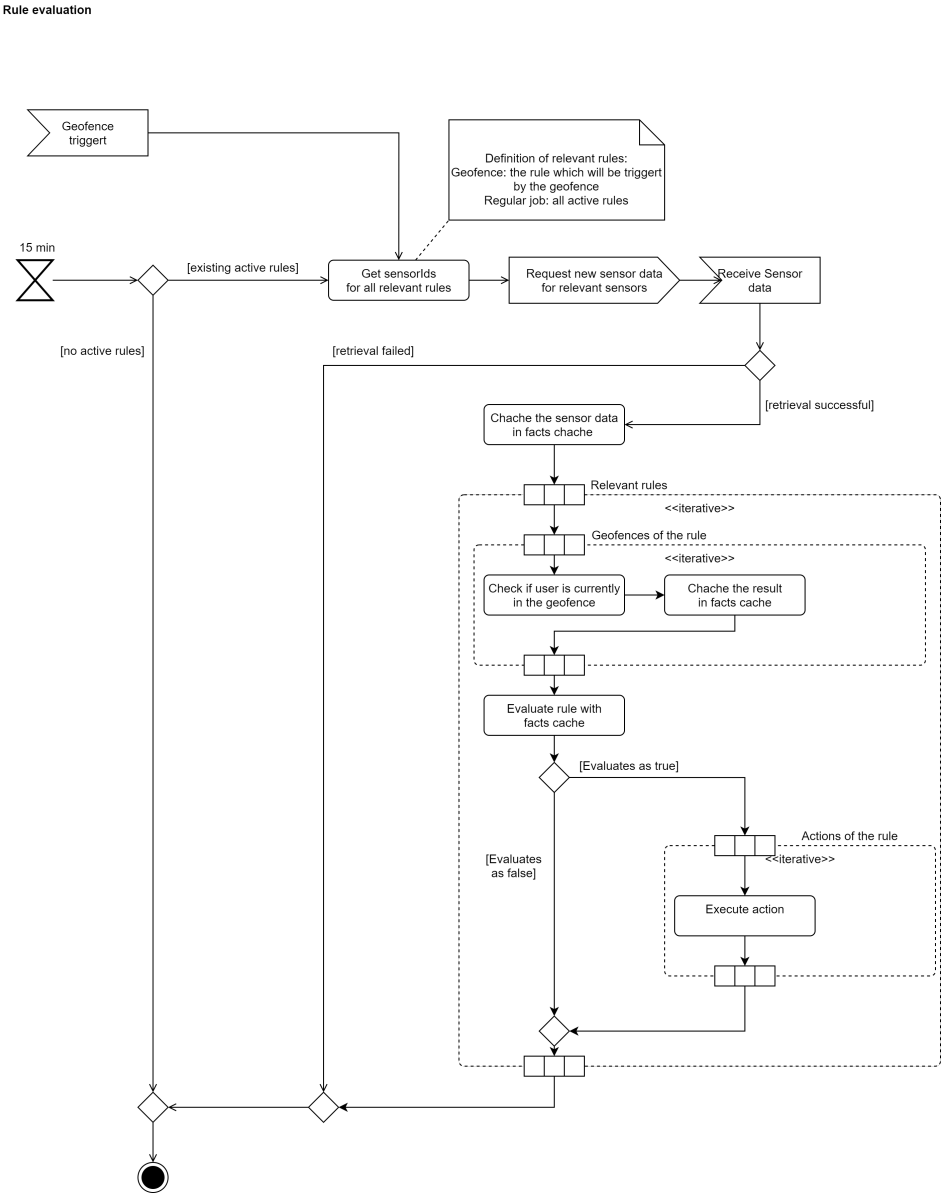

Figure 10: Activity Diagram of the Rule Evaluation.

Aside from the native supported operations of JsonLogic, two custom operators have been implemented:

sensor: returns the metric value using the domain, sensor id and metric.

geofence: returns whether the client is within a geofence.

Actions are simply a list of commands that should be performed. However actions can be nested, which allows more complex operations.

tts: The text-to-speech action reads out text using the native Android TTS engine. Supported parameters are text and language.

navigate: The navigate action opens Google Maps (either as an app if installed or in the webbrowser) with a search query[15]. The search query is specially crafted Google Maps URL that supports a variety of possibilities like search, directions, or displaying a map section.

notification: The notification action creates a notification in the operating system. Notification have a title, a text, and optionally up to three buttons that can be used to trigger further actions. Starting with Android 8.0, notifications must be assigned to a channel. If no channel is specified, a default channel is used.

3.7 Rule Editor Prototype

The current state of the architecture only allows to define a new rule by creating rule as a JSON object. In order to facilitate the creation of such rules we developed a prototype for a rule editor. The objective for this rule editor was to give a user the possibility to define their own custom rules and therefore a custom interaction with the LoRaPark.
The prototype was implemented using the Blockly library of Google. Blockly is web based visual code editor that uses blocks to generate algorithms. These blocks can be nested, which allows to create complex algorithms. There was already a Blockly version for JsonLogic[16] which we adapted to our needs. We added custom operators as well as the support for actions a rule can execute.
When opening the rule editor, the user is presented an empty rule. By dragging either condition blocks or action blocks from the toolbar into the rule, the users can modify the rule to their needs. The output is presented as JSON in a text box to the right of the workspace. An example for a rule is shown in figure 11.

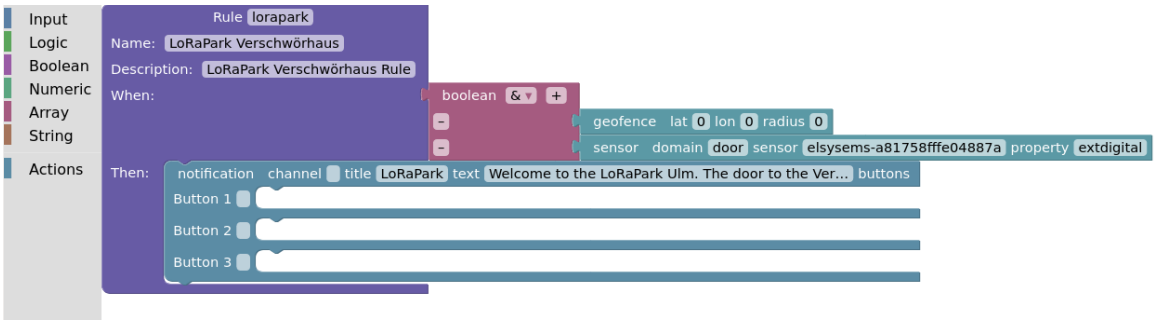

Figure 11: Rule Editor Prototype

[15] https://developers.google.com/maps/documentation/urls/get-started
[16] https://github.com/katirasole/JSONLogic-Editor

3.8 Result

The result of the implementation consists of several individual results. We designed and discussed different architectures which led us to the implementation of a demo Android application and a server based on this architecture. The Android application uses as client the data the server supplies, like the sensor values and rules.
The code for both applications was published publicly on Github[17].
In addition to the code of the application, another repository for documentation was created, which contains the main functionality in the form of UML diagrams[18]. As described in the architecture, the application has a user interface that consists of several independent views. An overview of the resulting UI with some example data is shown in figure 3.8 and 3.8.

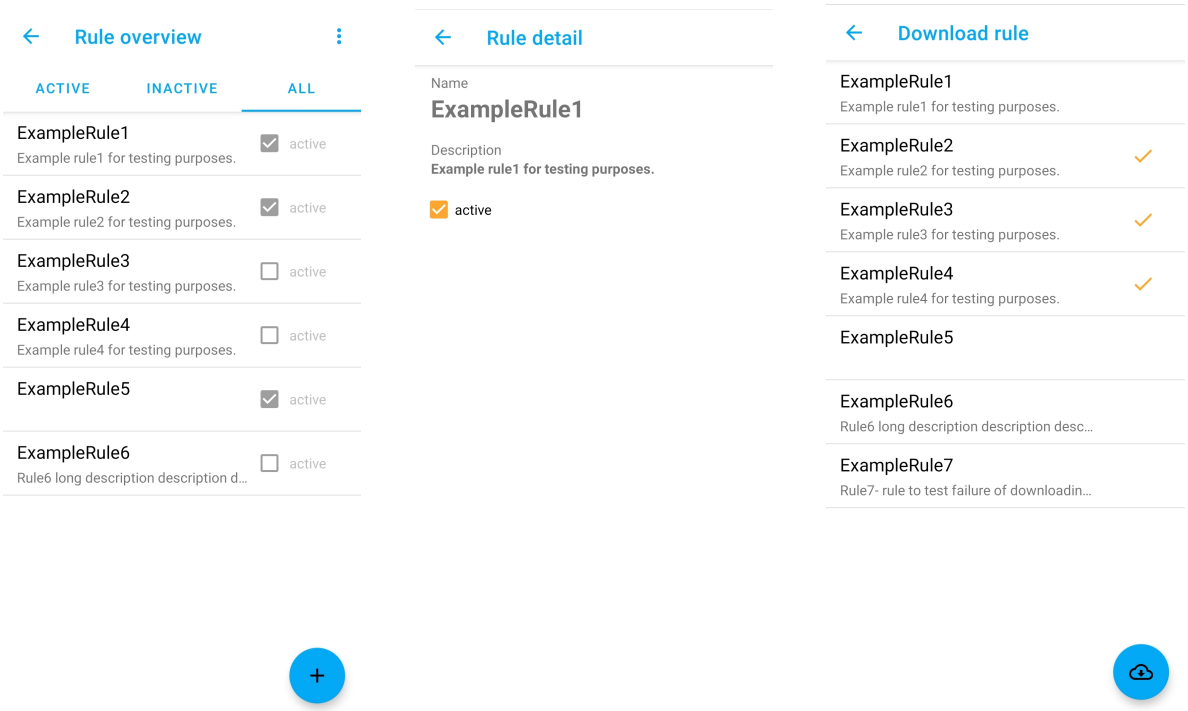

Figure 12: Rule related UI

[17] Android Application `https://github.com/frozzenshooter/LoRaParkApplication`
Server `https://github.com/oli-f/LoRaParkServer`
[18] Documentation `https://github.com/frozzenshooter/LoRaParkAppDocumentation`

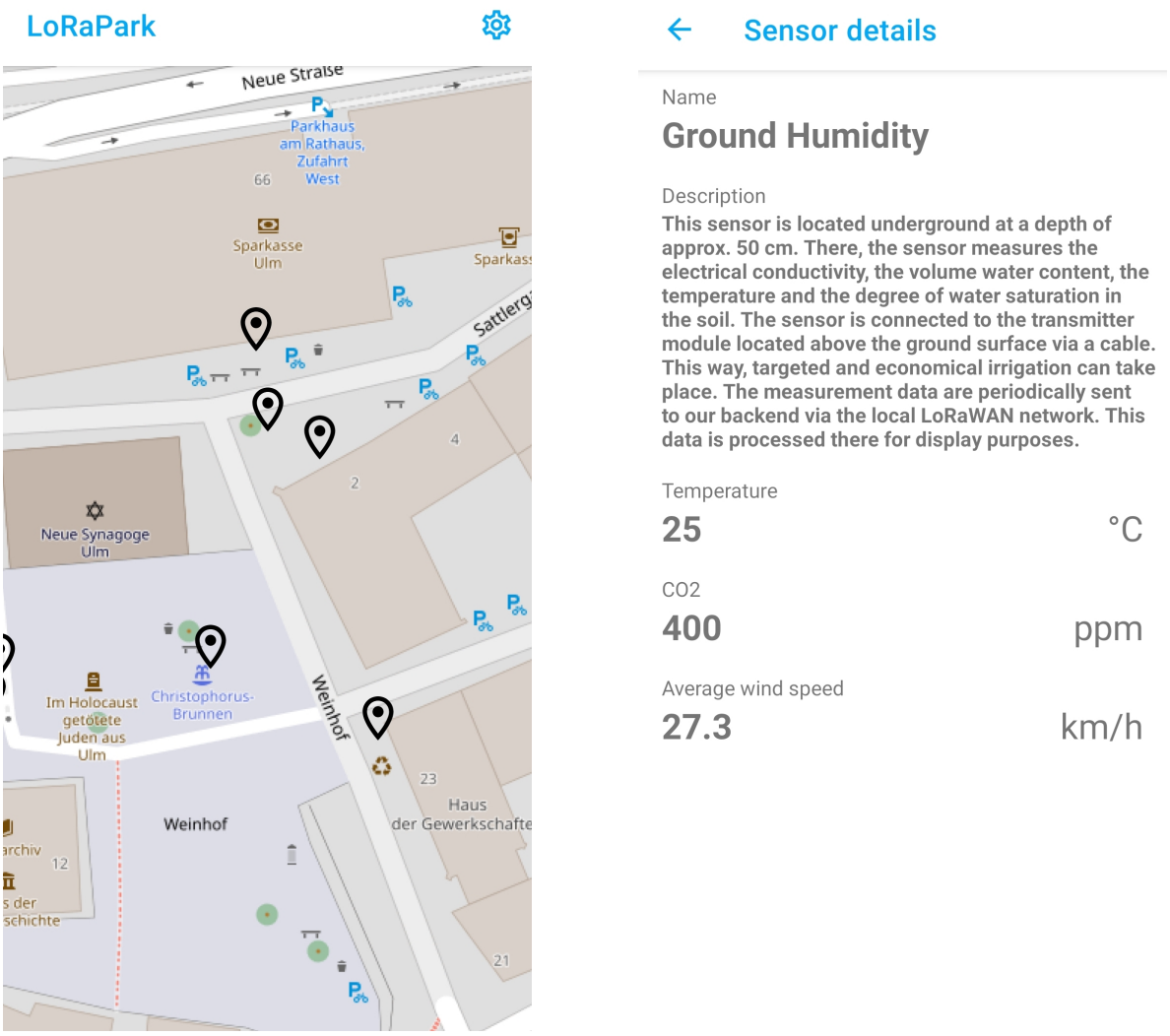

Figure 13: Sensor related UI

4 Discussion

In this section different aspects of the concept and the implementation are discussed.

4.1 LoRaPark API

The LoRaPark API uses OAuth, which requires credentials to access the data. These credentials have to be handed out by the API provider and have to remain secret. This means that the data can not be accessed by client directly and requires some middleware. If the data would be publicly available, the data could directly be used in the client which would diminish the need for a server. According to the operator of the LoRaPark API, there are plans in the future to make the data publicly available, but it is unknown when this will happen.
Another problem with the API is that it is a REST API which can only be pulled. The current solution is to periodically poll the data from the LoRaPark API and store it in a cache. A more elegant solution for our purpose would be an API which can also push data to the client if new data is available.

4.2 Background Execution Limit and Publisher-Subscriber Model

Android allows to run multiple applications simultaneously, which means that an application can not only run in the foreground and is visible to the user, but can also execute jobs in the background. Since Android Oreo, several restrictions and limitations were introduced for the processing in the background [4].
These restrictions help to reduce the resource usage, like the battery consumption, from applications and improve therefore the user experience, because the device can last longer without charging. These restrictions have a great impact on our client application.
This means that we can not simply start a background job and poll for example data at any point of time. The recommendation for Android is to use an automatic scheduler for such a periodic background job. If this job is scheduled too frequent, the system will begin to throttle the app [4] and the sensor data from the server might be delayed.
Android offers different ways to circumvent these restrictions and allow the development of fast-reacting applications. For this use case it would be possible to develop the client based on a publisher-subscriber model by using push notifications. The sensor data could be pulled by the server and if the data for a sensor has changed, the server could notify the clients that subscribed to the values of this sensor.
A common push notification system for Android is Firebase Cloud Messaging provided by Google, that allows to receive the data and start the app in the background. This has the advantage, that the Android OS will allow the execution of code in the background that was triggered by a Firebase Message and circumvents the limitations of the background processing. But since it is a proprietary solution, it was deemed unsuitable.
Another solution would be the implementation of a publisher-subscriber protocol based on websockets. This can be done by creating a websocket based Android service and

additional show a notification that is bound to the service and displays that the service is running. Such a service is called ForegroundService and allows an application to execute jobs in the background without the Background restrictions. We developed a proof of concept for such a websocket service to test the reliability and general usage[19]. The use of such a websocket connection has several disadvantages, because the user will always see the notification and can not remove it from the notification bar and additional the websocket will increase the battery consumption. Overall, the websocket solution promises a decrease in user experience regarding to the Android aspects of the application.
In essence, Android has several restrictions because of the resource consumption of applications. Despite these restrictions, in the future it is desirable to aim for a good response time for the rule evaluation. A further enhancement based on websockets is a good initial choice for this, but notifications from Firebase promise a better performance.

4.3 Geofencing and Awareness API

As mentioned earlier, since Android Oreo, there are certain restrictions on accessing and processing the location of a device in the background. The reason for this is that too frequent location requests lead to increased battery consumption, which negatively affects the user experience. These restrictions mean that geofencing is also affected accordingly: "The average responsiveness for a geofencing event is every couple of minutes or so"[5]. Based on this limitation, there may be several situations where geofences are not evaluated correctly. To evaluate how accurately geofences are measured, we had temporarily extended the demo application so that geofences can be logged and analyzed. With this setup, we then created and executed various test cases.
In the first step, we tried to test the geofences by using an app to fake the GPS position, but we discovered that this procedure forces the direct evaluation of the geofences. Reason for this is that the explicit access to the location circumvents the background restrictions [5, c.f.].
In order to test the functionality, it was therefore necessary to rely on a manual way of testing, which means planning a route based on GPS coordinates, creating corresponding geofences and walking along them. After many kilometers by foot, these tests showed a variety of results, some of which were quite disappointing. The accuracy strongly depends on the speed of the runner, as for example jogging was less accurate than slow walking. It also depends on the location, as for example in some regions not all location detection methods can be used, such as via WiFi networks.
Based on this, it is possible to deduce that the radius of the geofences plays a major role. If you choose a radius that is too small, the geofences are often not detected or only the entering but not the leaving is detected. For our test, we found that at least a radius of over 100m should be chosen, but preferably between 150 and 200m.
But as already mentioned, the results are still not that great, because many geofences

[19] Application: `https://github.com/frozzenshooter/WebsocketTestApp`
Test server: `https://github.com/frozzenshooter/WebsocketTestServer`

were not detected. To improve this in the future, there are several methods[5], such as implementing a ForegroundService that keeps the application in the foreground and thus is not restricted by the background limitations.

4.4 Rule Enhancements

We have developed the rules in such a way that they can be easily extended and also be used for other use cases. This section discusses some possible extensions to enhance the functionality of the application and the rules.

4.4.1 Rule Extension

As already mentioned, the Awareness API provides the possibility to create not only simple geofences, but also more complex fences that, for example, recognize the activity of a user.
Based on these more complex fences, the rules can be extended. This could be used, for example, for a fence that is only triggered when a user is in a certain area and wears headphones. This would be useful in a case where you want to make a direct output using text-to-speech. Another example would be activity detection, as a geofence could for example only affect a cyclist but not a pedestrian. This extension would allow the rules to have an even more precise context and thus an even better reaction to a context change.
Another way to make the context even more accurate would be to access a larger amount of data. This could be based on a received SMS, calendar entries or recent calls. However, the implementation of these additional data sources creates some challenges, because the content of the source has to be interpreted and additional Android permissions are required. Android requires initially a confirmation for each permission (e.g. calendar, telephone, and text messages). Asking for every permission is, on the one hand, bad practice and on the other hand might even prevent an application from being accepted in the Google Play Store, because of the excessive use of permissions.

4.4.2 Custom Rules

Rules are so far defined as static files, that can be downloaded from the repository onto the client. While this has the benefit of being simple, it restricts the usage for the user, because only predetermined use cases are possible.
In order to circumvent this restriction of the rules, in a first step it would be possible to introduce so-called placeholders. The introduction of placeholders allows a user to download a rule and then set a value for these placeholders. An example of this would be a temperature sensor, where the users decide at what temperature they want a notification.
The next possible step would be for users to design rules completely on their own. Such custom rules could be built directly on the mobile device and are then individually configurable. In order to make custom rules more accessible for users without computer

skills, a corresponding editor would be necessary. Such an editor could be developed based on a graphical interface, such as Blockly[20]. As described in section 3.7, we have already developed a first web-based prototype for this.

4.4.3 Dynamic Rule Creation

A further addition would be a more dynamic generation of rules. Up to this point, rules are handled in a static way, which means even the enhanced version with placeholders uses a predefined set of sensors and values. A more dynamic way of using rules would be the on-the-fly generation of a new rule.
Such a generation would allow to create rules based on a more complex and more precise context, because the rule would be created exactly for a specific use case. An example for this would be the computation of a rule based on a route. A user might want to navigate to a certain part of the city and passes several sensors on the way to her destination. The application can use the information about the route and create a rule that uses the sensors that are near this route.
This approach would for example allow to react on environment changes while the user is on the way. For example in case that the water level is rising and certain streets are not accessible, the application could react and update the route with an alternative route. Such an approach allows an application to react way more dynamic, because the rules are not a predefined context, but a correct and dynamic representation of the current user context.

4.4.4 Additional Actions

The design of the actions was chosen to be as flexible as possible. In the future new actions can be implemented and added to the already implemented actions. Such actions can implement a variety of functionalities. They can go from simple actions like sending an mail, a SMS or another messages to complex actions. An example for a complex action, is the connection with another API or the communication with smart home devices, for example to turn on a light when you enter a certain geofence. In combination with other extensions for the rules an almost endless amount of possibilities is created.
Another important point is a possible enhancement for impaired users. The application could offer a setting that enables a compatibility mode, that replaces an action with another action. An example would be a rule that has a notification as action. This action could be replaced by a a text-to-speech action and help a visually impaired user.

[20] `https://developers.google.com/blockly`

5 Conclusion

The objective of this project was to develop and implement a concept for an context-aware app using sensors from the LoRaPark. The most usefull context was found to be geolocation which was represented as geofences. Different use cases were defined and analyzed for their practicability. To define the context of a user based on the sensor values of the LoRaPark, a rule format was developed, that is based on a when-then pattern.
The concept was implemented as an Android application with a Python server as a middleware. The server regularly pulls the sensor values from the LoRaPark and provides all relevant information for the client. The Android application contains a map overview of all available senors with their metrics, a rule management component, and a rule engine.
The application allows to download rules from the server and saves them locally. The rules themselves are regularly checked if their when-condition is met and if so, the then-action is performed. The when-conditions support numerical, logical, array, and string operations as well as geofence and sensor operations. The then-actions support creating notification, navigation, and text-to-speech. Both when-conditions and then-actions are designed to be extendable and to support further operations.
Geolocation has proven to be an interesting and useful context. Further contexts such as calendar information, the user activity, or an aggregation of other contexts could be used to compute more complex contexts like a prediction of what the user is about to do and what sensors are relevant for that.
Additional to the context improvements we discussed in the previous sections several possible enhancements, that have to be further analyzed and could then be implemented in the future.
Overall the project shows the potential for the usage of the IoT and the potential of context-aware applications. They can not only support users in their daily life, but can be implemented in such a flexible way that even users with no previous knowledge can profit from it.

6 Repositories

This section will give an overview of all repositories containing the code and documentation of this project:

Android Application: `https://github.com/frozzenshooter/LoRaParkApplication`

Server: `https://github.com/oli-f/LoRaParkServer`

Documentation: `https://github.com/frozzenshooter/LoRaParkAppDocumentation`

Websocket Prototype: `https://github.com/frozzenshooter/WebsocketTestApp`

Websocket Test Server: `https://github.com/frozzenshooter/WebsocketTestServer`

References

[1] Gregory D Abowd, Anind K Dey, Peter J Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *International symposium on handheld and ubiquitous computing*, pages 304–307. Springer, 1999.

[2] Android Developers. Google Awareness API, 2019. URL `https://developers.google.com/awareness`. (last visited: 2021-03-31).

[3] Android Developers. Android Architecture Components, 2021. URL `https://developer.android.com/topic/libraries/architecture`. (last visited: 2021-04-01).

[4] Android Developers. Background Execution Limits, 2021. URL `https://developer.android.com/about/versions/oreo/background`. (last visited: 2021-03-31).

[5] Android Developers. Background Location Limits, 2021. URL `https://developer.android.com/about/versions/oreo/background-location-limits`. (last visited: 2021-03-31).

[6] Android Developers. Create and monitor geofences, 2021. URL `https://developer.android.com/training/location/geofencing`. (last visited: 2021-03-31).

[7] Android Developers. Guide to app architecture, 2021. URL `https://developer.android.com/jetpack/guide`. (last visited: 2021-03-31).

[8] initiative.ulm.digital e.V. LoRa, LoRaWAN und TTN Ulm, 2020. URL `https://lora.ulm-digital.com/`. (last visited: 2021-03-27).

[9] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: A survey. *IEEE communications surveys & tutorials*, 16(1):414–454, 2013.

[10] Bill N Schilit and Marvin M Theimer. Disseminating active map information to mobile hosts. *IEEE network*, 8(5):22–32, 1994.

[11] Semtech. LoRa® and LoRaWAN®: A Technical Overview, 2020. URL `https://lora-developers.semtech.com/library/tech-papers-and-guides/lora-and-lorawan/`. (last visited: 2021-03-26).

[12] The Things Industries. What is The Things Stack V3?, 2021. URL `https://www.thethingsnetwork.org/docs/the-things-stack/index.html`. (last visited: 2021-03-27).