TU WIEN Informatics

IoT and Edge Computing Technologies for Vertical Farming from Seed to Harvesting

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Master programme Software Engineering & Internet Computing

by

Adrian Jandl, BSc.
Registration Number 01527118

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Schahram Dustdar
Assistance: Pantelis Frangoudis, PhD

Vienna, 18th November, 2020

Adrian Jandl	Schahram Dustdar

IoT und Edge Computing Technologien für Vertical Farming vom Samen bis zur Ernte

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Masterstudium Software Engineering & Internet Computing

eingereicht von

Adrian Jandl, BSc.
Matrikelnummer 01527118

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Schahram Dustdar
Mitwirkung: Pantelis Frangoudis, PhD

Wien, 18. November 2020

Adrian Jandl	Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Adrian Jandl, BSc.

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, 18. November 2020

Adrian Jandl

Acknowledgements

I would like to thank my Advisor Schahram Dustdar for his support. Further, I want to especially thank Pantelis Frangoudis for his support and fast feedback on my thesis.

Kurzfassung

Der stetige Zuwachs der globalen Bevölkerung wird die Menschheit in Zukunft vor eine große Herausforderung stellen: alle Menschen auf der Welt mit genügend Nahrung zu versorgen. Eine mögliche Lösung für dieses Problem ist Vertical Farming. Dabei handelt es sich um eine Methode des Pflanzenanbaus, bei der einzelne Pflanzen übereinander angepflanzt werden, um so den Ertrag einer bewirtschafteten Fläche zu maximieren.

Durch IoT Technologien und Möglichkeiten der effizienten Verarbeitung von großen Datenvolumen, ist es möglich die Vielzahl an Parametern, welche das Pflanzenwachstum beeinflussen, zu analysieren und zu messen. Dies ermöglicht es uns aktives Monitoring zu betreiben, sowie während dem Betrieb unserer Plattform die Daten zu analysieren. Dadurch kann in Zukunft auf Langzeit-Speicher gebaut werden und durch Datenanalyse können neue Erkenntnisse gewonnen werden. In weiterer Folge kann dann eine Datenbank erstellt werden, in welcher optimale Wachstumsbedingungen und Parameter für eine Vielzahl an essbaren Pflanzen gespeichert werden.

In dieser Arbeit legen wir Grundbausteine für solche Mechanismen. Wir analysieren Anforderungen an Softwarearchitekturen, welche solche Informationen messen können. Weiters analysieren wir Netzwerktopologien und unterschiedliche Deployment Szenarien. Darüber hinaus präsentieren wir eine Proof of Concept Implementation eines solchen Systems, mit dem bereits wichtige Messdaten erfasst werden können. Unsere Softwarearchitektur legt dabei einen großen Wert auf Erweiterbarkeit und erlaubt es anderen, diese einfach auszubauen.

Außerdem führen wir eine Vielzahl an Experimenten durch um die Performance unserer Technik zu messen und um zu zeigen, dass sie für unterschiedliche Deployment Szenarien, und Netzwerk Architekturen und Topologien taugt. Dabei analysieren wir die Performance Limits von kostengünstiger Edge Computing Hardware. Dafür rollen wir unsere Plattform auf Single-Board Computern aus und zeigen, dass diese unseren Anforderungen standhalten können. Weiters testen wir die Performance Limits von LoRaWAN Geräten um auch von abgelegenen Farmen Daten erheben zu können. Wir zeigen, dass es zwar gewisse Performance Einschränkungen bei der Verwendung von LoRaWAN gibt, aber dass die verfügbaren Kapazitäten dennoch ausreichen für unser System.

Abstract

With the ever growing global population, a key challenge of the future will be providing enough food for people. One proposed solution to this is vertical farming, which is the practice of growing crops vertically, above each other in order to increase total crop yield for a given space. Supported by IoT technologies, such as sensors and artificial lighting, it allows farmers to utilize spaces that previously did not provide suitable growing conditions for crops, such as underground car parks, or abandoned underground tunnels.

By utilizing developments in IoT technologies, and mechanisms for efficient management of large volumes of IoT-generated data, such as stream processing, it is now possible to measure and analyse the vast amount of parameters that influence plant growth, such as humidity or temperature. This makes it possible to facilitate continuous monitoring and decision making at runtime towards a fully autonomic management of vertical farming processes. Importantly, these technologies can provide support for long-term storage and analysis of such data, in order to create new domain knowledge, such as by developing databases that contain optimal growth parameters for a majority of edible plants.

In this thesis we lay the groundwork for such mechanisms. We analyse the software design requirements for a system that can gather such information, as well as analyse various network topologies and deployment strategies for it. In addition, we show an implementation that already can be used to gather important data. Our software architecture lays a focus on extensibility and allows others to easily build upon it.

Last, we perform extensive testbed experiments to evaluate the performance of our scheme and to demonstrate its feasibility to operate in diverse deployment scenarios and network architectures and topologies. Aiming to identify the performance limits of low-cost edge computing equipment, we deploy the components of our architecture on low-end single board computers and show them to be able to support monitoring workloads that correspond to thousands of sensing devices. We further explore experimentally the performance and design implications of using modern Low-Power Wide Area Networking technologies, and in particular LoRaWAN, to collect monitoring data from remote vertical farms with minimal network infrastructure requirements. We show that while deploying and managing a full private LPWAN over edge computing devices may have significant effects on the overall system's workload processing capacity, it is still possible to support large-scale vertical farm installations.

Contents

CHAPTER 1

Introduction

By the year 2050 the worldwide human population is projected to be above 9 billion [Des11]. One of the key challenges for humanity as a whole for this century will be to provide all 9 billion people on earth with enough food to survive. This will prove especially difficult, as climate change will most likely reduce the amount of land that is farmable. For this, one proposed solution is vertical farming [Des11]. The basic principle behind vertical farming is growing as much food as possible on as small a space as possible, by growing crops above (or below) each other rather than next to each other. This can be done both outdoors by using natural light, or indoors by utilizing artificial lighting. Of course, both solutions come with their own advantages and disadvantages. For example, indoor solutions benefit from a very controlled environment, which allows farmers to control factors more fine-grained, thus increasing total crop yield. They also do not have to consider pest-control, as both herbicides and pesticides can be avoided, since only the desired crop will grow in the growing medium. It is also very efficient space-wise: farmers could grow their crops in underground tunnels or abandoned underground car parks, which would allow them to grow food close to or even in the city. Last, indoor solutions are far less impacted by climate change than outdoor solutions. However, it is still important to choose the location of a vertical farm wisely, to avoid other natural disasters such as earthquakes, hurricanes or tsunamis.

Outdoor solutions, on the other hand, are easier to build, as they do not need artificial lighting on the same level as indoor solutions do. Depending on the location of the farm it may, however, have a positive impact to use artificial lighting together with natural light. Moreover, outdoor solutions allow for the monitoring center of the farm to be further away. This can prove especially useful if the farm is in a rather remote location. Utilizing protocols such as LoRa [Aug+16] enables us to monitor a farm that is up to 15 km away [Ade+17].

It is important to note that for maximum yield, vertical farms usually do not have their crops growing in soil, but make use of aeroponics [WZ92; FMC06], fogponics [Mod+],

Figure 1.1: Breakdown of Vertical Farming Process

hydroponics [JJ16; Jen97] or aquaponics [DR00]. This is a method of growing plants with their roots hanging directly in a nutrient solution in the air, fog, water or a water tank with fish, respectively, rather than them being emerged in soil.

The vertical farming space faces some challenges with the big amount of parameters that influence plant growth and thereby total crop yield. These challenges include, but are not limited to 1. large amount of manual labour when sowing, harvesting or pruning plants, 2. difficulty to physically be able to reach all plants due to the fact that plants are grown vertically, 3. large amount of factors that impact plant growth, 4. hard to notice if a problem arises (again due to difficulty to reach plants). We believe that we can use technology to support the vertical farming space and to solve some of the problems that currently exist.

As such we want to highlight the vertical farming process from seed to fork and analyse where and how technology could be used to support the vertical farming process.

1.1 Vertical Farming Process

To illustrate the process and the decisions that need to be made, as well as which parameters are measurable and tunable, we split the vertical farming process in three main steps: seed to growing medium, growing space and harvest, packaging and distribution. The process, as we understand it is shown in Figure 1.1.

1.1.1 Seed to Growing Medium

We will split this section into three smaller parts: the seeds, the planting medium and transplantation into the final growing space.

Seeds

For the seeds there are a few points of seed quality, that can be measured or optimized. For example the germination rate of seeds is one key metric. This is usually given by the seller of the seeds but can be impacted by environmental factors such as light, humidity or temperature. One possible optimization is the speed with which the seeds germinate. By tuning the various environmental factors, faster germination of seeds can be achieved.

By taking into account the germination rate, the size of the fully grown plants, and how close we want plants to be to each other in one growing pod, we can measure and optimize the amount of seeds per m^2 or cm^2. This could result in 1. less waste of seeds and 2. less waste of seedlings, when we later need to cull some of the weaker plants to make more room for the healthier plants.

Planting medium

By using different planting media we can achieve different results. Depending on the type of plant, there will most likely be a different optimal planting medium - this is an area for research, where technology can assist in gathering the necessary data to achieve the highest yield through the best growing medium.

Transplantation

When culling the weaker plants and moving them to their final growing space, a large amount of manual labour is currently needed. While this could be achieved through robot assistance, we do not believe this to be feasible at this point in time.

1.1.2 Growing space and harvest

Since this is the step in the process where the most time is spent, it is clear that there are many parameters which can be optimized. The exact time that the plants will be in the growing phase depends on the type of the plant. For example, lettuce may take around 6 to 7 weeks from seeding to harvesting [Kra08].

Growing space

In general, in this step, plants will be stacked on top of each other in some form and will need light, water and nutrients. There are many different parameters which can be directly influenced by a vertical farmer and in this thesis we wish to provide a solution with which it is possible to measure exactly these parameters. We want to provide a proof of concept system which allows a vertical farmer to measure various data points while his plants are in the growing stage and be able to visualize them. For this we will specifically focus on a low-cost solution with regards to hardware and will create a software architecture that is easily extendable.

Harvest

To continue with our analysis of the process there comes a point when the produce is to be harvested. This is again a step which is quite heavy in manual labour. Depending on when the farmer decides to harvest his crops, his total annual crop yield can vary massively. By harvesting plants that are not yet fully ripe, a farmer may be able to achieve one more harvest in a year, which raises an interesting question: can the produce be harvested a few days earlier than usual, thus increasing the total amount of growing cycles in a year and thus increasing the total yield over the span of 12 months, while reducing the amount of produce harvested in a single harvest? While this is an interesting point, in our thesis we will not be focusing on it.

1.1.3 Packaging and Distribution

The packaging process is not much different from packaging in traditional farming, therefore we will not elaborate on it.

Distribution / Point of Sale

It is, however, very interesting to consider how we can influence distribution of produce that was grown in a vertical farming space. For example, vertical farming allows us to grow crops in or near the city, thus greatly reducing the time and distance that the produce has to travel. This would allow us to sell fresh vegetables in the city, minutes after they have been harvested - a feat that is unthinkable with traditional farming methods. It would also allow a vertical farmer to harvest "on demand", even further increasing the freshness of the produce.

1.2 Contributions

With so many different factors affecting plant growth such as water, lighting, temperature and others it is clear that this is an area for research. Further, the cost for a farmer is currently not clear and whether for a small farm it is even economically feasible to utilize a system to monitor a vertical farm. This raises three main research questions for our work.

RQ1: How can different factors that impact plant growth be monitored and how should such software architecture be designed to facilitate data gathering and processing?

RQ2: Is it feasible to operate a system for vertical farm monitoring and autonomous management using low cost edge computing hardware?

RQ3: What are the performance and capacity limits of edge computing hardware in terms of processing vertical farming monitoring data and what are their cost implications?

We identify five main contributions in our work, where each attempts to provide an answer towards one of our research questions.

1.2.1 Architecture

Our system is designed as a service oriented architecture (SoA) with separation of concerns and a clear focus on extendability. Specifically we build different modules that each are in charge of a specific part of the workload in our system. This aims to provide an answer towards the question how an architecture can look that supports the measuring of growth factors.

1.2.2 Deployment scenarios

We provide a comparison of different deployment scenarios such as indoor and outdoor. For this we will also highlight various networking technologies which our system can support such as LoRaWAN[1] or Wi-Fi. This further ties into the previous point, in that it is necessary to analyse deployment scenarios to be able to best design an architecture for our system.

1.2.3 Extensible Decision Engine

Next, our system allows for multiple decision modules to interface with it. These decision modules can read the data that is gathered by the sensors and decide whether the measured value is critical and if the vertical farmer should be notified of it. We provide two such decision modules in our proof of concept work to show that it is easy to add new decision modules. As our software architecture is heavily influenced by this requirement we explicitly list this as a contribution. Moreover, this requirement also influences the answer towards our first research question as our system architecture needs to be able to facilitate multiple decision engines.

1.2.4 Proof of Concept

Furthermore, we build a proof of concept solution that allows vertical farmers to use cost-effective single board computers to measure various data points of their vertical farm. These data points are then visualized in a user-interface where current and past data is shown. The system is extensible in a way that allows for new sensors to be added easily via a plug-and-play system without having to write any additional code. Our system allows for containerised deployment over different types of hardware, which simplified its setup and operation.

1.2.5 Performance Evaluation

Last, we perform extensive benchmarks on our proof of concept system to be able to answer research questions two and three. We first analyse the maximum load that low cost edge computing hardware such as Raspberry Pis can handle and then show which costs a farmer has to expect when planning to implement such a system.

[1]LoRaWAN is an open standard that specifies the communication protocols on top of the proprietary LoRa physical layer.

1.3 Structure

In Chapter 2 we give background information necessary to follow the rest of this work. In Chapter 3 we analyse the state of the art and focus on related work that is relevant to ours. In Chapter 4 we show the design and architecture of our proof of concept system. Chapter 5 presents different network topologies and deployment scenarios for a smart farming monitoring solution. Chapter 6 displays our implementation, as well as the interfaces that our platform provides, where others can extend it. In Chapter 7 we provide benchmarks of our system and also discuss the limitations of it. Finally, Chapter 8 concludes our chapter and reiterates our research questions and our findings.

CHAPTER 2

Preliminaries

There are a number of paradigms and technologies which are fundamental for our work. In this chapter we highlight the key concepts behind those necessary to understand the majority of this thesis.

2.1 Edge and Fog Computing

When the processing power requirements of programs started to exceed the computational power of personal computers, users started looking towards the cloud as a possible solution. Now that the data generated by the vast amount of IoT sensors and other small devices has risen much faster than the maximum throughput of communication networks towards the cloud, computations are being pushed away from the cloud, and closer to data generation [GD18; Sat17]. Figure 2.1 illustrates how the various devices communicate with each other. As can be seen, the edge devices act as both data producers and consumers, that is, for a given input the edge devices may also take an action. Edge computing use-cases are many and do not necessarily have to encompass cyber physical systems. A simplified example of edge computing is any calculation that is done on the end-device, rather than the cloud device, to save the round-trip time that a request to the cloud server would take. Key benefits include, but are not limited to, faster total processing time (due to the elimination/reduction of network traffic), privacy protection, data pre-processing or even caching.

While the terms edge computing and fog computing are often times used interchangeably in literature, we understand there to be a difference and understand it to be as follows. Where fog computing utilizes a number of devices such as cloudlets or foglets between the data generating devices and the cloud, edge computing goes a step beyond and performs calculations even closer to the edge of the network. Figure 2.2 shows the fog as a communication level between the cloud and the edge.

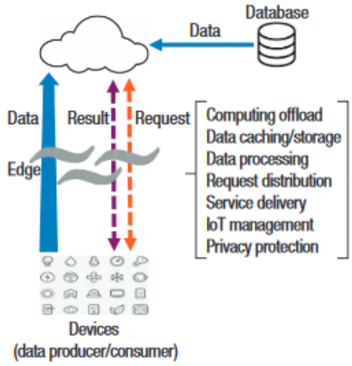

Figure 2.1: Communication channels between databases, cloud and edge devices [SD16]

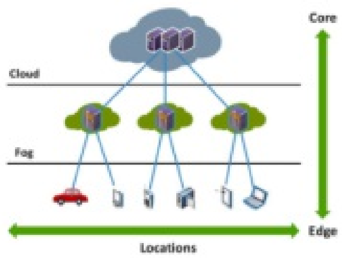

Figure 2.2: Communication between Edge, Fog and Cloud [DAM19]

Even though it may sometimes not be so clear whether a given architecture is an edge computing, or a fog computing architecture, the main difference between the two is the distance that data travels between generation and computation.

2.2 Service Oriented Architecture

Service oriented architectures essentially focus on a number of key points [HS05]:

1. Loose coupling
2. Implementation neutrality
3. Flexible configurability
4. Persistence
5. Granulartiy

First, this means that components should not be tightly coupled and only the high-level contracts between the various components should be considered. Next, it means that service functionality may not depend on implementation details such as programming languages, as it would violate the freedom of different implementation possibilities, and also may prevent inclusion of legacy applications. Third, the system can be configured dynamically and services adapt to the new configurations. Fourth, services do not require a long lifetime, as the computations are also in a dynamic environment. Last, the granularity of services is a coarse one, this means, that the dependencies between two services should be low and also reduces the amount of communications between two services, but makes the communication more meaningful, i.e. the services send few messages, which have big impact, rather than many messages with low impact.

2.3 Stream Processing

With the ever increasing amount of data generating devices, acting upon the generated data becomes a challenge. This is true especially if the data is time critical, and a result has to be presented in near real-time, due to the nature of the request. What we mean by this is that there exist questions that need to be answered quickly, or their result is no longer relevant. As an example, consider a system that analyses car traffic in a city. Various sensors send data of car positions and the system is supposed to tell us whether there is a traffic jam. If all the data is gathered to a central point and only then analysed, it may happen that the result of the calculation is no longer relevant, as the jam has already dissipated. Therefore it is clear that sometimes results need to be available quickly. [Akb+15] What this means, is that sometimes a result has to be given on a live stream of input data, rather than waiting for all data to be finished collecting. In fact, data may never finish collecting, as our system always answers a given question for a specific timespan. This is the practice of stream processing.

CHAPTER 3

Related Work

This chapter discusses related work in our space. It first introduces Precision Agriculture and explains that vertical farming is a type of precision agriculture, hence it is relevant to our work. The following section deals with work in the Edge Computing domain. Next, we show work that stands in connection with stream processing. Further, Section 3.4 shows networking technologies for the IoT. Section 3.5 highlights other works analysing software architectures for a vertical farming application, and Section 3.6 portraits some work that shows methods for image detection / classification of plants. Next, we show work that has either built proof-of-concept or beyond systems that are already in use. Last, Section 3.8 gives an introduction to smart cities, with an outlook of how (smart) farming can be moved inside cities.

3.1 Precision Agriculture

Agricultural advances have steadily increased the amount of produce that can be harvested from a field. As such, also the size of fields has increased. Before the mechanization of agriculture, the small size of fields allowed farmers to tend to the various parts of a field in a different way, and allowed them to vary the treatment that the various parts of the field received. Nowadays, however, the sheer size of farms makes this no longer feasible, and as such an entire field generally gets treated in the same matter. Precision agriculture can utilize advances in sensor technologies, robotics and others, to again reap the benefits of varying treatments depending on small differences with regards to environmental field circumstances. As such, if one part of a field receives partial shade, it might require less watering than a part that is fully in the sun. This is the practice of Precision Agriculture [ZWW02]. With the abundance in sensors and data available in vertical farming, it is clear that the vertical farming space is a type of precision agriculture.

Bongiovanni and Lowenberg-DeBoer [BLD04] explain precision agriculture and sustainability. Sustainability can be defined as the ability to constantly keep producing a steady amount of any given resource. In their case it is targeted specifically towards farmed crops. The challenges in traditional farming are clear, and classified as follows: 1. natural, such as heterogeneity of soil and topography of the land, 2. random, such as environmental impact by rain or drought and 3. managed, for example regarding the fertilization or the seeds. It is clear that vertical farming with its controlled environment is a perfect candidate for indoor precision farming.

3.2 Edge Computing

Chen et al. [Che+18] aim to improve our understanding of edge computing possibilities and room for improvement in that space. Projects such as ThriftyEdge propose resource-efficient edge computing solutions, in which each device attempts to minimize the amount of cloud resources it uses, but still satisfies its quality of service requirements. Their approach is novel, in the way they try to optimize the location at which a given calculation takes place. There is also various other work done in the optimization of edge computing networks, especially targeted towards task offloading algorithms. Other such work includes that done by Zhang et al. [Zha+18], You et al. [You+16] and Zhang et al. [Zha+16], whose focus is mostly on energy efficient task offloading algorithms.

A general overview of fog computing and its challenges, such as data stream processing and analytics can be found in the work done by Yang [Yan17]. Specifically, a number of different application scenarios was analysed: 1. IoT Stream Query and Analytics, 2. Real-Time Event Monitoring, 3. Networked Control Systems for Industrial Automation, 4. Real-Time Mobile Crowdsensing. They explain the typical application scenarios and what they entail. Our work is a mix of the first three scenarios, in that we gather stream data and analyse it, have real time event monitoring, such as alarms and automation in a way that we may tackle in future work, such as physically acting upon sensed data by e.g. notifying a motor to open a window, or increase the automated watering schedule. Moreover, they explain the models and architecture of typical fog streaming applications, which is related to our work specifically, since they discuss the data management layer, such as Apache Kafka, which is in use in our proof-of-concept system.

3.3 Stream Processing: Handling IoT Data

We have explained the key concept behind stream processing in Section 2.3. In this section we will present a number of related works in this field.

Akbar et al. [Akb+15] introduce a novel approach for context-aware stream processing. In their work they provide a method to make complex event processing (CEP) context aware, i.e. consider the season or the time of day in the calculation.

"The research area of CEP includes processing,analyzing and correlating event streams from different datasources using distributed message-based systems to extract high-level or actionable knowledge in near real-time [Luc02]." [Akb+15]

In their work, they present a distributed approach, which can run on embedded devices and can update rules during operation. Specifically, they propose an approach that finds optimized parameters by utilizing machine learning.

The topic of stream processing is also put into the context of smart cities by Tönjes et al. [Tön+14], which we elaborate on further in Section 3.8.

Extensive work on the benchmarking of IoT stream processing can be found in the work done by Shukla and Simmhan [SS16]. The authors go a step further in [SCS17], where they propose a new benchmarking solution called RIoTBench, which includes 27 common IoT tasks, that are implemented as re-usable benchmarks.

3.4 Networking Technologies for the IoT

Various communication technologies and network architectures have emerged that target IoT scenarios [KD20], both at connectivity [Din+20] and at the application levels [Diz+19]. One of the particular connectivity scenarios that we explore in Chapter 5 is based on Low Power Wide Area Networking (LPWAN) technologies, and in particular LoRaWAN.

LoRaWAN is an open specification supported by the LoRa Alliance[1]. In contrast to NB-IoT it uses unlicensed spectrum bands (ISM bands), whereas NB-IoT is a subset of the LTE-standard. An in-depth comparison of LPWAN technologies has been done by Mekki et al. [Mek+19].

LoRa devices can be categorized in three different classes: Class A, Class B and Class C devices. Class A devices have two receive windows in which a downlink message is possible. These receive windows only become active after an uplink message has been sent, and are typically 1 second and 2 seconds after the uplink message. A downlink transmission is only allowed, after a successful uplink message for Class A devices. If a downlink message is received in the first receive window, the second receive window of the initiating uplink message does not become active. Class A devices have the lowest power consumption of LoRa classes. Class B devices can be used for applications, which require more downlink transmissions. They do, however, have the trade-off of higher power consumption. Class C devices have the most downlink transmission possibilities, as they listen to the downlink transmission channel all the time, unless they are currently sending an uplink message themselves. The transmission rate depends on the desired transmission range and lies between 0.3kb/s and 27kb/s, whereas the range depends on environmental factors such as houses, mountains, forests and ranges from 2-5km in urban areas up to 15km in suburban areas [Ade+17].

A detailed technical analysis on LoRa for the IoT has been done by Bor, Vidler, and Roedig [BVR16] and an extensive evaluation on connectivity in cities and typical city environments such as buildings and underpasses has been performed by Wixted et al. [Wix+16]. Notably, LPWAN technologies in general and LoRaWAN in particular

[1]https://lora-alliance.org/

have recently emerged as interesting technologies to be studied in conjunction with smart-agriculture [Sin+20; Mil+20].

3.5 Software Architectures supporting Precision Agriculture

Sivamani, Kwak, and Cho [SKC14] present a potential REST API structure directly related towards vertical farming. They propose endpoints for both monitoring and controlling services, which they implement in a Jersey[2] implementation of JAX-RS[3]. It is not entirely clear which protocol is used to send the measurements from the sensors towards their database, which in turn is accessed by Java Services. However, they do not mention any streaming or low-power requiring protocols in their work.

An example of how FIWARE [4] can be used to build a system that already has performed experiments from 2013 to 2015 is presented in the work done by López-Riquelme et al. [LR+17]. "FIWARE is a curated framework of open source platform components To accelerate the development of Smart Solutions" and whose goal it is "to build an open sustainable ecosystem around public, royalty-free and implementation-driven software platform standards that will ease the development of new Smart Applications in multiple sectors"[Fiw]. The authors use the cloud computing possibilities provided by FIWARE to monitor outdoor farms, such as lemon trees and almond trees. The architecture they present consists of a hardware device that performs the measurements and sends them to the FIWARE server component, a software component that receives all the information gathered by their remote sensor nodes, as well as a graphic user interface. Each component is explained in detail in their work.

3.6 Image Detection of Plants

The work that is done by Tenzer and Clifford [TC20] and Lu, Chang, and Kuo [LCK19] could be relevant for extending our system, as their image detection done with neural networks could be implemented in our system to gather and analyse image data. This would allow us, in the future, to create a database in which we can link various growth parameters and the actual plant growth value. Both papers utilize neural networks to monitor plant growth. Tenzer and Clifford [TC20] combine past approaches to a new approach that allows them to measure plant growth even under bad light conditions. This is especially relevant since in a vertical farm there is a chance that it is very bright. Their solution can circumvent this by utilizing recent breakthroughs in deep learning for image segmentation. Additionally, their method is of particular interest because it does not rely on a high amount of manually annotated data. Lu, Chang, and Kuo [LCK19] on the other hand use deep convolutional neural networks to specifically measure lettuce

[2] https://eclipse-ee4j.github.io/jersey/
[3] https://projects.eclipse.org/projects/ee4j.jaxrs
[4] https://www.fiware.org/

growth. While this work does rely on manual annotations, it is still highly interesting, as after training, their CNN could estimate leaf area of lettuce with a mean accuracy of 97.63% and a maximum error of $0.25\,\mathrm{cm}^2$. Like the previous paper we could utilize this method to bridge the gap between the measured data and the actual impact it has on plant growth.

The work that is done by Tripicchio et al. [Tri+15] is interesting, as the use of drones to monitor fields could be a promising future step for our system. Specifically, drones are used to monitor the area used for agriculture to extract data about the depth of plowing, which is an indicator of the impact on the ecosystem that is done by farming. Such drones could be used in our system to generate data and eliminate the use of statically placed cameras. Due to the fact that our system will be run indoors, it may be possible to only use a single drone.

3.7 IoT Testbeds and Prototypes for Smart Agriculture

In the work done by Jayaraman et al. [Jay+16] a similar undertaking as ours is done. In it the authors focus on traditional smart farming that grows crops in soil, rather than vertical farming. They do, however, have similar requirements and goals as we do. SmartFarmNet, the IoT platform they present, can effortlessly integrate "virtually any IoT device", "Supports scalable data analytics" and allows plant biologists or farmers to analyse and visualise the data they have gathered. It is an extensive platform which outperforms all other platforms they have analysed and will allow biologists and farmers to perform the exact studies we aim to make possible. Their goal is to compare various different fertilizers, watering schedules, soil compositions and more. It has already been deployed across Australia and has been used to collect and analyse data from tens of thousands of sensors deployed in remote field locations. In their work, the authors present outcomes of four years of field research. However, the fact that the work deals with traditional farming brings with it other challenges than we face. One key difference for example is that we can tune the amount of light our plants receive through artificial lighting, whilst they have to rely on the sun and can only measure light, but not influence it. Additionally, they cannot stop the rain, so to say, while we can fully stop watering schedules.

Haris et al. [Har+19] use a more cloud-focused computing approach and are utilizing the strong computing capabilities of the cloud, rather than relying on resource constrained devices and edge computing as we are. This means that they probably can handle higher load than we can, although they provide no benchmarks for this. On the other hand, they are much more reliant on Internet connectivity and infrastructure than our solution is, as we do not need to transfer any data out of our system and can handle all calculations on-site rather than having to transfer data to the cloud. They make use of the Arrowhead framework[5] [Del17] to build a test bed that will, in the future, allow them and others to conduct further research in this area. The idea behind Arrowhead is to abstract IoT

[5] https://www.arrowhead.eu/arrowheadframework/this-is-it

devices to services and to reduce coupling between devices/services. In their work they also make use of Arrowhead's service registry and discovery mechanism, while in our implementation we rely on Apache Kafka and its topic handling mechanisms.

FarmBeats [Vas+17] provides a novel approach to a number of challenges in the agricultural space, all of which are technological solutions to existing problems. Their goal is, like ours, to keep the costs for the farmer as low as possible, while still providing an excellent service. However, since their farm is one that is outdoors, they do not combat exactly the same challenges as we do. For one, they have built a novel weather-aware IoT base station device, which utilizes weather forecast data to optimize battery usage and reduce downtime. In this, they have managed to reduce weather induced downtime - typically due to little to no sun, which prevented solar panels from properly charging the device - from 30% in an earlier year of measurement to zero. This is achieved through estimation of power generated through solar panels and duty cycling of the IoT base station. Next, they tackle the problem of generating a digital map of the farm by utilizing unmanned aerial vehicles (UAVs) such as drones. One challenge in this space is, that it is usually a very time consuming task, since the battery lifetime of a single drone is not so long. To reduce the downtime of the drones, they discuss two components: 1. a path planning algorithm for UAVs that utilizes wind to reduce battery usage, and 2. orthomosaic generation from UAV videos. The path planning algorithm aims to minimize the amount of waypoints in the path, and also exploits wind for acceleration, and drag for deceleration.

The work that is done by Davcev et al. [Dav+18] is similar to ours, in that they have built a testbed and a proof-of-concept system for gathering IoT sensor data. Their focus lies on the LoRaWAN protocol, with which they have built a system to monitor vineyards. Specifically, they monitor air temperature, humidity, leaf wetness and soil moisture readings. The various data collection nodes are positioned within 1km of a base station, a range that can easily be covered by LoRa, as it can cover up to 15km communication distances in suburban areas [Ade+17]. Others, such as Zamora-Izquierdo et al. [ZI+19], have also built solutions that can gather data using IoT technologies. They utilize an architecture that uses sensors on the local level to gather data, which is then transmitted to the cloud, where computation heavy tasks are performed. Additionally, they allow for emergency actions to be taken on predefined conditions, such as opening of windows and turning on ventilation. While, for example Ferrández-Pastor et al. [FP+18] shows the use of fog and edge computing paradigms to measure and monitor a greenhouse, but also uses a decision tree, that is created with expert domain knowledge to optimized water consumption of a given plant.

3.8 Smart Cities

Especially in the area of Smart Cities there are a number of different use cases which can be assisted through the use of edge or fog computing technologies, as well as IoT technologies. Such examples include smart parking or smart traffic lights. A general overview of smart cities, with its desired parameters and challenges is described in the work done by Tönjes et al. [Tön+14]. They highlight three main components in a smart city setup: 1. Large-Scale IoT Stream Processing, 2. Reliable Information Processing and 3. Real-Time IoT Intelligence. This can be seen as a good introduction to smart cities. The previously mentioned smart parking is a new concept, that may allow cyber physical systems to manage parking in a city. This can range from parking meters with smart routing - *Where is the nearest free parking spot?* - to fully managed smart parking lots, which may allow cities to better utilize the available parking space. One such solution is proposed by Khanna and Anand [KA16] in which a smartphone application is used to denote an occupied parking spot. However, it is worth mentioning that this is an area of research that has been receiving attention for more than a decade. Already in the year 2007 there have been proposed solutions to use RFID chips to reduce staff cost and to manage parking lots [PI07]. But the advent of IoT technologies and mobile network coverage in general will allow parking lot providers to increase the quality of service: drivers will be able to see the occupancy of a parking lot before going there.

There exist various works on the topic of smart traffic lights or smart traffic control systems. Such systems could utilize IR sensors to decide whether there is a car waiting at a red light [Gha+16] or use video/image processing technologies to achieve the desired result [KSS14]. The goal for all of these methods is clear: increase traffic flow and reduce inner city congestion caused by poorly timed traffic lights.

Most connected to us, however, is the idea of growing crops inside smart cities. This has a number of benefits, such as short supply chains and the creation of new jobs. Additionally it may make consumers more sensitive towards plants and plant growing cycles, which is an important step in sustainable farming practices.

CHAPTER 4

Architecture Design

In our work we want to analyse how the software architecture for a monitoring system for a vertical farm can look. For this, we will first analyse the requirements we have towards our system, and then propose a possible system design that fits these requirements and will allow us to perform the tasks we want to. We will then map the architecture towards possible deployment scenarios and network topologies in Chapter 5 and show our implementation of a proof-of-concept system that can perform the key points of data measuring and monitoring in Chapter 6.

Since different sensor technologies often times are best addressed by a specific device, protocol, or programming language, the multitude of devices that are actively communicating with the sensors are a very heterogeneous group. By considering this already in the design phase of our system, it allows us to be language and protocol agnostic as long as the interfaces between the various points of communication are well defined.

4.1 Design Philosophy

In this work we follow the Monitor-Analyse-Plan-Execute with shared Knowledge (MAPE-K) [Hor01; KC03; Mau+11; ARS15] design philosophy and provide the system support to enable the autonomic management and self-adaptive operation of a vertical farm. In the context of our work, but also for vertical farming in general, we can identify the following mapping of desired functionality to the elements of the MAPE-K loop. One other work that is highly relevant to ours is that done by Lee, Seo, and Kim [LSK19] in which a MAPE loop-based self adaptive system for the IoT is presented.

4.1.1 Monitor

Our sensors collect information about the vertical farm which is then delivered to components where data is processed. Monitoring in our case does not depend on a

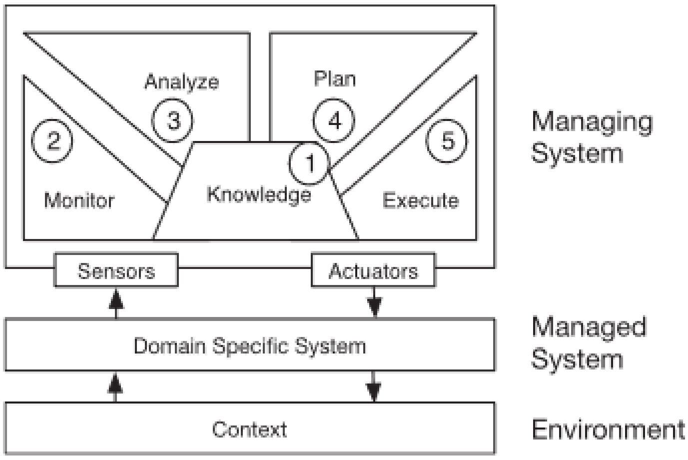

Figure 4.1: MAPE-K Model for self adaptive systems [IW15]

specific sensor technology, and can support various protocols for communication to the Analyse space.

4.1.2 Analyse

Readings and events gathered by sensors are processed and analysed. The analysis can consist in evaluating the collected values in terms of models of desirable vertical farm behaviour and compliance to operational requirements. These may range from simple checks of compliance with predefined threshold-based rules to more complex models.

4.1.3 Planning

Planning processes are responsible for identifying required changes and how these changes can be achieved. In our case this corresponds to sensor readings that are too high / too low and possible plans include opening of windows, or increasing water supply, as well as raising alarms when the farmer has to manually take action upon a metric.

4.1.4 Execute

The results of the planning stage are transformed to actions which are communicated to various components, such as the user interface or - potentially in the future - actuators in our farm which can perform tasks such as configuring air flow, water supply or temperature. In an early stage, the execute phase might simply consist of notifying the farmer that a manual action has to be taken.

4.1.5 Knowledge

Our system can be extended to allow knowledge extraction from the measured data in a straightforward way. Analysing historical data over the entire lifetime of the farm as well as potential results stemmed from previously taken decisions can be used extend domain knowledge of the system operator, i.e. the farmer. While this is outside the scope of this work, it is easily possible to add this functionality to our modular design.

4.2 Requirements

We want our system to be extensible and to allow others to expand our system without having to change much of the existing codebase. For this we designed our system with five main requirements in mind. It is important to note that these are hard requirements and all design decisions were made with this in mind, and technologies that were not suitable towards one of our requirements were decided against.

4.2.1 Different Sensor Technologies

We want to be able to handle different sensor technologies without having to change the system and its architecture. Specifically, the system should not care whether the sensor is, for example, a humidity or a temperature sensor. To this end our system will have an interface where sensors can register themselves and where they can send their measurements. It is important for this interface to be able to handle different units of measurements, such as relative humidity (in the range of 0-100%) or °C (for our use case typically between $-10\,°C$ and $40\,°C$). It is worth noting, that at the time of writing our proof of concept implementation of our architecture supports one measurement value per sensor. The system can, however, easily be extended to allow for a sensor to deliver multiple values in one reading.

The system also should not care which programming language, or sensor technologies are employed. The interface will expect the name of the sensor as well as the value measured in the data that is received.

4.2.2 Multiple Decision Modules

Depending on the value measured some action may have to be taken. If, for example, a sensor reports a value that is too high / too low, then the system will be able to raise an alarm. These alarms are to be raised in two different ways: 1. the system should push a message towards a notification system, on top of which a user interface can be built and 2. the notification system needs to be able to notify the sensors that an alarm should be raised. It is then the sensors' responsibility to raise an alarm, such as making a sound or turning on a warning light, on the sensor side of the system. Our architecture will allow multiple such decision modules to be used independently of each other. Most importantly, multiple decision engines have to be able to co-exist, without negatively impacting each other's performance and functionality.

4.2.3 Resource constrained devices

Our system should be deployable to many different locations. For this we will be making sure that our entire system architecture can run on resource constrained devices such as single board computers, which we expect to be typical of edge-assisted vertical farming installations. In Chapter 7 we will provide a benchmark to show that even on resource constrained devices, such as a Raspberry Pi our system can function properly. Additionally we will evaluate different deployment variants together with their maximum possible workload.

4.2.4 Service oriented architecture

Our system should follow service oriented architecture principles, such as decoupled components, well defined interfaces and separation of concerns. This allows us to put our focus on extensibility. What we mean by this, is that we can easily plug in new sensors and new decision modules with minimal disruption of the architecture and also with minimal disruption of an already running system. Specifically, we adhere to the previously mentioned principles as follows: our components are decoupled and do not require knowledge of each other's implementation details. The interfaces between them are well defined and can be found in Chapter 6. Also, the components do not share any responsibility with regards to functionality, i.e. no two components perform the same task.

4.2.5 Capability to configure different modules

Last, it is important that different modules can be configured independently of each other and also can be used independently of each other. That is to say that no two decision modules should be dependent on each other and also no two sensors should be dependent on each other. This is in line with the previous point regarding a service oriented architecture. However, there are various components of our system that are necessary for the rest of the platform to function.

4.3 System Design

Our system will be split into different modules which interface with each other over various protocols. In this chapter we will highlight the various modules and interfaces, as well as the requirements we have for the communication between them. We show the data-flow directions as well as a minimal setup of modules in Figure 4.2.

First there are one or more sensors which take a measurement of a given unit. These sensors need to publish the measurement to a Publish/Subscribe system which will allow the Controller and the various Decision Engines to subscribe to the measurements. The reason we chose a Publish/Subscribe model, is that it allows our decision engines to easily interface with an already running system and that the Controller needs no prior

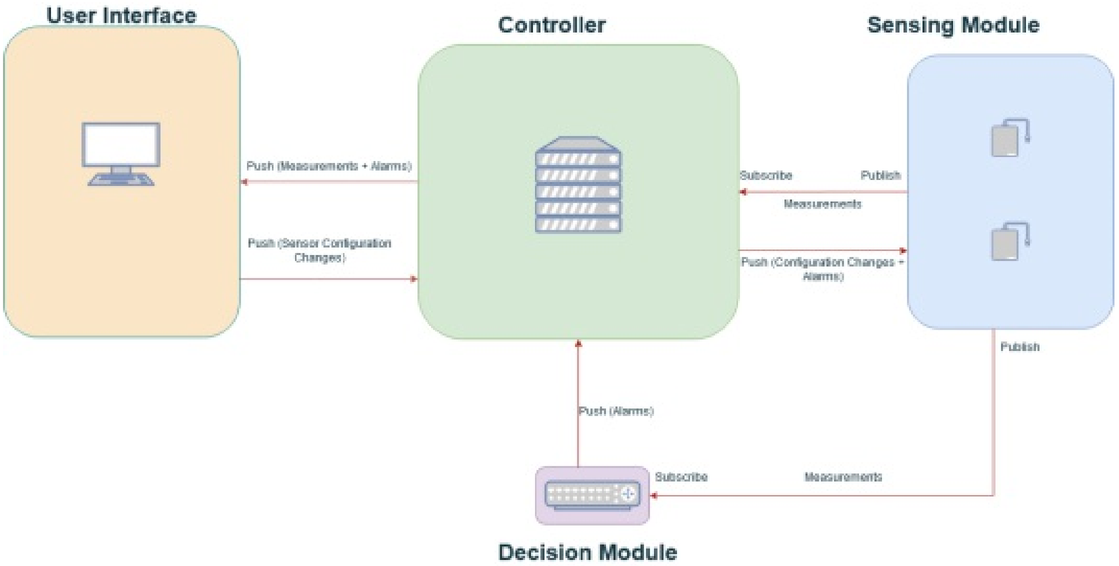

Figure 4.2: Architecture and data-flow directions.

knowledge of existing decision engines, with which it will later need to communicate. When a message is read by our Controller it

1. propagates the measurements to the User Interface and also
2. needs to be able to inform the sensors that the measurement is critical.

Next there can be zero or more decision engine modules which read the same measurements and decide, for a given value, if any and which action should be taken. Last, the already mentioned user interface displays current and past measurements and also informs the user of any alarms that have been raised by the decision engines.

In this section we will show various method calls between the components. These can be thought of as requests from one service to another, but should be considered protocol-agnostic at this point. Note that we will be supplying the various method calls that display the communication as shown in Table 4.1 where the header is the name of the method and further rows indicate one parameter each, together with an example value and an optional comment.

4.3.1 Sensor Gateway

The Sensor Gateway is in charge of feeding the sensor measurements into our system and into our stream processing pipeline. We have chosen a gateway approach for a number of reasons. First, this allows us to receive sensor readings from sensors which are controlled by different programming languages. This means, that one sensor gateway,

function name			
variable1	data type for variable 1	example value(s)	comment(s)
variable2	data type for variable 2	example value(s)	comment(s)

Table 4.1: Request specification example

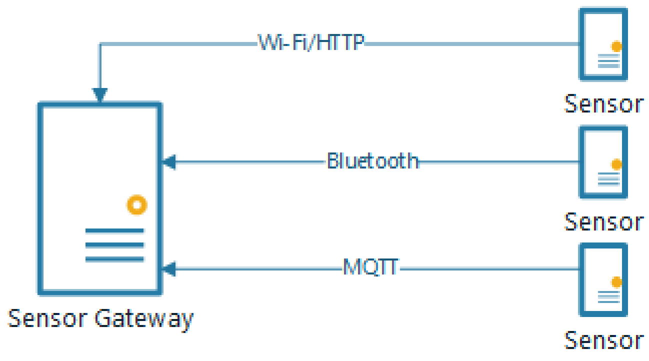

Figure 4.3: One sensor gateway can receive measurements via different communication protocols, which it then feeds to our stream processing system.

can for example receive measurements via HTTP, gRPC and MQTT, so that each sensor can use the technology and the protocol that is most suitable for it. This enables us to have a certain degree of heterogeneity in our farm, as it allows us to connect some sensors to the same wireless network as the gateway and use HTTP as the communication protocol, while another sensor, that is situated in a part of the farm where network coverage is not so good, could utilize WAN technologies such as LoRaWAN or NB-IoT. Next, it also provides a single point of entry into our data stream, which is especially useful when extending our system. This means that developers can either build their own Sensor Gateway, which feeds data into our stream processing network, or extend the given Sensor Gateway to also read data from a protocol that is not yet supported, such as Bluetooth. Figure 4.3 shows an example gateway, that can receive measurements via Bluetooth, HTTP, and MQTT, respectively.

Furthermore, the sensor gateway offers the possibility to configure various settings of the sensors and to raise an alert. Since the communication between the sensor gateway and the various sensor is not necessarily a duplex connection, it may not be possible to propagate the configuration changes towards the sensors. This is a limitation, that is not fully circumventable, as we are limited by the specification of the chosen protocol.

sendMeasurement			
sensor	string	temperature sensor	-
value	number	23.2	decimal values and negative values need to be supported
unit	string	°C	should support full unicode

Table 4.2: Sensor gateway sendMeasurement request specification.

changeMeasurementInterval			
measurementIntervalInMs	number	5000	data should be sent in ms

Table 4.3: Sensor gateway changeMeasurementInterval request specification.

Data Propagation

As mentioned above, data propagation is one of the tasks of the sensor gateway. As such, it is in charge of reading the sensor measurements and publishing them.

The values that are measured and published should contain the fields as shown in Table 4.2.

Configuration Module

The configuration module needs to be able to receive messages sent by the Controller and then change the configuration of the sensor as required. Depending on how the gateway communicates with the sensors this may or may not be possible. At the moment our proof of concept implementation only supports changing the measurement interval of sensors. The request should contain only the fields as shown in Table 4.3.

Further, to receive such change notifications the sensor needs to first register with the Controller. This is done via the interface that is shown in Table 4.4.

Alert module

The alert module is, like the configuration module, dependent on receiving messages from the Controller. When the sensor receives an alert message, it means that an alert should be raised. It is then the sensor module's task to raise an alarm. The sensor module may, however, decide to not raise an alarm, if this sensor is not required to perform any action when an alarm is raised, i.e. does not have any actuators.

4.3.2 Controller

The Controller lives at the heart of the system. It is in charge of propagating data from the user interface to the sensors and vice-versa. This module will be heavily intertwined with the networking protocol chosen, since it, in essence, needs to transform the values it receives into a format and protocol that can be read by the user interface module.

registerSensor			
ip	string	example.com/sensor or 192.0.2.1	either hostname or ip address
port	number	4060	-

Table 4.4: Sensor gateway registerSensor request specification.

There are three sub-modules in the controller: 1. the configuration module, which propagates configuration changes to the sensors, 2. the alarm module, which propagates alarms raised by a decision engine to the sensors and the user interface, and 3. the data propagation module, which sends the data that is received from the sensor gateway to the user interface.

Configuration Module

Starting off, the controller pushes configuration changes it receives from the user interface to the sensors. For this, it expects a request in the same format as the sensors do.

Furthermore, it provides an interface where the sensors can register themselves to receive alarms and configuration changes. The parameters can be found in Table 4.4.

Alarm Module

Next, when an alarm is raised by a Decision Module, it is the controller's responsibility to

1. raise an alarm in the user interface and
2. raise an alarm in the Sensor Modules.

Data Propagation Module

This sub-module is in charge of transforming the data that is received from the sensor gateway to a format and protocol that is understood by the user interface module. Depending on the technologies chosen, this may be a trivial, or not so trivial task. For example, it would be possible that the user interface module is written in a programming stack that can listen directly to the stream processing technology. While we do not recommend this, as it means that the user interface relies heavily on the backend technologies chosen, it is, generally, a possibility.

4.3.3 User Interface Module

This module is in charge of displaying the measured values. It is important that the Module does not require any prior knowledge to the amount of sensors that are used.

For this, the module dynamically adjusts its display to be able to show all the necessary data.

Second, this module also allows the vertical farmer to configure the Sensor Modules. For this, it uses the previously shown interface to notify the controller, which in turn notifies the Sensor Module of the desired configuration changes.

It should be noted that the chosen technology stack for this module, may put restraints on the rest of the entire system, so also the technology choices for the user interface should be made with extensibility and SoA principles in mind.

4.3.4 Decision Module

The decision module subscribes to the same Publish/Subscribe system as the controller and to which the sensors publish. The data that is consumed adheres to the same format as previously shown.

After these values are consumed and the decision module decides that an alarm should be raised, it publishes an alarm in the format as display in Table 4.5.

raiseAlarm			
sensor	string	temperature sensor 1	-
value	number	23.2	-
type	string	one of: max, min, low, high	the type of alarm raised

Table 4.5: raiseAlarm request specification

In general each decision module can also have various interfaces of its own. We leave these to the architects of the various decision engines. In Section 6.5.2 we show the interface of one of our decision engines we built to highlight a possibility where the decision module could take decisions based on domain knowledge that is manually entered by the vertical farmer, thus essentially having a configurable decision engine.

4.3.5 Summary

To summarize, the system has the following data flow directions.

1. Sensors publish data which is consumed by the Controller and the Decision Modules. In other words the sensors only publish the data once, which is then read by multiple consumers.

2. The measurements read by the Controller are pushed to the User Interface.

3. If the Decision Modules deem that an alarm needs to be raised, they push information to the Controller.

4. If an alarm is to be raised, the Controller pushes this information both to the User Interface and to the Sensors.

5. In case of a configuration change, the request is pushed from the User Interface to the Controller and from there it is further pushed to the Sensors.

CHAPTER 5

Network Topologies and Deployment Strategies

In this chapter we will provide different deployment scenarios and network topologies, in which our system could be run. We will also highlight key advantages and disadvantages of the various deployments. In total we have identified four main topologies which make sense for our application.

It is worth noting, that the user interface requires a bit more explanation. In general the user interface needs to be available somewhere. This user interface (HTML + JavaScript) is then downloaded to the device which is used to view it, where it is then executed. The end-device then communicates with the server to receive and send data. For this, the end-device needs to know the location (host name or IP address) of the server. This location is hardcoded in the user interface, which has been downloaded to the end-device. So, while the user-interface always **runs** on the end-device, it is **hosted** somewhere else, and depending on from where one needs to access the server, the user interface also needs to be accessible. We will therefore always display the user interface as hosted in the cloud if at least one other component is cloud hosted, and hosted inside the farm if no other component is cloud hosted.

5.1 On-Site Hosted Local Area Network

In a small-scale farm that has good power supply possibilities, the system can utilize Wi-Fi internally and rely on the power supply that is available. That is to say, the communication between the various parts of the system is done over Wi-Fi. In our architecture this means that the user interface, the controller, decision modules and the sensor modules communicate over Wi-Fi. The communication between the sensors and the sensor gateway can be done over Bluetooth, Wi-Fi or a wired connection.

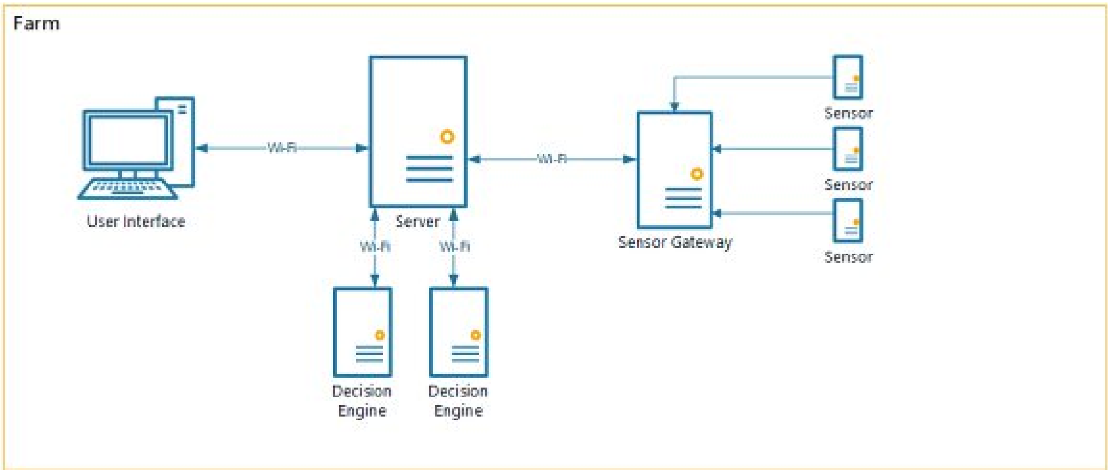

Figure 5.1: On-site hosted local area network deployment scenario.

5.1.1 Advantages

Ease of setup This setup can be done by a vertical farmer with little to no technical expertise. The communication channels can be hardcoded, and there is little setup to be done.

Ease of configuration Depending on how much of the stack the farmer wants to use, there will also be little to no configuration needed. This will allow the vertical farmer to start monitoring his or her farm without spending a lot of time on the initial setup.

Security Security will not prove a big issue on the application level, since the system does not expose any data to the public Internet. Standard security measures, such as securing the wireless network and using standard encryption levels should, of course, be applied.

No reliance on external entities As mentioned above the setup for this scenario is minimal. Additionally, there is no reliance on external entities, such as cloud providers. This also means that no cloud hosting fees apply.

Disconnected operation The fact that the platform can (with limitations) allow monitoring, to an extent, without reliance on an Internet connection, is an obvious advantage of this deployment scenario. We highlight various disadvantages connected to this in the following subsection.

5.1.2 Disadvantages

No remote monitoring Due to the fact that this setup is not connected to the public Internet, the farmer has to be on-site to monitor the farm. This, of course, means, that

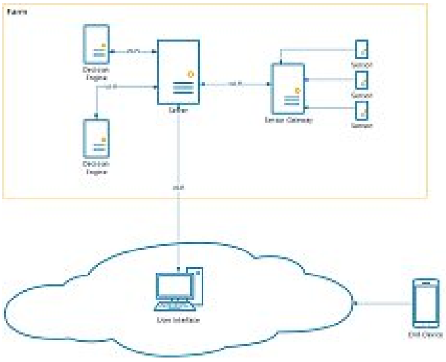

Figure 5.2: LAN exposed to public Internet deployment scenario

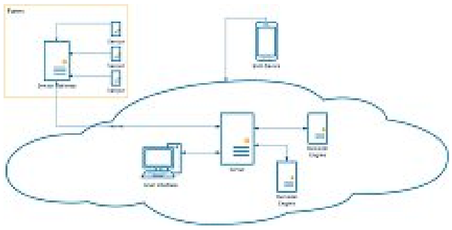

Figure 5.3: Gateway based LAN exposed to public Internet deployment scenario

should an alarm be raised when the farmer is not on-site, he or she will not be notified of it.

Dependent on existing infrastructure However, this setup requires pre-existing infrastructure, such as a wireless network, and power supply possibilities spread throughout the entire farm. This is needed, since the communication relies on a wireless network. So therefore, the wireless network needs to exist, and depending on the size of the farm, it may also be required to have multiple wireless routers spread around the farm. Additionally, the sensors require a power supply, that also needs to be present. Wi-Fi comes with more power consumption than LPWAN-based alternatives, which limits the possibilities of operating on a battery.

5.2 LAN exposed to public Internet

This scenario is the same as the above one, with the difference that the data can be monitored from outside of the local area network. This is helpful when the farmer wants to monitor their farm remotely. This can be done in two ways:

1. The network has a public IP-address that is exposed, over which the user-interface can communicate with the controller directly or

2. there exists a gateway between the sensors and the controller, and the latter, as well as potential decision engines are deployed to the cloud.

The gateway then only propagates the measurements to the controller, which in turn can process them. Depending on the technologies used, as well as the setup in the farm, the gateway may also be omitted.

5.2.1 Advantages

Remote monitoring Due to the fact that the data is exposed to the public Internet, it now allows the farmer to monitor it remotely. This is of course useful, since it no longer means that the farmer has to be physically present in the farm to monitor it.

5.2.2 Disadvantages

Setup difficult The added functionality, unfortunately, also brings added complexity. Specifically the communication channel between the parts of the platform that are running in the farm and the parts that are running somewhere else, e.g. the cloud, needs to be configured. This task requires a certain amount of expertise and is best performed by an IT-professional.

Dependence on existing infrastructure Again, the sensors require some sort of power supply to allow them to function. Additionally, depending on the exact setup chosen in the farm, there will also be some kind of wireless network required, together with an active Internet connection, that allows the data to be sent to the monitoring device. Like before, the dependency on Wi-Fi limits the possibilities of operating on battery.

Security Opening up the platform to the Internet brings security concerns. Specifically, the data now needs to be protected in a way that prevents potential attackers from gaining access. This can no longer be done only on a physical level, but needs to be done on application level as well. Depending on the exact setup and technologies chosen, there may be multiple candidate solutions for securing the application. Exploring these is not in the scope of this work.

5.3 Long Range wireless technologies

If the farm is in a remote location it may be useful to utilize long range wireless technologies such as LoRa. This would allow us to put the monitoring system into the cloud, and reduces infrastructure requirements in the farm. This scenario would mean that the sensors use LoRa to send their measurements into our system, where they are processed as required. Most of these communication technologies need very little power supply and can run for months or years at a time with a single battery. A key point is obviously that no longer is the farm reliant on pre-existing infrastructure such as a network or power supply.

Since this scenario has some similarities, but also some key differences to the previous ones we want to expand on the explanation of this scenario and compare it with a Wi-Fi based solution. For this we will compare LoRaWAN to a standard Wi-Fi based solution. Most importantly, while operating in unlicensed bands (Industrial, Scientific, Medical - ISM), LoRa is subject to *duty cycling* regulations. One is required by law to not utilize the LoRa bandwidth continuously. Instead, there is a requirement such as of 90% sleep time and 10% transmission time[1]. What this means, is that only 10% of the time the bandwidth can actually be used. For us, this means that we have a limit on how many measurements we can send in a given timeframe. This, of course, means that we have to be careful with how many measurements we send. While the time it takes for our measurements to be sent is quite low, it is still something we have to be wary of. Uplink latency for LoRa based solutions is definitely higher than that of Wi-Fi based solutions, but for our use-case we are not dependent on near-instant communication and latency in the range of a few seconds is sufficient. When requiring downlink communication in LoRa based solutions, it is not so trivial and there are some considerations to be taken.[2,3]

One key example for downlink messages in our system is the change of configuration values, such as the measurement interval. For this, the LoRa Class A will be enough, and also a good choice with regards to the low battery requirements. Class A LoRa means that for a short interval of time after an uplink message, the device opens a receive window. Specifically, 1 s and 2 s after an uplink, the device is ready to receive messages from the server. In this communication window the device will be ready to receive the communication changes. However, this also means that the latency for configuration changes is determined by the uplink interval, i.e. if the uplink interval is 5 minutes, it may take up to 5 minutes for the configuration change to take place. As can be seen in Figure 5.4 after a transmission from the device to the server, it opens up two receive windows. The server can then send the downlink configuration change in the window as shown.

This is in contrast to Wi-Fi solutions, where both directions of the communication are always open.

5.3.1 Advantages

Allows remote monitoring Also in this deployment scenario, the farmer can monitor his farm remotely. In fact, even the modules that receive the sensor readings can be situated remotely, since the communication between sensor devices and the processing system is no longer bound by a LAN network's range.

Not reliant on pre-existing infrastructure There is no longer a need for a pre-existing wireless network infrastructure on-site, or power supply. However, in this scenario every communication adapter has to be bought or rented, dependent on the chosen long range communication protocol.

[1] The time on air restrictions may vary per country and frequency band used.
[2] https://www.thethingsnetwork.org/docs/lorawan/limitations.html
[3] https://www.thethingsnetwork.org/docs/lorawan/classes.html

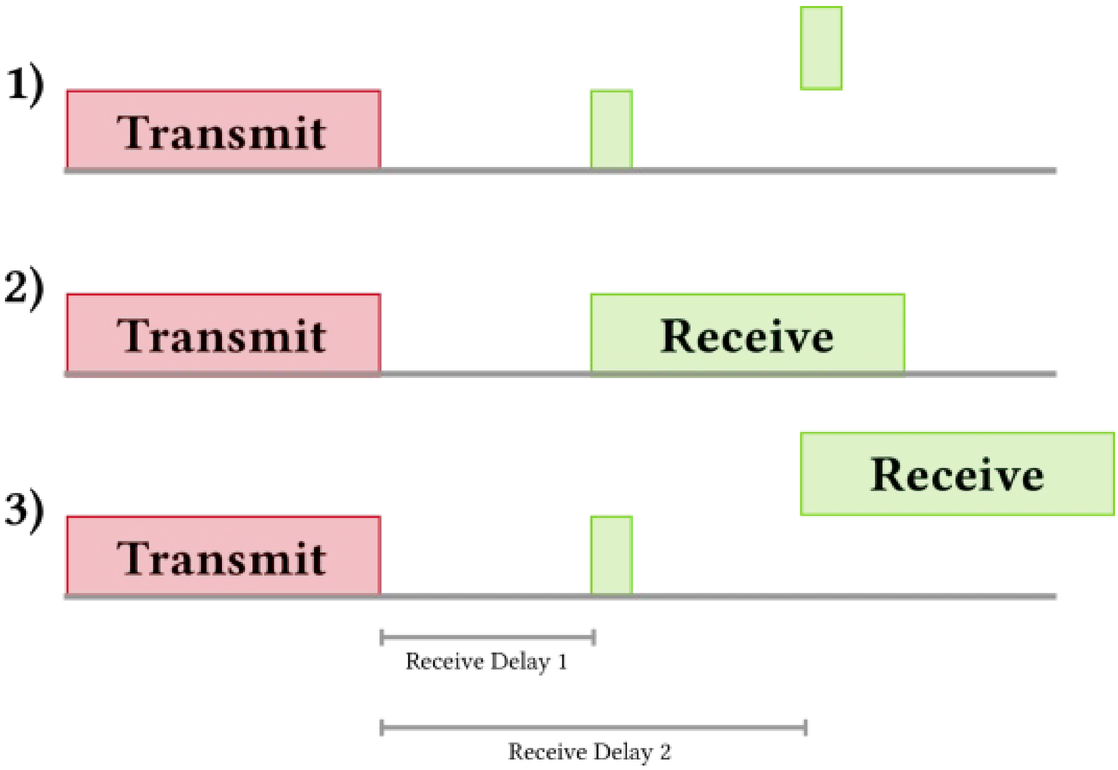

Figure 5.4: LoRa transmit and receive window

5.3.2 Disadvantages

Setup difficult Depending on how customized the farmer wants his or her farm to be, a lot of setup may be required. Specifically, the communication channels need to be configured by an IT-professional.

Security Depending on which long range transmission technology is chosen, different concerns regarding security arise. It is important to note that wireless technologies are not inherently insecure but implementation as well as the hardware used matters when trying to secure such a network.

5.4 Computation intensive decision modules

If the decision modules required to properly monitor a given farm are computationally expensive, it may make sense to utilize the strong scalability of the cloud, as well as the cheap processing power there. This scenario can again be split up to allow the monitoring to occur remotely or not and also allows the farmer to decide how the communication between Controller and User Interface occurs.

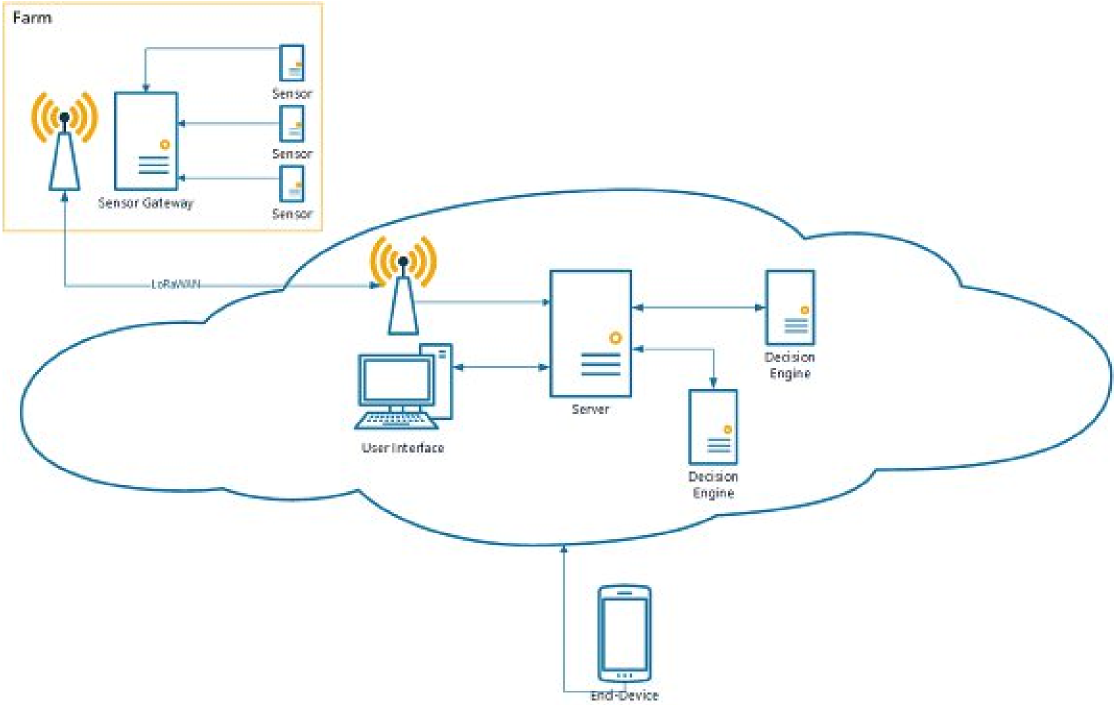

Figure 5.5: Long Range wireless technologies deployment scenario.

5.4.1 Advantages

Possibility to remotely monitor The farmer will be able to monitor the farm remotely. The exact details depend on the specific deployment strategy chosen, but in any case it will be possible for a farmer to have access to the data that is measured remotely.

Utilize computational power of cloud If the decision engines require a lot of computational power, as the processing that is performed on the data is computationally expensive, the cheap and near-endless processing power of cloud computing providers can be utilized. This will allow the farmer to cost-effectively perform calculations that are not feasible on low cost end devices. Additionally, this relieves the farmer of potential overhead of maintaining edge computing equipment for computational tasks, resulting in less single board computers on-site.

5.4.2 Disadvantages

Setup difficult Obviously with the increased complextiy in the communication channels, also the setup difficulty rises. Also for this scenario, it is doubtful whether a farmer can perform the initial setup on his or her own.

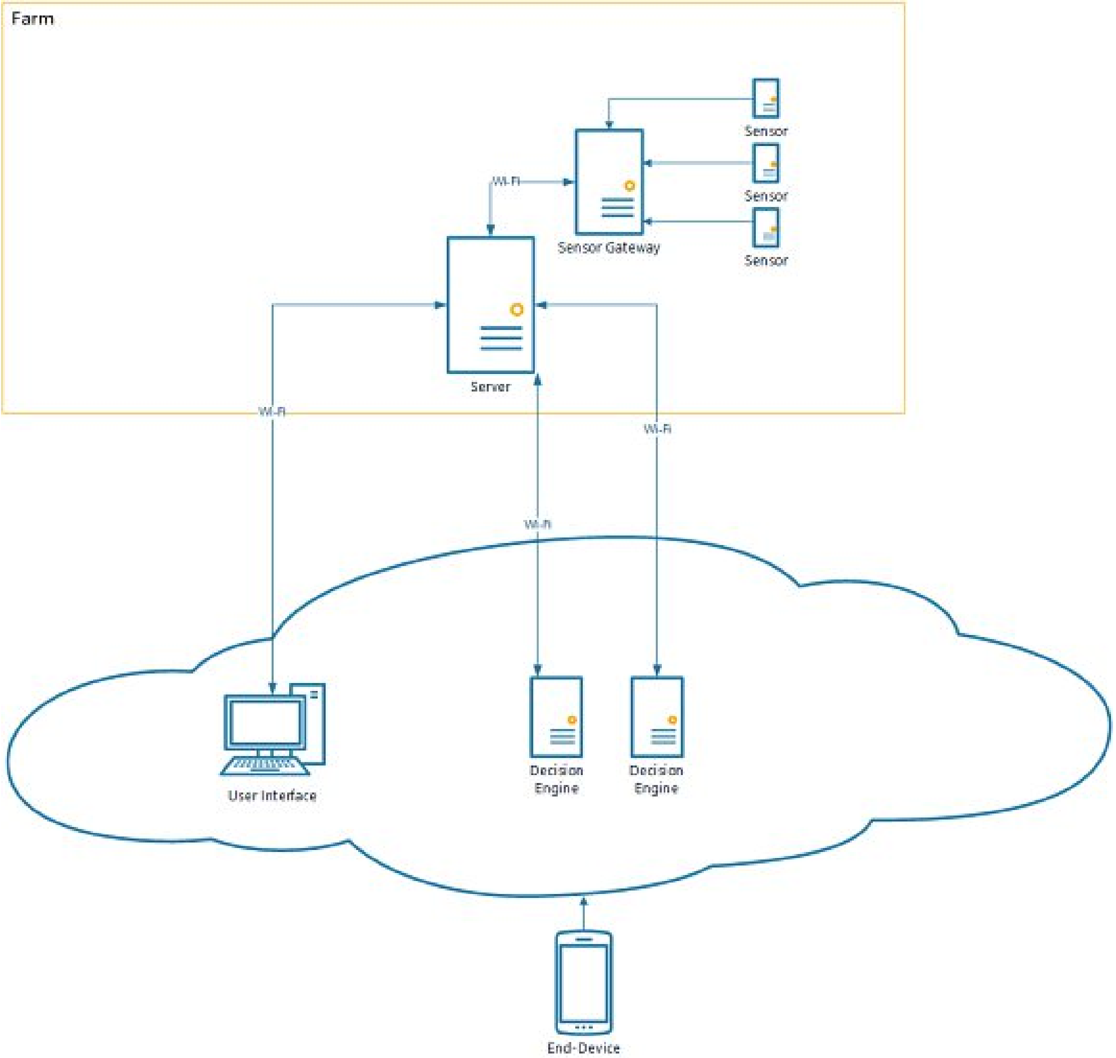

Figure 5.6: Computation intensive decision modules deployment scenario.

Security The security considerations from 5.2.2 apply here.

5.5 Comparison

In this section we will compare the various topologies/scenarios we have explained under the chapter. We rate the various scenarios in the following five categories on a five-point scale, where, unless otherwise noted, 1 represents the poorest performance, and 5 represents an excellent one. Table 5.1 shows our grading.

Ease of setup How easy the setup and configuration of the deployment scenario is. This takes into account how much configuration is necessary, if any cloud hosting options are required, and if we believe that a layperson could perform an initial setup of the platform.

Infrastructure dependence How dependent upon infrastructure the solution is. This takes into account power supply, and a wireless network connection, as well as the possibility to connect a potential cloud service to the system. We do not take communication technologies into account that require a special type of communication model. This means that for example, the dependency upon a Bluetooth or LoRa receiver/transmitter does not mean a worse grade.

Security risks How hard will it be to secure the system against malicious access, malicious generation of readings or general interference. A lower grade means that it is more difficult to secure the system, either through the sheer amount of attack vectors, or the fact that technologies have to be employed that are inherently difficult to secure.

Remote monitoring capabilities If it is possible / easy to remotely monitor the farm. What we mean by this is how the farmer can monitor his or her farm. We chose not to rate this as a simple "yes/no" category, as we believe that there exists a difference between the various types of monitoring capabilities, which we will expand upon later.

Maximum computation power How high the computation power can become in the specific scenario. While it is clear that it will depend on the devices used, specifically in the scenarios that have a big part of the system running physically in the farm, we still grade the scenarios based on the maximum computation power that can be achieved with a reasonable cost.

We will now elaborate on the reason for our grading and explain why we have given the grades as we did.

It is clear that deploying the system only in a private LAN requires the least amount of setup. The reason for the low grade in the Long Range wireless scenario depends entirely upon the fact, that often times these technologies require extra setup such as gateways, which need to be installed and configured by an IT-professional.

However, the fact that these technologies usually do not rely on a dedicated power supply, but can rely on batteries for quite a long time gives them a low dependency on existing infrastructure. Cloud solutions on the other hand require connectivity towards a cloud provider.

The reason we have given all scenarios, other than the private LAN one, a security risk rating of 1 is, that the amount of attack vectors is quite large: the wireless network in the farm, the communication between the farm and the system in the cloud, application level security such as passwords / key-based security schemes. This does not mean that

Deployment Scenario	Ease of Setup	Infrastructure dependent	Security risks	Remote monitoring capabilities	Maximum computation power
Private LAN	⋆ ⋆ ⋆ ⋆ ⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆ ⋆		⋆ ⋆
Publicly exposed LAN	⋆ ⋆ ⋆ ⋆	⋆ ⋆	⋆	⋆ ⋆ ⋆ ⋆	⋆ ⋆ ⋆
Long Range wireless technologies	⋆ ⋆	⋆ ⋆ ⋆ ⋆	⋆	⋆ ⋆ ⋆ ⋆ ⋆	⋆ ⋆ ⋆ ⋆
Cloud-based decision engines	⋆ ⋆ ⋆	⋆	⋆	⋆ ⋆ ⋆ ⋆	⋆ ⋆ ⋆ ⋆ ⋆

Table 5.1: Grading of Network Topologies

securing such a system is impossible, just that it will require more effort than securing a system that only runs in a private network.

The reason for rating this section not as a simple "yes/no" category is that we feel that it would not give credit to the fact that Long Range wireless technologies really allow a farmer to have access to data that is not only generated some place else, but that multiple physical locations can feed into one single system. In other words, there does not need to be a physical connection between the various sensors that in the end make up one "farm". Instead, one "farm" can be made up of various smaller "farms", which in turn would reduce management overhead and reduce setup cost. Therefore we have decided to grade this scenario higher than those that depend on physical connectivity of one farm.

The computation power of systems running in the cloud is near endless, but since the focus for the Long Range wireless scenario is not upon this, we grade it one grade lower, as it is not entirely clear that the processing is performed on a cloud level and the focus is on long range data delivery. The private LAN and publicly exposed LAN scenarios, on the other hand, are limited mostly by cost factors. With unlimited monetary resources, (almost) unlimited computation power could also be achieved for these scenarios. However, we believe that this is not realistic. Instead, it is more realistic to use single board computers or stronger desktop computers for this.

CHAPTER 6

Implementation

Our proof of concept system is comprised of various standalone projects. In this chapter we will present the various components of our system as well as highlight the communication strategies between components. Figure 6.1 shows our components and the network protocols we use. We have performed various experiments on our system, specifically in regards to single-board-computers, which can be found in Chapter 7.

6.1 Communication Mechanism

To enable the communication of the various loosely-coupled components of our design, there are several application-layer protocols that we have considered. We have identified the following networking protocols which fit some of our criteria to implement specific communication tasks: Apache Kafka[1], WebSockets[2,3], HTTP, gRPC[4,5], MQTT[6]. Which protocol is suitable for which use-case is shown in Table 6.1.

6.1.1 Apache Kafka

Apache Kafka is an event streaming platform, that uses a publish/subscribe model to send events from data producers to a set of data consumers interested in the data. Events are assigned to a topic, to which the consumers can subscribe to receive the data that was published for a given topic.

[1] https://kafka.apache.org/
[2] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[3] https://tools.ietf.org/html/rfc6455
[4] https://grpc.io/
[5] https://grpc.io/blog/pubsub/
[6] https://mqtt.org/

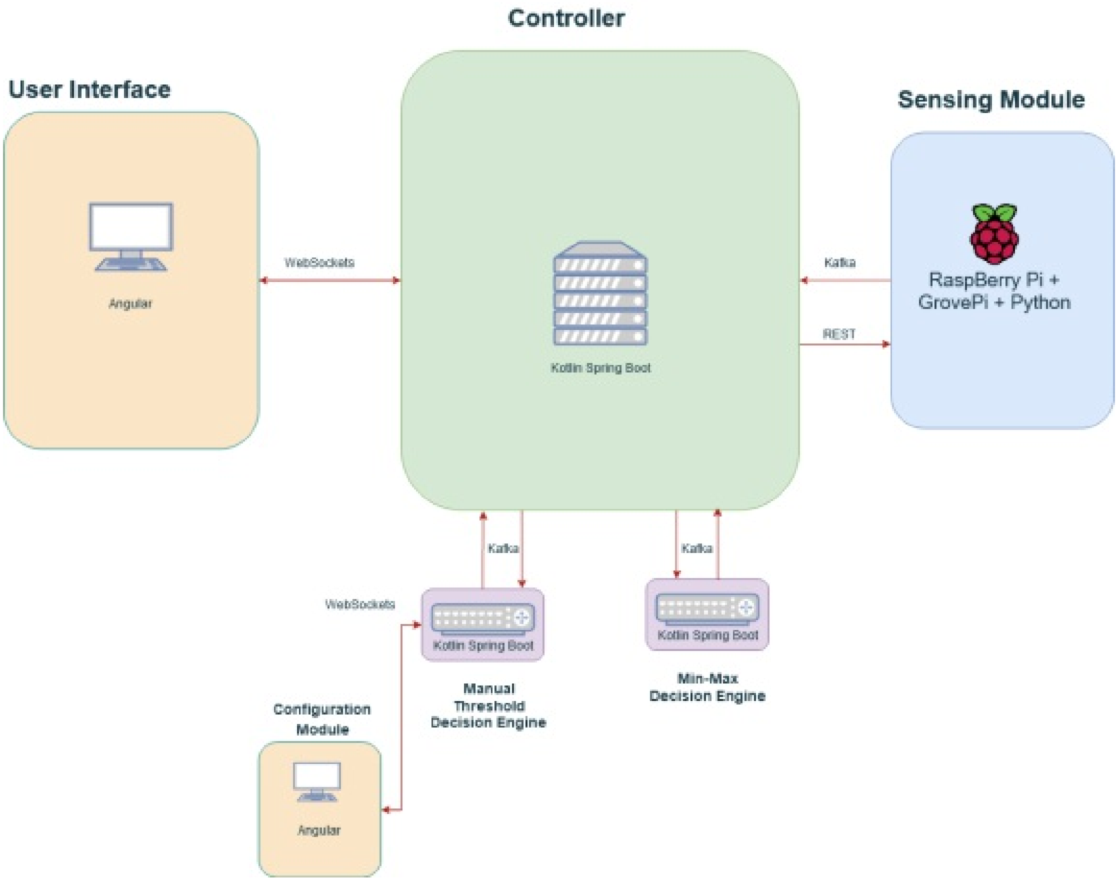

Figure 6.1: Implementation and selected protocols of our proof of concept implementation.

	Apache Kafka	WebSockets	HTTP	gRPC	MQTT
Push to Client Devices		✓			
Request/Response to Server devices			✓	✓	
Pub/Sub	✓	✓		✓	✓

Table 6.1: Network protocol considerations, where a checkmark implies that a given protocol satisfies the requirements.

Zookeeper

Kafka needs a mechanism to coordinate between the different nodes in a potentially distributed Kafka cluster, to configure various topics, to monitor access control, and for controller election. With the current version 2.6.0 of Kafka, Zookeeper is still required to run Kafka. However, there is a recent development effort to remove this dependency and use a Self-Managed Metadata Quorum[7]. Note that we will be using Zookeeper for our system and our tests in this work, but will omit mentioning it explicity from here on. When we talk about the "Kafka Stack", we are talking about the Apache Kafka Server and Zookeeper from this point onwards.

"ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services."[Apa]

6.1.2 WebSockets

WebSockets can be used to synchronously send push notifications to a browser. This is often used in applications featuring (or requiring) real-time interaction, such as chat applications, or tools that allow people to collaborate on a piece of work. For us, WebSockets could be used to push the measurements that are being taken towards the user interface, and also to push potential alarms to the user interface.

6.1.3 HTTP

In our case it would make sense to use HTTP requests in the communication, that is either initiated by the user interface, or a request in the back end of our system. One such request for which HTTP would make sense as a technology choice is the change of configuration parameters. This is initiated by the user interface, and then propagated to the sensing module by the controller.

6.1.4 gRPC

gRPC was initially developed by Google, but is now Open-Sourced under the Apache License. It is a method of communication that is similar to HTTP, but requests are not human readable and are utilizing a binary message format called Protobuf, resulting in smaller payload sizes. While there exists gRPC for the browser, it currently does not support pushing data towards browsers. In other words, for us it can only be used instead of HTTP, but not instead of WebSockets, since we have some functionality that requires us to send data to the browser. It could, however, be used for communication between some of the backend services, such as the sensing module and the controller.

6.1.5 MQTT

The MQTT protocol is a lightweight publish/subscribe protocol that, similar to Apache Kafka, allows us to open various communication channels, to which clients can subscribe and data producers can publish data. In comparison to Kafka, MQTT is much more lightweight, but does not provide scalability on the same level as Apache Kafka does. MQTT is tailored to IoT applications, and is particularly suitable for small IoT devices due to its minimal resource requirements as well as code and memory footprint.

[7] `https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum`

6.1.6 Selected Protocols

Table 6.2 shows the functionality, as well as the protocol we have decided to use for communication between the different modules.

Functionality	**Chosen Protocol**
Sensors publish data that is read by multiple consumers	Apache Kafka
Controller pushes measurements to User Interface	WebSocket
Decision Modules push alarms to Controller	Apache Kafka
Controller pushes alarms to User Interface	WebSocket
Controller pushes alarms to Sensors	HTTP
User Interface pushes configuration changes to Controller	WebSocket
Controller propagates configuration changes to Sensors	HTTP

Table 6.2: Selected protocol for a given communication type.

The reason we have decided to rely on Apache Kafka for our Pub/Sub requirements is that it is backed by a strong corporation, a strong community and is widely used. This will 1. allow developers to find many resources, should they wish to extend the system and 2. allow us to have good opportunities for support from the community. Additionally, it allows us to have a strong stream processing engine at the core of our system, that is easily scalable. This is also useful, as we can have a multitude of decision engines which need to interface with our stream processing engine.

The choice for WebSockets was also clear, as we needed to easily push the measurements towards the user interface, i.e. receive push notifications in the browser.

Last, we chose HTTP as it is the natural choice for REST-style communication, as is the case for the functions we use it for. Additionally, HTTP is easy to use, test and debug.

6.2 Sensing module

Our sensing module is a python project that is run on a Raspberry Pi and can interface with various sensors. Once a measurement has been taken or received it is published to the Kafka topic `vf-sensor-topic`. To publish, the sensing module utilizes the `kafka-python`[8] python library.

The sensors used in our system all register with their IP address to an HTTP endpoint of the controller, so that they are notified of alerts and configuration changes. The endpoint in question is `/sensor/register`. To receive any notifications - either configuration changes or alerts - the sensors must register their IP address if they want to receive information such as alerts, or configuration changes.

[8] `https://github.com/dpkp/kafka-python`

We show two different types of sensor communication in our proof of concept system. First, we show one that uses GrovePi[9] sensors and one that builds upon libelium[10] devices and communicates over LoRaWAN.

6.2.1 GrovePi

We implement one solution in which the sensor is bundled on the same hardware as the sensor gateway. For this we use GrovePi sensors. Utilizing the GrovePi sensors we can also raise alarms by using an actuator, such as a buzzer or a light, to notify the farmer if an alarm is raised.

6.2.2 LoRaWAN-based sensor module

To illustrate the flexibility of our system, we also implement a solution that is based on LoRaWAN. For a discussion on the functionalities and limitations of LoRaWAN, we refer the reader to Sections 3.4 and 5.3, respectively.

Our proof of concept implementation uses a Libelium Plug & Sense! Smart Water device[11] to measure air temperature. This reading is then transmitted over LoRaWAN to a gateway hosted by a Raspberry Pi. The full LoRaWAN network server stack is hosted on the same device. For this purpose, we are using ChirpStack, an open-source LoRaWAN network server stack[12]. Let us first give an overview of the ChirpStack, and how we can utilize it to provide a communication between our Sensor Gateway and the temperature sensor. Figure 6.2 shows the ChirpStack architecture and the interfaces and protocols used for communication.

Essentially, LoRaWAN devices which are not pictured in Figure 6.2, send broadcast data over the LoRa physical layer, which are received by LoRaWAN gateways. This Gateway, then forwards the data to the Gateway Bridge, which in turn uses MQTT to transfer the data to a Pub/Sub broker and consequently to the Network Server. The Network Server uses gRPC to send the data to the Application Server, in which we can add various "Integrations", which are another form of forwarding, that allows us to easily test our setup and propagate measurements to our system.

LoRaWAN Gateway

The LoRaWAN gateway runs the Packet Forwarder, which is responsible for receiving and sending. Two common implementations are the Semtech UDP Packet Forwarder[13] and the Semtech Basic Station Packet Forwarder[14].

[9] https://www.dexterindustries.com/grovepi/
[10] https://www.libelium.com/
[11] https://development.Libelium.com/smart_water_sensor_guide/
[12] https://www.chirpstack.io/
[13] https://github.com/Lora-net/packet_forwarder/
[14] https://doc.sm.tc/station/

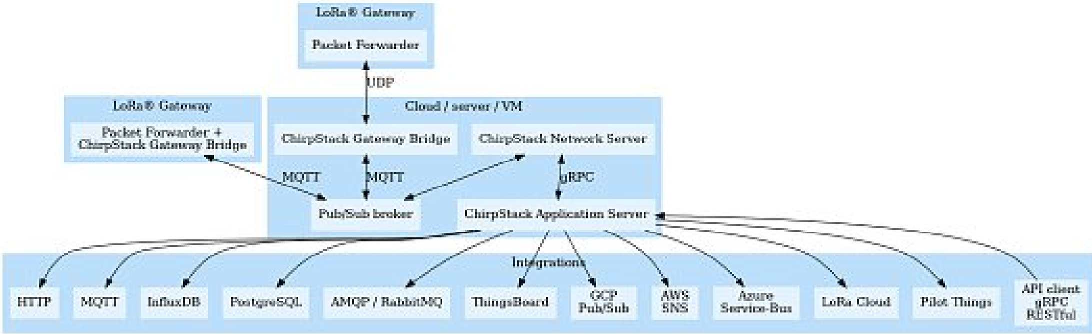

Figure 6.2: ChirpStack implementation: the Gateway forwards packets to the Gateway Bridge, which in turn forwards them to a Pub/Sub broker over MQTT. The ChirpStack Network Server then uses gRPC to send the measurements to the Application Server, where we can use Integrations to feed the measurements into our system. Specifically, we will use an HTTP integration. [Chi]

Gateway Bridge

The Gateway Bridge is located between the Gateway and the MQTT broker. It transforms the data received from the Gateway into a data-format that is used by the rest of the ChirpStack.

Network Server

The Network Server is responsible for managing the state of the network. It is also in charge of handling join-requests, when devices want to join the network. When the network topology calls for multiple gateways to be used, the network server is in charge of de-duplication and forwards data to the ChirpStack Application Server. When downlink communication is necessary, the Network Server is also in charge of performing the downlink communication towards the Gateway(s).

Application Server

The Application Server provides a web interface as well as an API that allows for management of the stack. Furthermore, it provides the interface to configure Integrations, which can be seen in Figure 6.3, with which the data can be forwarded to an end-application - such as our system.

Downlink

The ChirpStack implementation of a LoRaWAN application server exposes interfaces to enqueue downlink messages in a straightforward way. For example, we can use the UI of the Application Server to enqueue downlink payload, but also MQTT, REST and other

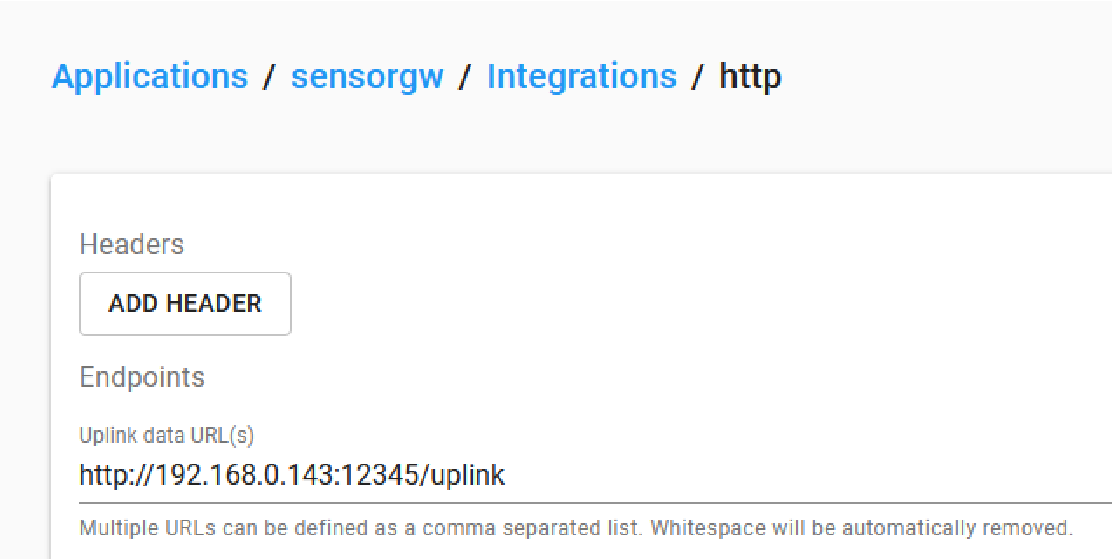

Figure 6.3: The ChirpStack Application Server allows us to define so-called Integrations, with which we can forward data that is received from the LoRa Gateway.

endpoints are provided. The required parameters are the port, as well as the message to send encoded in Base64.

The Libelium device now allows us to receive data that contains the payload we previously enqueued in HEX representation. Converting this to ASCII is trivial, and shows that we can easily send downlink messages as well.

However, for use during deployment, this is obviously not suitable. We will therefore show that it is also possible to use the ChirpStack Application Servers API and enqueue a downlink message programmatically. For this, refer to the API as can be found in `http://application_server_ip:application_server_port/api`. First, we require a JWT token, that can be received by using the `/api/internal/login` endpoint. Next, we can POST the data we want to enqueue - again in Base 64 - to the `/api/devices/{device_queue_item.dev_eui}/queue` endpoint. We can then read the value we received, and convert it to ASCII, and then sleep for the desired new amount, rather than the previously set default value.

This shows that we can also change the measurement interval when our sensor communicates over LoRa, and that we can use the LoRa communication to send downlink messages programmatically, during runtime.

We show the results of our experiment, in which we measure the round trip time it takes for a LoRa measurement to be transmitted between the Libelium device, our LoRa Application Server and a potential downlink back to the Libelium device again in Section 7.6.1.

6.3 Controller

Our Controller is a Spring Boot[15] Kotlin[16] project which consumes the previously mentioned `vf-sensor-topic` Kafka topic. This is done via the official Kafka Consumer Java API[17]. This data is forwarded to the user interface over a WebSocket topic called `/message/sensor-reading`.

The Controller also provides an interface to change the measurement interval of the Sensing module. The WebSocket topic used for this is called `/config/config/change`. For this it expects a request in the format as shown in Table 6.3.

changeMeasurementInterval			
measurementIntervalInMs	number	5000	data should be sent in ms

Table 6.3: Controller changeMeasurementInterval request specification.

On top of this it is also responsible for alert handling. If an alert is raised via the `vf-alert-topic` the controller propagates this information to the user interface over another topic: `/message/alert` and it also notifies the sensors that an alarm is to be raised via the REST interface explained in Subsection 4.3.1.

The controller also provides an HTTP endpoint `/sensor/register` where sensors can register themselves. A sensor needs to register itself so that changes of the `measurementIntervalInMs` parameter can be propagated to it.

6.4 User Interface

The user interface is an Angular[18] project that displays the value received over the `/message/sensor-reading` WebSocket topic. It uses the `ngx-charts`[19] package to display the sensor measurements. It can also be used by the vertical farmer to configure the sensors, over the `/config/config/change` WebSocket topic.

For this it sends an object in the JSON format as shown in Table 6.4.

changeMeasurementInterval			
measurementIntervalInMs	number	5000	data should be sent in ms

Table 6.4: User Interface changeMeasurementInterval request specification.

[15] https://spring.io/projects/spring-boot
[16] https://kotlinlang.org/
[17] https://kafka.apache.org/documentation/#consumerapi
[18] https://angular.io/
[19] `https://github.com/swimlane/ngx-charts`

changeThreshold			
sensor	string	temperature sensor 1	the sensor for which the threshold should be changed
upperThreshold	number	120	-
lowerThreshold	number	80	-

Table 6.5: Manual Threshold Decision Engine changeThreshold request specification.

6.5 Decision modules

For our proof of concept we have developed two decision modules. One of them raises an alarm whenever a new maximum or a new minimum is reached for any given sensor. The other one allows users with domain knowledge (in our case the vertical farmer) to manually set threshold values over (or under) which the measurement should not be.

6.5.1 Min-Max Alarm Decision Engine

This decision engine is another Spring Boot Kotlin project which consumes the Kafka topic to which the sensors publish and keeps track of the current min/max values for each sensor. If a new value is measured, an alarm is raised to the `vf-alert-topic`.

6.5.2 Manual Threshold Decision Engine

This engine, as previously mentioned, allows the vertical farmer to set manual thresholds for all sensors. This allows a farmer with domain knowledge to set critical values. For example, the farmer could set a threshold for the water temperature sensor that it should not be above 20 °C, as that would promote algae growth. The module has a WebSocket connection to the frontend, over which the frontend is notified if a new sensor has been detected. The name of the WebSocket topic is `/config/sensor`. Over the same connection the server receives threshold values. The expected request is in JSON format with the schema as shown in Table 6.5.

Configuration Interface

The user interface for the threshold based decision engine is an Angular project. It communicates via WebSockets with the decision module over the `/config/sensor` topic and receives current sensor information from the module. The format for the given JSON request is shown in Table 6.6.

It is worth noting that the sensor and unit variables are required only for displaying purposes.

getCurrentThresholds			
sensor	string	temperature sensor 1	only used for display purposes
upperThreshold	number	30	-
lowerThreshold	number	18	-
unit	string	°C	only used for display purposes

Table 6.6: Manual Threshold Decision Engine Configuration Interface getCurrentThresholds request specification.

CHAPTER 7

Evaluation

In this chapter we will benchmark the various components of our system and perform an analysis of how big of a farm a farmer can monitor with our system. This is important, as it will allow us to deduce what exactly we can expect performance wise from our single board computers. It will also allow us to draw conclusions how many devices we need to be able to handle realistic monitoring workload, and will allow us to generate a cost estimate.

Specifically, we will analyze the following seven test scenarios.

1. Latency measurements when Kafka is running on a strong device, such as a PC.
2. Latency measurements when Kafka is running on a single Raspberry Pi.
3. The maximum amount of measurements that one Raspberry Pi can handle running a Kafka Server and a Consumer.
4. Whether a Raspberry Pi can handle our entire platform, running both Kafka and our system.
5. The maximum amount of data that a single Raspberry Pi can produce.
6. Latency measurements for a LoRaWAN device.
7. The maximum throughput we can achieve with one ChirpStack Application Server running on a Raspberry Pi.

7.1 Hardware

The hardware we have used for our tests is shown in Table 7.1.

Raspberry Pi	Raspberry Pi 2 Model B 900MHz quad-core ARM Cortex-A7 CPU 1GB RAM
PC	AMD Ryzen 5 3600X 6x 3.80GHz 32GB RAM

Table 7.1: Hardware specification used in tests.

7.2 Latency

In this section we will present the results of the latency benchmarks. Specifically, we show the time it takes for a measurement that is emitted by a device to be processed by Apache Kafka (running on a different device) and then read by the same device again. For this test we perform four different case studies. For all of them we emitted 500 measurements and received all 500 measurements again - that is to say that no measurement was lost, even though there were some connectivity issues for the tests done over Wi-Fi.

The four test cases are as follows.

1. Kafka on PC, Consumer on RPi, Producer on RPi, connected over Wi-Fi.
2. Kafka on PC, Consumer on RPi, Producer on RPi, connected over Ethernet.
3. Kafka on RPi, Consumer on PC, Producer on PC, connected over Wi-Fi.
4. Kafka on RPi, Consumer on PC, Producer on PC, connected over Ethernet.

These tests will allow us to deduce if the time that it takes for one measurement to go between data generation and reading it is low enough to support the monitoring activities that we require.

7.2.1 Kafka on PC, Producer + Consumer on RPi

For this test the PC was running the Kafka Server, while the Raspberry Pi was running a Producer script, as well as the Consumer. We emitted a total of 500 measurements on the Producer and received all 500 measurements on the Consumer, so no measurement was lost.

Wi-Fi

Min 16ms, Max 2902ms, Median 29ms, Average 58.796ms

Interestingly enough, the time that it takes for a measurement to be emitted and read is, on average, very low. It is worth noting that the three measurements towards the right side of the diagram are directly following each other. This is probably caused by a temporary Wi-Fi connectivity issue, as the rest of the measurements, and also the average and median are significantly lower. All readings can be found in Figure 7.1.

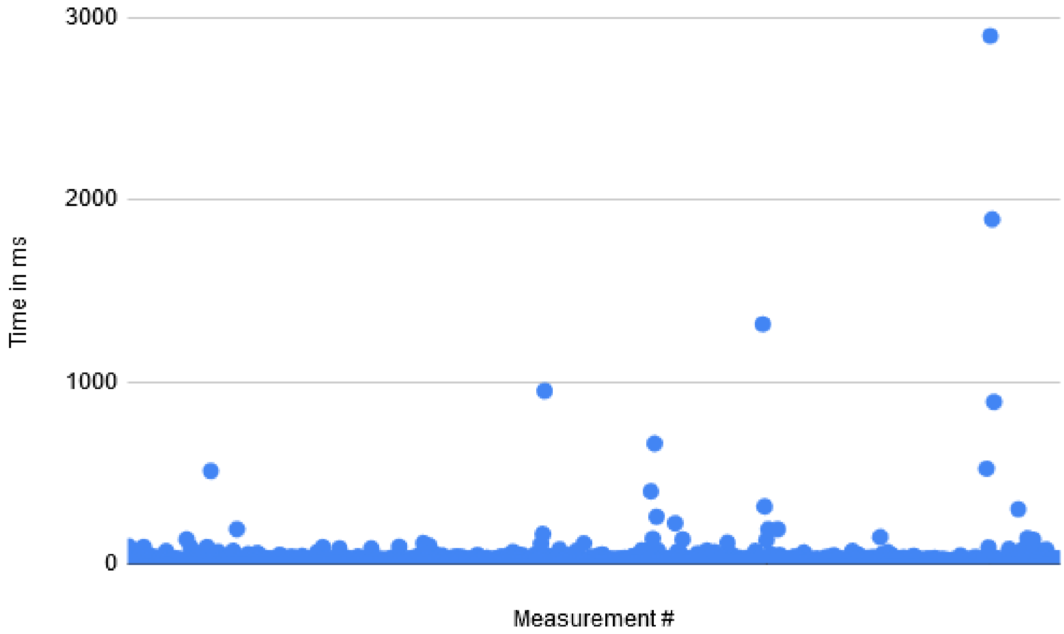

Figure 7.1: Latency per measurement when Kafka is deployed on a PC and a Producer and Consumer are running on a Raspberry Pi, Wi-Fi connected.

Wired

Min 14ms, Max 76ms, Median 20ms, Average 19.582ms

As expected the wired connection performs better than the wireless one. It is especially interesting to note that the first measurement is the slowest one. Furthermore, all other measurements are in the range of 14-27ms. This, together with the data from the wireless experiment, lead us to believe that the processing time required by Kafka is almost constant, and the various outliers from the previous experiment stem from wireless connection issues, rather than Kafka being slow. All readings can be found in Figure 7.2

7.2.2 Kafka on RPi, Producer + Consumer on PC

For this test the Raspberry Pi was running the Kafka Server, while the Raspberry Pi was running a Producer and a Consumer. The Producer emitted 500 measurements, of which all 500 were read by the consumer, so no measurement was lost. It is worth noting that we could not run Kafka with the default settings on our Raspberry Pi device. Instead, we had to change the following two settings, to allow Kafka to start and to avoid out of memory errors.

```
-Xms256m
-Xmx256m
```

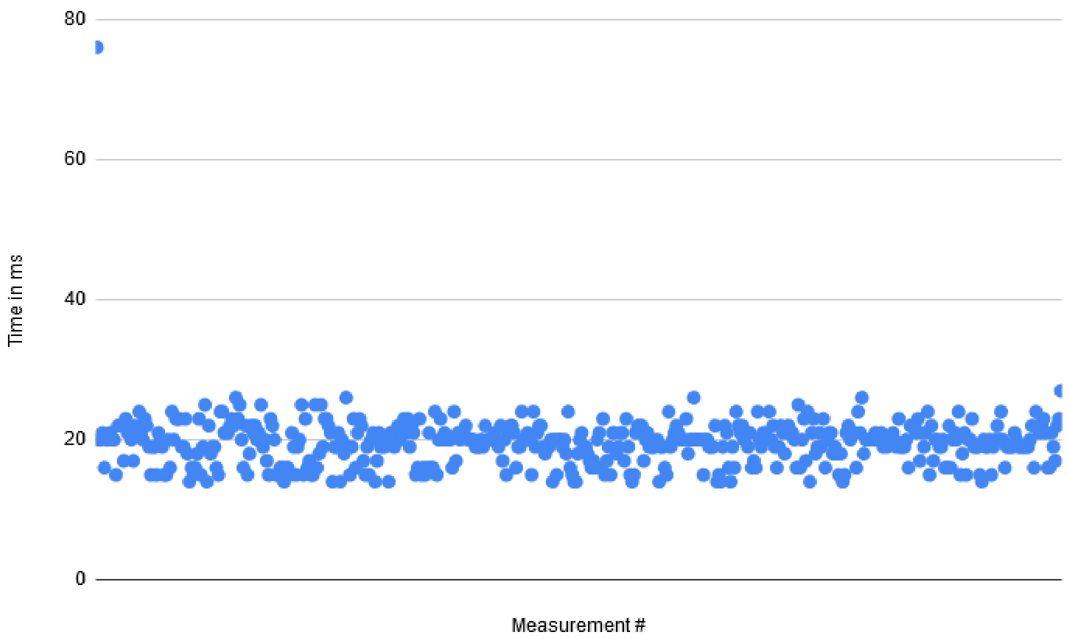

Figure 7.2: Latency per measurement when Kafka is deployed on a PC and a Producer and Consumer are running on a Raspberry Pi, Ethernet connected.

These settings could be changed further, and can be tweaked so that the Raspberry Pi really comes to its limits with regards to processing power, but does not exceed them. We did not perform any experiments with increasing the memory footprint of Kafka, as we feel that already for this low memory consumption, the results are sufficient for our use case. This would allow us to use the rest of the available processing power on the Raspberry Pi to run other components of our system.

Wi-Fi

Min 17ms, Max 2171ms, Median 39ms, Average 82.594ms

As expected, the time it takes for Kafka to process an entry, increases when Kafka is running on a weaker device. The data for all readings can be found in Figure 7.3.

Wired

Min 16ms, Max 513ms, Median 30ms, Average 30.666ms

Again, it is noteworthy, that the median and average time for a measurement to be processed has increased, with the weaker device running Kafka. All readings can be found in Figure 7.4.

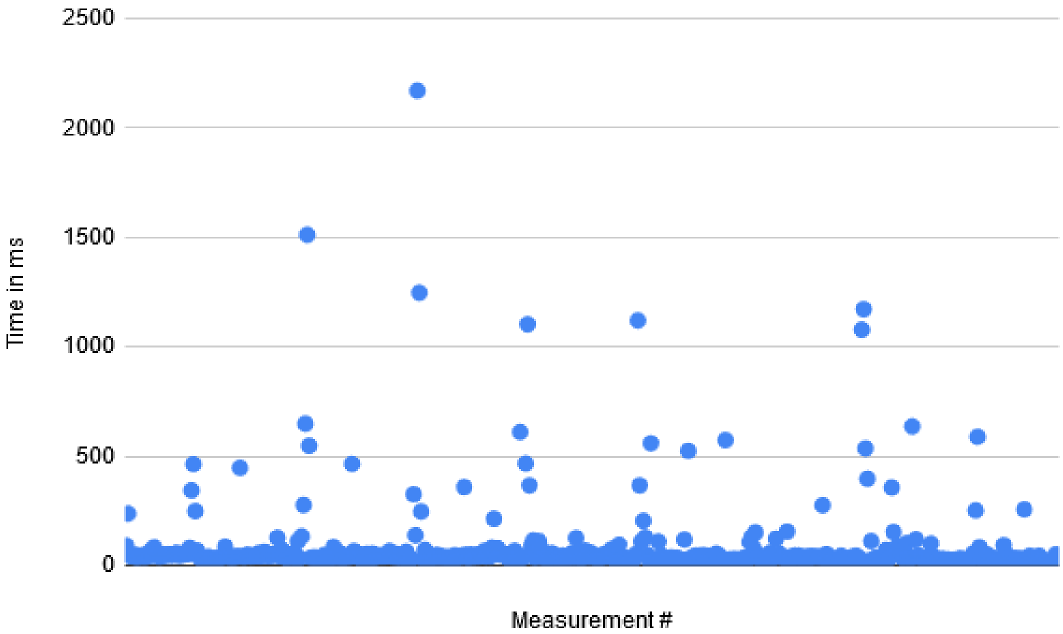

Figure 7.3: Latency per measurement when Kafka is deployed on a Raspberry Pi and a Producer and Consumer are running on a PC, Wi-Fi connected.

7.2.3 Summary

We have found that, while the average time to process a measurement is definitely influenced by the processing power of the device running Kafka, rather than the device producing the data, even running Kafka on a Raspberry Pi yields good enough latency for our use cases. Additionally, as expected, even temporary Wi-Fi problems do not impact the functionality in the system, as the retry mechanisms that Kafka employs ensure that the data is processed by consumers in any case. Figures 7.5, 7.6, and 7.7 show a direct comparison between the 4 previously mentioned scenarios, where the labels on the x axis correspond to the following numbers. Specifically, Figure 7.5 shows the whole y-axis, while Figures 7.6 and 7.7 cut off the y-axis at $600ms$ and $100ms$ respectively.

1. Kafka on PC, Consumer on RPi, Producer on RPi, connected over Ethernet.
2. Kafka on PC, Consumer on RPi, Producer on RPi, connected over Wi-Fi.
3. Kafka on RPi, Consumer on PC, Producer on PC, connected over Ethernet.
4. Kafka on RPi, Consumer on PC, Producer on PC, connected over Wi-Fi.

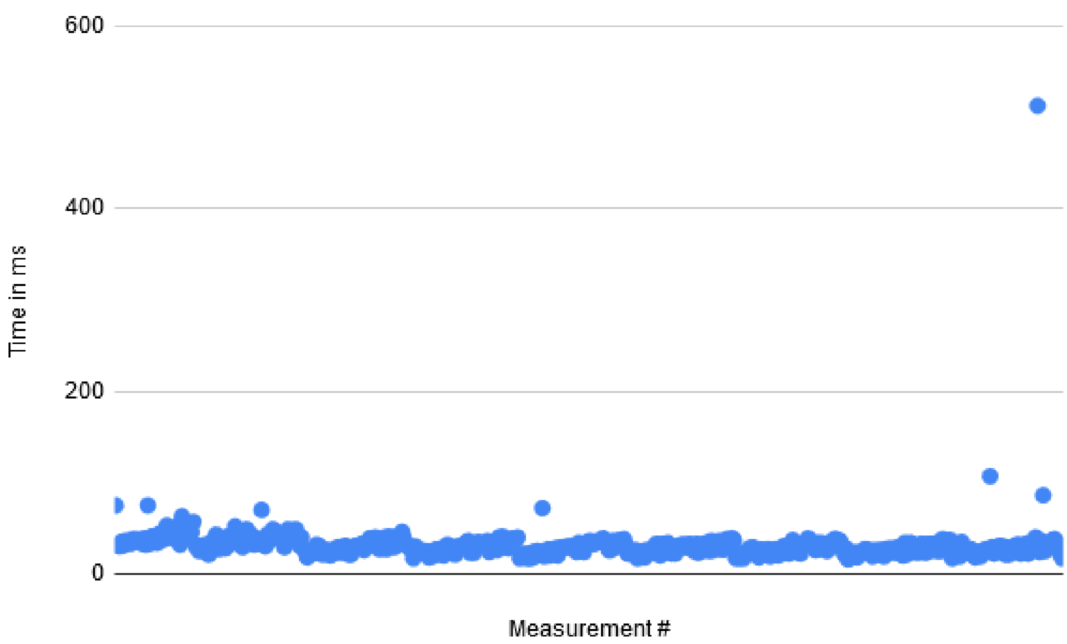

Figure 7.4: Latency per measurement when Kafka is deployed on a Raspberry Pi and a Producer and Consumer are running on a PC, Ethernet connected.

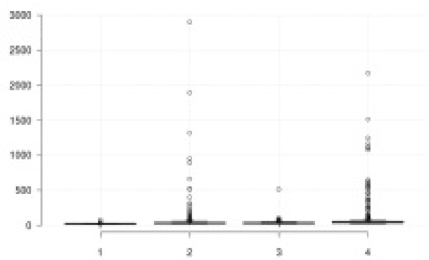

Figure 7.5: Comparison of the four communication strategies, scaled to full.

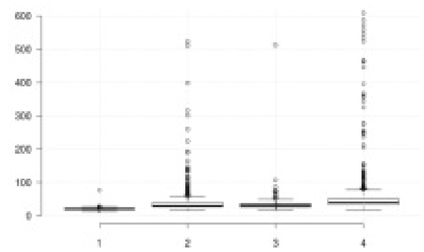

Figure 7.6: Comparison of the four communication strategies, scaled to maximum of 600ms.

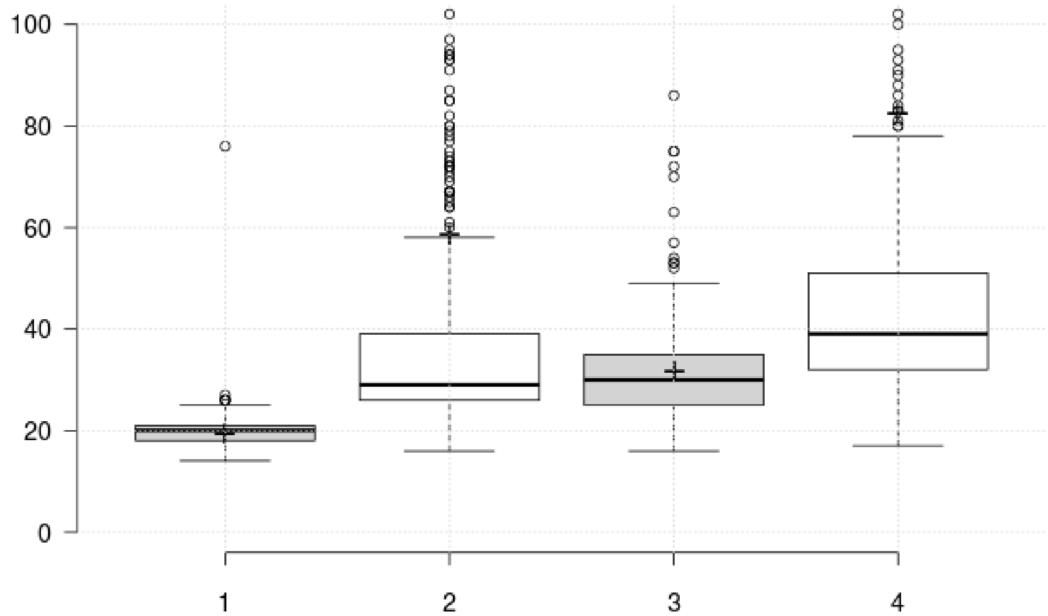

Figure 7.7: Comparison of the four communication strategies, scaled to maximum of 100ms.

There are a two important takeaways from this comparison. First, Wi-Fi generally only performs marginally worse than an Ethernet connection does but has much higher latency spikes. Second, generally Kafka performs better on a strong device, as was expected.

7.3 Maximum Kafka load on one Raspberry

This benchmark will show how the Kafka Stack (Kafka Server + Kafka Consumer) performs on a single Raspberry Pi. For this, the Kafka Server runs on the Raspberry Pi, the Consumer runs on the Raspberry Pi and a Producer runs on a separate device. The connection between devices was a wired one.

We increased the amount of items that were published by factors of 10, and then measured how long it took to publish all items. Interestingly, the amount of items we can publish per second almost converges towards 30,000 items published per second. Also, even with one million emitted items, no item got lost or was read twice, but the time it took for all measurements to be read increased consistently.

This test can help us give an answer on how many items we can expect to process on a single Raspberry Pi. This will allow us to answer the question: How many sensor devices can we serve with only a single Raspberry Pi device running a Kafka Server. This test shows us that we can process 30.000 items per second.

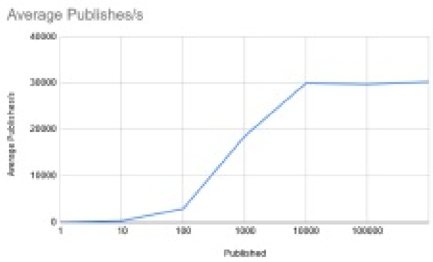

Figure 7.8: Average publishes per second on PC, calculated

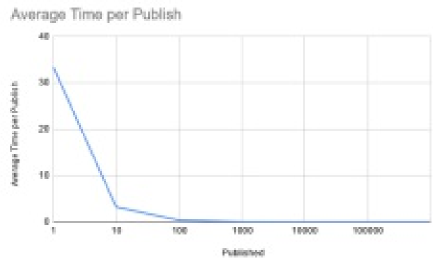

Figure 7.9: Average time per publish for a single measurement on PC

We believe that this test demonstrates that one Raspberry Pi is definitely enough to monitor a farm, as it is highly unlikely there will be more than thirty thousand measurements per second. In fact, the amount of measurements taken will be significantly lower.

7.4 Maximum total load of one Raspberry

This benchmark will run our entire software stack on one Raspberry Pi and will show that a single Raspberry Pi can run our entire stack, without problem.

Our Raspberry Pi 2 Model B with 1GB of RAM, could support our entire Stack, requiring around 700MB of RAM in a minimum stack setup: Kafka, as well as the Controller and one decision engine.

7.5 Maximum data produced by one Raspberry

In this benchmark we will have our entire software stack minus the Producer running on a separate device, while on the Raspberry Pi only the Producer is running. This will allow us to deduce how many sensors can be handled by a single Raspberry. It's important to note that we performed this test using a wired connection, since using a wireless connection would only produce noise in the testdata, that would make it even harder to form conclusions. It is also worth noting, that we have set the retries parameter of the consumer to 5.

Essentially, this test measures the amount of measurements we can send from a single Raspberry Pi, while still guaranteeing 100% delivery rate. Based on our experiments it is not clear where the point is, at which the delivery can no longer be guaranteed, but for us it is sufficient to say, that the ranges we have shown are high enough, to satisfy our use case.

We perform this test by synchronously sending one measurement after another without any explicit wait times in between. It is worth noting, that we used the default parameters

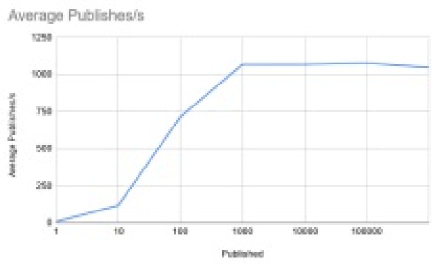

Figure 7.10: Average publishes per second on Raspberry Pi, calculated

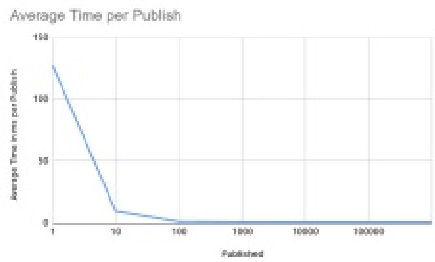

Figure 7.11: Average time per publish for a single measurement on Raspberry Pi

of the Kafka python Producer library[1], as mentioned earlier. This means, that the `batch_size` parameter is set to 16384. This is also the reason, why the time for a single measurement consistently decreases (up to a given point) - as the overhead generated by a single batch is bigger, than the processing time it takes to add the various measurements to the batch.

Based on the data shown in Table 7.2, the amount of measurements per second should not prove a performance bottleneck for the Kafka Producer that we employ in our test setup. We have shown that the Kafka Producer can safely emit up to one million items in rapid succession, with more than 1000 items per second with 100% transfer rate and no lost readings.

This test is an indication of how many sensors we can attach to a single Raspberry Pi and still expect all sensor readings to get sent within a reasonable amount of time. The answer being 1000 items per second that we can read and safely transmit.

7.5.1 Reducing the Buffer Size

As mentioned, the experiment we describe above uses the default buffer size of 16384. We performed other tests with the buffering completely turned off. This can be achieved by setting the buffer size parameter to zero. While we understand that the parameter space is vast, and it is not the goal of this work to fully benchmark the Kafka python library, we still feel that it is important to shortly summarize the results of this experiment, as it raises interesting questions.

Our testbed is the same as before, except for the reduced buffer size. At this point it is important to make the distinction between measurements, records and messages/requests. A measurement is one value that we have read, such as one reading on a temperature sensor "24 °C", whereas one record is a Kafka entity that already has various metadata attached to it and, in our case, encapsulates one measurement. Last, one request is the entity that we send into our Kafka topic and contains one or more records. In the

[1] `https://kafka-python.readthedocs.io/en/master/apidoc/KafkaProducer.html`

Items	Time in ms	Received	Success Rate in %	Time per Publish in ms
1	71	1	100	71
1	167	1	100	167
1	72	1	100	72
1	167	1	100	167
1	160	1	100	160
10	173	10	100	17.3
10	74	10	100	7.4
10	77	10	100	7.7
10	76	10	100	7.6
10	77	10	100	7.7
100	128	100	100	1.28
100	136	100	100	1.36
100	161	100	100	1.61
100	162	100	100	1.62
100	122	100	100	1.22
1000	858	1000	100	0.858
1000	957	1000	100	0.957
1000	842	1000	100	0.842
1000	1075	1000	100	1.075
1000	990	1000	100	0.99
10000	9266	10000	100	0.9266
10000	9207	10000	100	0.9207
10000	9512	10000	100	0.9512
10000	9243	10000	100	0.9243
10000	9600	10000	100	0.96
100000	95295	100000	100	0.95295
100000	98685	100000	100	0.98685
100000	89640	100000	100	0.8964
100000	87923	100000	100	0.87923
100000	93286	100000	100	0.93286
1000000	955973	1000000	100	0.955973

Table 7.2: Test results producer load test on Raspberry Pi

Published	Consumed	Success Rate in %
10	10	100
10	10	100
10	10	100
100	100	100
100	100	100
100	100	100
1000	1000	100
1000	1000	100
1000	1000	100
10000	8942	**89.42**
10000	9012	**90.12**
10000	8890	**88.90**

Table 7.3: Success rate for non-batched stress test

previous experiments, one request contained up to 16384 records. In this experiment, one request contains exactly one record. This test shows two things: 1. the previous tests highly benefit from the batching and 2. the overhead generated by one request can not be underestimated. Table 7.3 shows the results of our experiment. What we mean by Published and Consumed items, is the number of items we publish into the Kafka topic, and the number of items we consume. Where the items that can not be consumed are lost is not clear at this point in time. We do, however, expect some internal buffer of the Kafka Server to be overflowing, and thus not correctly forwarding all measurements to the Consumer. It is worth noting, that the "retry" parameter, is no help in this experiment, since as far as the publisher is concerned, the items were published correctly.

The takeaway from this test is twofold: 1. batching has a positive impact on publish-performance (at the cost of memory) and 2. the batch size is a parameter we can tune and optimize. Specifically, it might make sense to adjust the batch size in a way, that uses domain knowledge to 1. increase the time between measurements and 2. to batch measurements if possible.

An interesting aspect would for example be, to increase batch size (but not the measurement interval) during the night hours and actively flush the buffer when a measurement is taken that is higher than the rest of the measurements. This could be done to reduce power consumption. However, as we already have stated, the parameter space that can be optimized is large and the optimization of it is not the goal of this work, even though it is clear that it is an area for future work.

In this work, it is enough for us to say that the default parameters are sufficient for our use case and that we have shown that with minimal acquisition cost, a vertical farm can be monitored.

7.6 Vertical farm monitoring over a private LoRaWAN network

In this section we will perform two general test scenarios.

1. We will test the time it takes for a measurement from a LoRaWAN sensor to reach our system and until a potential confirmation has been received on the sensor device.
2. We will perform a load test on the LoRa Application Server to analyse how many devices we could potentially serve with one Application Server.

The testbed we use for these tests is shown in Figure 7.12. We should note that apart from the LoRaWAN network server stack, the Raspberry Pi shown in the figure also hosts the Controller module.

7.6.1 Latency

We performed a similar experiment as the one done in Section 7.2 regarding LoRaWAN. Our testbed setup consists of a Libelium Plug & Sense! Smart Water device[2], which communicates with a ChirpStack[3] Application Server. Specifically, we measure the round trip time a measurement takes from the Libelium device to the ChirpStack, and back to the Libelium device.

In total we have performed four experiments, where we compare the four following methods: `sendUnconfirmed` and `sendConfirmed` both with a downlink frame enqueued and with no downlink message enqueued on the application server.

A Confirmed LoRa message, is an optional protocol functionality, that allows the gateway to send a reply that the message was received. This is useful, as during wireless transmission, the sender can not be sure that the message was received by the receiver. There are two ways to achieve higher reliability of reception on the receiver side

1. The sender sends more than one message, in the hope that at least one of these messages reaches the receiver.
2. The receiver informs the sender that it has received a reading.

It is clear, that only the second case allows the sender to be certain that the message has been received. LoRaWAN allows for both of the above modes of operation. Both of these methods have their advantages and disadvantages, and one cannot be chosen over the other before considering a concrete use-case. Battery lifetime and airtime usage

[2] https://development.Libelium.com/smart_water_sensor_guide/
[3] https://www.chirpstack.io/

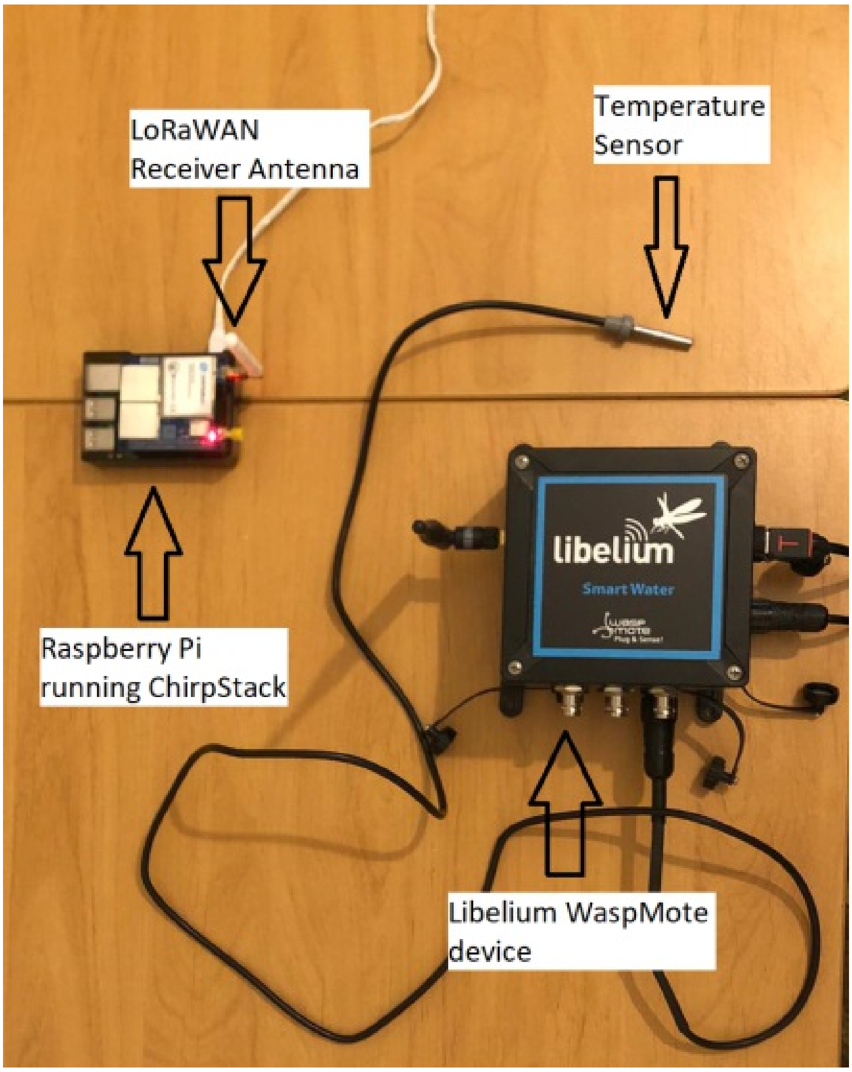

Figure 7.12: LoRaWAN testbed setup.

are two important factors to consider, especially since generally sending requires more battery than receiving.

```
1 void transmit()
2 {
3   Water.ON();
4   TemperatureSensor.readTemperature();
5   Water.OFF();
6   LoRaWAN.ON(socket);
7   LoRaWAN.joinABP();
8
9   LoRaWAN.sendConfirmed() or sendUnconfirmed();
10
11  if type == "confirmed"
12        convert received value as required
13 }
```

Listing 7.1: Pseudocode for taking a measurement with a Libelium device

Consider the pseudocode shown in Listing 7.1. The "total" readings, encompass reading from lines 3 to 12, both inclusive, whereas the "send only" readings, encompass only line 9 with the respective method used. Our results for the total reading can be seen in Figure 7.13, while the results for only the call to the `send` method are shown in Figure 7.14. The labels on the x-axis represent the following four methods of communication.

1. sendConfirmed No Downlink
2. sendConfirmed Downlink
3. sendUnconfirmed No Downlink
4. sendUnconfirmed Downlink

For each of the shown transmissions, we have performed 14 measurements, which we deem enough to show our point, as there is not much variance to be seen. These box plots show the minimum and maximum values, as well as the first and third quartile together with the median and mean values. The median is denoted by a horizontal line, while the mean is shown by a plus sign. The color of the box plots is only for display purposes and has no meaning regarding data.

These experiments have given us a number of interesting points to discuss. First, the variance in the various measurements is extremely low, usually in the range of only a few ms, except for one test: the sending of confirmed messages where no downlink is enqueued. The reason why this test has a variance of approximately 100ms is not clear, and has not been further investigated in this thesis, as the variance is so low that it does not impact our work.

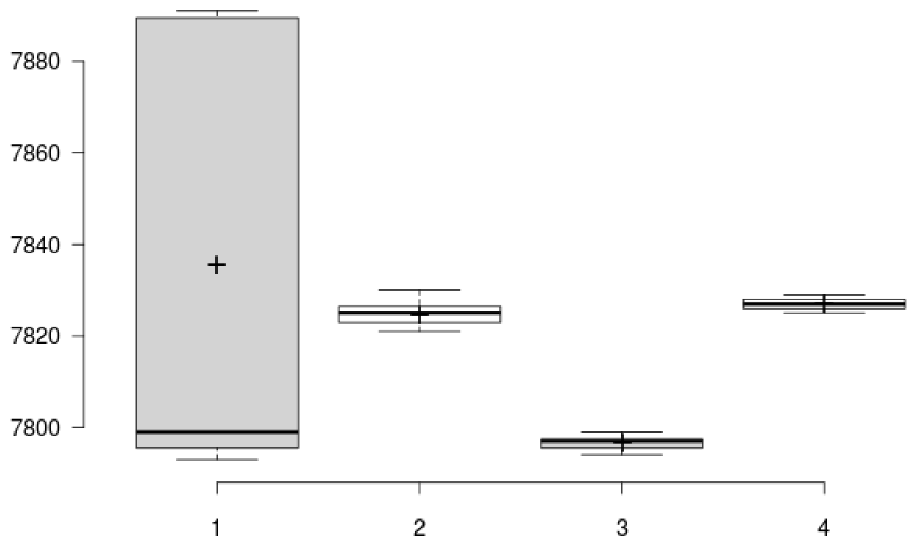

Figure 7.13: Total measurements for LoRa experiment.

Next, the actual send call of the LoRaWAN library provided by the WaspMote, is not impacted by using a confirmed, or an unconfirmed message, as can be seen both in Figure 7.13 and Figure 7.14.

Last, it is interesting to see, that the 2s receive window we have mentioned in Section 5.3 and further shown in Figure 5.4, can clearly be seen as included in the $2.5s$ it takes for the `send` call to finish.

The big discrepancy between the time it takes for the entire LoRa call to complete and only the actual sending, raises the question, where the rest of the time is spent. Figure 7.15 shows that the majority of the time ($\sim 38\%$) is spent in the `joinABP` call, narrowly followed by the real transmission of data at $\sim 33\%$. While these are all functions that are implemented in the LoRaWAN library provided by Libelium and we do not have the possibility to easily optimize these, it is still important to understand where the majority of the time it takes for an entire reading to be done and transmitted is spent. In fact, there is a non-negligible amount of time spent in `sleep` commands in the Libelium library before each command sent from the host MCU to the radio module. This communication takes place over a UART (Universal Asynchronous Receiver/Transmitter) interface and the `sleep` guards are in place to ensure that commands have been executed properly. It may be possible to reduce these sleep times and optimize the function calls, but this would be out of the scope of this work.

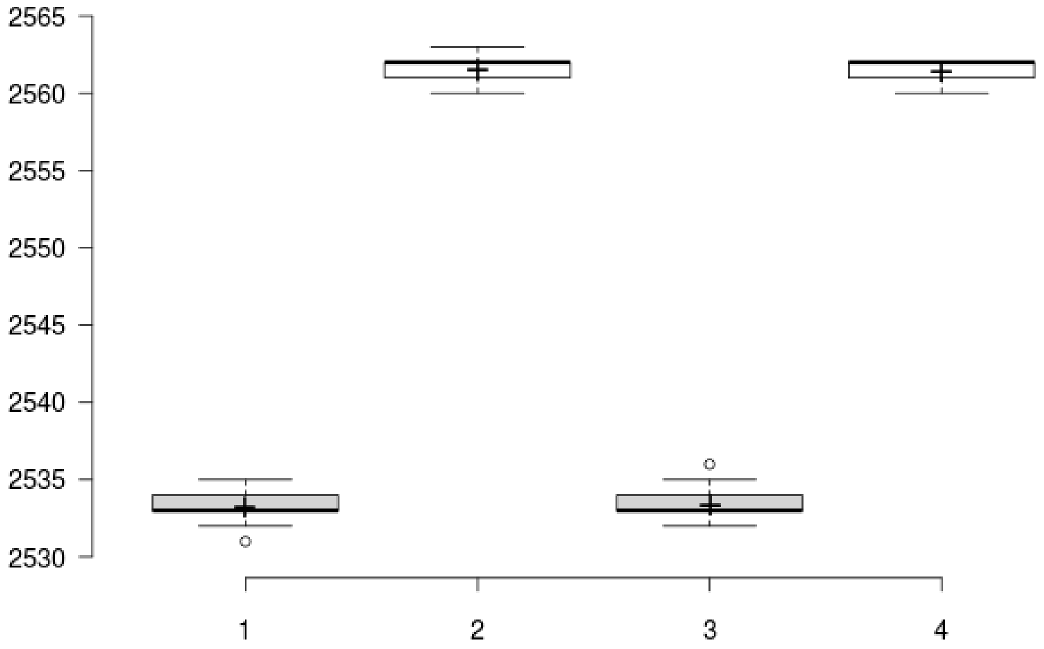

Figure 7.14: Measurements for only the send function call

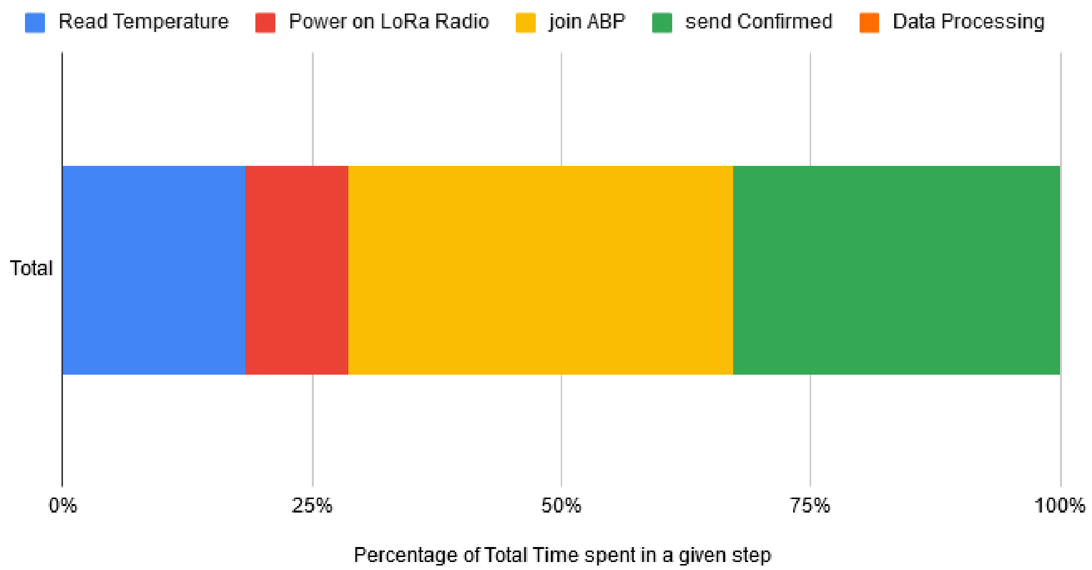

Figure 7.15: Breakdown of LoRa communication of a sendConfirmed call with downlink enqueued. Less than 0.1% of time is spent on data processing post receive.

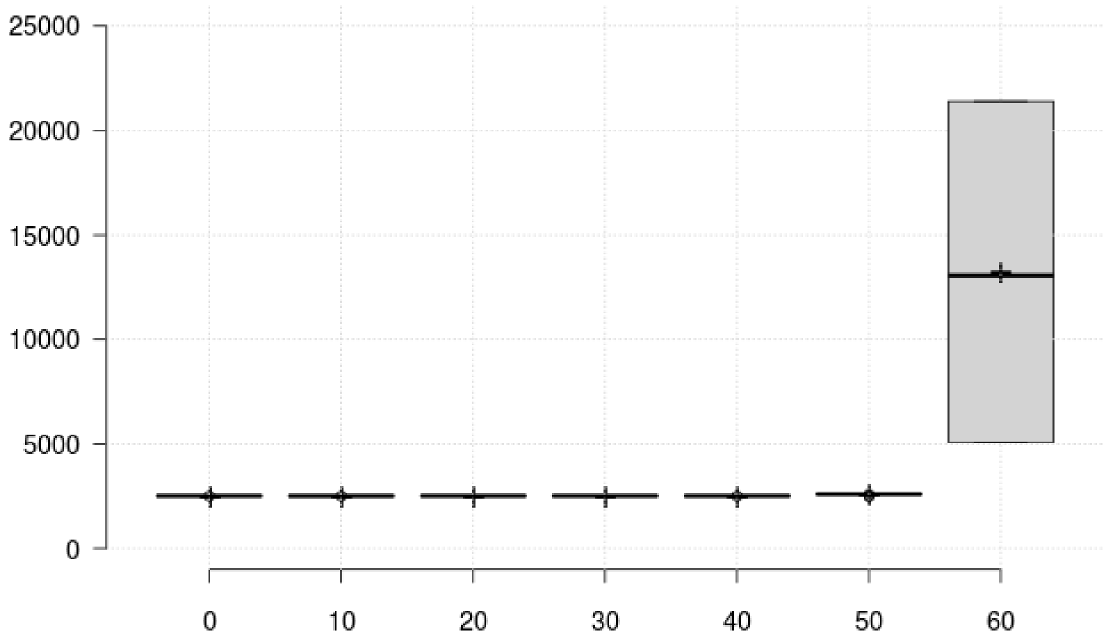

Figure 7.16: Time it takes for the sendConfirmed call to finish under load of the Application Server.

7.6.2 Maximum Load

We used the ChirpStack Simulator[4] to simulate increasingly heavier load on our Application Server. Specifically, we increase the amount of uplink messages per second the simulator generates by increments of 10 and then analyse the amount of time it takes for the entire `transmit` method, as shown in Listing 7.1, to complete. We found that only the actual `sendConfirmed` call in line 9 is impacted by increased load, and the `joinABP` call is not impacted.

Our tests have shown there appears to be a hard limit at around 60 uplink messages per second received. Figure 7.16 clearly shows that there is no slow degradation of service quality, but rather a hard limit at which the Application Server cannot handle the required load anymore. Additionally, at 60 uplink messages per second, we first experienced that some messages did not receive any confirmation from the LoRa stack. We measured that approximately $\sim 25\%$ of messages were not confirmed.

[4] `https://github.com/brocaar/chirpstack-simulator`

7.6.3 Summary

The takeaway from these tests, is that we can definitely use LoRaWAN devices to measure a vertical farm for our use-case, as a roundtrip latency of approximately 7s, is fine for our use-case. These 7s were measured when using the Libelium WaspMote device and more efficient communication between the device host and the LoRa radio module would help reduce this time.

About downlink communication a few things have to be said. We do not provide an experiment on the time it takes for downlink messages from the LoRa application server to reach the device. The reason is twofold: 1. we show that the round-trip time for uplink and downlink messages is sufficient to satisfy our use-case and 2. more importantly: the downlink latency, is fully dependant upon the uplink frequency. As we have already mentioned in Section 5.3, Class A LoRa devices only can receive downlink messages after an uplink, so if we only send uplink messages in a 30min interval, then the worst-case expected downlink latency also has to be approximately 30min. Of course, as also mentioned we can utilize Class B or Class C LoRa devices, if downlink messages become important. However, 1. we do not deem the downlink messages to be of such high relevance for our current setup and 2. Class A is the default type for devices and the minimal behaviour that has to be supported and therefore each device will definitely be able to handle this type of communication.

We have further shown that there is a hard limit of 60 uplink messages per second for the ChirpStack Application Server. However, 60 uplink messages per second corresponds to 600 LoRa devices, if each device sends 1 message every 10 seconds. In practice, and for various vertical farming scenarios, event generation frequencies could be much lower. Importantly, this bottleneck is associated with the processing capabilities of our reference edge device, namely a Raspberry Pi. Executing the same experiment on a more advanced computational platform, as would be the case for a powerful x86-based host or VM or a VM executed at edge or centralized clouds, would yield significantly better capacity. Furthermore, it is possible for the LoRaWAN stack to be scaled horizontally, provided that more edge compute resources are available. Our results can help the system designer to directly derive the amount of such resources necessary (e.g., number of Raspberry Pis at a local edge cluster) to support a given desired workload.

We deem this enough at this point in time, and conclude that the Libelium WaspMote device and the LoRaWAN network stack, as provided by the ChirpStack implementation, deployed on a Raspberry Pi provide sufficient processing power and bandwidth for our current use-cases.

7.7 Cost analysis

By taking the knowledge we have gained from the previous sections in this chapter, we believe that a farmer will be able to monitor a farm of any size with just one single Raspberry Pi. Eventually, the problem will become one of the connectivity between components, rather than of computing power. This raises the question when the farm becomes too big, how can the various sensors be connected to the Raspberry Pi? We do not supply an answer to this question at this point in time.

Note that all prices were extracted on 31.10.2020. The prices and also the contents of the GrovePi+ Base Kit may have changed since the time of writing.

We calculate the setup cost for a monitoring system for a farm as follows:

- 1 Raspberry Pi[5] 40€
- 1 GrovePi+ Base Kit[6] 130€

The GrovePi+ Base Kit includes sensors such as temperature, humidity and light.

So for the price of only 170€ a farmer can start monitoring his farm for temperature, humidity and light. Additional costs such as a wireless network connection or power supply throughout the farm will arise.

Should the farmer wish to use a LoRaWAN based installation, the cost for LoRaWAN communication devices can be approximated at less than 10 €. This is only the price for the LoRaWAN communcation, and sensor devices have to be bought in addition.

7.8 Summary

In this section we provide a short and to the point summary of the key takeaways from our experiments.

Takeaways

- Kafka employs good enough error handling methods to reliably transmit measurements over both Ethernet and Wi-Fi connections.
- A Raspberry Pi can handle both the Kafka stack and our system simultaneously.
- The batch size parameter of the Python producer heavily influences the maximum throughput we can achieve.
- LoRaWAN is a suitable communication technology for our system.
- There appears to be a limit of 50 measurements per second for the ChirpStack Application Server.
- The latency for LoRaWAN devices depends on the type of device as well as the library that is used. In our case we achieved round-trip latency of approximately 7 s.

[5]https://www.raspberrypi.org/products/raspberry-pi-4-model-b/?resellerType=home
[6]https://www.dexterindustries.com/store/grovepi-base-kit/

CHAPTER 8

Conclusion and Discussion

In this work we have shown our Proof of Concept implementation of a platform that allows a vertical farmer to monitor his farm with various sensors. We have furthermore shown that our system can be extended to add multiple decision engines that notify the farmer if an error has occurred. Additionally, our design, implementation and our evaluation results are in line with the design requirements we had on our system as shown in Section 4.2. Furthermore, we analyse the different network topologies that a system to monitor a (vertical) smart farm can have, and highlight the advantages and disadvantages of each. We show two different deployment scenarios for our system and also show that we can handle various wireless technologies such as LoRa and Wi-Fi. We also provide a cost analysis for our system that allows a vertical farmer to evaluate how much a setup of our system will cost for a farm of a given size. Specifically we have calculated that for less than 200€ a farmer can monitor his farm using our system.

8.1 Adherence to Design Requirements

When considering the Design Requirements we defined at the beginning of this work in Section 4.2 and the final result of our work, it can be seen that we have satisfied all requirements that we had.

8.1.1 Different Sensor Technologies

Our Proof of Concept system can handle different sensor technologies, as we require of it. Additionally, it can handle multiple units of measurement and negative and decimal values. Further, we also show that we can handle transmission of data over the LoRaWAN protocol.

8.1.2 Multiple Decision Modules

In our PoC system we show that there can be multiple decision engines working simultaneously without interrupting each other, as the stream processing capabilities of Apache Kafka support multiple consumers.

8.1.3 Resource constrained devices

Again, we have demonstrated that a Raspberry Pi device can handle enough load to support tens of thousands of sensors when using a GrovePi module. However, there are other limitations, which will limit this number before the Raspberry Pi will become the bottleneck. Additionally, we have shown that we can support approximately 500 LoRa devices, with a single ChirpStack Application Server running on a Raspberry Pi.

8.1.4 Service oriented architecture principles

The architecture we designed follows SoA principles. Moreover, it is easy to extend as we have also shown, by implementing various additional modules, such as extra decision engines or a LoRa sensor device.

8.1.5 Configure different modules

Modules can be configured independently of each other if necessary. However, our PoC implementation shows that very little configuration is needed.

8.2 Revisiting Research Questions

In this section we will revisit the three research questions we presented in the beginning of our work, and attempt to either answer them directly or point towards the given part of our work where we answer them.

RQ1: How can different factors that impact plant growth be monitored and how should such software architecture be designed to facilitate data gathering and processing?

In Chapter 4 we show our suggestion for a software architecture, which we later implement in Chapter 6. Further, in Chapter 5 we show the network topologies that impact the architecture design decisions.

We show a method to measure such factors by using low-cost edge devices, like Raspberry Pis and the GrovePi module or the Libelium WaspMote device. Further, we suggest a service oriented architecture as it allows extensibility as well as scalability. This can prove especially useful if the amount of data gathered and the processing power requirements increase.

RQ2: Is it feasible to operate a system for vertical farm monitoring and autonomous management using low cost edge computing hardware?

As we have shown in Chapter 7 and further expanded upon in the various sections of the chapter, we definitely deem it feasible to operate such a system by using low cost edge computing hardware, such as Raspberry Pi computers.

RQ3: What are the performance and capacity limits of edge computing hardware in terms of processing vertical farming monitoring data and what are their cost implications?

The cost implications of such a system are highlighted and presented in Section 7.7 and the performance limits are shown in the various Sections in Chapter 7.

Specifically we have measured a Raspberry Pi to be able to run our system and to perform an amount of measurements which are definitely enough to monitor a farm sufficiently. First, we have determined that the latency between a Raspberry Pi and another device is low enough to support our use-case. Second, we have measured that a single Raspberry Pi is able to transmit up to 30000 measurements per second. Next, the maximum amount of LoRaWAN sensor devices we can support with one Raspberry Pi hosted ChirpStack network server is approximately 500. The costs of entry are low for a farmer who wishes to start monitoring his or her farm, as we have calculated the cost to be below 200 €.

8.3 Key Takeaways

In this section we wish to highlight the results of our experiments and reiterate the key points that we have discovered.

1. One Raspberry Pi can handle enough sensor load to support monitoring of a vertical farm.
2. One Raspberry Pi can handle the Kafka Stack.
3. Different Network Topologies possible or necessary depending on the situation in the farm such as pre-existing infrastructure and network connectivity.
4. Small price of entry (∼200 €) for farmers to start monitoring their farm.
5. Domain knowledge necessary for threshold values that are critical.

8.4 Limitations

As we have only provided a Proof of Concept implementation as well as a substantial amount of analysis of the process and potential pitfalls in regards to architectures or software design, our system in its current form still has some limitations. We wish to mention these limitations in this chapter, and provide an outlook of how these limitations can be circumvented in Section 8.5.

8.4.1 Hardware requirements

The proof of concept implementation in its current form has a few requirements: power supply, hardwire connection between sensors and the GrovePi module, Wi-Fi connection between the sensor modules, decision modules, controller, user interface. These limitations could, however, be circumvented by using different technologies. For example, there exist other single board computers that have a lower power consumption and can be run on portable batteries for a multitude of weeks/months.

8.4.2 Persistent data retention

Furthermore, the system in its current form does not have a way of saving historical data. We do, however, consider adding a locally run database connected to the data processing modules a straightforward task. Using a MongoDB in a Docker Container[1] is a good point of reference for starting this implementation. Additionally, due to the fact that there is no persistent data retention method, the user interface component can not "catch up" to already measured data, but instead can only show the live values that are being measured.

8.4.3 Domain knowledge required

Since there currently is no technology-supported learning implemented in our system, there is an amount of domain knowledge required when setting up decision engines that rely on domain specific knowledge, rather than just statistical data.

8.5 Future Work

There are a number of features we have identified in the course of our work as suitable extensions and next steps for our system.

One feature which we have mentioned throughout our work is the handling of image sensors and their processing. This is an obvious candidate for future work. Due to the extendibility of our system and our design, it will be easy to integrate such sensors. Specifically, one would only have to adapt the data transmission between the sensors and the controller. We can imagine two scenarios for this: 1. the image processing is done on the sensor device or 2. the image processing is done in the controller. For the first scenario the sensor will have to be adapted and the interface between sensors and controller will only have to be marginally adapted to allow for the new data types. For the second one, the interface will have to be adapted, and also the controller has to be

[1] https://hub.docker.com/_/mongo

adapted to allow handling of image files. This will require new benchmarks and possibly even new networking technologies to allow for transmission of much bigger message sizes.

In Chapter 6 we have shown the interfaces that our platform supplies. By using the given interfaces it is easily possible to add further decision engines that can further assist the vertical farmer. One such extension could be the implementation of more sophisticated decision engines. Specifically, the decision engines we implemented for demonstration purposes do not make use of the full capabilities that Apache Kafka would provide to us. For example, one could implement a decision engine that takes a specific time window together with some parameter. Concretely this could mean using a window of 15 minutes and measuring whether a parameter has exceeded a given value a given amount of times during this time frame. It will especially be interesting to implement decision engines that require a bigger amount of processing power, as this would allow us to make use of other deployment scenarios we have presented such as the one shown in 5.4.

Next, as mentioned in Section 8.4, a persistent data retention mechanism should be added for use in production environments. This will further allow the system to rely on machine learning algorithms to learn from the data that has been gathered in the past. This is necessary to be able to give better recommendations towards farmers, which treatment a specific plant should get under the given circumstances. Additionally, extending the system as touched upon in Section 4.1 will allow the system to adapt on measured data and possibly 1. reduce the manual input required by domain experts and, in turn, 2. increase the domain knowledge of the domain expert, by allowing a farmer to get better insight to historical and current data that is measured through our system.

Additionally, we believe that there should be more evaluations done with regards to sensors connected to the GrovePi device and further research is needed, if the GrovePi module can be switched out for a different one. The reason for this is, that roughly 75% of costs currently stem from the GrovePi module, while the Raspberry Pi has a much smaller financial impact on a farmer.

Next, it should be analysed and measured how much pre-existing infrastructure is required in a farm before our system can be set up. This includes things such as a power supply, network connectivity and also methods to keep sensors dry in case our system is utilized for an outdoor farm.

Last, we would like to build a multitude of testbeds and focus on one crop to build a database of the various external factors such as humidity, light, water, and pH and analyse the plant growth that comes with changing of these factors. Building upon this, we could maximize the potential yield of a vertical farm, and possibly even allow individual households to have their own small vertical farm that can be monitored by our cheap monitoring solution.

List of Figures

List of Tables

Bibliography

[Ade+17] Ferran Adelantado et al. “Understanding the limits of LoRaWAN”. In: *IEEE Communications magazine* 55.9 (2017), pp. 34–40.

[Akb+15] Adnan Akbar et al. “Context-aware stream processing for distributed IoT applications”. In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE. 2015, pp. 663–668.

[Apa] *Apache Zookeeper Confluence*. `https://cwiki.apache.org/confluence/display/ZOOKEEPER/Index`. Accessed: 2020-11-11.

[ARS15] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. “Modeling and analyzing MAPE-K feedback loops for self-adaptation”. In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE. 2015, pp. 13–23.

[Aug+16] Aloÿs Augustin et al. “A study of LoRa: Long range & low power networks for the internet of things”. In: *Sensors* 16.9 (2016), p. 1466.

[BLD04] Rodolfo Bongiovanni and Jess Lowenberg-DeBoer. “Precision agriculture and sustainability”. In: *Precision agriculture* 5.4 (2004), pp. 359–387.

[BVR16] Martin Bor, John Edward Vidler, and Utz Roedig. “LoRa for the Internet of Things”. In: (2016).

[Che+18] Xu Chen et al. “ThriftyEdge: Resource-efficient edge computing for intelligent IoT applications”. In: *IEEE network* 32.1 (2018), pp. 61–65.

[Chi] *ChirpStack Homepage*. `https://www.chirpstack.io/`. Accessed: 2020-11-11.

[DAM19] Schahram Dustdar, Cosmin Avasalcai, and Ilir Murturi. “Edge and fog computing: Vision and research challenges”. In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE. 2019, pp. 96–9609.

[Dav+18] Danco Davcev et al. “IoT agriculture system based on LoRaWAN”. In: *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*. IEEE. 2018, pp. 1–4.

[Del17] Jerker Delsing. *Iot automation: Arrowhead framework*. CRC Press, 2017.

[Des11] Dickson Despommier. “The vertical farm: controlled environment agriculture carried out in tall buildings would create greater food safety and security for large urban populations”. In: *Journal für Verbraucherschutz und Lebensmittelsicherheit* 6.2 (2011), pp. 233–236.

[Din+20] Jie Ding et al. “IoT connectivity technologies and applications: A survey”. In: *arXiv preprint arXiv:2002.12646* (2020).

[Diz+19] Jasenka Dizdarević et al. “A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration”. In: *ACM Computing Surveys (CSUR)* 51.6 (2019), pp. 1–29.

[DR00] Steve Diver and Lee Rinehart. *Aquaponics-Integration of hydroponics with aquaculture*. Attra, 2000.

[Fiw] *FIWARE Homepage*. `https://www.fiware.org/`. Accessed: 2020-11-11.

[FMC06] Imma Farran and Angel M Mingo-Castel. “Potato minituber production using aeroponics: effect of plant density and harvesting intervals”. In: *American Journal of Potato Research* 83.1 (2006), pp. 47–53.

[FP+18] Francisco Javier Ferrández-Pastor et al. “Precision agriculture design method using a distributed computing architecture on internet of things context”. In: *Sensors* 18.6 (2018), p. 1731.

[GD18] Marjan Gusev and Schahram Dustdar. “Going back to the roots—the evolution of edge computing, an iot perspective”. In: *IEEE Internet Computing* 22.2 (2018), pp. 5–15.

[Gha+16] Bilal Ghazal et al. “Smart traffic light control system”. In: *2016 third international conference on electrical, electronics, computer engineering and their applications (EECEA)*. IEEE. 2016, pp. 140–145.

[Har+19] Isakovic Haris et al. “CPS/IoT Ecosystem: Indoor Vertical Farming System”. In: *2019 IEEE 23rd International Symposium on Consumer Technologies (ISCT)*. IEEE. 2019, pp. 47–52.

[Hor01] Paul Horn. “Autonomic computing: IBM’s perspective on the state of information technology”. In: (2001).

[HS05] Michael N Huhns and Munindar P Singh. “Service-oriented computing: Key concepts and principles”. In: *IEEE Internet computing* 9.1 (2005), pp. 75–81.

[IW15] Didac Gil De La Iglesia and Danny Weyns. “MAPE-K formal templates to rigorously design behaviors for self-adaptive systems”. In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 10.3 (2015), pp. 1–31.

[Jay+16] Prem Prakash Jayaraman et al. “Internet of things platform for smart farming: Experiences and lessons learnt”. In: *Sensors* 16.11 (2016), p. 1884.

[Jen97] Merle H Jensen. “Hydroponics worldwide”. In: *International Symposium on Growing Media and Hydroponics 481*. 1997, pp. 719–730.

[JJ16] J Benton Jones Jr. *Hydroponics: a practical guide for the soilless grower.* CRC press, 2016.

[KA16] Abhirup Khanna and Rishi Anand. “IoT based smart parking system”. In: *2016 International Conference on Internet of Things and Applications (IOTA)*. IEEE. 2016, pp. 266–270.

[KC03] Jeffrey O Kephart and David M Chess. “The vision of autonomic computing”. In: *Computer* 36.1 (2003), pp. 41–50.

[KD20] Wafa’a Kassab and Khalid A Darabkh. “A–Z survey of Internet of Things: Architectures, protocols, applications, recent advances, future directions and recommendations”. In: *Journal of Network and Computer Applications* (2020), p. 102663.

[Kra08] BA Kratky. “Three non-circulating hydroponic methods for growing lettuce”. In: *International Symposium on Soilless Culture and Hydroponics 843*. 2008, pp. 65–72.

[KSS14] Anurag Kanungo, Ayush Sharma, and Chetan Singla. “Smart traffic lights switching and traffic density calculation using video processing”. In: *2014 Recent Advances in Engineering and Computational Sciences (RAECS)*. IEEE. 2014, pp. 1–6.

[LCK19] Jie-Yan Lu, Chung-Liang Chang, and Yan-Fu Kuo. “Monitoring Growth Rate of Lettuce Using Deep Convolutional Neural Networks”. In: *2019 ASABE Annual International Meeting*. American Society of Agricultural and Biological Engineers. 2019, p. 1.

[LR+17] JA López-Riquelme et al. “A software architecture based on FIWARE cloud for Precision Agriculture”. In: *Agricultural water management* 183 (2017), pp. 123–135.

[LSK19] Euijong Lee, Young-Duk Seo, and Young-Gab Kim. “Self-adaptive framework based on MAPE loop for Internet of things”. In: *sensors* 19.13 (2019), p. 2996.

[Luc02] David Luckham. *The power of events*. Vol. 204. Addison-Wesley Reading, 2002.

[Mau+11] Michael Maurer et al. “Revealing the MAPE loop for the autonomic management of cloud infrastructures”. In: *2011 IEEE symposium on computers and communications (ISCC)*. IEEE. 2011, pp. 147–152.

[Mek+19] Kais Mekki et al. “A comparative study of LPWAN technologies for large-scale IoT deployment”. In: *ICT express* 5.1 (2019), pp. 1–7.

[Mil+20] Badreddine Miles et al. “A study of LoRaWAN protocol performance for IoT applications in smart agriculture”. In: *Computer Communications* (2020).

[Mod+] Falmata Modu et al. “A Survey of Smart Hydroponic Systems”. In: ().

[PI07] Zeydin Pala and Nihat Inanc. “Smart parking applications using RFID technology”. In: *2007 1st Annual RFID Eurasia*. IEEE. 2007, pp. 1–3.

[Sat17] Mahadev Satyanarayanan. "The emergence of edge computing". In: *Computer* 50.1 (2017), pp. 30–39.

[SCS17] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. "Riotbench: An iot benchmark for distributed stream processing systems". In: *Concurrency and Computation: Practice and Experience* 29.21 (2017), e4257.

[SD16] Weisong Shi and Schahram Dustdar. "The promise of edge computing". In: *Computer* 49.5 (2016), pp. 78–81.

[Sin+20] Ritesh Kumar Singh et al. "Leveraging LoRaWAN Technology for Precision Agriculture in Greenhouses". In: *Sensors* 20.7 (2020), p. 1827.

[SKC14] Saraswathi Sivamani, Kyunghun Kwak, and Yongyun Cho. "A design of web-based services using RESTful API for vertical farm". In: *Life Science Journal* 11.7 (2014), pp. 763–767.

[SS16] Anshu Shukla and Yogesh Simmhan. "Benchmarking distributed stream processing platforms for iot applications". In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2016, pp. 90–106.

[TC20] Mark L Tenzer and Nicholas C Clifford. "A Digital Green Thumb: Neural Networks to Monitor Hydroponic Plant Growth". In: *2020 Systems and Information Engineering Design Symposium (SIEDS)*. IEEE. 2020, pp. 1–6.

[Tön+14] Ralf Tönjes et al. "Real time iot stream processing and large-scale data analytics for smart city applications". In: *poster session, European Conference on Networks and Communications*. sn. 2014.

[Tri+15] Paolo Tripicchio et al. "Towards smart farming and sustainable agriculture with drones". In: *2015 International Conference on Intelligent Environments*. IEEE. 2015, pp. 140–143.

[Vas+17] Deepak Vasisht et al. "Farmbeats: An iot platform for data-driven agriculture". In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 515–529.

[Wix+16] Andrew J Wixted et al. "Evaluation of LoRa and LoRaWAN for wireless sensor networks". In: *2016 IEEE SENSORS*. IEEE. 2016, pp. 1–3.

[WZ92] PJ Weathers and RW Zobel. "Aeroponics for the culture of organisms, tissues and cells". In: *Biotechnology advances* 10.1 (1992), pp. 93–115.

[Yan17] Shusen Yang. "IoT stream processing and analytics in the fog". In: *IEEE Communications Magazine* 55.8 (2017), pp. 21–27.

[You+16] Changsheng You et al. "Energy-efficient resource allocation for mobile-edge computation offloading". In: *IEEE Transactions on Wireless Communications* 16.3 (2016), pp. 1397–1411.

[Zha+16] Ke Zhang et al. "Energy-efficient offloading for mobile edge computing in 5G heterogeneous networks". In: *IEEE access* 4 (2016), pp. 5896–5907.

[Zha+18] Ke Zhang et al. “Mobile edge computing and networking for green and low-latency Internet of Things”. In: *IEEE Communications Magazine* 56.5 (2018), pp. 39–45.

[ZI+19] Miguel A Zamora-Izquierdo et al. “Smart farming IoT platform based on edge and cloud computing”. In: *Biosystems engineering* 177 (2019), pp. 4–17.

[ZWW02] Naiqian Zhang, Maohua Wang, and Ning Wang. “Precision agriculture—a worldwide overview”. In: *Computers and electronics in agriculture* 36.2-3 (2002), pp. 113–132.