FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Teemu Varsala
Vili Pelttari

INTERNET OF THINGS BACKEND FOR EMBEDDED SYSTEMS TEACHING

Bachelor's Thesis
Degree Programme in Computer Science and Engineering
June 2021

Varsala T., Pelttari V. (2021) Internet of Things Backend for Embedded Systems Teaching. University of Oulu, Degree Programme in Computer Science and Engineering, 45 p.

ABSTRACT

In this work, the Internet of Things system is implemented for enabling distance learning and laboratory work for an embedded systems programming course at the University of Oulu. The system must meet the following three requirements. The system receives and visualizes sensor data from the embedded device. The system enables two-way communication with the cloud application and the embedded device. The system can be connected via the public Internet, where the system is managed through Kubernetes.

The architecture of the system is described in three different layers. The perception layer contains embedded devices used to produce sensor data. The components of the network layer process and transmit data. The cloud layer includes data storage and further processing in the application, as well as data visualization. The architecture of the implemented system consists of distributed microservices that are deployed using container technology.

The system was tested on the basis of feedback collected from the beta version implemented in autumn 2020, as well as use cases defined by the developers, which were constructed from previously known problem areas. As a result, modular and scalable future distance learning system for embedded systems was developed.

Keywords: Internet of Things, embedded systems, engineering education, microservices, virtualization

Varsala T., Pelttari V. (2021) Esineiden internetin taustajärjestelmä sulautettujen järjestelmien opetukseen. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 45 s.

TIIVISTELMÄ

Tässä työssä suunnitellaan ja toteutetaan etäopiskelun sekä -laboratoriotyön mahdollistava esineiden internetin järjestelmä sulautettujen järjestelmien ohjelmoinnin kurssille Oulun yliopistossa. Järjestelmän tulee toteuttaa seuraavat kolme vaatimusta. Järjestelmä vastaanottaa ja visualisoi anturitietoa sulautetulta laitteelta. Järjestelmä mahdollistaa kaksisuuntaisen viestinnän pilvisovelluksen ja sulautetun laitteen kanssa. Järjestelmään saa yhteyden julkisen Internetin kautta, jossa järjestelmään hallinnoidaan Kubernetesin avulla.

Järjestelmän arkkitehtuuri kuvataan kolmena eri kerroksena. Havaintokerros sisältää sulautettuja laitteita, joita käytetään anturitiedon tuottamiseen. Verkkokerroksen komponentit käsittelevät ja välittävät dataa. Pilvikerros sisältää tietojen tallennuksen ja jatkokäsittelyn sovelluksessa, sekä tietojen visualisoinnin. Toteutetun järjestelmän arkkitehtuuri koostuu hajautetuista mikropalveluista, jotka otetaan käyttöön konttiteknologian avulla.

Järjestelmää testattiin perustuen syksyllä 2020 toteutetusta kokeiluversiosta kerättyyn palautteeseen sekä kehittäjien määrittelemiin käyttötapauksiin, jotka luotiin hyödyntäen entuudestaan tunnettuja ongelma-alueita. Työn tuloksena valmistui modulaarinen ja skaalautuva tulevaisuuden etäopetusjärjestelmä sulautettujen järjestelmien ohjelmoinnin kursseille.

Avainsanat: esineiden internet, sulautetut järjestelmät, insinöörikoulutus, mikropalvelut, virtualisaatio

TABLE OF CONTENTS

FOREWORD

Foremost, we both would like to thank our supervisor, Dr. Teemu Leppänen, who was actively involved in the project from the very beginning. Providing guidance, and helping us through the most difficult phases. Professor Timo Ojala deserves thanks as the second examiner. Also, we would like to thank Mr. Rouhollah Ehsani for giving us a better understanding in container orchestration as well as in Kubernetes itself. Lastly, thanks to the faculty of Information Technology and Electrical Engineering for the summer job 2020, which through all different phases led us to finish this thesis.

Oulu, June 4th, 2021

Teemu Varsala
Vili Pelttari

LIST OF ABBREVIATIONS AND SYMBOLS

API	Application Programming Interface
CPU	Central Processing Unit
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identification number
IoT	Internet Of Things
QoS	Quality Of Service
RDBMS	Relational Database Managements Systems
RTOS	Real-Time Operating System
SPA	Single-page application
TLS	Transport-Layer Security
TSDB	Time Series DataBase
TUI	Terminal User Interface
UART	Universal Asynchronous Receiver-Transmitter
UI	User Interface
USB	Universal Serial Bus
VM	Virtual Machine
6LoWPAN	IPv6 over Low-Power Personal Area Networks

1. INTRODUCTION

The Internet of Things (IoT) has seen growing popularity over the last decade, with even more connected devices present in everyday human life. Different sensors and smart devices are used for example in home, industry, organizations, and health-care, to monitor, manage and automate task that has traditionally required human actions. [1]

Due to the COVID-19 pandemic, teaching courses with extensive lab work became more difficult. Thus, the person in charge of the course had a plan of a system that allows the course to be completed without restrictions imposed by COVID-19, while providing jobs for students.

In this thesis, we design and implement a container-based solution for a cloud application receiving sensor data from students embedded devices and enabling two-way communication with the devices for course *Computer Systems* at the University of Oulu. This platform will make it possible to deploy a course project environment for remote learning. At the end, the application will be deployed publicly. Furthermore, containers are orchestrated by Kubernetes distribution, RedHat OKD [2].

The rest of the thesis is organized as follows. Chapter 2 is about the background of technologies used in this project. In Chapter 3 we present the application architecture and Chapter 4 we present its implementation. All the testing is performed in Chapter 5, and the results are presented with a comprehensive data analysis. In chapter 6, we discuss the realized design and implementation as well as the reasoning leading to the presented solution and propose directions for future work. Lastly, Chapter 7 concludes this thesis.

2. RELATED WORK

Due to the pandemic, starting in December 2019, schools and universities in Finland were temporary closed in March 2020, including University of Oulu. The abnormal situation forced us to come up with better ways to carry out distance learning. The need to make the course, *Computer Systems*, suitable for remote studying became evident. During the course, students complete their own embedded programming project in groups using the Texas Instruments SensorTag embedded device, requiring classroom presence during exercises to enable interaction, including wireless communication, capabilities of the device with its environment. Creating a cloud application for this purpose makes it possible to comply with social distancing rules enforced by the public institutions by spreading the gateway devices for testing wireless communication to a larger area. It also makes it possible for some students to participate remotely, by using their own computer as the gateway device.

This project also serves the purpose of finding what it takes to create an IoT application for educational purposes, and how all this is utilized in education. Creating new ideas for future courses or for adapting existing ones. The current development in web technologies seems to be toward edge computing, and this gradual change should also be visible in education.

During this section, the most relevant concepts to the project are described in glancing detail to provide a view of where the project is situated in the present field. The major concepts include the IoT, microservices, and web technologies, or in other words, the web stack.

2.1. Internet of Things

IoT, a complex entity where numerous objects form a mesh to communicate with each other and the outside world [3]. These objects, small pieces of well structured network, are not always thought of as computers. They can be sensors, actuators, small devices, and etc. The whole idea behind this, however, is to make various different tasks automated and function without human intervention [1]. IoT is around us every day, even if we do not think about its existence. Novel examples of IoT surrounding us could be a smart lock at home, which keeps track of the state; locked or unlocked. Another well illustrating example from IoT world is the weather aware thermostat, which is able to adapt, no matter if it is a cold or warm day outside [4]. IoT is not standardized, and there are not just one unique and universal definition for IoT; it is rather context dependant [1].

The concept of IoT has been around for a long time, however, the term IoT was first coined in 1999 by British technology pioneer, Kevin Ashton. Ashton wanted to show the power of connected Radio Frequency IDentification tags. Since then, IoT has become the one and only term for a system described above [5]. We have come a long way since IoT started to gain popularity. In 2020, there are more IoT devices than non-IoT, approximately 12 billion IoT devices are connected to the Internet. The growing popularity of IoT shows no signs of slowing down. In fact, 30 billion devices is expected in 2025 [6], whereas the most optimistic believe the number could be much greater, even 100 billion devices has been proposed. The time will show the growth,

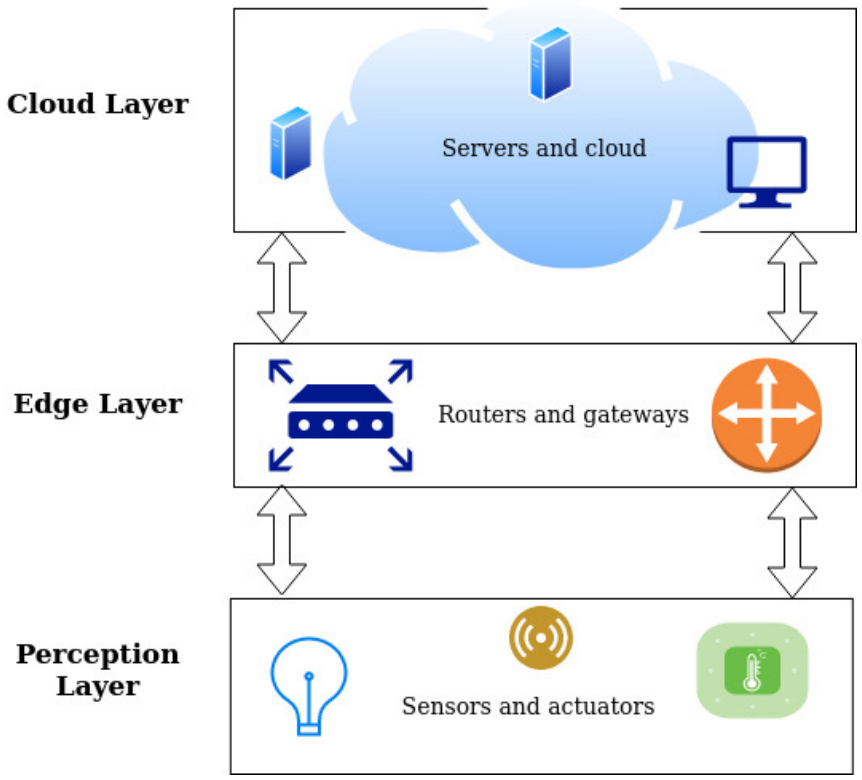

Figure 1. The layers of IoT

but it is clear that new solutions for processing and transferring huge amounts of data in a sustainable way is needed.

2.1.1. Layers

In IoT, there are more than just one definition what comes to the architecture models. The most fundamental way to structure IoT is the protocol-based three layer model. The five layer model is more suitable for researchers due to its more detailed way for separating functionalities [7].

Cloud layer

The top layer of this three layer IoT architecture model. Data travels from the perception layer, through edge layer, and finally reaches the cloud layer. This layer contains powerful servers that are used to run services, and to process and store data. These services are, for example, applications where to request data using a browser.

Edge layer

Edge layer is a part of the network layer, which consist of a core network layer and edge layer, a middle man of IoT architecture. Data travels in both directions through edge layer devices such as gateways, servers and access points. [8] Massive amounts of data produced by IoT devices has forced to come up with new ways to approach data

processing. For a long time, huge data centers located usually far from these physical IoT devices, were responsible for all computing. Edge brings more computing power closer to the place where it is needed, to the source of data, and allows real-time data to be used in time critical operations.

In march 2015, the Internet Architecture Board released a set of networking guidelines regarding smart objects.

In *device-to-device* model, at least two devices communicate directly using protocols such as Bluetooth. Commonly used in smart homes, for example, from light bulb to light bulb.

Alternatively, a smart device can connect with a cloud service directly, using Wi-Fi for instance, allowing the user to establish connection from a client application to the smart device. This is called *device-to-cloud* model.

Device-to-gateway connection is required in case where a smart device is not able to connect directly with a cloud service, or if a local gateway is needed for some other reason. Very common use-case is a watch that needs a smartphone to work at full capacity.

In more complicated scenarios, data produced by a sensor might need some supplement data from another sensor which may be located far away from sensor one. At this point *back-end data-sharing* model comes into play. These backend applications can share data, and produce wanted result by combining data from multiple different sources. [1]

Perception layer

On the bottom of IoT network architecture, there is a layer containing a miscellaneous set of heterogenous devices, “things”. This layer contains physical devices, and it is called a perception layer as well as device layer. This layer is included in every IoT-model. It consist of objects made specifically to produce data. The most common device in this layer is a smart phone, including various different types of sensors such as light intensity sensor, movement sensor, and microphone. [7]

2.1.2. Data Processing

IoT devices produce data more than ever, and the data has to be processed somehow. A way that has prevailed for a little longer is the cloud computing model, which is now losing its popularity to the fog computing model.

Fog

For the last couple of years, some changes towards the decentralized architecture have been made already. Fog (Figure 3, a newcomer that emphasizes distributed data processing is able to eg. offer more accurate real-time data by leaving out the phase where all the data is transferred and processed in a cloud [7]. The word “fog” for this concept was first presented by Cisco in 2014, a word that describes decentralized network as an extension to a cloud computing paradigm [9].

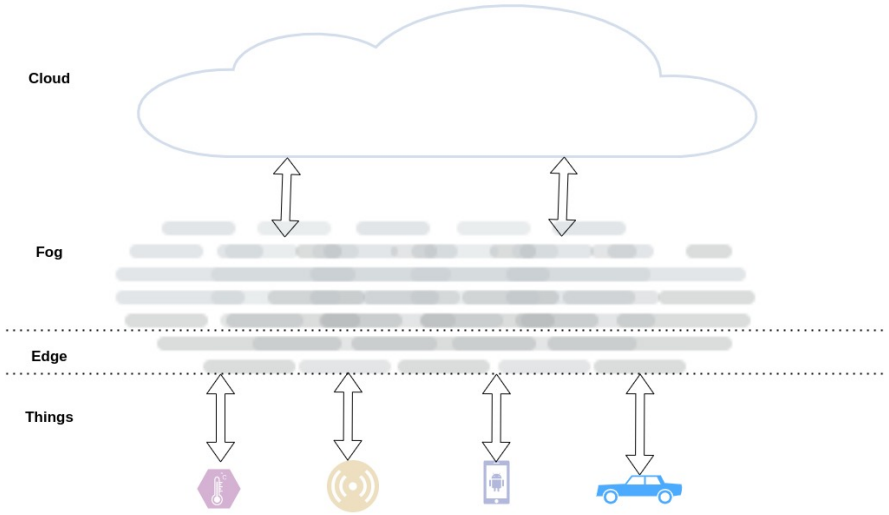

Figure 2. Fog computing architecture

Nodes of a fog computing network are usually located in wide area, and the number of nodes is much greater than cloud data centers. Furthermore, they have multiple advantages compared to cloud computing. For example, continuous internet connectivity is not mandatory, and by using smaller hardware as gateways, switched, servers, etc; it is possible to reduce power consumption. In addition, widely spread fog nodes can be located close to perception layer devices. [10]

One important metric for fog computing is Quality of Service (QoS), which is measured by four different parameters: connectivity, reliability, capacity, and delay. [11]

Cloud

Private cloud was the predecessor of public cloud, which means that the services can be bought with money, whereas, private cloud was usually located somewhere on the company's premises. There are significant advantages in using public cloud providers from user's perspective [12]:

- Availability of computing resources.
- Scalability of the services and applications.
- No need to invest in expensive hardware.
- Features such as security and reliability could be better.

These advantages are achieved by using virtualization techniques; virtual machines (VMs) and containers [12]. VMs guarantee better availability, and easier maintenance as well as installation of multiple different operating systems, and so, better utilization of the hardware.

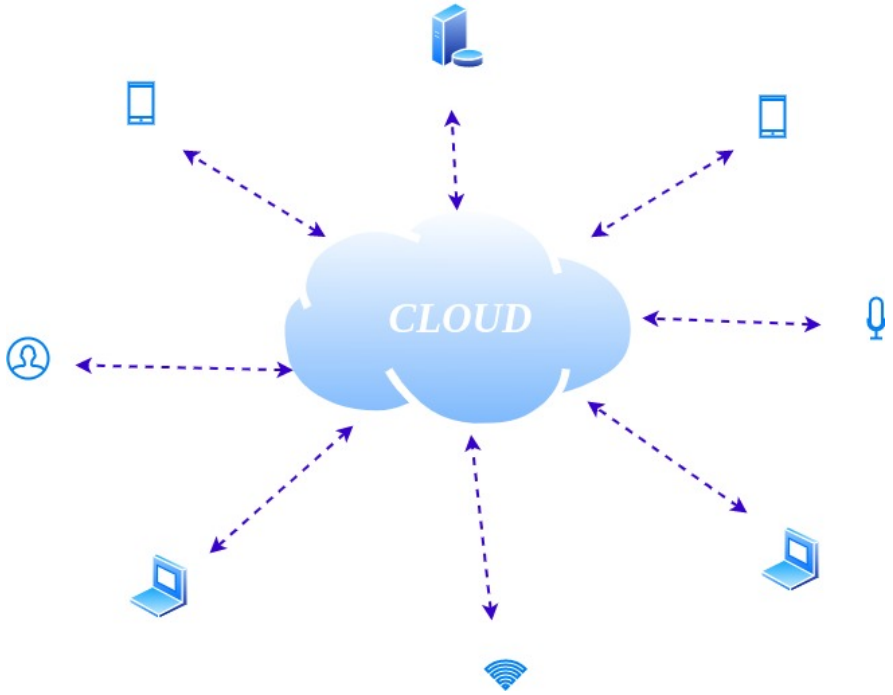

Figure 3. Cloud computing architecture

2.2. Microservices

Microservices are independent small executable modules that communicate with other microservices using messages, capable of being executed independently from other modules. A single microservice typically performs a single well-defined task, and many microservices can be used as application modules to create an application, called a microservices architecture. This differs from traditional monolithic applications because the traditional modules communicate using shared machine resources, and are not independently executable after development. [13]

Microservices and the IoT share some goals: lightweight communication, independent deployable software, a minimum of centralized management, and independent development techniques and technologies [14]. The message-based communication makes the microservices architecture easily distributable since the messaging can naturally be done over the network. These distributed applications can be easily scaled, maintained, and containerized [13]. The traditional server-side applications run in centralized cloud, but the utilization of microservices allows getting the data processing done closer to the end user in the edge or fog layer, decreasing network load, shortening latency, and improving security.

2.2.1. Containers

Containers are also called operating system-level virtualization. They are based on a concept of namespaces, which means that the resources available to a process appear to them as if they have their own instance. These resources are only visible to the processes within a namespace and processes from other namespaces cannot interfere

with other namespaces. Nevertheless, processes still run on the host system and thus all containers use the same kernel, so containers run only on specific hosts. [14]

Containers can be deployed with all of the required dependencies enclosed, without worry of conflicting library versions. In addition, the container is the only environment where the enclosed processes run, so running tests on the whole container prevents execution environment problems. Scalability is also better with containers, since starting and stopping multiple instances of each service is possible due to low overhead of container virtualization, and microservices are highly decoupled. Due to these reasons, containers are a good choice to meet the fluctuating demand in cloud and edge layers. Solutions for managing containers, like Docker, also can fully automate deployment. [14]

2.2.2. Orchestration

Container orchestration is the method which allows the user to define how the containers should be coordinated in the cloud when a multi-container application is deployed [15]. It defines the initial deployment of containers and the management of multiple containers, meaning the user can create more instances of a service on the fly to replace crashed containers or to balance traffic, and in the distributed setting, orchestration also can move containers from server to server. In addition, orchestration tools like Kubernetes abstract many aspects of deployment, and for example, abstract the network aspect by providing the concept of a cluster. The containers within a cluster essentially belong to the same subnet, and every container is reachable from every other container within a cluster, even though they may not be normally reachable from the servers running the cluster.

2.3. MERN

Web-based client applications for IoT systems are typically built using several different technologies. One popular web technology combination is MERN, which stands for MongoDB, Express.js, React.js and Node.js. This technology stack allows working with only one programming language, JavaScript [16], by making the development work easier due to focusing only on a single programming language.

React.js is an open-source *'JavaScript library for building user interfaces'*. Originally created by Jordan Walke, a software engineer at Facebook. Nowadays, React has thousands of contributors to keep up with the pace [17].

Before User Interface (UI) frameworks, or libraries such as React.js, web applications followed a round-trip model. While browsing through a website, all the new content was fetched from a server every time the user, for instance, requested a content update. In this way, the most of the logic were on servers, and browser was used to render HyperText Markup Language (HTML), and execute small scripts. There are still many applications following principles of this older round-trip model.However, a new, and prevailing trend is to build applications to follow Single-Page Application principles. Once a HTML document is retrieved from the server, it will not be reloaded. For example, in React, Virtual Document Object Model is a

Javascript object, and it is modified by JavaScript as well. Hypertext Transfer Protocol (HTTP) to requests to Application Programming Interfaces (API) returns data, which is then used in rerender of a content [18].

One very convenient feature in React is Javascript Extensible Markup Language (JSX), which is a syntax extension to JavaScript, but looks very much like Hypertext Markup Language (HTML). In this way, embedding JavaScript variables inside JSX tags could be done, as well as functions, conditions or other JavaScript. In addition, JSX includes built-in functionality to prevent injection attacks. Every time JSX is run in browser, it has to be compiled down to a React function call, and finally React elements [17].

MongoDB is a database where data is stored in documents, or in objects, instead of rows and columns as in traditional Relational Database Managements Systems (RDBMS). Collections in MongoDB is equivalent to tables in RDBMS. NoSQL databases, like MongoDB has created for the need, and to perform better in some areas than RDBMS, for example, scalability and speed [19].

One popular backend web application framework these days is minimal and flexible Express.js, which is built on node.js, and can be used to share static content as well as in routing. Express uses middleware functions for handling request objects, and sending results back in response object [16].

Node.js is an open-source, cross-platform runtime environment for running, and interpreting Javascript code. Utilizing the same engine that is running in Google Chrome's. Node.js is single threaded, and can work in a blocking, or non-blocking way. Blocking means that a thread that starts an operation must wait until the operation is finished. Non-blocking is asynchronous, which leads to better performance and speed, since other operations get executed while waiting one to get finished [20].

2.4. Teaching IoT

Digitization of campuses and other educational institutions is propagating all over the world. One form that embodies this is IoT, furthermore, there are also various different ways to utilize it. For example, to empower professionals, or students in academic fields. IoT may even be a start of something remarkable in educational institutes by reforming old methods of teaching, and therefore, leaving the valuable time of professors in something important, and by automating these everyday tasks. From the students' point of view, learning can boosted with a simple add on for the course, and make learning compelling rather than by offering only the necessary to survive the course. [21]

As an example of an IoT infrastructure, IoT platform was developed for thousands of students in [22]. The hardware was a tailored sensor device, *SenseBoard*, containing various electrical components such as LEDs, sensors, and a button. *Sense*, a brand new visual programming language and environment, giving students a flying start. And lastly, cloud-based application, using servers of university. That further connects all sensing boards, and enable messaging between these devices. *Sense* environment helps students without prior knowledge in programming to develop their first program in short time, and rapidly scale up. As a result, students found good understanding in

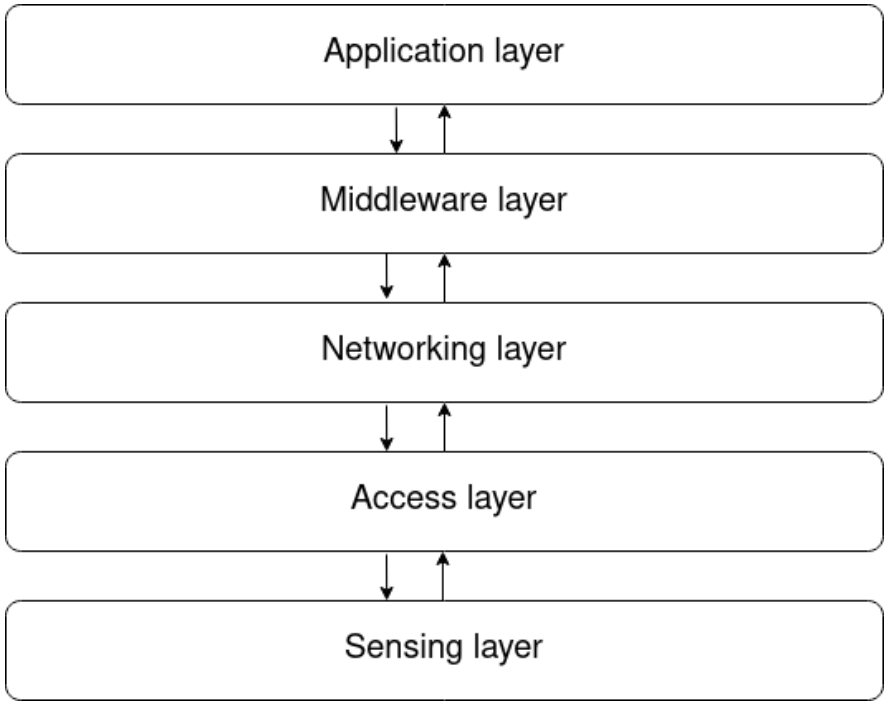

Figure 4. Five layer architecture used in [23]

programming basics at the end, and the course was highly valued based on the surveys. Even though, the drop rate remained the same than in earlier years [22].

Since IoT is not just a one thing defined in a one way, there can be several ways to form IoT systems. In [23], a system with five layers was built for educational purposes. Depicted in Figure 4.

Functional blocks of IoT system defined in [24]:

1. IoT devices for producing data, or to function based on the data.
2. Servers for processing data.
3. Databases for storing data.
4. Applications for visualizing data.
5. Communication between these blocks, or within a block.

An example of key components covered in two courses that were offered in Lander University. *Embedded programming* and *Computer hardware* play a huge role in IoT, thus getting familiar with the basics of programming and hardware functionality will help a lot in further studies. In addition, since IoT system are distributed, *Networking and communication* as well *Distributed computing* have taken their places as a key topics to cover for a comprehensive understanding of IoT systems. [25]

Due to the size, and constantly changing concept of IoT, it might be difficult to choose the most valuable topics to cover in a course. A list created by [26] is a good place to start, and it provides a great foundation for a course. The first subject to cover is finding a way for students to think about IoT applications. A simple card game may do the job, and help students get started. Other subjects to consider are the

hardware to use, communication protocols, programming languages, non-traditional user interfaces, cloud provider, and data analysis. In a list created by the authors in [26], advise to go with widely used technologies that has large communities.

Unfortunately, IoT is not just an awesome, or world saving new thing, but challenges within the field of IoT can be exigent. This includes areas such as cloud computing, instructional technologies, mobility applications, security and privacy, research computing, quality and ethics, and financing [21].

3. DESIGN

In this section we present the project use cases and their requirements, project design divided into two sections, and a short analysis of the design.

3.1. Project Objective

The main objectives defined for the project are:

- Receiving and visualizing multimodal sensor data from an IoT device.
- Two-way messaging between backend and an IoT device.
- Deployment into a public IoT infrastructure.

Analysis of these requirements leads to IoT system architecture with the characteristic three layers. The cloud layer saves and visualizes sensor data, and holds client applications. The edge layer holds gateways that relay messages between the perception layer and the cloud layer. Finally, the perception or device layer holds various IoT devices that send data to higher levels in the architecture.

3.2. System Functionality

Functionality of the system is largely governed by the use cases, briefly described in the table below.

1. Students can use their embedded device to control their own avatar in a game by sending wireless messages.
2. Messages that are sent from the client application can be received by the student's device.
3. It is possible to use an embedded device to wirelessly send sensor data to the client application.
4. Sensor data can be visualized, for example, to help design a simple artificial intelligence for gesture detection.

Next, the different design requirements will be discovered based on the objectives and use cases, in the cloud, edge, and device layers.

The messages and sensor data need a database for storage, from which it can then be viewed. A very flexible way to view the sensor data and various client applications would be a web UI. For ease of use, the web UI will have to have simple user authentication to select which messages should be viewed. Finally, the third requirement about easy deployment can be satisfied by running the cloud layer in Kubernetes, for instance.

In the edge or network layer, there are gateways and other possible gateway-related infrastructure. A gateway is the component that interprets messages between

two networks. Multiple gateway devices are needed if the deployment is required to cover a large area, because of limited signal range. The responsibility of the support infrastructure for gateways is to provide a way for them to easily and scalably send messages between system components, and this can be achieved using a publish/subscribe mechanism.

The final layer, the perception or device layer, has multiple IoT devices that contain some sensors, and are capable of two-way communication with the gateway. With this project, we will create a simple application in this layer to test various aspects of the resulting system.

3.3. Analysis

The system is designed by following the latest state-of-the-art, thus one huge backend service in the cloud layer is divided into multiple smaller and independent microservices. To facilitate the workload caused by maintenance, and the process of developing new system features in the future, we aimed to find as many entities as possible. These entities can be scaled up or down, as well as removed without causing the system to crash. Furthermore, new ones can be created and integrated into the system. Despite of the advantages it is not the most efficient solution for a system of this size to be divided into so many components. Luckily, the only focus was not to build as efficient a system as possible, but increase the educational value.

As mentioned earlier; four different networking models for IoT have been released in 2015. [1] One of these models is *device-to-gateway* model, which is the best fit for this project. The gateway is possibly a software running on a piece of hardware. However, the gateway can also be run on a laptop without a need to have hardware for wireless communication, since the IoT devices can also use the Universal Serial Bus (USB) connection. Nevertheless, the gateway component is used as a common point for perception layer devices to send data.

The users will be programming the perception layer devices, and so are creating the messages to communicate with the gateway. There is no given format for these messages, but in this case it will have to be as user friendly as possible, for both the ease of use and the resulting reduced chance of user error. Plain text format comes at a cost of longer messages in a memory and capability constrained environment. For this reason, the usage of existing formats, like JavaScript Object Notation (JSON), is not preferred due to large overhead compared to a more custom approach. When there are multiple gateways within signal range from each other, there is some concern about where the messages should come from and go to. This could cause message duplication if not handled correctly.

3.4. Architecture

Based on the analysis, and topics covered earlier, the architecture will be as follows. There are three layers of IoT depicted in fig. 5 that represents the project architecture. These layers can be divided into three different locations as well. The *perception-*

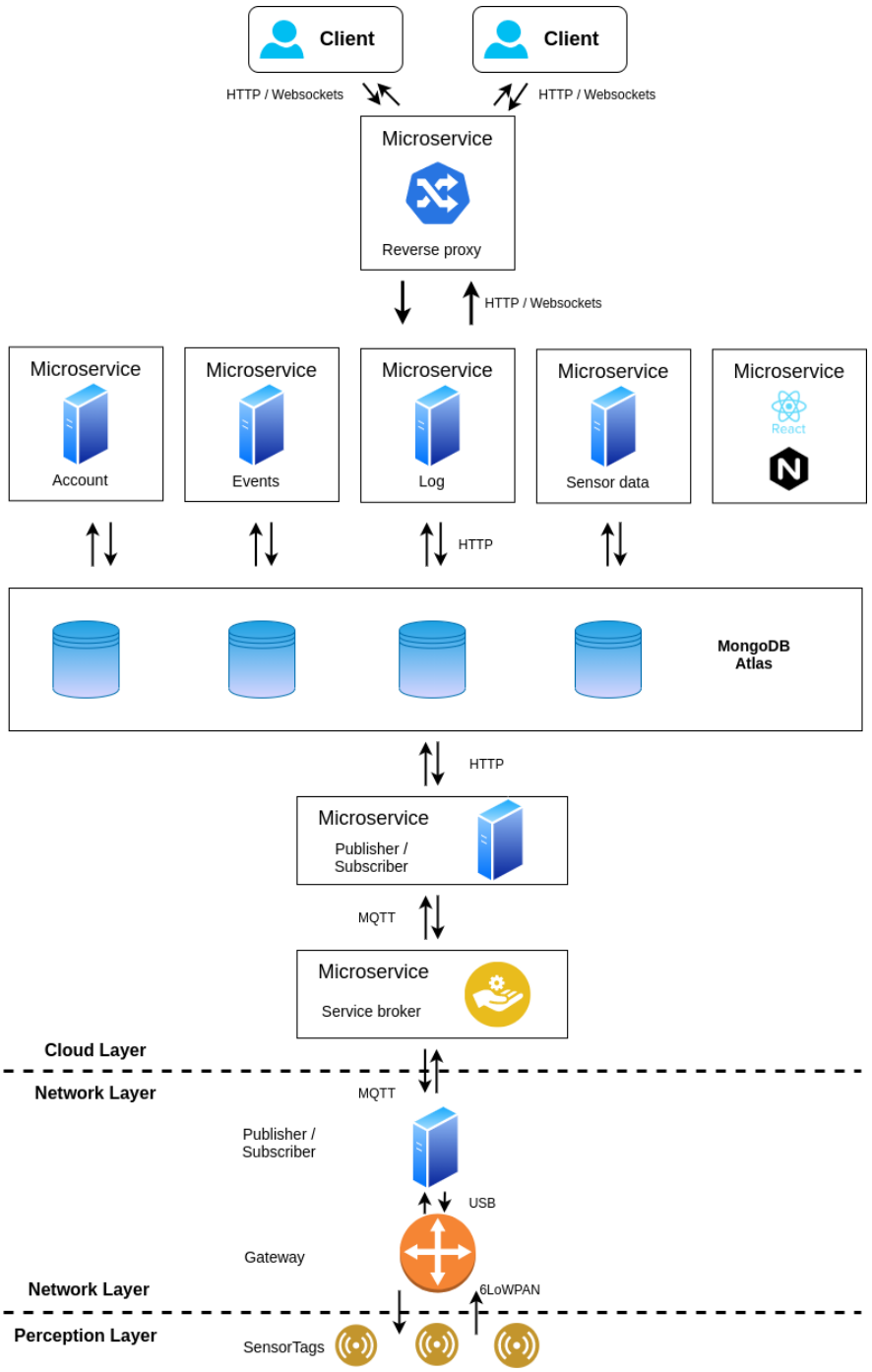

Figure 5. Project architecture

layer devices and *network-layer* components are local, and the rest of the components, *cloud-layer*, are located much further.

The topmost layer in the project architecture is a React.js web application (Figure 5), which runs in a browser. Nginx, a web server among other things, is responsible for sharing React.js static files. Nginx ingress controller is a gate to the application, it exposes HTTP routes allowing an access to the Kubernetes cluster from outside world. Furthermore, Ingress controller is responsible for routing request to a correct microservice as well. [27]

The second layer that contains the back-end server, is not just one server, but many. *Account, Events, Log* and *Sensordata* services. Each different concept forms its own entity, a microservice, which is directly connected to the databases.

In the middle (Figure 5), there are databases. Each microservice will have its own database, which is one of the principles when building microservice-based applications. Databases are hosted by MongoDB Atlas, and further, AWS (Amazon Web Services). The amount of data will be relatively small, at least at enterprise-level, hence in this project each microservice has its own data collection, but not database. In this way, scalability will remain good, and later if needed, switching from this model to multiple database model, can be accomplished with little effort.

Below the databases, there are two components. Both components are able to pass data in both directions, however, they focus on different types of data. The last cloud layer component is a service broker node for managing traffic between publisher/subscriber nodes as well as for managing access into the system.

Only two components are located on the network layer, right on the edge of the network. A publisher/subscriber, which is able to process data, and therefore decide whether it is worth sending data any further. The second component is a gateway that works as an point for the data to move to and from the perception layer. The number of gateways is not limited, but there has to be at least one.

On the bottom in the project architecture is the *perception layer*, or simply, the layer that contains sensors, actuators, or any other smart devices. They can communicate with the gateway, or with other devices. Devices in this layer are the source of data.

Each microservice (Figure 3 5 will be wrapped in a container, which later can be scaled up or down by some orchestration tool. Or even create a new service instance to replace the crashed one.

4. IMPLEMENTATION

After the design phase comes implementation, but even before that, we had to have a comprehensive conversation with the customer about technologies, devices, and patterns to be used in this project. After this conversation, we ended up to choose CSC Rahti container cloud to be the backend service provider in the *cloud layer*. On the *network layer*, MQTT event-based communication was selected as the winner, since its popularity and widespread use in IoT. A bit challenging part was to decide how the gateway will be distributed to students, or should it be something else from a distributable package. The following describes the implementation of the backend in general, but since the focus of this thesis is in IoT and communication between components, these areas are in covered in more detail.

4.1. Web Client

The client application is built using the web UI library React.js. The appealing look comes mostly from the components of the popular UI framework, Material-UI. In addition, several other libraries will be utilized in this project as well. React-router, a collection of navigational components, added to this application for making components render in the wanted order, as well as giving the correct URL to avoid an otherwise mandatory refresh on the browser while navigating. Because of the nature of React.js, it is a single-page application (SPA), meaning that there is only one actual HTML file, embedded with JavaScript. As a result, moving from one view to the next is technically just navigating in one HTML file.

To display personal data, it is required to have some way to link a user, or a group, with a corresponding SensorTag. Even though the data produced by the SensorTag is not sensitive, we decided to implement password authorization. As mentioned earlier, nothing sensitive or identifying information is stored, but proper access control reduces misuse of accounts, and therefore, students are able to finish their projects with little unnecessary hassle. The downside here is that the registration is not limited to students only, but every internet user is able to make an account, and login to the application. This could be fixed by adding an extra field, where every student should input a passcode given by the person in charge of the course, or every student gets a pre-created account. But on the other hand, we do not want to make this application to be an extra nuisance for the teacher.

The feature of sending events back and forth between a SensorTag, and a server located in class, has been a part of the testing phase of the course in previous years. That feature was implemented this year as well, but in a different way. This time, every event will be pushed from a server to the client application using websockets, and this all happens almost in real time. Like all the data retrieved from the database, client application stores log messages in Redux store. The store is available for every react component, and messages can be re-used in multiple different components.

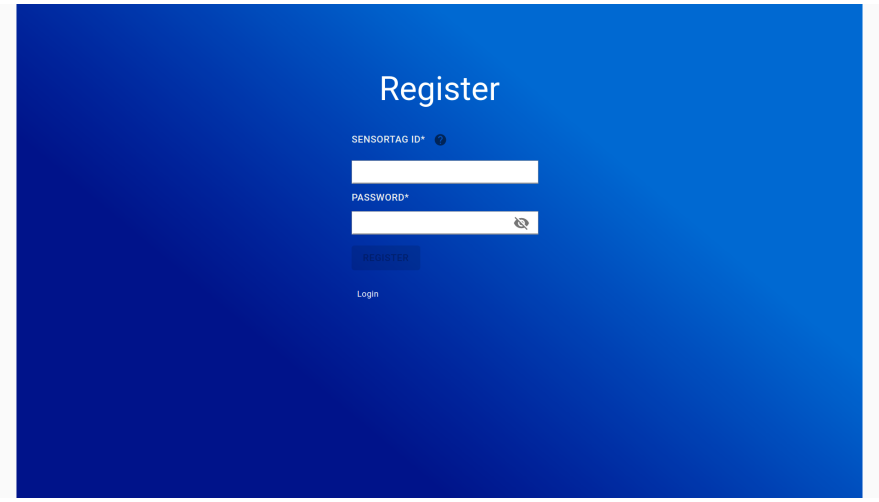

Figure 6. Application registration

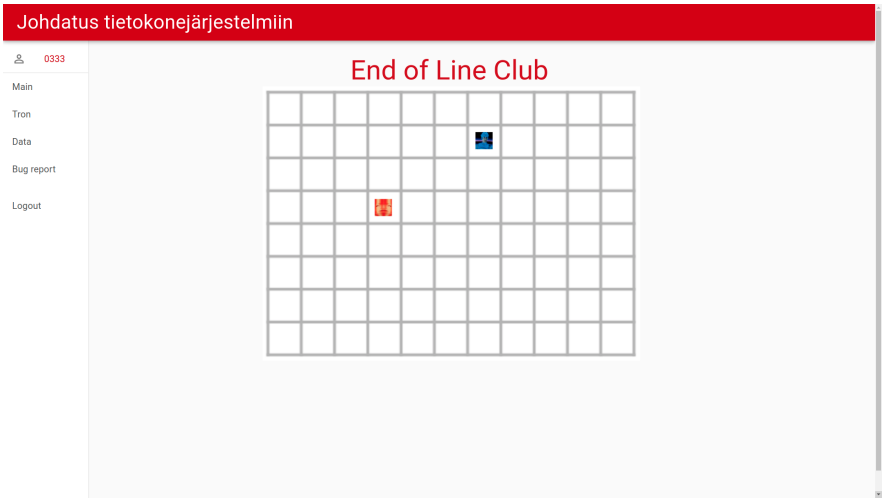

Figure 7. A simple game for having good time while testing

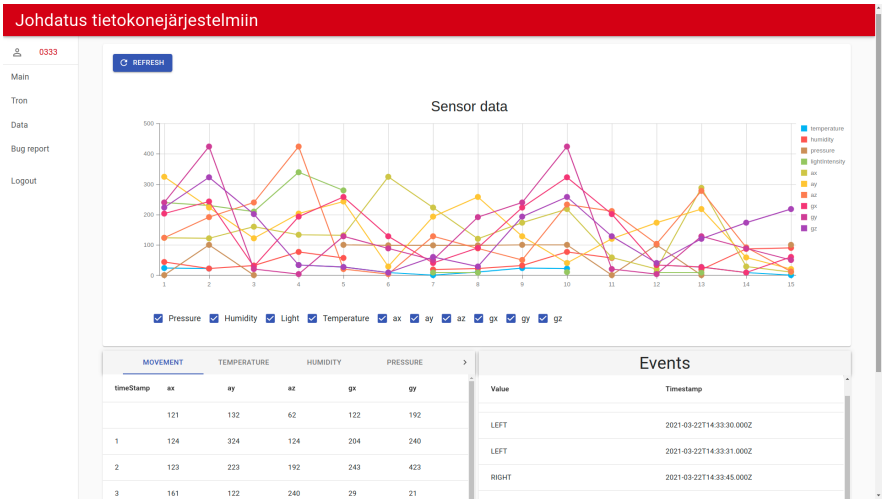

Figure 8. Data diagram, data grid, and a list of events

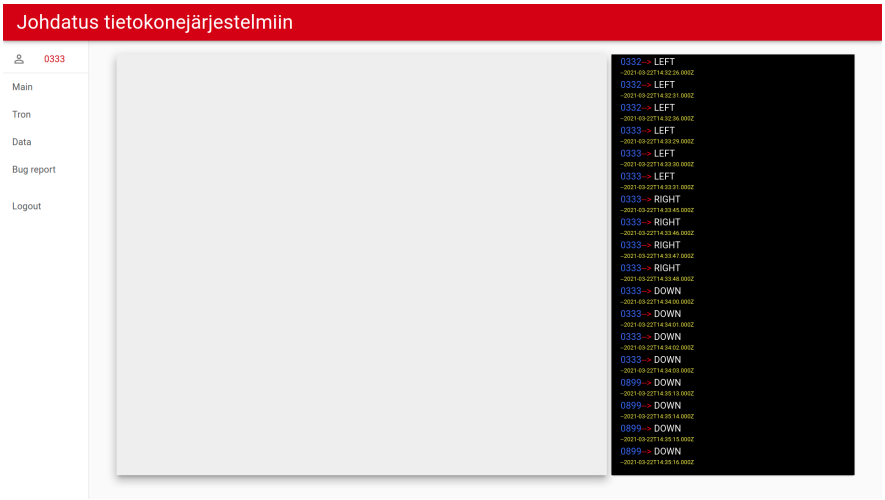

Figure 9. Message log on the right side, and a drawing board on the left

4.2. Backend

Design and implementation of a microservice-based system describes this project well, thus we wanted to find as many entities as possible that could be separated reasonably to be a microservice, even if the task was not that large. All the microservices in this layer have an Express.js server running, and the server is listening constantly incoming HTTP messages on the port dedicated to them. Even though all our server side code is written in JavaScript, one of the good things in a microservice-based architecture is the ease of switching programming languages and technologies between microservices. In addition to the table below, authentication is implemented in all microservices. Instead of calling users server for every protected endpoint, Passport.js is implemented to check the validity of a JSON webtoken. Other features are listed and explained in the table below.

All microservices listed above have their own database collection for storing and retrieving data. Some of these collections are accessible from subscriber microservices as well. Although all microservices should have a database that is not used by any other microservice, in this case each microservice has a collection or two in some cases.

4.3. MQTT

There are two subscribers and one publisher directly connected to the database. Functionality of these microservices is briefly described in the table below.

4.4. Gateway

In this chapter, the implementation of the gateway and the corresponding gateway publisher/subscriber belonging to the network layer is described. For abbreviation, the gateway publisher/subscriber shall be called the interface during this chapter. First, the message formats between each component starting from the perception layer, up to the

HTTP Microservices	
Events	Broadcasts SensorTag events to all clients that have established a connection with Events-server as well as forwards game results to the database. Socket.io library is the tool used to achieve real-time broadcasting. The library utilizes websockets, and therefore, enables data to be pushed in real-time to all clients.
Eventlog	Ultimately, the only task for this service is to keep listening changes in the message collection, and almost immediately respond to that change by pushing the data to all clients.
Sensordata	Retrieves sensor data from the database.
Users	Authorization and Authentication server, which is responsible for registration, and login by creating web tokens. These tokens contain encrypted information about that account.

Table 1. HTTP microservices

MQTT Microservices	
Sensordata	Responsible for storing raw sensor data in a database. And it subscribes a message with a topic of *sensordata*
Eventlog	Stores events (UP, DOWN, RIGHT, LEFT) to the database. Retrieves game results from the database, and forwards to broker. And it subscribes a message with a topic of *event*
Broker	Forwarding of all the valid data.
Gateway pub/sub	Forwards and parses data to and from the gateway connected via a serial connection. Has a rudimentary UI to show state and errors, and to allow manual intervention.

Table 2. MQTT microservices

broker, are given. Second, the main functionalities of the gateway and the interface are explained. Finally, there are some notable mentions of the implementations of both components.

The main flow of data goes from the perception layer to the cloud layer. The data format for sending data from the perception layer to the gateway components was selected to be the user friendly comma separated key-value pairs, described in Appendix 1. The messaging between the gateway and the interface is done using the Universal Asynchronous Receiver/Transmitter (UART), and messages from gateway use a delimiter to signify end of message. In the gateway, the address of the sending device is prepended to the message as a 16 bit little-endian integer, and then the compound message is encoded using an escape character scheme to ensure the UART delimiter character only appears at the end of the message, where it will be appended to after encoding. The gateway is made to forward arbitrary bytes, so the escape character scheme works by selecting two characters, which shall be called the escape character, and the substitute for the UART delimiter. In the encoded message, an UART delimiter preceded by n escape characters will be replaced by $2n + 1$ escape characters and a substitute character. A substitute character preceded by n escape characters will be replaced by $2n$ escape characters and a substitute character. Any other string of character preceded by escape characters will be unchanged. Messages sent in the other direction have a fixed length and no encoding.

The interface component receives messages from the gateway, and sends the data via MQTT to the MQTT broker in JSON format. The game control events are sent right away with the device ID, message, and a corresponding time stamp according to the interface, to the MQTT topic 'event'. The sensor data from multiple messages is collected in the interface to a session, and which is then formed into a message when the session is ended. This message is sent to the MQTT topic 'sensordata', containing the device ID, the time stamp for when the session has been started, lists of values for each possible sensor for when it was recorded and a none value where the value is missing, and a list of time stamps for each row of data, either sent from the device, or otherwise relative to the session time stamp. Messages that come from the broker to be sent via the gateway, are received on MQTT topic 'game', describing the ID of the target device, and boolean values for all messages that can be sent to the device.

There are a few important functionalities in the gateway. First of all, it is possible to ask the gateway to respond with information about the gateway, and this is used to identify a gateway device and make sure the serial connection works. The second response request is the heartbeat functionality, where the gateway is periodically asked to respond with a certain message to detect crashes, and the gateway uses it in reverse to find if the interface is unreachable. Any other messages that don't start with a specific string of bytes are sent via 6LoWPAN to the address given in the 16 bit little-endian integer in the beginning of the message. The final major function is the message buffer. It is used to store messages, to make sure the probability of a message being dropped is fairly low, and temporarily store messages if the interface is unreachable, for example.

The interface component does many tasks. A major part of the design of the interface is that it will have to work connected with both the gateway and a client device if no gateway is present, enabling a student to use the system with a single device. We will begin by describing how the interface works when the gateway is connected, and then conclude with how it works with only a perception layer device. First off, in the

same order as during startup, the interface automatically connects to the gateway by listing all serial ports with the associated Plug'n'Play (PnP) IDs and automatically tries the ports with PnP IDs matching predefined patterns. Each matching port is tried by requesting identification from the connected device, and if identification is received and matches a pattern, the connection is completed. If the connection to the gateway is broken at any time, the interface automatically attempts to find the next gateway device.

After connection, the interface listens simultaneously for the gateway serial connection and terminal input from user. The user can send messages to perception layer devices manually, force reconnection, and change some other visual parameters. Meanwhile, the messages from the gateway are decoded and displayed in a human readable form, and then parsed according to the data format. Any errors from the parsing phase are displayed in the terminal. Otherwise, the data is either stored in the session cache or sent to the MQTT broker. Meanwhile, once every 15 seconds, the heartbeat request is sent to the gateway to check if it has stopped working, and any message send requests that come from the broker are forwarded to the gateway if the interface has recently received messages from the device described in the send request.

As mentioned, the interface behavior changes slightly when the user wants to connect a perception layer device to it, instead of the gateway. The automatic port connection still is used if the user wants so, but identification and heartbeat messaging are not used because they likely are not implemented in the device, and they are not required in such a manual use case. Also, since the messages don't come via 6LoWPAN, the device ID has to be added to each message as a key-value field. This way, the interface works very much like a simple MQTT enabled serial terminal with message parsing.

We will finish this chapter with some implementation details, starting with the gateway. In our case, the course for which this backend is created for, uses the Texas Instruments CC2650 SensorTag as the perception layer device. The SensorTags use a custom communication library, so the natural choice for the gateway is another SensorTag. Briefly, assuming some knowledge of the SensorTag code, to make it possible to receive messages while sending data over UART, two different tasks are used. The Texas Instruments Real-Time Operating System (TI-RTOS) implementation of a ring buffer is used to store messages, which creates a small possibility for a race condition, solved using semaphores. Because TI-RTOS is roughly a real-time operating system, event handler functions have a very limited execution time, which is solved by setting the program state machine to different states in the handler, and the set state can then be responded to in a task.

The asynchronous nature of the interface program makes JavaScript a very attractive language. The actual program is divided into five modules: The UART module, MQTT communicator, port finder, message parser, and utility module. To increase the usability of the interface, a simple configuration file is used to store important constants and the different types of data fields, which can be easily customised to match the theme of the final project of the course.

4.5. SensorTag

Crucial to the evaluation of this project is the SensorTag in the role of the perception layer device, which was implemented during this project. The code is based on the course template, and includes a gesture detection algorithm used to play the game in the web UI. During this chapter, the different features of the SensorTag are described.

The SensorTag is a state machine with six states for sending data at different rates. These include two fast modes for sending sensor data every 50 milliseconds from the pressure sensor, or from the pressure sensor, thermometer, accelerometer, and gyrometer. The others include one slow mode for sending pressure data over a longer period of time, a mode where hard coded test messages can be cycled through manually, and a mode to try to use the gateway ping function to send data sequentially when messages are lost to interference. The rest of the messages are sent using the four-way gesture detection: These normally send the avatar movement commands to the game, but with a modifier they send the sensor data session control commands, and the ping message. At any time, the SensorTag can receive and display wireless messages to see that the messages sent from the gateway are received.

5. EVALUATION

What comes to the evaluation, the pandemic has taken us to a situation where we need to make a lot of changes, especially in evaluation. In this chapter, we cover the project evaluation, and all the tests performed. We have managed to divide these in two different categories; system performance testing, and usability testing.

5.1. Evaluation Plan

A major part of evaluation are the test cases. Test cases will have to initially demonstrate that the use cases are satisfied. To be able to test the use cases, the registration and login has to be done. After this, use cases 1 and 2 can be tested by attempting to play the game. The player should see the game on the website and be able to win, lose, and receive the respective messages from the game. The testing of use cases 3 and 4 is then done by sending sensor data, and seeing it displayed on the website. The rest of the use cases can be tested by picking messages that will cover much of the code associated with each case, and sending the messages from a client SensorTag. The test cases are given below.

1. Registration and logging into the website.
2. Use cases 1 and 2: Send control messages, win and lose the game, see if loss and win events can be received from game.
3. Use cases 3 and 4: Send sensor data. View sensor data on website. Is the graph usable?
4. Sensor data session start and end functionality, with and without ping.
5. Error handling of mistyped message format.
6. Handling of unintended input messages.

During evaluation, the server components will be deployed to CSC Rahti and the gateway will be running with a Raspberry Pi 3 model B.

5.2. Experimental Setup

The setup for tests and evaluation is as much as possible the same than it will be later in production. On the cloud layer, every container that are communicating using HTTP, are running in the container cloud CSC Rahti (Figure 5). Databases, or in this case, database collections of this system rely on MongoDB Atlas. On the cloud layer, publisher/subscriber components are running on RPi 3 (Raspberry Pi 3).

For testing purposes, gateway, and the publisher/subscriber component attached to it are utilizing the RPi 3 as well. This component is located within a radius of ten meters from the sensor device.

At the *perception layer*, test are performed using SimpleLink CC2650 SensorTag. To be exact, three of them sending messages at the same.

First three test-cases are carried out by a system user. Which is us in this case. Login and registration was tested by using the application UI, and manually check valid and invalid credentials. Test-case number 2 included one of us to use Sensortag to produce some data, and another one inspecting result as well as in test-case 3.

The sensor data session message functionality (4), gateway interface error handling (5), and UART delimiter escape character handling (6), are tested by sending various predefined messages from a client SensorTag. The effects are then evaluated by checking the results on the interface against the intended behavior described in attachment SensorTag message format and having the received messages match the sent ones. The test messages are selected to test the detection of errors in the key a few times because it is the same for any key-value pair. Value error handling of each different key is tested as each one uses a separate handler. Lastly, the UART message delimiter escaping is tested with known problematic patterns. The wireless library used by SensorTags has worked well, but the maximum message length will be good to know for future reference. Finally, some messages are selected to try to intentionally cause problems to the gateway.

5.3. Results

A major use case for the system is to view sent sensor data on the web UI and use it to gain insights for tasks such as developing automated gesture detection. To test this aspect, three graphs are generated and analyzed for usefulness in this section. The time scale is represented in milliseconds.

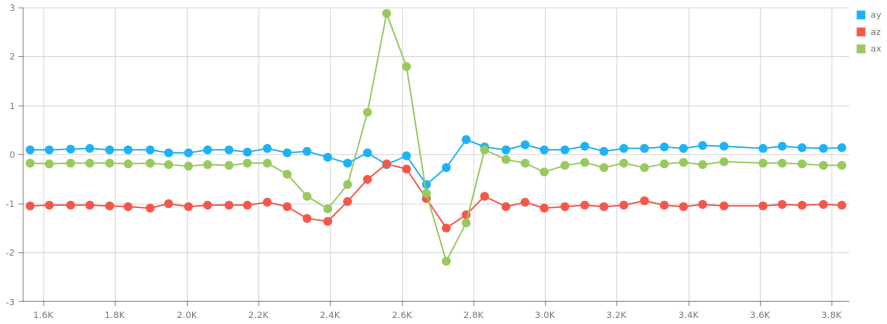

Figure 10. Accelerometer values when quickly swung right and back again

In the first demonstrative graph (Figure 10), the SensorTag is in quick succession moved rightward and returned to starting position, with the screen always pointing upward. During this maneuver, the SensorTag is sending sensor data in about 50 to 60 millisecond intervals, surprisingly well corresponding with the task sleep being set to 50 milliseconds in between each run. As the keen-eyed reader you might have noticed the missing point at roughly 3.55K, which is due to the 6LoWPAN packet being dropped between the SensorTags, as errors are quite frequent there.

The graph clearly shows constant acceleration before and after the gesture, with the z axis prominently showing a value of -1, meaning that the gravitational acceleration is along the z axis, and the unit of measure is acceleration divided by standard gravity.

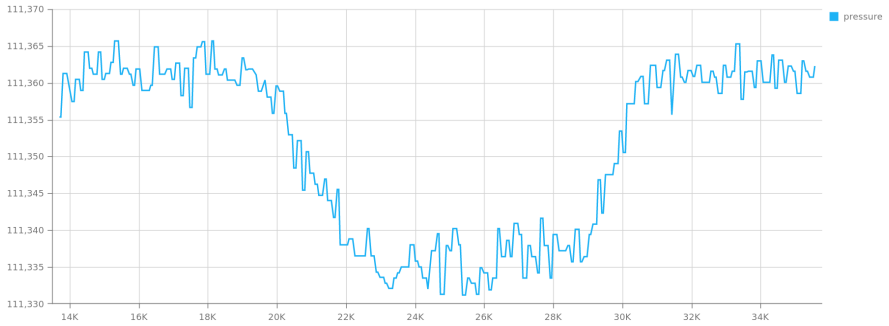

Figure 11. Pressure change when SensorTag is lifted 2 m and down again

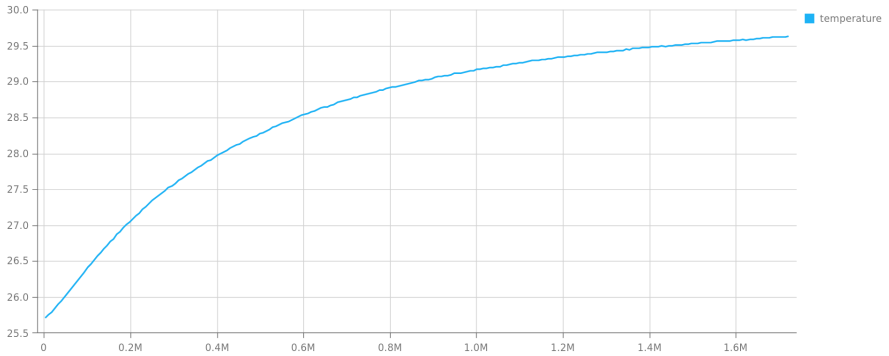

Figure 12. SensorTag temperature curve

In the actual gesture, the largest reaction is seen along the x axis. The initial negative acceleration is due to the SensorTag experiencing a pull to the left while it is being pushed to the right, and this motion is arrested, and reversed, with the sharp peak in the middle of the motion corresponding to the SensorTag experiencing a pull to the right. The SensorTag is then brought to rest with the last negative experienced acceleration roughly matching the first negative peak. This analysis tells us that the positive x axis could be thought of as pointing to the right using the SensorTag's frame of reference. It can also be observed that the middle peak is approximately twice the height of the smaller peaks, which is largely due to it being easier to apply larger force while the movement is larger and more uncontrolled compared to the start and stop situations. Finally, the correlating gesture along the z axis suggests that the movement was done at a slight upward angle.

The SensorTag also has a barometer, which can be used to sense changes in altitude. This is demonstrated in the second graph (Figure 11), which shows pressure data from the SensorTag initially resting on the ground and then being lifted two meters and being brought back down. While the barometer is not calibrated, as witnessed by the abnormally large values on the left, the increase in altitude is clearly visible as the drop in pressure. Representing this data visually would give the user an idea of the dispersion of the measurements, vital for using the sensor in this way.

In the final graph (Figure 12), the SensorTag's internal temperature is plotted to show how it stabilizes. The data is gathered over a period of 28 minutes, which is closer to using the system as a real IoT backend for a SensorTag that is monitoring room temperature, for instance. How ever, this sort of graph could be used to predict the time a SensorTag has been running, if the graph is predictable while the SensorTag

is being used, or to find a model which could be used to predict the final running temperature.

When the user comes to the web UI, they are greeted by a simplistic log in page. A link on this page leads to the registration form, which is very similar to the log in page, only requiring the SensorTag ID and a password. Currently only IDs four numeric characters long, starting with a zero, are allowed, and there are no restrictions for the password since the service is not hiding anything sensitive. On every tab of the web page there are the unchanging elements familiar to many users: The page title, username of the current user, and a vertical menu for different tabs. The main tab has a clear log of recently received game events, which can be used to detect being logged on using the wrong ID or sending messages using a different ID.

Continuing to playing the actual implemented game, the avatar does indeed move when an event is sent from the client SensorTag. The delay between sending the message, and seeing the avatar react to it is surprisingly small when taking the architecture into account: Approximately under half a second, feeling almost real-time. When an event causes the avatar to hit a wall, a subtle pop up is shown in the UI, and a message is received by the client SensorTag informing about a lost game. When the avatar moves to the villain's space, the user is congratulated in the UI and with a message received by the client SensorTag about the won game. What happens after a game has been won might confuse the user because the game board is not randomized again, and instead the avatar is left in the villain's square. This happens because then being able to have the game resend the win message makes it is easier to debug the SensorTag, and the board is randomized only when a wall is hit or the page is refreshed. It is notable, that if the game is running on multiple browsers, the SensorTag receives messages from all instances, which are initialized at a random position. Currently the backend only shows error messages in the interface which isn't normally visible to the end user, so debugging can be difficult if messages are sent in wrong format.

Test case three is passed: It is easy to send sensor data to the backend, and display and use it effectively. The graph is clear albeit the point markers sometimes are too large, the graphs of different sensors can be selected using radio buttons to help view data from different scales, and the values of points can be examined by hovering with a mouse and by using the data table. As seen in the data analysis section, the graph is highly usable for examining data in order to find properties of gestures and other interesting aspects. Notably, using wireless messages to log sensor data is much faster than the previously used methods: Printing values to the debug console or even utilizing the debugger watchpoint functionality. For example, this helps in gesture detection design because with it the seen sensor data is gathered closer to the true measurement frequency of data seen by the gesture detection algorithm.

The interface program between the gateway and the MQTT network is meant to be somewhat user friendly. It automatically finds usable ports, and if required, automatically connects to potential gateway SensorTags. This has been tested to work on Linux and Windows 10, and the Plug'n'Play ID detection can be easily refined at a later date. After the gateway is connected, the user can type and send messages globally or to specific SensorTags, and the interface receives messages from the serial port. If the SensorTag is disconnected, the port to it will be automatically reopened when the SensorTag is connected again, making it easy to reboot a crashed gateway. The received messages are shown completely, and message format errors are visible.

The sensor data session logic works as intended with the ping. The results of the mistyped message error handling are shown in appendix 2. To summarize, many surprises came from the incorrect usage of two JavaScript functions that convert strings to integers and floating point numbers, as they read into a number only the first numeric characters before encountering a problem. In addition, two related problems were found in the key-value parser: Passing two values to a key value handler passed an undefined object into it.

Appendix 2 also has some messages used to test the wireless library and the escape character scheme used for guaranteeing the UART message delimiter will not occur inside the message body. It seems that the wireless library can send messages that are only at most 116 bytes long, and this inadvertently revealed that the method of suggesting valid keys when an unknown key is received doesn't find a suggestion when the key is too long. The most critical bug found was that the gateway SensorTag crashes when a message of zero length is received. Lastly, the UART delimiter escape character is accidentally removed from the end of message because it is mistakenly not added there in the decoder.

6. DISCUSSION

In this chapter we will discuss the limitations of the current implementation, reflect on what we have learned during the project, and introduce possible future work. Furthermore, go through changes and enhancements we could have done better, or just differently.

6.1. Limitations

This section will discuss about the limitations of this system. There has always been some main task in the course, *Computer systems*. This year it was a game where an avatar is moved by SensorTag messages. Everything works well if just one client application is open. In other words, if the same account is active in two different windows, a feedback is sent back two times. This same behaviour causes SensorTag to get message if the player actually did not hit the wall or caught the villain. This disruptive bug occurs when a player does either of those two things. The one to blame is an window that has accidentally left open. Therefore, this system does not allow more than one client at a time to work properly.

A use with a mobile device is possible, but it comes with great limitations. Only some of the UI components are responsive, and looks good on a small screen. Some of the components are not even visible, but unintentionally slides behind of an another component. Hence, it is recommended to use monitor resolution of at least 1920 x 1080 to get the full experience.

Up to this point we have not encountered any intentional misuse, but there is always a risk. Everything relies on an ID of SensorTag. That ID has to be there during registration, as well as every time when sending SensorTag messages. It is a four digit code, and malicious person could create multiple accounts by using some four digit codes. It would not cause any serious harm, but it would waste precious time of the person in charge of the course, since that person would the one deleting those fake accounts to allow real students to register their devices. The idea of two-stage registration, which would have prevented such a misuse, was under consideration for some time, however, that would have complicated things more than the benefits we would have gained from it.

Real-time data is a common thing in IoT systems. This system differs from those system by allowing data to be manually retrieved from the database, whilst a real-time system would automatically push the data to the client application.

The fact remains that the ultimate goal was in achieving to build system which allows students to carry out all assignments given during the course remotely. However, it can be argued that this remote system requires too much extra effort from students to get it up and running on their own hardware. But a little closer look reveals that the amount of extra work is not overwhelming at all. In the contrary, we have written configuration files to start all services with a single command. Thus, only task left for students is to install Docker on their computers, and kick off the containers. A preconfigured gateway could be obtained from the university, and all that remains is to plug it into a computer.

6.2. Reflections

To begin with, we started this project without having an exact plan. A plan that would include functional requirements and visual appearance of the UI, endpoints, or the number of microservices. We have been adding more features on the fly, which has caused the system be unstable from time to time. However, we have been able to restructure the project, and fix all the bugs that have appeared during testing. A great help has been *Computer Systems* last fall, and students who actively gave us constructive feedback. Without this prior knowledge of the potential problems we may encounter, we would probably have implemented many things in a different way. During fall semester, the beta version was tested by students. Back then, we were able to receive all kinds of feedback regarding usability, system errors, visual appearance, etc. We took this information, and made necessary changes to reach the goal. One these fixes targeted to register form. Notwithstanding it was clear to us that registration requires number zero (0) to be the first number, several students tried to do the registration without the leading zero. Thus, we added that to the input validation.

There are several ways to achieve results that look the same to users as they do now. However, we decided to approach this with the architecture described above. Performance test results are promising; data sent from SensorTag is able to reach the client application almost in real-time even though the distance can be hundreds, or even thousands of kilometers. Furthermore, a use of multiple devices simultaneously does not jam any of the components, which has been one of the issues in earlier years.

Here, we are handling time series data, so probably the best option would have been to use a time series database, for example, InfluxDB. Currently the way we send and store data is inflexible, and therefore, it forces us to make trade-offs. The data message includes fields for every sensor, even if they are not needed at the moment. Those fields get a value of null if that sensor is not producing any data. And the reason for that is a difficulty of managing timestamps.

Another tricky choice to make was the need of real-time data, and how beneficial it is for students. As it appears, we ended up to implement partially real-time system, which enables some messages to be inspected in real-time using client application, and some data to be requested manually from servers. In this way, students are able to take their time with the data just created, without losing the data while new data is constantly being received. Or another option is to do a manual lookup to the database.

Original design, and our local test environment included an Ingress controller standing between the outside world and our microservices. While deploying this to production servers, we found out that the production server of our choice does not support Ingress controllers so far. However, there was another option for exposing services externally, Routes. Hence, all services have an endpoint, which can be reached directly without a proxy server. Nevertheless, we do not see that as a big downgrade for security, since we are not handling personal data, or any sensitive data at all. Nothing would have stopped us to run reverse-proxy in front of the services, but after having to drop the Ingress controller, we just chose not to. Proxy would have give us a change for better control of the system, for example, limit the number of requests, and hiding real endpoints of services. Despite of exposing services, most of the data lies behind a token-based authorization, and the data is not accessible without a valid token.

One thing that was not entirely clear in the beginning is responsibilities of the authentication service. Currently it is responsible for creating users, and assigning tokens. It is not responsible for token validation. We chose to have a token validation in every service, to avoid all the traffic going through the authentication service. In next version, we probably would change that, and make one service to handle all of that, and reduce the duplicate code in different services.

At the end, a small disappointment was encountered. The intention was to have Service broker component running in cloud, but our container cloud provider did not have support for that. Therefore, the responsibility for running the service broker locally is left to the students. This does not affect to the goal of the project, but will require a little extra work from students who want to do this at home.

What comes to the challenges, we had a fair share of them (listed in chapter 2.4) during the project as well. From the very beginning, everything has been quite straightforward. For example, the financial part is covered by the university of Oulu itself. Therefore, quality of the course, and the cost of hardware will not be the anchor here. Furthermore each student that would like to give a change to IoT is able to do so. Instead of deploying the system on servers in university, we chose to utilize CSC Rahti container cloud, free of charge for academic purposes. After all, only encountered challenge from list is *security and privacy*. The way we would be able to identify students was unclear for a long time. At the end, to solve multiple related issues we chose to go with an approach where none of the students cannot be identified.

In chapter 2.4, there are presented possible topics for IoT courses. Compared to *Computer Systems*, we are lacking with automated data analysis and IoT application design. Nonetheless, complete project work requires data analysis indeed, but for now, it has to be done manually. Although, it is better to keep it intact since the analysis is important part of it. The rest is covered comprehensively.

6.3. Future Work

Future of this project worries us a little. After we are done with this project, who will manage these services and databases, or create new add-ons every year. How the monitoring will be carried out, and who is going to act if errors, or even warnings occur. We do not know the future of this project, but there are a lot of things that could be done to ease the workload in future.

Current documentation is written using inline-style, which is not a bad thing. Although, it is clear, and enough in many cases, sometime it may be better for the documentation to be written with a professional tool in addition to the inline-style. This would allow the project to be outlined and get started quickly.

There still are many options left to improve the usability of the system. The students will be most interested in what goes wrong with the messages they have sent, so the error messages should be visible in the web UI, or, the lowest effort change would be to have the gateway send abbreviated error messages back to the SensorTag that sent the message. The original plans for the backend included having multiple gateways to balance load and to lessen the impact of crashed gateways. The available gateways and their locations could be best advertised on the web UI if multiple channels are being used, but manually sent broadcast pings from SensorTags work as well. This

could also be used to alert the course staff of malfunctioning gateways. Currently the number of different names for sensors is limited, and some are missing, like the magnetometer. This is because the other sensors require some special adjustments to the sensor driver library code given on the course, and the students are not expected to utilize them. However, there is a possibility that, with some changes to the database and other components, the limited selection of sensor names could be relaxed by having the user choose names for the sensors they will use. Finally, an undoubtedly wanted feature would be a button to export the sensor data from the web UI as a file.

Furthermore, a pipeline would have been a major enhancement for the development workflow by leaving the developer out of the process where the latest software version is delivered. Even though this project may not get a full-time, or even part-time developers after us, preconfigured pipeline would help the course staff a lot, so the staff themselves could make quick patches to the code if necessary. In this way building, testing, merging, delivery, and the rest of the wanted steps could be achieved with very little effort, and production. Having these environments at each stage reduces the time required for configuration. For example, URL is different in production than it is in development, and switching URLs back and forth is absolutely redundant.

Even though loggers are implemented in microservices, they may not provide all information that we might find useful, and so, we have pruned this section a bit. The reason has been the lack of automation. Without any automation, reading logs would require probably too much man power. Rahti offers a web-based application for managing Kubernetes cluster. Application logs can be read using those tools while pods are running in production. And for further use, logs can be directed to a file, or automate a process that alerts if an error occurs.

7. CONCLUSIONS

The goal was to develop a distributed microservice-based backend system for the course *Computer Systems*. Not only because the course absolutely requires a real life IoT system, but rather giving an opportunity for students to complete the course remotely, including laboratory exercises as well as the project work. This work also seeks to provide for a better overall picture about IoT, and to increase an educational value of the course. Another aspect here was to offer something special, a game for example. So far, the game is quite simple, but by just building an own controller using SensorTag can enthuse students to keep creating new features for the controller.

Even though we reached the goal, sometimes every step felt like a new challenge, and even made us question the design. One of the challenges we struggled with a quite some time was switching from development to production, and all the things we probably should consider felt a bit unclear. Mostly all this was caused by a huge range of possible solutions to carry it out. At the end of the day, all microservices are running firmly in the cloud without a need of constant checking, and ready for the actual target group.

At the beginning of the project, there was discussion about utilizing the backend created for this project as a start of a smart campus system. Indeed, in its current form the system has limited usefulness but works on excessively powerful concepts, meaning that the current backend could work as a proof-of-concept for developing a larger application.

8. REFERENCES

[1] Rose K., Eldridge S. & Chapin L. (2015) The internet of things (iot): An overview–understanding the issues and challenges of a more connected world. Internet Society , pp. 1–80.

[2] CSC (2021), Csc:n pilvipalvelu rahti on avoimessa beta-käytössä. URL: https://www.csc.fi/-/csc-n-pilvipalvelu-rahti-on-avoimessa-beta-kaytossa.

[3] Atzori L., Iera A. & Morabito G. (2010) The internet of things: A survey. Computer networks 54, pp. 2787–2805.

[4] Simoniot (2021), The importance of iot in your daily life. https://www.simoniot.com/internet-of-things-daily-lives/.

[5] Ashton K. et al. (2009) That 'internet of things' thing. RFID journal .

[6] Lueth K.L., State of the iot 2020: 12 billion iot connections, surpassing non-iot for the first time. URL: https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/.

[7] Sethi P. & Sarangi S.R. (2017) Internet of things: architectures, protocols, and applications. Journal of Electrical and Computer Engineering 2017.

[8] Huang M., Liu Y., Zhang N., Xiong N., Liu A., Zeng Z. & Song H. (2018) A services routing based caching scheme for cloud assisted crns. IEEE Access PP, pp. 1–1.

[9] Chan S., Fog brings the cloud closer to the ground: Cisco innovates in fog computing. URL: https://newsroom.cisco.com/feature-content?type=webcontent&articleId=1894659.

[10] Yousefpour A., Fung C., Nguyen T., Kadiyala K., Jalali F., Niakanlahiji A., Kong J. & Jue J.P. (2019) c. Journal of Systems Architecture 98, pp. 289–330.

[11] Yi S., Li C. & Li Q. (2015) A survey of fog computing: Concepts, applications and issues. In: Proceedings of the 2015 Workshop on Mobile Big Data, Mobidata '15, Association for Computing Machinery, New York, NY, USA, p. 37–42. URL: https://doi.org/10.1145/2757384.2757397.

[12] Milan M. (2020) Internet of Things: Concepts and System Design. Springer, Cham.

[13] Dragoni N., Giallorenzo S., Lafuente A.L., Mazzara M., Montesi F., Mustafin R. & Safina L. (2017) Microservices: yesterday, today, and tomorrow. Present and ulterior software engineering , pp. 195–216.

[14] Butzin B., Golatowski F. & Timmermann D. (2016) Microservices approach for the internet of things. In: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), IEEE, pp. 1–6.

[15] Pahl C., Brogi A., Soldani J. & Jamshidi P. (2017) Cloud container technologies: a state-of-the-art review. IEEE Transactions on Cloud Computing 7, pp. 677–692.

[16] Subramanian M. (2017) Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node. Apress.

[17] React (2021), A javascript library for building user interfaces. https://reactjs.org/.

[18] Freeman A. (2019) Pro React 16. Apress.

[19] Subhashini Chellappan D.G. (2020) MongoDB Recipes: With Data Modeling and Query Building Strategies. Apress.

[20] Wexler J. (2019) Get Programming with Node.js. Manning Publications.

[21] Aldowah H., Rehman S.U., Ghazal S. & Umar I.N. (2017) Internet of things in higher education: A study on future learning. Journal of Physics: Conference Series 892, p. 012017. URL: https://doi.org/10.1088/1742-6596/892/1/012017.

[22] Kortuem G., Bandara A.K., Smith N., Richards M. & Petre M. (2013) Educating the internet-of-things generation. Computer 46, pp. 53–61.

[23] Dobrilovic D. & Zeljko S. (2016) Design of open-source platform for introducing internet of things in university curricula. In: 2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI), pp. 273–276.

[24] Khanafer M. & El-Abd M. (2019) Guidelines for teaching an introductory course on the internet of things. In: 2019 IEEE Global Engineering Education Conference (EDUCON), pp. 1488–1492.

[25] Ali F. (2015) Teaching the internet of things concepts. In: Proceedings of the WESE'15: Workshop on Embedded and Cyber-Physical Systems Education, WESE'15, Association for Computing Machinery, New York, NY, USA, pp. 1–6. URL: https://doi.org/10.1145/2832920.2832930.

[26] Burd B., Barker L., Divitini M., Perez F.A.F., Russell I., Siever B. & Tudor L. (2018) Courses, content, and tools for internet of things in computer science education. In: Proceedings of the 2017 ITiCSE Conference on Working Group Reports, ITiCSE-WGR '17, Association for Computing Machinery, New York, NY, USA, p. 125–139. URL: https://doi.org/10.1145/3174781.3174788.

[27] Kubernetes. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/.

9. APPENDICES

Command fields sent from a client SensorTag	
Key	**Value type and key explanation**
id	At most four hexadecimal characters. When the sending SensorTag is directly connected to the interface, this must be used to express the ID of the sending SensorTag. In wireless communication, adding this field to the message is not required.
event	String (UP, DOWN, RIGHT, LEFT). Game avatar control event.
time	Integer. Timestamp of the current message. Can be used to give a custom time for when a sensor value was recorded according to the sending SensorTag.
ping	Respond with a 'pong' to the sending ID. Can be used to signify that the message was handled without errors. This command is the only exception to the key-value scheme.
session	String (start, end). Start a sensor data session to collect sensor data in the interface, with the x value being `time` timestamp, or milliseconds from session start. The collected data is sent to the database when the session ends and can be viewed by refreshing the graph. The session can be started and ended in the same message. Starting the session after a session has already been started will empty the data buffer. The session will not end on its own and an empty session cannot be ended.

Sensor data can be sent only when a session is open, among other fields. The available sensor names are `temp`, `humid`, `press`, `light`, `ax`, `ay`, `az`, `gx`, `gy`, `gz` and all are floating-point valued. Only required sensor values should be sent, or missing values can be set to a constant like zero. One message corresponds to one row of data, and if fields are repeated, only the last value will be recorded.

Message examples	
event:DOWN	If the game is visible in a browser, this will move the avatar down by one step.
ping,event:UP	If game is visible, move avatar up by one step. Replies with 'pong' once the command has been correctly executed in the interface.
session:start,temp:27.82, session:end,ping	Start a sensor data session, write temperature value 27.82 into the open session, end the session and write it to the database, and reply with 'pong' once the command has been correctly executed in the interface. This will always work, regardless of any previous session state.
event:RIGHT,light:208	Move the avatar right and record light level 208 into an open sensor data session. Sending sensor data with every event will give a graph of sent events once the session ends.

Any error in the message format will result in the rest of the message contents being lost. If this happens, the `ping` will not be responded to. The maximum message length is dictated by the wireless communication library used on the course: 127 bytes.

The following messages were used to try to test the key-value error handling during unintended usage.

Message	Result (Default: Appropriate error message is seen)
"time:e34"	
"tiem:34"	
"i:0024"	
"id:00g4,ping"	Ping is sent to id:0000. This is due to JavaScript Number.parseInt
"id:test"	
"session:stop"	
"sessio:start"	
"event:Up"	
"temp:a55.3"	
"humid:52\""	No error is caused. Number.parseFloat reads all it can
"hmid:12.7"	
"press:2f"	No error is caused
"light:0s0.0"	No error is caused
"ax:"	
"ay:−1"	
"az:2\n"	No error: Whitespace is trimmed intentionally
"gx:2\t"	No error: Whitespace is trimmed intentionally
"gx: 2.1"	No error: Whitespace is trimmed intentionally
"gy:,ping"	
"gz:2.1:UP"	Interestingly this results in an error message about the value, but the value is undefined
"event::DOWN"	Throws a TypeError about property 'trim' of undefined
"event:UP,,ping"	
"ping,,"	
"ping,"	
":5"	

The following messages were used to test the UART connection between the gateway and the interface. Long messages are abbreviated using Python list element repeat syntax. The characters \xf0 - \xf2 are used for the escape character scheme to have the character \xf2 only occur at the end, signifying end of UART message.

Message	Result (Default: No errors were made)
"a"*116	Key suggestion is not found and "undefined" is wrongly shown
"a"*117	Message was not sent. This likely means that the wireless library used on the SensorTag cannot send messages larger than 116 bytes
"\xf0"*115 + "\xf2"	
"\x01"	
"\x00"	Crashes the gateway. The method of testing accidentally sends a message of length zero, leading to this discovery
"p\xf2ng"	
"p\xf1i\xf1\xf2ng"	
"p\xf1\xf0i\xf0\xf1\xf1\xf0\xf1\xf2ng\xf2"	
"\xf2"	
"\xf1"	
"\xf0"	Not received. The decoder incorrectly removes it in the interface
"\xf0test"	
"\xf0\xf0\xf0"	Not received. Removed by decoder
"press:8\xf0"	Last byte is removed by decoder

STAGE 1

Student	**Hours**	**Contributions**
Teemu Varsala (PM) ("BSc thesis")	30 h, 60 %	Research, presentation, writing the report
Vili Pelttari ("ACP1 course")	24 h, 40 %	Research, writing the report and presentation

STAGE 2

Student	**Hours**	**Contributions**
Teemu Varsala (PM) ("BSc thesis")	95 h, 50 %	Backend and frontend design, and implementation, writing report
Vili Pelttari ("ACP1 course")	106 h, 50 %	Gateway design, implementation, report writing

STAGE 3

Student	**Hours**	**Contributions**
Teemu Varsala (PM) ("BSc thesis")	70 h, 50 %	Research, Kubernetes implementation, testing, writing
Vili Pelttari ("ACP1 course")	66 h, 50 %	Testing, writing test code and report and presentation

STAGE 4

Student	**Hours**	**Contributions**
Teemu Varsala (PM) ("BSc thesis")	80 h, 55 %	Research, testing, writing, presentation
Vili Pelttari ("ACP1 course")	80 h, 45 %	Code improvement, writing report and presentation