Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Andreas Chanon Arnholm
Mathias Neslow Henriksen

Combining Industry 4.0 and 5G Connectivity with Robots in Digital Production Factories

Master's thesis in Engineering & ICT
Supervisor: Amund Skavhaug
Co-supervisor: Stig Petersen and Adam Leon Kleppe
June 2021

NTNU
Norwegian University of
Science and Technology

Andreas Chanon Arnholm
Mathias Neslow Henriksen

Combining Industry 4.0 and 5G Connectivity with Robots in Digital Production Factories

Master's thesis in Engineering & ICT
Supervisor: Amund Skavhaug
Co-supervisor: Stig Petersen and Adam Leon Kleppe
June 2021

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU

Kunnskap for en bedre verden

Preface

This thesis is the final product of our master's degree in Engineering ICT at the Norwegian University of Science and Technology in Trondheim. This thesis is written for the robotics and automation group within the course "TPK4960 - Robotics and Automation, Master's Thesis" at the Department of Mechanical and Industrial Engineering.

We want to thank our supervisor Amund Skavhaug for his support, guidance, and mentoring before and during the project. We would also like to thank our co-supervisors Stig Petersen and Adam Leon Kleppe for their technical guidance and technical support in the laboratory. Lastly, we would like to thank the Department of Mechanical and Industrial Engineering at NTNU for letting us use their facilities during the COVID-19 pandemic.

This thesis assumes the reader has basic knowledge related to Industry 4.0, programming, communication protocols, and robotics. We hope this report will benefit anyone interested in Industry 4.0, 5G, or the other components used and discussed!

Andreas Chanon Arnholm & Mathias Neslow Henriksen
Trondheim, 2021–06–10

Abstract

This thesis explores whether 5G can complement and be used as part of or in conjunction with today's other major industrial driving forces, mainly Industry 4.0 initiatives such as Asset Administration Shells. This is done in light of two use cases proposed by the 5G-SOLUTIONS project, both of which aim to further develop digital factories with advanced communication solutions and real-time, remote operation. An architecture is designed and proposed as a way to tackle the use cases. The implementation of this architecture is a system with an accompanying AAS, which utilizes 5G communication, OPC UA, and ROS 2. The system's functionality revolves around auto/re-configuration, rapid deployment, and remote control of an AGV. The thesis presents the work carried out for setting up a testing environment at NTNU Gløshaugen's Industry 4.0 laboratory and the fundamental approach to implementing the system based on the proposed architecture. Each component in the system is thoroughly discussed in light of Industry 4.0 standards, and their feasibility is reviewed with regard to the use cases. The results of this are used to determine the system's and 5G's feasibility in an industrial environment based on Industry 4.0 initiatives. The end goal of developing a system meeting the standards of the use cases and Industry 4.0 is partly accomplished. A substantial research foundation and a component-based architecture are established and tested on many levels. The system with the AAS is used to control an AGV remotely, over the internet, and with the use of 5G. Rapid deployment and auto/re-configuration are tested and found plausible. To some extent, the system does support the idea of getting a new robot introduced to a factory or laboratory and tested within a short amount of time. The latency results required by an Industry 4.0-compliant real-time robotic system are not achieved, which is partly due to external delays associated with the installation of a local 5G node at NTNU Gløshaugen's Industry 4.0 laboratory.

Sammendrag

Denne masteroppgaven undersøker om 5G kan komplementere og brukes som en del av eller i forbindelse med de andre store industrielle drivkreftene i dag, hovedsakelig Industri 4.0-initiativer som Asset Administration Shells. Dette gjøres i lys av to "use case"-er som er foreslått av 5G-SOLUTIONS-prosjektet, som begge tar sikte på å videreutvikle digitale fabrikker med både avanserte kommunikasjonsløsninger og sanntidsfjernstyring. En arkitektur er designet og foreslått som en måte å takle "use case"-ene på. Implementasjonen av denne arkitekturen er et system med et tilhørende AAS, som bruker 5G-kommunikasjon, OPC UA og ROS 2. Funksjonaliteten til systemet dreier seg om auto-/re-konfigurasjon, rask utrulling og distribusjon, og fjernstyring av en AGV. Oppgaven presenterer arbeidet som er utført for å sette opp et testmiljø ved NTNU Gløshaugen's Industri 4.0 laboratorium, samt den faktiske tilnærmingen til implementering av systemet basert på den foreslåtte arkitekturen. Hver komponent i systemet blir grundig diskutert i lys av Industri 4.0-standarder, og gjennomførbarheten av dem blir vurdert med hensyn til "use case"-ene. Resultatene av dette blir brukt i et forsøk på å bestemme systemets og 5Gs verdi i et industrielt miljø basert på Industri 4.0-initiativer. Det endelige målet med å utvikle et system som oppfyller standardene for brukstilfellene og Industri 4.0 er delvis oppnådd. Et betydelig forskningsunderlag og en komponentbasert arkitektur er etablert og testet på mange nivåer. Systemet med AAS-en brukes til å kontrollere en AGV eksternt, over internett og med bruk av 5G. Rask utrulling og auto-/re-konfigurasjon er testet og funnet sannsynlig. Systemet støtter til en viss grad ideen om å få en ny robot introdusert til en fabrikk eller laboratorium og testet innen kort tid. Den aksepterte maksimum forsinkelsen som kreves av et Industri 4.0-kompatibelt sanntidsrobotsystem oppnås ikke, noe som delvis skyldes eksterne forsinkelser knyttet til installasjonen av en lokal 5G-node ved NTNU Gløshaugens Industri 4.0-laboratorium.

Acronyms

AAS Asset Administration Shell

AGV Automated Guided Vehicle

AP Access Point

API Application Programming Interface

ASGI Asynchronous Server Gateway Interface

CLI Command-line Interface

DDS Data Distribution Service

DDSI Data Distribution Service Interoperability

GNSS Global Navigation Satelite System

HTTP Hypertext Transfer Protocol

IDL Interactive Data Language

KMP KUKA Mobile Platform

KMR KUKA Mobile Robot

KPI Key Performance Indicator

KUKA Keller und Knappich Augsburg

LAN Local Area Network

NR New Radio

OPC UA Open Platform Communications Unified Architecture

PIN Personal Identification Number

PLC Programmable Logic Controller

QoS Quality of Service

REST Representational State Transfer

ROS Robot Operating System

RTPS Real-time Publish-Subscribe

SIM Subscriber Identity Module

SSH Secure Shell

TTI Transmission Time Interval

URLLC Ultra-Reliable Low-Latency Communications

eMBB Enhanced Mobile Broad-band

iiwa intelligent industrial work assistant

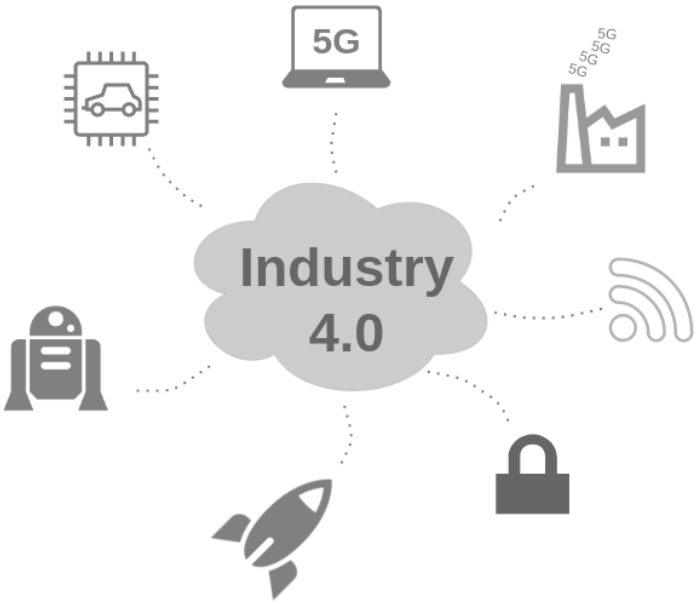
mMTC Massive Machinetype Communications

List of Figures

Contents

Chapter 1

Introduction



1.1 Motivation and Problem Description

In today's vastly expanding digital society, innovative and more efficient solutions are needed to be competitive in our new technological world. Throughout modern history, three industrial revolutions have taken place, and many believe that we are now witnessing the fourth. The term describing today's revolutions and development is referred to as Industry 4.0. Every piece of equipment one can imagine is connected to the internet in some way. It could be our curtains, our fridge, or even our car. In order to facilitate and sustain this rapid development, a solid industrial infrastructure with an underlying foundation of telecommunication

is needed.

5G-SOLUTIONS is a 5G project financed by the EU. The project's main objective is to conduct advanced field trials in innovative and thematically diverse digital services that require 5G capabilities.

"The fifth generation of telecommunication systems, or 5G, will be one of the most critical building blocks of the European digital economy and society in the next decade."
- 5G-SOLUTIONS [1]

As seen from their quote, 5G-SOLUTIONS believes that 5G will heavily affect the next decade. Many groups and corporations, expanding several European countries, have joined the project, NTNU being one of them. In collaboration with the 5G-SOLUTIONS project, this thesis has explored whether 5G can complement and be used as part of or in conjunction with the other major industrial driving forces of today, mainly Industry 4.0 initiatives such as Asset Administration Shells (AAS). The objectives related to this problem description are described in section 1.3.

1.2 Previous Work

Since the 5G-SOLUTIONS project commenced in 2019, fundamental work and a foundation upon which to further improve have been outlined. The fundamental work mostly surrounds defining, grouping, and assigning the different use cases of 5G. In total, it has been determined that more than 140 target KPIs are going to be validated per the 3GPP release. It has also been determined that the business potential of each use case will be validated by the end-users to guarantee that each of the use cases is well-positioned to respond to the needs of the market [2]. Figure 1.1 displays a map of the project locations and their related groups and corporations.

The blue node in Figure 1.1 represents 5G-VINNI (5G Verticals INNpcation Infrastructure), which is another project financed by the EU. Since 2018, the project participants have been working on establishing an "end-to-end facility that validates the performance of new 5G technologies by operating trials of advanced vertical sector services" [3]. The project is coordinated by Telenor, and with main facility sites in Norway, UK, Spain, and Greece, they are currently using advanced 5G technologies and automated testing to make sure that the new technology works as expected when it is implemented into various industry verticals [4]. In 2020, Telenor launched a commercial 5G network in Norway that is available for the general public. Their headquarters in Fornebu includes a 3.6 GHz 5G base station used for the commercial network, as well as a 26 GHz 5G base station used for 5GVINNI testing. In addition to this have companies such as Ericsson, which is also a part of the

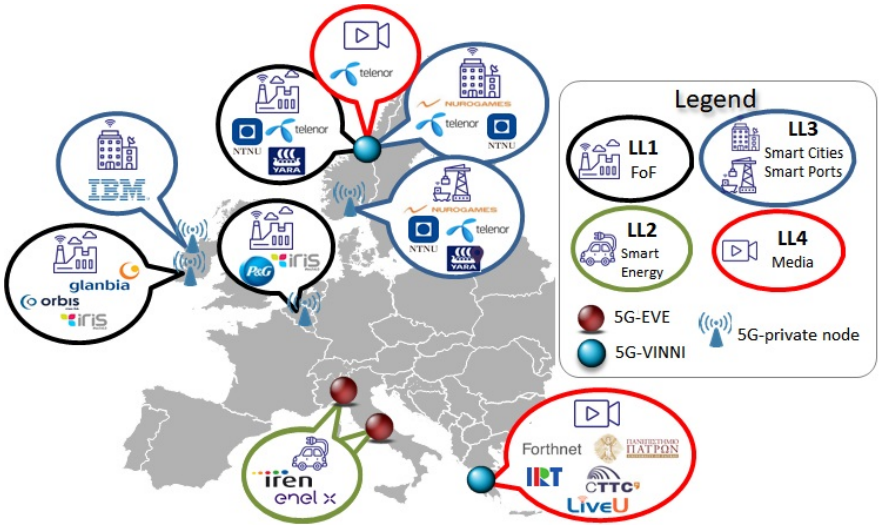


Figure 1.1: Map of the 5G-SOLUTIONS project with related groups and corporations [2].

5G-VINNI project, performed extensive research and testing of 5G in relation to industrial robotics and automation. This research includes performance testing with the use of remotely controlled drones and cars using 5G. A detailed look at some of Ericsson's endeavors is found in subsection 2.1.1.

The scope of this thesis is not solely focused on 5G and its potential, but also on presenting a system that satisfies the objectives of the project. The implemented system described in this thesis uses some of the results from the specialization project conducted by the authors in the fall of 2020. The objective of the specialization project was to set up an automatic configuration solution between a KMR iiwa robot entity, a workspace computer, and a web-based AAS. Even though automatic configuration was not achieved, a prototype of the AAS capable of controlling the KMR iiwa on a local Wi-Fi network was developed as a proof of concept with key features of Industry 4.0 in mind. Through both research and testing of the communication layers in the system, it was determined that technology such as ROS 2 and OPC UA are powerful and adaptable tools when working with autonomous robots.

1.3 Objectives

The trials described in the 5G-SOLUTIONS project are presented with the use of "Living labs". The purpose of the living labs is to validate the proficiency of both the use cases of the vertical industries and the 5G technology [2]. Each living lab consists of several use cases specialized for their specific role in the project. The living lab that is in focus for this thesis is "LL1: Factories of the Future (FoF)". LL1 aims to improve digital factories with increased sensing, faster and more reliable robotic operation, and real-time, secure connectivity. The

scope of this thesis is defined by two of the use cases in the LL1 initiative [2]:

- **Use Case 1.5** - Rapid deployment, auto/re-configuration, and testing of new robots.
- **Use Case 1.3** - Remotely controlling digital factories.

The objective of the project related to this thesis is to provide a solution for an architecture based on the relevant use cases. The purpose of the thesis is to study the results of this project and discuss its value with regard to the standards of Industry 4.0, 5G, and the relevant use cases.

1.3.1 Use Case 1.5

The context of Use Case 1.5 is NTNU Gløshaugen's Industry 4.0 laboratory[1]. This lab has a combination of stationary production robots and two automated guided vehicles (AGVs). The intent of the use case is to use 5G as the communication infrastructure in a system that handles control and data exchange between robots, AGVs, and the control room. For this, a system that supports rapid deployment, auto/re-configuration, and testing of new robots needs to be developed. A critical task for this use case is to accomplish "Initial configuration of control room video feeds". In other words, establish a reliable video stream from the production facility. Wired communication is not an optimal solution for production robots, and the current generation wireless communication protocols do not support the latency requirements of standards of Industry 4.0, nor the data rate for multiple high-quality video streams in a production area.

1.3.2 Use Case 1.3

Use Case 1.3 is focused on the idea of remotely controlling industrial production factories. This can be achieved by giving the operators the ability to control parts of the production using tablets, PCs, or smartphones from a remote location. Being able to remotely control a factory would give operators enhanced flexibility and improved efficiency, as it would impact and change a large part of their daily operations for the better. Use Case 1.3 is a standalone use case, but it is interesting to see if the system developed with regard to Use Case 1.5 can function as a solution for this use case. The end goal is to use the developed system to achieve real-time remote control of the AGVs, and for the system to meet the performance requirements posed by the use case. The maximum acceptable latency from a user device to a robot is 10 ms for Use Case 1.3.

[1] NTNU Gløshaugen's Industry 4.0 laboratory is a subpart of the MANULAB: Norwegian Manufacturing Research Laboratory project (269898) [5]

1.4 Contributions

This thesis attempts to contribute to society by offering a system architecture that can potentially be used to further the development of digital factories. With a component-based, flexible, and portable system for controlling robots using 5G, new and innovative strategies can be created to push the boundaries of Industry 4.0. With this project and its research, the world is hopefully one step closer to faster and more precise deployment, configuration, and control of industrial robots. Even if the complete architecture lacks certain features and performance results with regard to the requirements of Industry 4.0, the thought process behind these solutions can be of benefit. The successful parts of the implemented system can also be extracted and reused in other systems. Telenor has utilized some parts of the architecture presented in this thesis in order to operate a remote-controlled car with 5G at their test site in Fornebu.

1.5 Limitations

As mentioned in section 1.2, the project conducted for this thesis is related to the 5G-VINNI project. Telenor has a vital role in this project, and one of their tasks was to install a 5G node providing a local 5G network in NTNU Gløshaugen's Industry 4.0 laboratory. At the beginning of the project, it was expected that this node would be installed with sufficient time remaining to perform experiments. Due to delays from Telenor, this node is now scheduled to be installed after the delivery date of this thesis. As a consequence of this, experiments are instead performed on Telenor's public 5G network.

1.6 Outline

The report is structured in nine chapters. Chapter 2 introduces the background theory with the concepts and technologies that were researched and used for this thesis. This chapter delves into technical theory about the concepts, hardware, and software components, and it is meant for those who want to understand the underlying theory of the system presented in this thesis and evaluate it against the standards of Industry 4.0.

Chapter 3 outlines the architecture that is presented as a solution to how 5G can be incorporated into an industrial robotic lab facility. The system design is introduced and described in relation to the use cases explored in this thesis and the design principles of Industry 4.0. This chapter shows the most important segments and gives insight into how each of these plays an important role in realizing a robust and effective system within the realm of Industry 4.0. It is intended for those who wish to understand why the system is designed the way it is and

the thought process behind each major design decision.

Chapter 4 describes an implementation of the architecture on a technical level. Here, the technical details and implementation-related specifications are thoroughly explained with the use of code snippets and detailed architectural illustrations. For those who wish to recreate the system, it serves as a detailed guide on how each part of the system is implemented. The chapter is therefore intended for those who wish to recreate parts of the system or the system in its entirety and those who want to know how or why each specific implementation is done.

Chapter 5 outlines how the hardware and software components are used to set up the experimental prototype of the system. Using terminal commands and installation instructions, it details how to get every component up and running. The chapter is meant as a guide both for those who wish to recreate either the entire system or parts of the system used for this project, those who want to perform the tests displayed in chapter 7, and for those who are interested in the technical details surrounding specific components.

Chapter 6 describes the feasibility experiments conducted in order to ascertain whether or not certain components are usable as a part of the system and to make certain design choices. It is therefore meant for those who wish to see the reasoning behind some of the decisions that were made during the implementation of the system. The performance experiments conducted to benchmark the finalized system are described in chapter 7. Both chapters are also intended for those who might want to build similar systems, improve the one presented, or for those who are interested in knowing what kinds of tests they themselves can use for comparison. These chapters are structured with sections containing a complete experiment description, results, and a discussion. Each experiment is presented together with its results and discussion in order to increase readability and make it easier for the reader to obtain the relevant information from each individual experiment.

Chapter 8 discusses how the architecture and the implemented system solves the tasks related to the objectives, and whether or not the proposed solution has merit with regard to today's industrial standards. The discussion also surrounds the most important parts of the system, their advantages and disadvantages, and whether or not they help the system meet the requirements of Industry 4.0. This chapter is meant for those who want to understand the underlying principles of the system. For those who might want to build similar systems, this chapter also sheds light on which aspects should be improved. The report ends with a conclusion in chapter 9.

Chapter 2

Background Theory - Concept and Technologies

This chapter introduces the most important parts of the various concepts, software technologies, and hardware technologies that are relevant for the research, implementation, and experimentation related to this thesis. The system architecture that is described in this thesis is intricate and involves several different technologies that are implemented to work in unison. Because of this, it is important to describe the fundamentals of each part in detail to get an understanding of how and why they are chosen as part of the system. Section 2.1 covers the concepts that are researched and utilized in the project. These are the parts of the system that are not viewed as tangible software tools, but rather as means to fulfill the purpose of the system. Section 2.1.1 explains the main properties of the newest generation within cellular networking, namely 5G, and it outlines its place in Industry 4.0. Real-time systems in general, the operating system Linux and Linux Kernels' real-time properties are described in subsections 2.1.2, 2.1.3 and 2.1.4, respectively. The section also outlines different protocols used throughout the system, such as OPC UA in subsection 2.1.7, TCP/IP in subsection 2.1.5 and the closely related WebSockets in subsection 2.1.9. Section 2.2 covers the hardware technologies that are used to test the implementation. This includes the Raspberry Pi 4, the SIM8200EA-M2 and 5G HAT, and the KMR iiwa AGV, which are explained in subsections 2.2.1, 2.2.2 and 2.2.3, respectively. Lastly, Section 2.3 covers the various software technologies that were used during both implementation and testing. The information in these sections is important for those who want to understand the underlying theory of the architecture presented in this thesis and evaluate it against the standards of Industry 4.0. [1]

[1] Some sections in this chapter are heavily inspired by the related specialization project completed by the authors during the fall of 2020.

2.1 Concepts

2.1.1 5G Networks

Because of massive data growth over the last ten years, the resulting capacity demands have brought forth a rapid development in mobile access technology [6]. The newest revolutionary change is the 5th generation (5G) mobile network, or New Radio (NR). With this new global wireless standard comes even greater requirements than that of its predecessors in terms of wide-area coverage, reliability, latency, and availability.

2.1.1.1 Fundamentals

The 5G communications technology supports three main services shown in Figure 2.1: enhanced mobile broadband (eMBB), ultra-reliable low-latency communications (URLLC), and massive machine-type communications (mMTC) [7]. The first, eMBB, provides stable connections with high peak data rates as well as enhanced wireless access and connectivity. URLLC provides low-latency transmission of data with high reliability, and it makes a network able to adapt to large amounts of changing data in real-time. Lastly, mMTC supports connecting large numbers of devices that intermittently transmit small amounts of data.

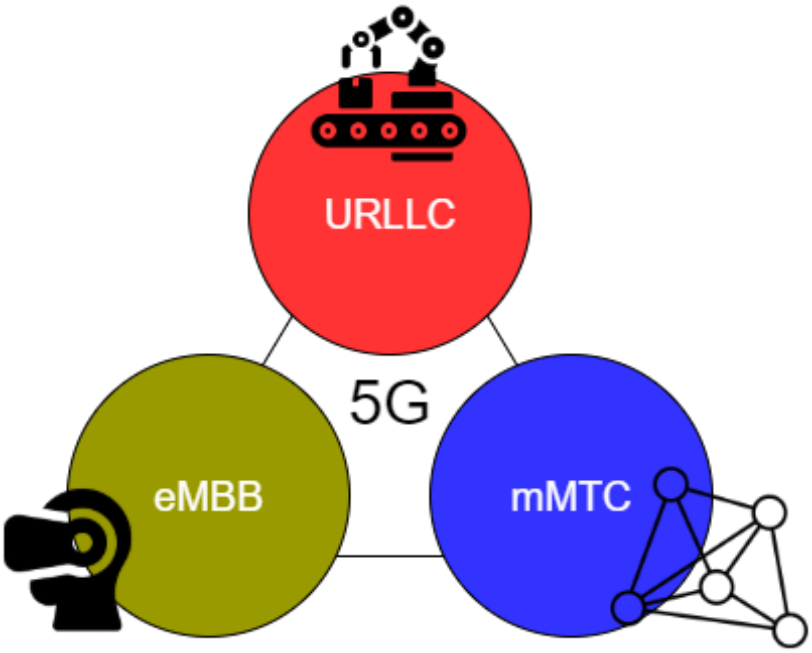


Figure 2.1: 5G service types.

New applications and uses cases are expected with the advent of 5G, like virtual and augmented reality, autonomous vehicles, tactile internet, and Internet of Things (IoT) [7]. These all have the potential to be great assets in the factories of the future, and the use of 5G

is essential, not only to propel the development of these technologies but also to make sure that they meet the requirements of the new standards of Industry 4.0. Because the demand for low-latency computation is increasing rapidly, and because low latency is a fundamental metric for network performance and required by the emerging applications and use cases, the services that 5G provides are invaluable. Stable, high bandwidth connections can allow for large-scale video streaming, low-latency data transmission with high reliability can ensure the safety and effectiveness of industrial automation, and increased mass-connectivity can let factories connect an almost unlimited amount of systems, devices, and sensors.

2.1.1.2 Performance

In a study presented at the 2019 International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), the performance of 5G was tested in a simulated Industry 4.0 scenario. The scenario included 280 wirelessly connected robots, each equipped with URLLC transceivers, distributed in 5 production lines. The URLLC transceivers communicated with a central controller through a set of gNodeBs, which are 3GPP-compliant implementations of the 5G base station [8]. The tests were conducted with different configurations on three separate frequencies: 700 MHz, 3.5 GHz and 26 GHz. At 700 MHz and 280 robots, the latency results showed that, at each TTI (transmission time interval), a limited number of packets were transmitted compared to the total number of packets generated. This suggests both lack performance for short delays (below 1 ms) and channel congestion with many interconnected robots. If the number of communicating robots was reduced to 20, the latency results increased significantly, with 100% of the packets being transmitted with latency below 1 ms. At 3.5 GHz, the latency with 280 robots never exceeded 1 ms on certain scheduling configurations. At 26 GHz, the latency with 280 robots was below 1 ms for 100% of packet transmissions, with one particular scheduling configuration achieving a delay lower than 0.5 ms [8].

Although Industry 4.0 is more of a conceptual trend rather than a standard and absolute quality of service (QoS) requirements have yet to be established, there are already agreements around what emerging technologies need to deliver in order to realize the goals of Industry 4.0. In a discussion paper published by the Federal Ministry for Economic Affairs and Energy (BMWi) in collaboration with the Plattform Industrie 4.0 network, a quantitative range of relevant QoS requirements is presented in relation to network-based communication. Here, the latency requirement for motion control is proposed as 0.25-1 ms, while the latency requirements for condition monitoring and augmented reality is proposed as 100 ms and 10 ms, respectively [9]. The 5G performance results suggest that it is able to comply with the proposed QoS requirements for Industry 4.0, at least for motion control, as this is the property with the least packet transmission volume and thereby closest, in terms of similarity,

to the Industry 4.0 scenario used for the testing.

2.1.1.3 5G in Robotics and IoT

An interesting note, especially within the context of this thesis, is that 5G is more suitable for robotics and automation than 4G LTE, as it can provide a more reliable connection with lower latency. Ericsson, one of the main drivers of the 5G era, is at the forefront of development research of 5G use cases within Industry 4.0, IoT, and robotics, and they have already been able to show the power of the new cellular network through high-accuracy control and manipulation of physical objects, a concept that is known as *tactile internet* [10]. In collaboration with KTH Integrated Transport Research Lab (ITRL), they have managed to remotely operate a 5G concept car located 50 km away with a control latency of 4 ms and video acquisition and rendering latency of 20 ms. Before this, similar demonstrations using drones and an ABB industrial robot teleoperated over 4G produced significantly less satisfactory results, with average control latencies of 40 ms and average video acquisition and rendering latencies of 300 ms [10].

2.1.2 Real-Time Systems

A definition of a real-time computer system is a system that "controls an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time" [11]. Real-time computer systems are often a part of larger systems called real-time systems. These real-time systems are often separated into a set of self-contained subsystems called clusters. The clusters outline the independent parts of a real-time system, e.g. the physical robot to be controlled (controlled cluster), the real-time computer system (computational cluster), and the human operator (operator cluster). The connection between the different clusters within a real-time system is obtained using specific interfaces [12]. See Figure 2.2.

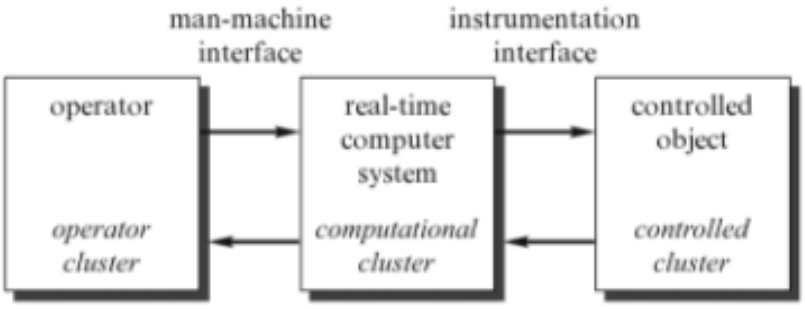


Figure 2.2: A real-time system consisting of three clusters and two interfaces [12].

Real-time systems are defined by a deterministic schedule—not by low-latency with which

they are often associated. The systems must guarantee that they finish certain tasks in a certain amount of time. As a result, it is important that a maximum allowable latency for tasks is set and that the latency is measurable. In order to avoid interfering with temporal properties and rather aid with the implementation, a real-time computer system needs an operating system that has real-time capabilities. In addition, it needs user code that supports deterministic execution [13].

A real-time computer system must respond to events from the environment, either from the controlled cluster or the operator cluster, within time intervals imposed by the environment. A "deadline" is the instance when a result must be produced, and is classified as either "soft", "firm", or "hard". If the result is useful after the deadline has passed, it is classified as "soft", otherwise it is "firm". If a firm deadline is missed and results in "severe consequences", the deadline is classified as "hard". A real-time computer system that must fulfill at least one hard deadline is called a hard real-time computer system. If no hard deadline exists it is called a soft real-time computer system. The designs of these two systems are fundamentally different. Soft real-time computer systems can miss deadlines occasionally, while hard real-time computer systems must have a guaranteed temporal behavior under all specified fault and load conditions [12].

2.1.3 Linux

Linux is a collection of open-source, Unix-like operating systems based on the Linux Kernel. It has been around since the 1990s and has a user base that spans the entire globe [14]. Most larger IoT devices, such as phones, infotainment systems, and specific television brands, run on some kind of distribution based on the Linux Kernel. Distribution, or "distros", is the collective name for different versions of Linux. There exist many Linux distributions, each developed to serve a certain type of user or system. Ubuntu and Debian are examples of Linux distributions.

Linux has zero cost of entry and is one of the most reliable computer ecosystems on the planet. It is distributed under an open-source license, which means that every user has the freedom to run the program for any purpose, study how the program works, realize changes to make it do what the user wishes, and redistribute custom-made copies. One of the key aspects of Industry 4.0 is the focus on interoperability, and as a result of Linux being open-source, it is highly scalable and provides a great platform for connectivity between different devices.

2.1.4 Real-Time Linux Kernels

An optimized kernel designed to maintain low latency, determinism, and consistent response time is in the Linux world often referred to as a real-time Linux kernel. A real-time kernel is not "better" or "superior" compared to a standard kernel, they just serve different goals and meet other system requirements [15]. There exist other operating systems built with specialized real-time capabilities in mind, but these often have little support for third-party software compared to Linux. According to redhat.com [15], a real-time Linux kernel includes the following important mechanisms:

- Under heavy load the priority of the tasks are checked.
- High priority tasks are given preference for CPU execution.
- Abolishes the use of Completely Fair Scheduling.
- Maintains low latency execution time.
- Provides the ability to measure, record, and configure response time.
- Uses the scheduling policies SCHED_FIDO or SCHED_RR.

As an example, the *PREEMPT_RT* patch gives Linux several essential properties of real-time operating systems. PREEMPT_RT lets the interrupts run as threads, which makes it possible for the user/developer to modify the priorities of the interrupts and to prioritize the interrupt handlers even when the hardware does not support them. A big advantage of PREEMPT_RT is that it makes the Linux kernel more capable of task-handling in real-time, as it does not prevent the developer from creating an application that meets real-time requirements. PREEMPT_RT reduces the maximum latency experienced by the user by changing the internal code. This code change involves changing *spinlocks* to *mutexes* [16]. A spinlock will simply wait for a resource indefinitely until it is available, i.e. the thread is not preempted, while a mutex will put a waiting thread to sleep. Mutexes do not, unlike spinlocks, waste CPU cycles.

2.1.5 TCP/IP

Transmission Control Protocol/Internet Protocol, or TCP/IP, is a suite of data communications protocols, with a name that refers to two of the protocols that belong to it. It is the leading communications software for connecting devices on local area networks and is the foundation of the worldwide Internet [17].

TPC/IP has important features that have made it capable of meeting the many needs of worldwide data communication since its inception in the 1970s, and that makes it capable of realizing the challenging implementations of new Industry 4.0 standards. TCP/IP has

open protocol standards, is available for free, and developed independently from any specific hardware or operating system [17]. This significantly simplifies the process of connecting different hardware and software components, even without the use of the Internet. TCP/IP is also independent of specific physical network hardware [17], which allows it to integrate and run over many different types of networks including Ethernet, Wi-Fi, optical networks, and many more. Another handy feature is the common addressing scheme that allows any computer to uniquely address any other computer in the entire network [17], making the passing of packets of data between TCP/IP devices safe and reliable.

The structure and function of TCP/IP are described using an architectural model called the *Open Systems Interconnect Reference Model*, or OSI Reference Model [17]. This model contains seven abstraction layers, each of which defines the function performed when data is transferred between computing systems. Two of these layers are of particular interest in the context of this thesis, namely the Network layer and the Transport layer.

2.1.5.1 Network Layer

In order to fully understand TCP/IP, it is important to know the details of how addressing and delivery of data is handled. IP addresses are 32-bit numbers normally expressed as four decimal numbers separated by periods, e.g. 192.168.231.123, with one part being the network address and the other being the host address. The format of these parts is not the same in every IP address, and the number of bits used to identify them varies according to the prefix length of the address, which is again determined by the address bit mask more commonly known as a *subnet* mask. A bit in the mask can either be on (255) or off (0). If a bit is on in the mask, the bit in the IP address belongs to the network part, and if it is off the bit belongs to the host part. With a subnet mask of 255.255.255.0, the IP address 192.168.231.123 would have 192.168.231.0 as the network part and 0.0.0.123 as the host part.

The inclusion of subnet masks is significant because it allows decentralized management of host addressing [17]. By splitting a network into smaller sub-networks it is possible to control the amount of host devices on the network, and thereby also the amount of congestion. With no subnetting, each device would be able to see packets broadcast from all other devices, and the network switches would be heavily overloaded with having to move traffic to the appropriate ports.

2.1.5.2 Transport Layer

The transport layer protocols are what provide the end-to-end communication between two hosts on a network [18]. The two core protocols of the transport layer are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol), each of which has its own advantages

and use cases. The choice of transportation protocol is dependent on the QoS requirements for the application in question, as it affects performance measures such as delay, jitter, packet loss, and throughput.

TCP is a popular and robust connection-oriented protocol that supports reliable delivery. Data is transmitted in the form of segments, each of which has a sequence number that helps to order messages and eliminates duplicates. Reliability is maintained by having the receiver of transmitted segments send an acknowledgment (ACK) back to the sender on successful delivery. If an ACK is not received by the sender within a specified time interval, the sender attempts to transmit the segment again [18]. Congestion control is done by having the receiver return a "window" together with the ACK that specifies how many sequences it can handle. This way, the receiver is not overwhelmed by more data than it can handle [18]. The characteristics of TCP make it suitable for applications that have stringent reliability requirements. Applications that make use of TCP include HTTP, WebSockets, and SSH (Secure Shell).

UDP is a more simple transport protocol than TCP, and it does not provide the same reliability or ordered delivery [18]. Instead of segments, UDP delivers data in the form of datagrams, which are made up of headers and payloads [18]. Once a datagram is sent to a receiver, the sender will never know whether or not the datagram was received. There is no metadata indicating the capacity of the receiver, and there is no mechanism for ordering of datagrams [18]. These aspects make UDP fast but unreliable. The characteristics of UDP make it suitable for applications that have stringent latency requirements and where packet loss is not an issue. Such applications include video and audio streaming.

2.1.6 AAS - Asset Administration Shell

The Asset Administration Shell (AAS) is an important concept in the world of Industry 4.0. It is used to describe an asset electronically in a standardized manner [19]. The purpose of an AAS is to exchange asset-related data among and between industrial assets and engineering tools or production orchestration systems. Rather than reading individual states and sensor data from many different entities, an AAS in Industry 4.0 proposes to represent data as a combined asset.

To this date, there is no detailed specified standardization of an AAS. However, the structure of an AAS must be standardized in such a way that it enables exchanges. OPC UA, a communication protocol that is explored later in this chapter, is a format for exchanges and communication that is regulated and compliant with Industry 4.0 standardizations.

To enable an autonomous and self-managing network, an AAS has to provide information about its capabilities. Standardization requirements are also needed. Industry 4.0 requires

an AAS to communicate in an Industry 4.0-compliant manner. In other words, there are rules and requirements specified by industry 4.0 that the AAS needs to follow. These rules surround quality, efficiency, scalability, security, flexibility, etc. These standardizations are set to reduce the overall complexity and allow for greater interoperability.

For scalability and flexibility, the important aspect of an AAS is the requirement of an internal interface to the entity and a standardized interface for external communication [20]. This requirement needs to be satisfied in order for the standardized external communication to not interfere with the functionality of the entity. This way, an AAS can safely and easily be changed or expanded upon with, say, the ability to administer multiple entities without having to change any of the software or hardware related to the entity itself. These requirements are also tightly linked to the concept of modularity—an essential property of any robust software system—which aims to separate the functionalities of components so that they can be removed, added, and recombined to suit the needs of the user.

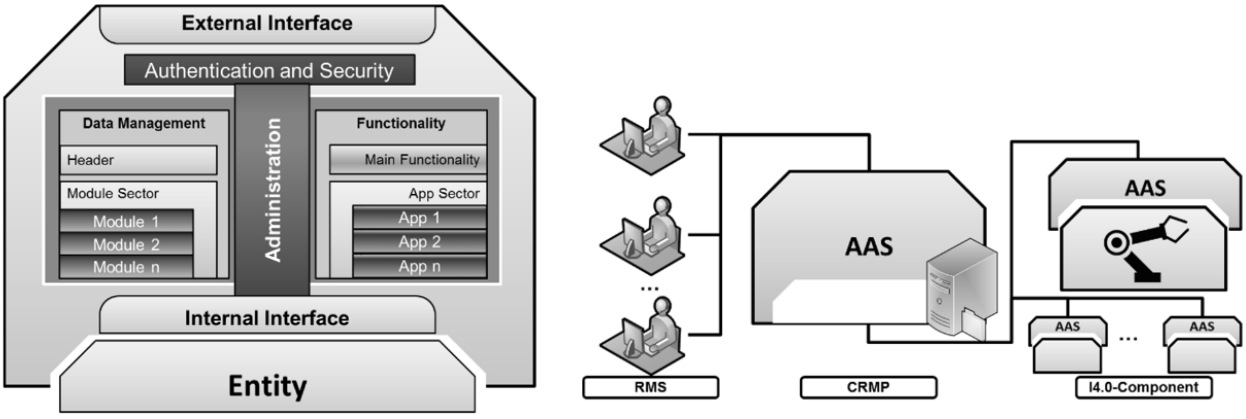


Figure 2.3: Proposed structure of an I4.0-compliant AAS with separated modules and functionality applications (Left) and the AAS's integration with a proposed overall system (right) [20].

In addition to an external and internal interface as mentioned above, an Industry 4.0-compliant AAS should also need segments for authentication and security, data management, functionality, and administration [20]. The external interface handles the flow of data from external clients, and in order to assure interoperability, it must be highly standardized. As of 2017, service-oriented paradigms like RESTful and SOAP are proposed for communication for achieving this [20]. Furthermore, a segment for authentication and security needs to be connected to the external interface. This segment is dependent on requirements related to the entity and the technical specifications of the AAS implementation. The data management segment consists of several modules. The main module is mandatory and contains information about the aspects of the AAS, and the other modules either contain data about the components managed by the AAS or provide additional data about an application within

the functional segment [20]. Because of the amount of data that needs to be handled in this segment, it could be beneficial to utilize some sort of data management software like a database management system. The functionality segment contains the different applications within the AAS. In a complex system with many Industry 4.0-components, having multiple applications with different functionality can greatly improve the quality of the AAS. The administration segment handles the data flow between the other segments, and the internal interface deals with the communication between the AAS and the entity.

As stated, there is no specific standard for the requirements surrounding an AAS; however, in order to be Industry 4.0-compliant, the system should follow the basic structure outlined in this section. Additionally, there is not yet a technical definition of structure for an AAS [20], which means that the implementation can be done in a way that suits the needs of the project and users.

2.1.7 OPC UA - Open Platform Communications Unified Architecture

OPC UA is a machine-to-machine communication protocol developed by the OPC Foundation. OPC UA was first released in 2008 and was designed to enhance and surpass the capabilities of the classic OPC [21]. It has today become one of the key standards of industry 4.0, especially in Europe. The focus of OPC UA is on communication with industrial equipment and systems for data collection and control.

Industry 4.0 often enhances the idea of connectivity and interoperability between the different equipment found in the manufacturing industry [22]. OPC UA's main intention is to solve the issue of poor interoperability between equipment that comes from a wide pool of different manufactures. To achieve this, the development has been highly affected by the use of standardized methods and abstractions. OPC UA has a multi-layered approach, see Figure 2.4, which integrates the functionality of the individual OPC Classic specifications into one extensible framework.

This approach realizes the original design specification goals of OPC that are listed below [21].

- *Functional equivalence* - all COM OPC Classic specifications are mapped to UA
- *Platform independence* - from an embedded micro-controller to cloud-based infrastructure
- *Secure* - encryption, authentication, and auditing
- *Extensible* - ability to add new features without affecting existing applications
- *Comprehensive information modeling* - for defining complex information

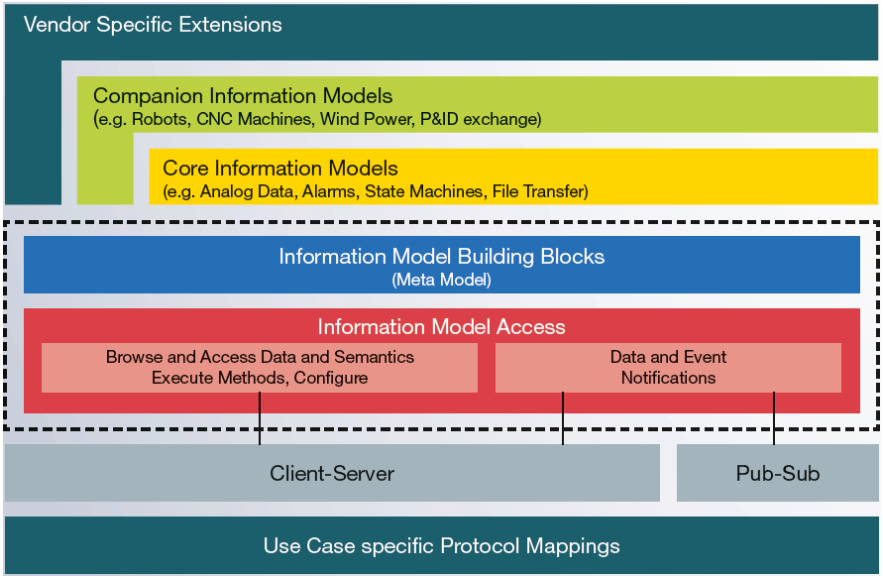


Figure 2.4: OPC UA's multi layered approach[21].

OPC UA also surpasses the capabilities of the classic OPC in the different categories by:

- *Functional equivalence* - adding discovery, address space, on-demand, subscriptions, events, and methods
- *Platform independence* - adding more hardware platforms and support for more OS's
- *Secure* - adding transport protocols, session encryption, message signing, sequenced packets, authentication, and user control
- *Extensible* - using the highly modifiable multi-layered approach

Object-oriented structures are used to turn data into information. With this, even the most complex multi-level structures can be modeled and extended [21]. This object-oriented framework is the most fundamental and crucial part. It defines the rules and the base building blocks that are necessary to expose an information model with OPC UA.

OPC UA supports both the client-server pattern and the publish/subscribe pattern with regard to communication between different components in a computer network [21]. Client-server communication means that the users of a service, called *clients*, send requests to the service provider, called a *server*. After the server receives a request from a client, it processes the requests and sends back a result with a response. With publish/subscribe there are no requests or responses, and the communication happens strictly one-way. The sender of a message is called a *publisher*, and instead of programming the messages directly to the receiver, it publishes the message to a named resource called a *topic*. It does so without knowing which, if any, receivers are out there to read the message. These receivers are

known as *subscribers*, and they subscribe to topics and simply wait for messages without any knowledge of potential publishers.

2.1.8 REST - Representational State Transfer

Representational State Transfer, commonly known as REST, is an architectural style that defines a set of software engineering guiding principles for web services. RESTful systems, which are REST-compliant systems, are characterized by their nature of being stateless and that they separate the concerns of client and server [23]. By separating the user interface from the backend, REST improves the portability of the user interface and the scalability of the server and backend. The server and clients in RESTful systems communicate using HTTP. Clients send HTTP-Requests and the server responds with HTTP-Responses. A request must contain a path to the resource that the operation should be executed on. Figure 2.5 outlines the basic structure of RESTful communication.

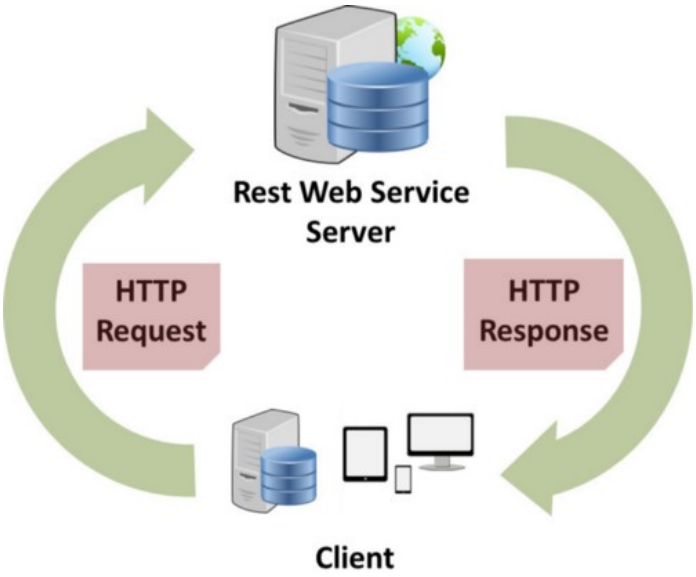


Figure 2.5: RESTful Web Services Communication Architecture [24].

There are four base HTTP verbs used in requests in RESTful systems [23]. These are:

- GET — retrieve a specific resource or a collection of resources
- POST — create a new resource
- PUT — update a specific resource
- DELETE — remove a specific resource

Other available methods are PATCH, HEAD, CONNECT, OPTIONS and TRACE.

2.1.9 WebSockets

The most common way for web applications to receive and transmit data is through HTTP polling, i.e. sequences of HTTP requests and responses. The problem with polling is that each request and response contains header information that might not always be important, especially in real-time scenarios. Thus, when the rate of requests increases, the repeated header information produces significant overhead, to the extent that real-time, full-duplex communication is possible only with increased latency and high network traffic [25]. Another way is through HTTP long polling. As Figure 2.6 shows, long polling means that a connection is held open between the server and the client, and after a request from the client, the server responds only when there is data to respond with [25]. This eliminates empty responses from when the client requests something that is not available, but since it only reduces network traffic in scenarios where the client needs data from the server, there are no real benefits for applications in which a client might want to send real-time commands to the server with no need for a response. Also, in applications in which the server and its clients need to exchange continuous information, the repeated header information will produce significant overhead even for long polling.

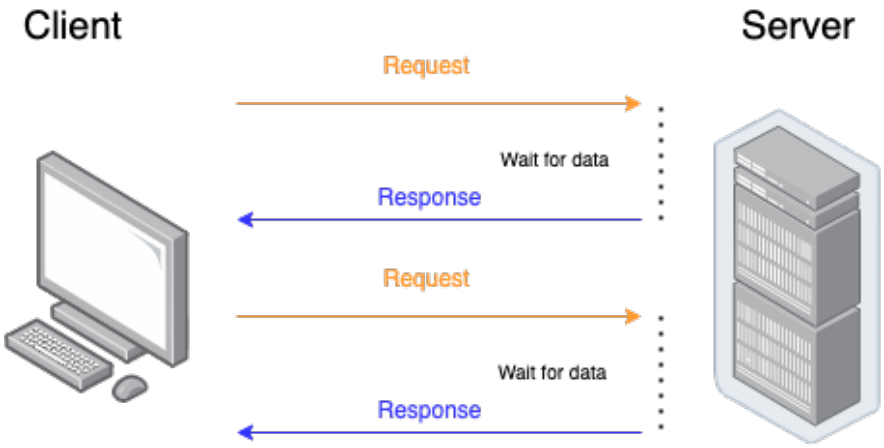


Figure 2.6: HTTP long polling. For every request, the server responds with data when its available.

The solution to this is using the WebSocket protocol. This protocol provides "a full-duplex, bidirectional communication channel that operates through a single socket over the Web and can help build scalable real-time Web applications" [25]. As seen in Figure 2.7, the server receives a message from the client and responds with a *handshake*. After this, a persistent TCP connection is established that allows full-duplex messaging, meaning that the client and server can stream messages to each other simultaneously. This two-way communication method is less resource-intensive on the server than long polling, as only one connection needs to be held open per client rather than one per request from every client.

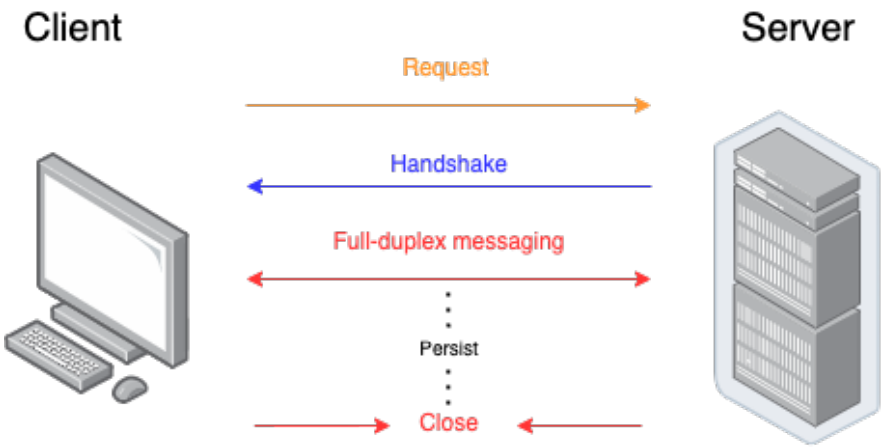


Figure 2.7: The WebSocket protocol.

2.1.10 DDS - The Data Distribution Service

DDS is a *middleware* protocol and API standard for communication in distributed systems. It is managed by the Object Management Group, and they boast "low-latency data connectivity, extreme reliability, and a scalable architecture that business and mission-critical Internet of Things (IoT) applications need" [26]. In a distributed system, a middleware is the software layer between the operating system and the applications that enable easier communication between the system's components. DDS uses the peer-to-peer architecture, which means that data is shared among the components ("peers") in the system instead of through a centralized server or cloud, and the communication between these components is based on the publish/subscribe pattern explained at the end of subsection 2.1.7. Using peer-to-peer and publish/subscribe together provides great scalability, reliability, and flexible network topology. Furthermore, DDS is uniquely data-centric, meaning that it knows what data it stores and how to share that data. Essentially, this reduces the amount of code that needs to be written by developers, as DDS takes care of data-sharing implementations like serialization/de-serialization and maintaining state. This is different from message-centricity, which requires programmers to write the code that sends data messages.

DDS uses RTPS (Real-Time Publish Subscribe) as a wire protocol, i.e. for getting data from one point to another. RTPS is a "field-proven technology that is currently deployed worldwide in thousands of industrial devices" [27]. This protocol enables reliable publish/subscribe communications for real-time applications, and it is designed to automatically discover new applications and services that can join and leave a network without having to be reconfigured. The discovery process happens between domain participants, called Endpoints, with the use of UDP multicast. Multicast is available within local subnets, and with the use of *multicast routing*, it can be used over certain types of internetworks and extended LANs [28]. Since

most inter-domain routers do not implement multicast routing, many networks, like the Internet, do not support UDP multicast [29].

2.2 Hardware

2.2.1 Raspberry Pi Model 4B

Raspberry Pi 4B, displayed in Figure 2.8, is a credit card-sized, single-board computer developed by the Raspberry Pi Foundation. It was first released in 2019 and is a part of the 4th-generation of Raspberry. Ubuntu Core, Raspberry Pi OS, and Windows IoT Core are some of the operating systems that the Raspberry Pi supports. The key feature of this model is the high-performance 64-bit quad-core processor (Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz), and the memory capability of 8 GB [30]. The Raspberry Pi also has several built-in interfaces (see Table 2.1) and 802.11ac Wi-Fi support. The performance-to-size ratio makes the Raspberry Pi suitable for robot projects and experiments.

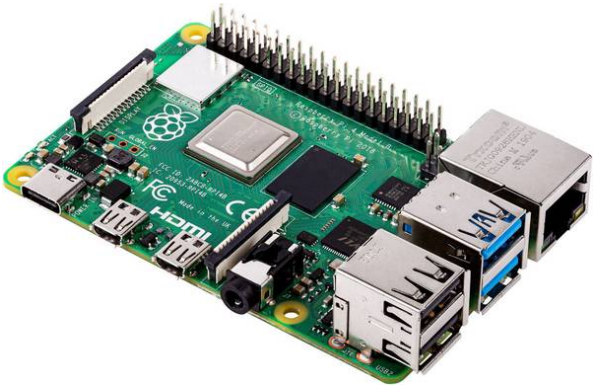


Figure 2.8: Raspberry Pi 4B.

LAN (Gigabit Ethernet) - RJ-45
2-lane MIPI CSI camera port
Raspberry Pi standard 40 pin GPIO header
2-lane MIPI DSI display port
4-pole stereo audio and composite video port
2 x HDMI-output - micro HDMI
2 x USB 2.0 - Type A
2 x USB 3.0 - Type A
USB-C (power only)
Micro-SD card slot

Table 2.1: Raspberry Pi 4B interfaces [31].

According to benchmark experiments conducted by Gareth Halfacree, the Raspberry Pi model 4B sees significant improvements in many critical areas compared to its predecessors [32]. The interesting aspects in the context of this thesis are the "Ethernet throughput" and "USB throughput". There are major improvements in both of these fields, and by looking at them in isolation (see Figure 2.9) one can deduce the increased data bandwidth of this Raspberry Pi compared to the other versions. Halfacree shows that this increased bandwidth comes at the cost of higher power usage and higher thermal temperatures.

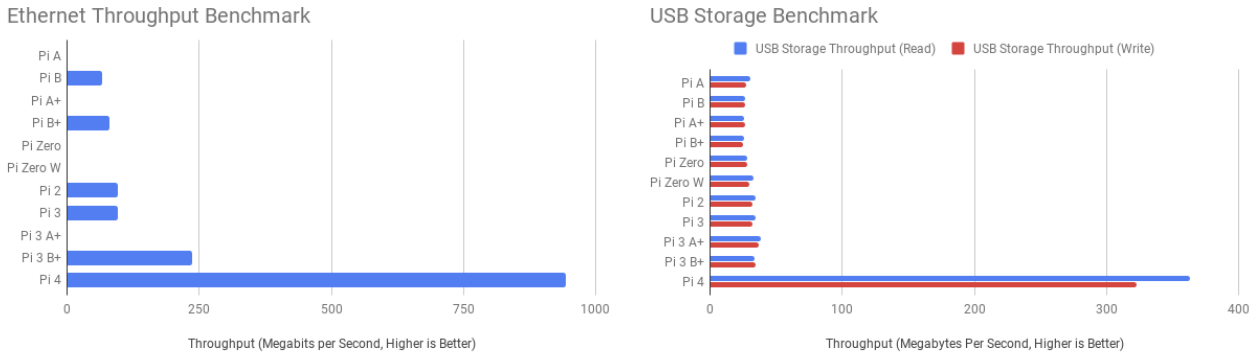


Figure 2.9: Ethernet and USB throughput benchmark of different versions of the Raspberry Pi measured by Gareth Halfacree [32]. (Note: Pi 4 and Pi 4B are synonymous.)

2.2.2 SIM8200EA-M2 and 5G HAT

SIM8200EA-M2, shown on the left side of Figure 2.10, is a wireless cellular communication module developed by SIMCom focusing on the utilization of 5G. It has support for multi-air access technology such as 5G NR and 4G LTE, and communication protocols including TCP/IP and HTTP. The module also has a built-in, global navigation satellite system that integrates various GNSS systems [33].

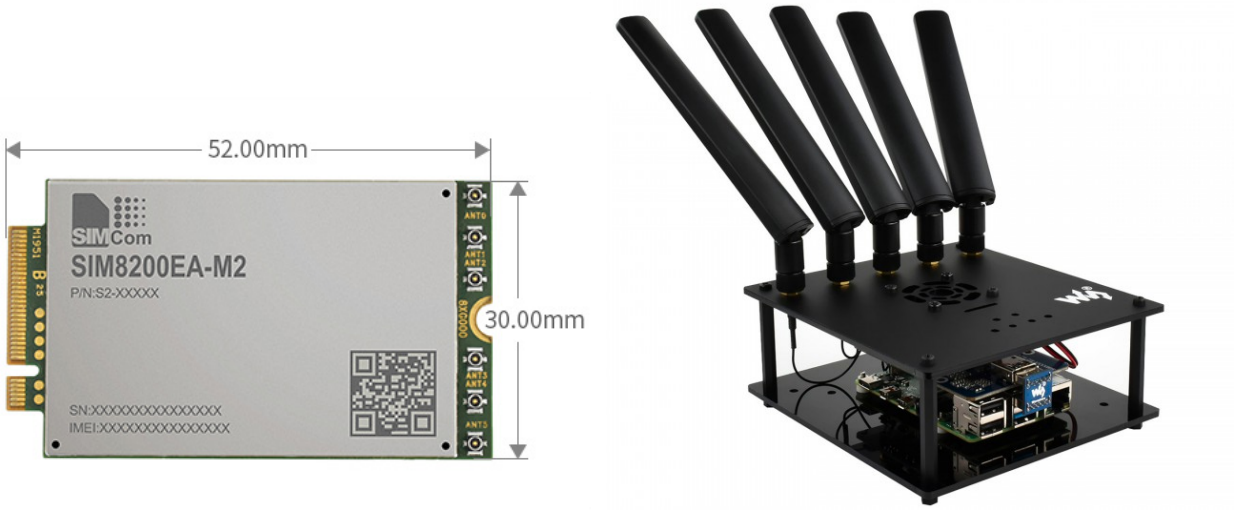


Figure 2.10: SIM8200EA-M2 and 5G HAT for Raspberry Pi [34].

The communication flow of the module is shown in Figure 2.11. The module can be connected to a Raspberry Pi with the use of a SIM8200EA-M2 5G HAT, shown on the right side of Figure 2.10, through USB. The 5G HAT discussed in this thesis is developed by Waveshare, and is, when installed correctly, capable of establishing a 5G connection for a Raspberry Pi with download speeds up to 4 Gbps [34].

The connected device can control the SIM8200EA-M2 module through *AT*-commands. An AT-command (short for attention-command) is used to control a modem [35]. Selecting frequency band and entering PIN codes can be done by sending the corresponding AT-commands to the cellular modem on the SIM8200EA-M2 module.

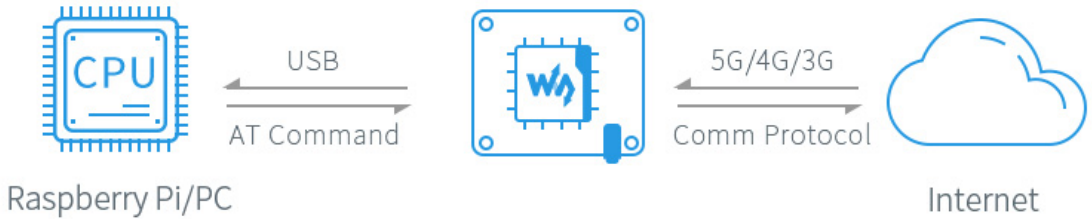


Figure 2.11: Simplified diagram of SIM8200EA-M2 communication flow [34].

The 5G HAT has an LED that indicates the network status of the SIM8200EA-M2 module. If the LED is off, the 5G HAT is either powered off or in sleep mode. IF the LED is always on (no blinking), the SIM8200EA-M2 module is searching for a network. The LED will blink with intervals of different length depending on network registration:

- **100ms on, 100ms off**: data is being transmitted on a 5G-registered network.
- **200ms on, 200ms off**: data is being transmitted on a 4G-registered network.
- **800ms on, 800ms off**: data is being transmitted on a 3G-registered network.

A guide explaining how to assemble the 5G HAT is found in Appendix A.

2.2.3 KMR iiwa

The KMR iiwa, shown in Figure 2.12, is a mobile, autonomous platform consisting of the LBR iiwa—a collaborative robot arm—and the KMP 200 omniMove vehicle. The main purpose of the unit is to navigate a factory floor and picking up, transporting, and placing workpieces and devices. In an effort to adhere to the standards of Industry 4.0, the KMR iiwa is both location-independent and highly flexible [36].

2.2.3.1 KMP 200 omniMove

The KMP 200 omniMove is the vehicle that houses the drive battery and the controller—called the Sunrise Cabinet [37]. The number 200 means that the max payload of the platform is 200 kg.

Figure 2.12: KMR iiwa [37].

2.2.3.2 LBR iiwa 14 R820

The LBR "intelligent industrial work assistant" is mounted on top of the KMP omniMove. The robot arm can only move if the vehicle is stationary, and it needs to be in a specific position for the vehicle to move, referred to in this thesis as the "Drive" position. The jointed-arm robot has seven controllable axes. The maximum payload for the arm is 14 kg, and the maximum reach is 820 mm [37].

2.2.3.3 Technical Specifications

The KMR vehicle cabinet is what contains all the technology that is needed for starting, connecting to, configuring, and operating the KMR iiwa, and this section introduces the parts most relevant for the goal of achieving automatic configuration of the robot.

The front of the vehicle includes an emergency stop device ①, two WLAN antennas ②, a laser scanner for detecting movement in front of the robot ④, a radio receiver antenna ⑤ for an optional radio control unit (a wireless handheld device), and an interface panel ⑥. The WLAN (Wireless LAN) antenna installed on the WLAN access point device that is located on the controller unit inside the KMR cabinet, and its purpose is to reinforce signal strength. The antenna radiates signal in the air at a specified frequency—either 2.5 GHz or 5 GHz. The WLAN AP and the antennas are what allow other devices, such as a workspace

computer, to connect wirelessly to the robot through a Wi-Fi router within a given coverage area. The coverage area in which the KMR iiwa is reachable as an access point is 25 to 50 meters – depending on the structure of the environment.

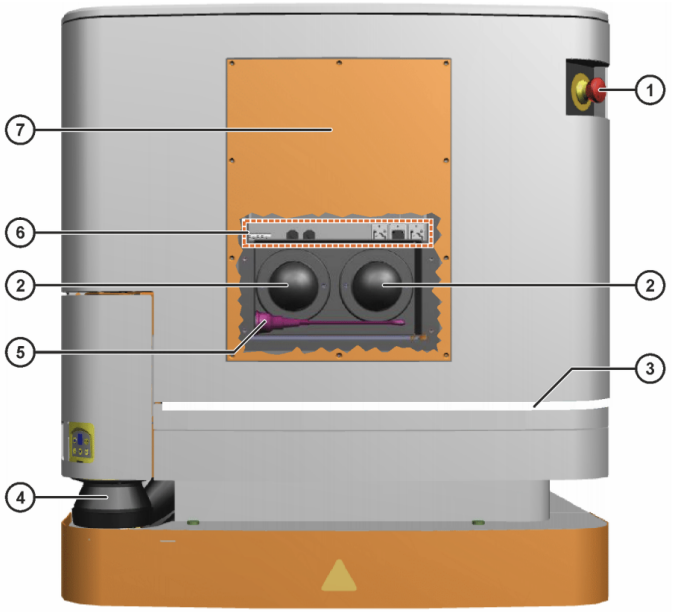


Figure 2.13: KMR iiwa front panel [37].

The rear of the vehicle includes an emergency stop device ①, a control panel with operator control and indicator elements ②, an infrared receiver ④ for establishing a connection with the optional radio control unit, a charging socket ⑤, a laser scanner for detecting movement in front of the robot ⑦, the main switch for turning the KMR on and off ⑧, and an interface panel ⑪. The interface panel ⑪ includes the KLI network connection that is used to connect the KMR to a workspace computer, a Wi-Fi router, or a Raspberry Pi. It is through this connection a Sunrise project is installed or a Sunrise application is synchronized. Information about how to configure the software on the Sunrise Cabinet is found in section 5.4.

2.2.3.4 Operation

The KMR iiwa has three operating modes, T1, T2, and AUT. T1 and T2 are both test modes, while AUT is the automatic (autonomous) mode. The different modes can be selected by using the SmartPAD.

The test modes are equal, with the only exception being that T1 sets the manual max velocity of the KMR to 250 mm/s. In these modes, the robot is operated manually, either by “jogging” or executing an application. Jogging means to move the KMR iiwa according to its own coordinate system [38]. The KMR is fully operational when in test mode, however

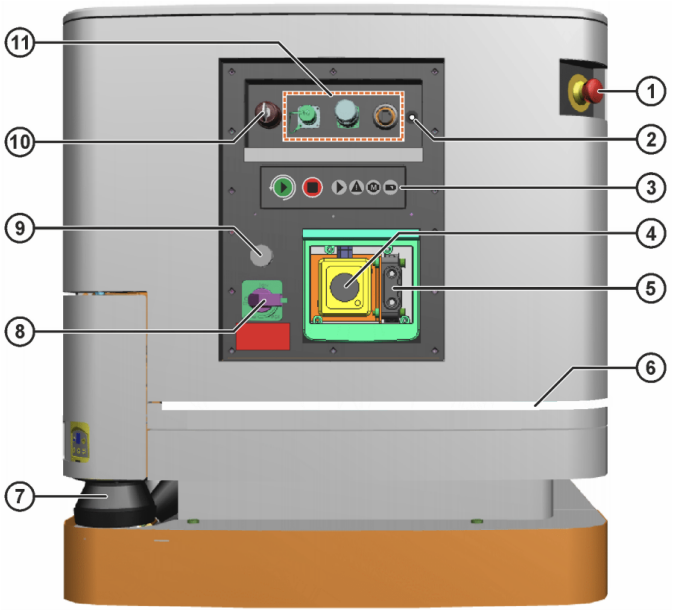


Figure 2.14: KMR iiwa rear panel [37].

the enabling switch on the SmartPAD needs to be pressed. The next subsection explains how to operate the SmartPAD.

In AUT mode the robot can be controlled via KUKA NavigationSolution or by an application. A program called *PositionAndGMSReferencingprogram* needs to be launched, in order to calibrate the sensors of the LBR, before the vehicle can be set in AUT mode [38]. This program is included as an extra safety measure as it checks whether the stored zero position corresponds with the mechanical zero position.

2.2.3.5 SmartPAD

The KUKA SmartPAD is a wired hand-held control panel with a touch-sensitive display that allows the user to operate the KMR iiwa (or other KUKA-robots) manually. It has all of the display functions and operator controls required in order to operate the KMR iiwa. KUKA Sunrise.Mobility software is the user interface of the SmartPAD.

The SmartPAD has multiple enabling switches which have three positions, “Not pressed”, “Center position”, and “Fully pressed”. At least one of the enabling switches must be held in the center position in operating modes T1 and T2 in order to be able to jog the KMR iiwa. These switches serve no use in automatic mode.

The SmartPAD interface is found at the rear of the KMR iiwa. If disconnection of the SmartPAD is configured as allowed, the SmartPAD can be disconnected while the vehicle controller is running. The application which is active will continue to run while the SmartPAD

Figure 2.15: KUKA SmartPAD [39].

is disconnected.

2.2.3.6 Workspace computer

A workspace computer is a necessity when working with KMR iiwa entities. This computer needs to be running Windows 10 with Java JDK and Sunrise Workbench installed. It is important that the workspace computer is on the same network as the KMR iiwa entities. This can be achieved by either connecting directly with an ethernet cable or using a wireless router. When the workspace computer is correctly configured one can create, install and run Sunrise applications, descriptions for which can be found in subsection 5.4.1, subsection 5.4.2 and subsection 5.4.3.

2.3 Software

2.3.1 ROS 2

ROS, or Robot Operating System, is a set of open-source software libraries and tools for creating robot applications [40]. The development of ROS 1 was started in 2007 and has since then been at the forefront of providing software tools for research and development on all types of robots. ROS 2 is a parallel set of packages that was developed as an improvement of the original system but with the intention of being able to be installed alongside ROS 1 rather than replacing it. The latest set of ROS 2 packages (distribution), and the one used in this project, is ROS 2 Foxy Fitzroy [41].

There is growing industrial support for ROS 2, and because it is built on widely deployed standards like IDL, DDS and DDSI-RTPS [42] there is little friction when using it in industrial settings. An example of ROS 2 in the world of Industry 4.0 is the development of 5G- and ROS 2-based Factories of the Future that was started in 2018 by ADLINK Technology together with Fair Friend Group [43]. The aim is to combine 5G network technology with ROS 2 as data-exchange middleware to create a smart robotics industry ecosystem to face the challenging implementations related to Industry 4.0 and future factories.

The main goal of ROS 2 is to improve user-facing APIs while still maintaining the key concepts from ROS 1, which include distributed processing, publish/subscribe messaging, language neutrality, etc. [42]. There are also many important changes from ROS 1 that makes ROS 2 more compatible with Industry 4.0 in general and this project in particular. First of all, it targets newer versions of programming languages like C++11 and Python 3.5 and above. Also, as shown in Figure 2.16, all ROS 2 middleware implementations are based on the widely supported DDS standard, as opposed to ROS 1 which only supports the custom serialization formats TCPROS and UDPROS for message handling [44]. ROS 2 includes a DDS abstraction layer that not only optimizes the use of DDS, but also removes the need for ROS 2 users to be aware of the DDS APIs. In a ROS 1 system, a so-called master process has to be started before anything else. This process is based on the master-slave protocol and works as a DNS server that makes it possible for the nodes to communicate with each other. The implementation of DDS as a middleware in ROS 2 entails the use of a peer-to-peer protocol instead (see subsection 2.1.10 about DDS).

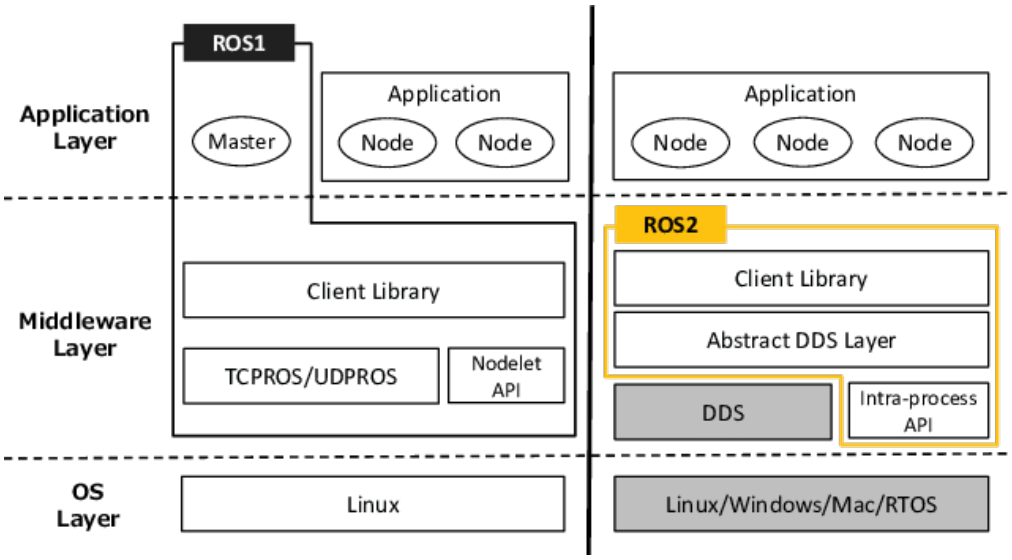


Figure 2.16: Comparison of ROS 1 and ROS 2 architectures [44].

In ROS 2, a *node* is an independently executable process that performs some computation

and is responsible for a single purpose within a system (e.g. one node to control the wheel motors for an AGV and one node to control the robot arm stationed atop the AGV). Nodes are connected in a peer-to-peer fashion, and they are made aware of each other's existence through the DDS discovery process described in subsection 2.1.10. After discovery, the nodes communicate with each other via *messages*, *topics*, *services*, and *actions*. Communication via topics is based on the publish/subscribe pattern, while communication via services is more similar to the client-server pattern. Both of these are explained at the end of subsection 2.1.7. As portrayed in Figure 2.17, a publisher node publishes a message to a topic, while one or more subscriber nodes can subscribe to the given topic and process the message that is published. These messages are simple data structures defined by .msg files. With services, messages are only sent if a *client* node specifically requests them from a *server* node [45]. With this model, two nodes in communication would require three services: a client service at the client node, a server service at the server node, and a service for handling the requests and responses.

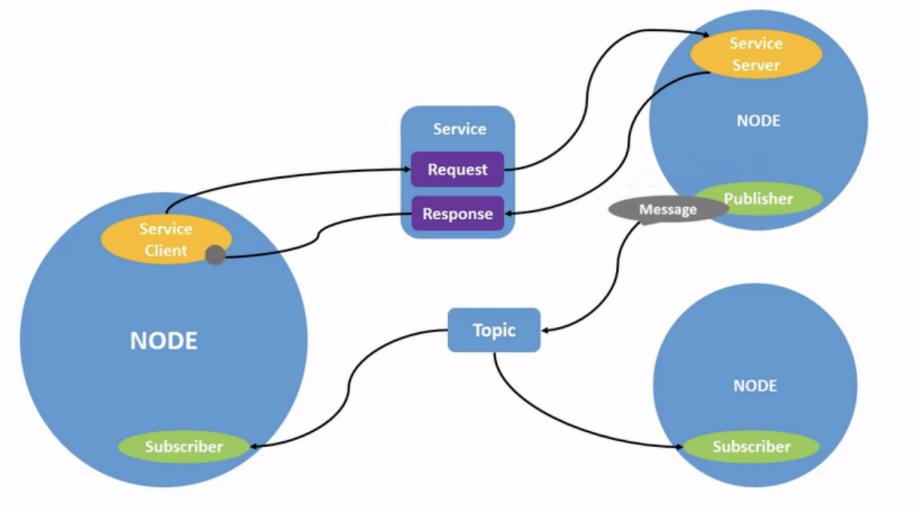


Figure 2.17: Illustration of how communication happens between nodes in a ROS 2 system [46].

The third communication type is called action. Actions are built on top of topics and services, and they consist of three parts: a *goal*, a *result*, and *feedback* [47]. They use topics for feedback and goal status, and services for setting goals and requesting results. The action services are asynchronous, meaning that action clients do not have to wait for results, and they can cancel goals mid-execution. An action client sends a goal request (e.g. move joint on robot arm) to an action server, and the server replies with a response saying that it received the request. The action server then publishes a stream of feedback on a topic that the action client subscribes to until the goal is complete. After this, the action client requests a result from the action server and gets a response (e.g. robot arm movement completed/failed).

2.3.2 Flask

Flask is an extensible "micro web framework" written in Python that provides a solid core for basic web services as well as support for third-party extensions that offer other desirable features. The three main dependencies of Flask include the routing, debugging, and Web Server Gateway Interface (WSGI) subsystems from Werkzeug, the template support from Jinja2, and the command-line integration from Click [48] [49] [50]. Flask has no native support for high-level tasks such as accessing databases, validating web forms, and user authentication, and it is up to the developer to select and use the extensions which satisfy their project [51]. The lightweight nature of Flask makes it easy to implement as a back-end to a web application, and the extensibility provides significant power potential. A developer can easily make their application asynchronous by replacing the Werkzeug WSGI subsystem with any supported Asynchronous Server Gateway Interface (ASGI) extension, and real-time, bidirectional communication can be enabled by simply importing the correct library and implementing its functions. Flask is also widely supported, and thus there are many extensions that provide similar functionality, meaning that a developer can carefully choose the ones that meet their specific needs. Figure 2.18 shows how the Flask environment supports interchangeable components. The figure shows an example with only some of the components that are usable with Flask. Netflix, Twilio, and Uber are just a few of the many popular services that are Flask-powered [52] [53] [54].

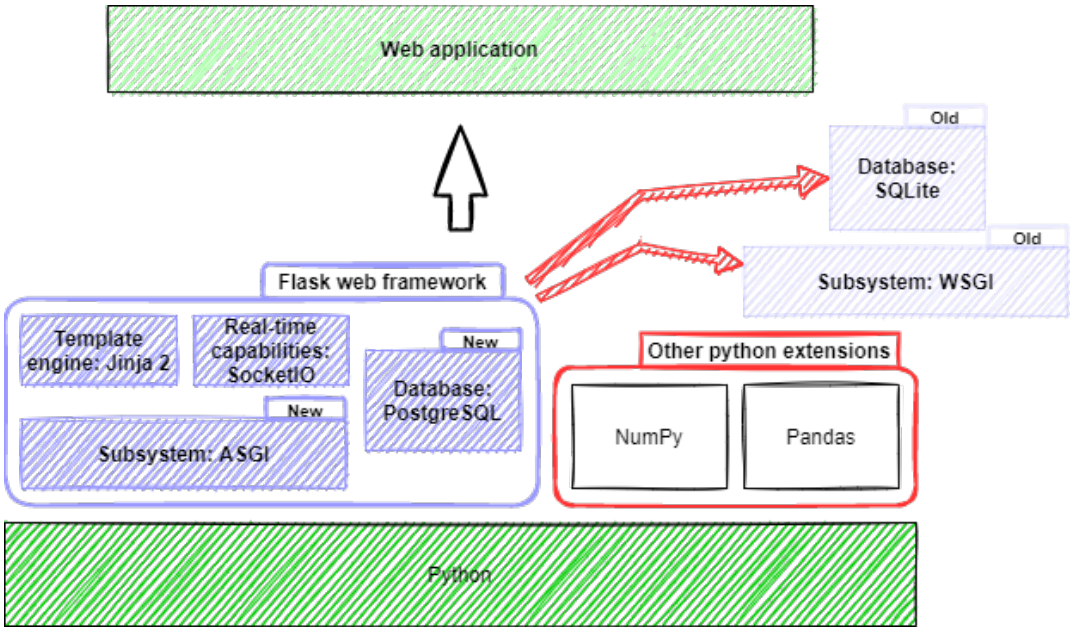


Figure 2.18: The Flask web framework supports a variety of different components, and replacing a database or subsystem can be done without major impact on the underlying structure of the application.

2.3.3 React

React is a declarative, open-source frontend JavaScript library used for creating user interfaces and components. React was released in 2013 and is maintained by Facebook Inc and a group of individual developers [55]. It is mostly used in the development of single-page websites or mobile applications. Important features in React include the ability for developers to create reusable UI components and change data without reloading entire pages. This makes web applications developed with React both scalable and fast. Today, React is being used on popular websites such as Instagram and Facebook.

2.3.4 WebGear

WebGear is an ASGI Video-Broadcaster API used for transmitting Motion-JPEG-frames from a single source to multiple recipients via the browser [56]. WebGear is written in Python and is suitable for projects of the same nature as described in this thesis, as it is easy to set up and use. A computer running Python with an input stream source and an output IP address is all that is needed in order to start a video broadcast. Under the hood, WebGear makes use of an intraframe-only compression scheme where the incoming image feed is first encoded as JPEG-DIB (JPEG with Device-Independent Bit Compression) and then streamed over HTTP using an ASGI Server [56]. This method is not the best for top image quality, but it imposes lower processing and memory requirements, making it suitable for real-time applications. Figure 2.19 shows a snippet from an active broadcasting session using WebGear.

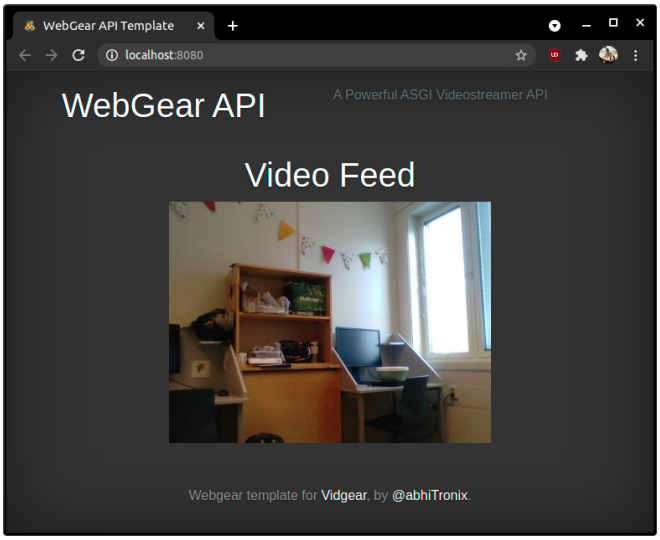


Figure 2.19: A snippet from an active broadcast using WebGear [56].

2.3.5 Hawkeye

Hawkeye is a web-based active monitoring and testing platform used to continuously manage performance and connectivity across networks and systems. It can monitor and test data centers, cloud services, and remote sites all from a single source and tool [57]. Several platforms such as Windows, Linux, iOS, and Android are supported. Hawkeye lets the system architect deploy endpoints on critical parts of the architecture. These endpoints can either be dedicated hardware endpoints or software endpoints deployed through certain applications.

Hawkeye consists of a wide range of pre-calibrated tests such as "Speed test from site to site with advanced configuration on traffic profiling", "Bandwidth availability or verification with TCP-based testing", "IP network SLA verification (one-way delay, jitter, loss)", and many more [58]. These tests can be performed in conjunction with the endpoints mentioned above. Hawkeye provides a web-based console that can be accessed from many devices given correct authentication. This console gives the system architect access to the test results and other useful tools regarding the specific endpoints.

2.3.6 Sunrise.OS

KUKA Sunrise.OS [59] is the system software for KMR iiwa and other industrial robots that are operated with the robot controller KUKA Sunrise Cabinet. This software system provides functions for programming, planning, and configuring lightweight robot applications. Contrary to its name, it is not an actual operating system, but rather a Java application that is run on top of Windows 7, which is the base operating system for KUKA Sunrise Cabinets.

2.3.7 KUKA Sunrise Workbench

KUKA Sunrise Workbench is the development environment for KUKA robots [60]. It offers functionalities for start-up and application development. With regard to start-up, it offers the following abilities:

- Installing the system software on a KUKA robot
- Configuring the robot cell (station)
- Editing the safety configuration
- Creating the I/O configuration
- Transferring the project to the robot controller

While with regard to application development it offers the following abilities:

- Programming robot applications in Java
- Managing projects and programs
- Editing and managing runtime data
- Project synchronization
- Remote debugging (fault location and elimination)

2.3.8 KUKA Robotics API

KUKA Robotics API is an object-oriented Java programming interface for controlling KUKA robots and peripheral devices. It can be imported directly and fully utilized within Sunrise Workbench. As seen in Figure 2.20 Robotics API is separated into two layers, namely the Activity Layer and the Command Layer. It uses Robot Control Core to communicate with the specific hardware components on the robots. Robot Control Core is responsible for real-time hardware control and is integrated with the Realtime Primitives Interface. This interface is used for tasks that need to be executed with real-time guarantees, like robot movements [61]. From a developer's perspective, KUKA Robotics API is essentially a java library consisting of functional methods used to operate KUKA robots.

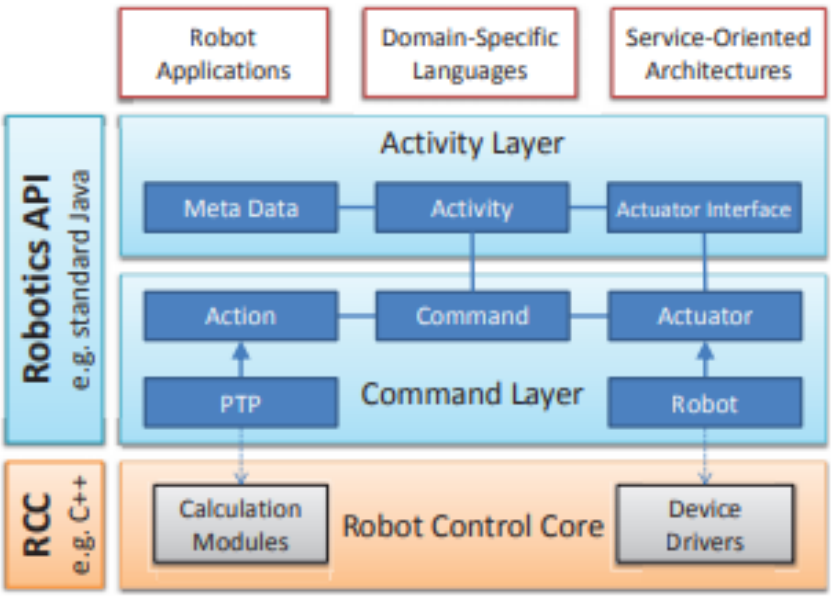


Figure 2.20: Overview of KUKA Robotics API [61].

Chapter 3

Architecture

This chapter explains the system architecture that is presented as a solution to the use cases introduced in section 1.3. The chapter offers a simplified view of the overall system design in an attempt to show the most important segments, and to give insight into how each of these plays an important role in realizing a robust and effective system within the realm of Industry 4.0 and IoT. It is therefore intended for those who wish to understand why the system is designed the way it is and the thought process behind each major design solution. Section 3.1 explains each segment of the system and how they work together, section 3.2 describes how the architecture compares to the design principles of Industry 4.0, while section 3.3 outlines how the design solution aims to realize the goals of the project.

3.1 System Design

The industrial factories of the future will most likely consist of industrial facilities with multiple different robots and a solid 5G infrastructure. In order to support this reality, an architecture needs to be scalable and flexible with regard to different robots, new methods of communication, and the number of concurrent users. Figure 3.1 displays a simplified illustration of a system design that aims to build a foundation for innovative strategies that help reach these goals. From this specified point of view, the system is separated into the four distinct segments "Network", "Industrial Lab", "Control Room", and "User Devices". Each segment contains important features, and the system design is reliant on all four of them to fully operate. In order to maintain scalability and flexibility, each segment is operational independently.

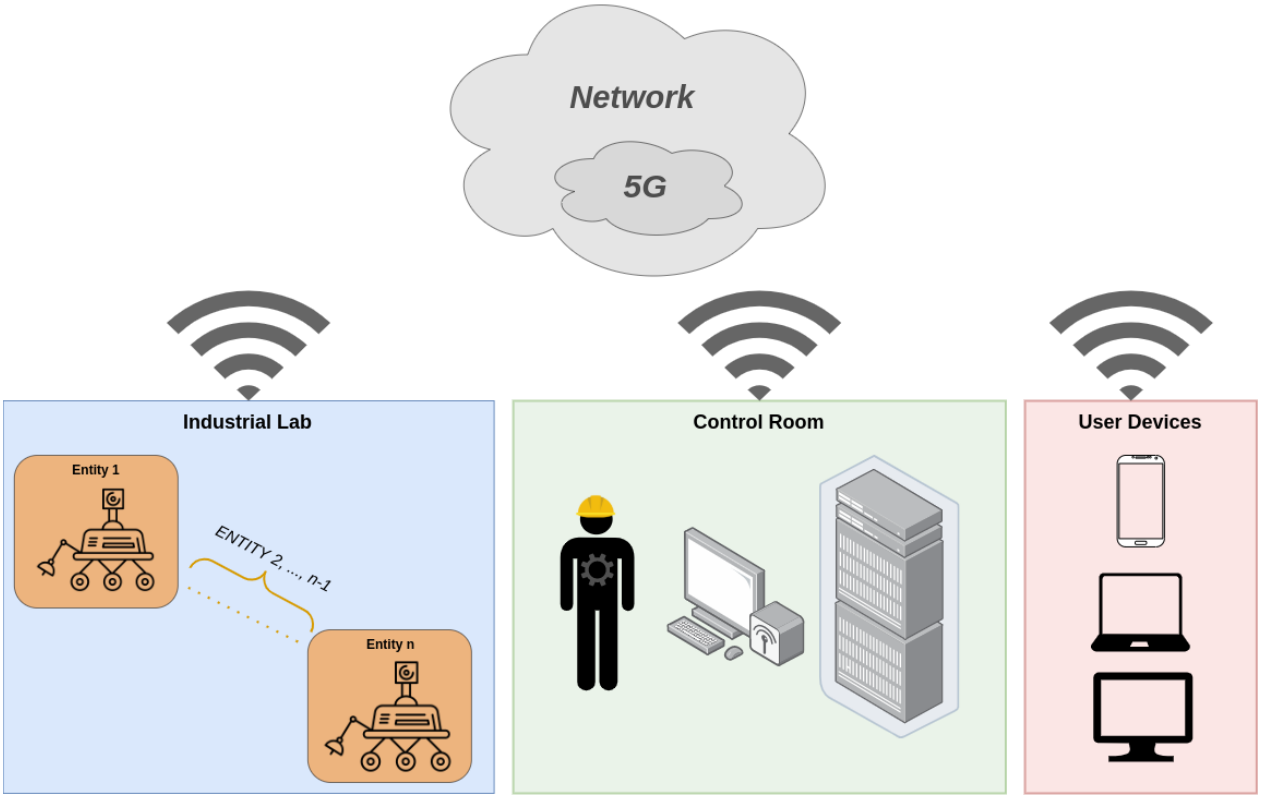


Figure 3.1: Proposed system design with the four distinct segments "Network", "Industrial Lab", "Control Room", and "User Devices".

3.1.1 Network

The network segment is visualized by a *Network* cloud encompassing a smaller *5G* cloud. This represents the fact that, while 5G is an important part of the system, not every part of the system is integrated with 5G-capability—most of it is reliant on Wi-Fi and Ethernet connections. The reasoning behind this separation is explained in more detail in chapter 4. The network segment is vital because it serves as the connection between the other segments. Every segment can, as previously mentioned, operate on its own, but in order to fulfill the requirements set by the use cases, they need to communicate with each other.

The limitations surrounding the public 5G cellular network, outlined in section 1.5, make this the most challenging segment to solve with regard to implementation. This is because a public 5G network is quite restricted in terms of access, and there is little chance of integrating a system with the core functionality of the network within a reasonable amount of time. Chapter 4 describes a solution with workarounds that tackle this limitation. An example of a workaround is not using 5G on server components, since hosting web servers on a public 5G network is deemed nearly impossible.

3.1.2 Industrial Lab

The "Industrial Lab" segment is visualized by a facility consisting of n robots, or, as they are referred to throughout this thesis, *entities*. Communication solutions are developed and implemented to ensure seamless, real-time, and secure connectivity between the entities. This segment is a vital part of Use Case 1.5 and 1.3, as it is where the development made in relation to these use cases will be most apparent. Each entity is connected to 5G and will communicate wirelessly with the components in the "Control Room".

As seen in the illustration of the architecture, no human workers are located in the "Industrial Lab". This segment should be completely autonomous. This design choice is made to emphasize the importance of Use Case 1.3. Human workers should only enter the "Industrial Lab" for edge cases such as maintenance, fault repair, and initial configuration of new entities.

3.1.3 Control Room

The "Control Room" segment is visualized by a human worker, a workstation computer, and a server rack. It is at this location that technicians and "on-site operators" are located. They will remotely monitor the "Industrial Lab" and oversee the work performed by the entities. Databases and other configurations are stored at this location, as opposed to the cloud, for security and latency reasons. The "Control Room" can either be close to or far away from the "Industrial Lab", and each "Control Room" can operate several "Industrial Labs".

3.1.4 User Devices

The "User Devices" segment is visualized by a smartphone, a laptop, and a desktop monitor. These visualizations represent the remote, "off-site" control features and the multi-platform support that the implementation of the architecture offers. The possibility to remote-control a factory would be a part of daily operations and give factory operators enhanced flexibility and improved efficiency. In order for this solution to be viable, a robust security system must be in place. As a security measure, the operators working off-site do not have the same capabilities as the operators in the "Control Room" but instead have specified and targeted functions that suit their tasks.

3.2 Design Principles of Industry 4.0

The design principles of Industry 4.0 aim to ensure that all machines, work pieces, and systems are computerized and that the manufacturing processes are connected through intelligent networks in which they can control each other autonomously. In order for the

architecture to properly comply with Industry 4.0-standardization, it needs to satisfy the following basic principles: interoperability, information transparency, technical assistance, and decentralized decision making.

3.2.1 Interoperability

Interoperability is the ability for machines and humans to connect and interact through the Internet of Things (IoT), which can be said to be the essence of the entire architecture presented in this chapter. With a system implemented based on this architecture, a human can easily interact with a complex robot over the internet and 5G without having to worry about any robot-specific hardware. Multiple entities from different manufactures can be introduced without making any large changes to the system.

3.2.2 Information Transparency

Information transparency is the ability of an information system to use sensor data to create a virtual copy of the physical world. This is fully realizable given an industrial facility with enough entities and sensors, and it would significantly expand the potential of the architecture's functionality. A complete virtual copy of the "Industrial Lab" would mark a significant step towards completing Use Case 1.3. Althought this is an interesting concept to pursue, such features are not within the scope of this thesis.

3.2.3 Technical Assistance

This principle addresses the ability of a cyber-physical system to support humans by conducting tasks that might be unpleasant, too exhausting, or unsafe. Technical assistance is one of the primary purposes of the architecture, as it is supposed to provide easy access to robots and their functionality. The architecture can be considered to satisfy this principle, as it aims to assist operators by providing the ability to control robots safely from a remote location using real-time technology.

3.2.4 Decentralized Decisions

Decentralized decisions refer to an industrial robot's ability to independently make decisions and perform tasks as autonomous as possible. Artificial intelligence and machine learning are methodologies that would satisfy these requirements. Implementing artificial intelligence and machine learning concepts are outside the scope of this thesis. However, since the architecture is designed using generalized principles, there is no reason why future implementations cannot incorporate such solutions.

3.3 Realization of Use Cases

The use cases studied in this thesis are listed below.

- UC 1.5 - Rapid deployment, auto/re-configuration and testing of new robots.
- UC 1.3 - Remotely controlling digital factories.

Creating an architecture that supports the realization of these is a primary focus in this thesis. The architecture outlined in this chapter focuses on separating the different tasks into suitable segments. Choosing to segment the architecture is done because low coupling between components often means increased cohesion throughout the system, and because it is a suitable approach to UC 1.5. Rapid deployment, auto/re-configuration, and testing of new robots would be significantly more difficult to achieve with a highly coupled design. Segmentation allows for high scalability, and further development in each individual segment can be done without jeopardizing the integrity of the system as a whole.

Being able to remotely control digital factories yields many advantages. Hours otherwise spent on manual labor can be used elsewhere, dangerous activities can be performed by replaceable robots, and travel costs for workers can be saved. Remote control of a factory can become a part of daily operations and give factory workers enhanced flexibility and improved efficiency. “User Devices” is designed as its own segment to emphasize the possibilities of remote control. The “User Devices” segment does not have a direct connection with the “Industrial Lab” because of safety concerns. Instead, chosen functions are exposed by the “Control Room” to the devices with the use of an interface.

Chapter 4

Implementation

This chapter describes the implementation and technical details of the system architecture. It serves as a detailed guide to how each part of the system is implemented and works in conjunction with the rest. The chapter is intended for those who wish to recreate parts of the system or the system in its entirety and those who wish to know how or why each specific implementation is done. The chapter includes various snippets of code that serve as additional resources for describing the inner workings of the system. In an attempt to not overwhelm the reader with exceedingly long code listings, parts of the code are omitted. These parts are indicated in the listings as "...". The remaining parts of the code are those deemed most important for understanding the basic idea behind each implementation. Section 4.1 provides a brief overview of how the implementation corresponds with the architecture described in the previous chapter. Section 4.2 outlines the implementation of the "Industrial Lab" segment. This section explains the robot application on the KMR iiwa, the ROS 2 program on the Raspberry Pi that is mounted on top of the KMR iiwa, and the ROS 2 program on the Raspberry Pi that connects the entity to the "Control Room". Section 4.3 describes the implementation of the "Control Room". This section details the various components of the AAS and the implementation of the AAS Video Streaming service. Lastly, an outline of the most important data flow is provided in section 4.4.

4.1 Implemented architecture

Figure 4.1 shows a detailed illustration of how the architecture is realized with a component-based system. The use of several Raspberry Pis allows for flexibility and adaptability, and the software and networking solutions make it so that the components can communicate in real-time. Like the architecture, the system is separated into the four distinct segments "Network", "Industrial Lab", "Control Room", and "User Devices". This implementation satisfies the requirements specified in chapter 3 and is a proposed solution to the problem outlined in chapter 1.

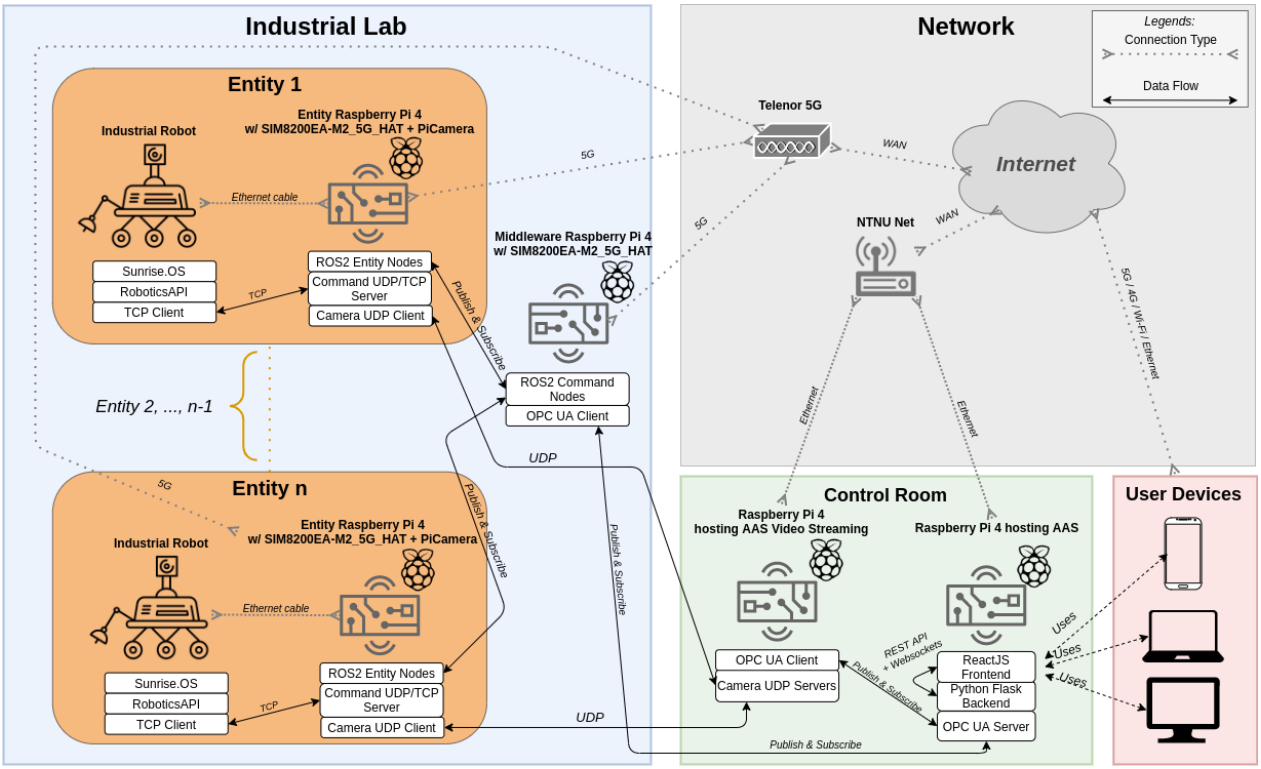


Figure 4.1: Implemented architecture with connections and significant software.

4.2 Industrial Lab

A KMR iiwa AGV, provided by KUKA, is located at NTNU Gløshaugen's Industry 4.0 laboratory and is therefore used in this project. The aim is, by exploring the possibilities with this robot, to establish a broader understanding of the capabilities that can be developed and implemented at other factories with other kinds of robots. The KMR iiwa robot is connected with Ethernet to a Raspberry Pi, which has a 5G HAT and Raspberry Pi Camera Module V2, hereby referred to as the "Entity Raspberry Pi". The Entity Raspberry Pi communicates with ROS 2 over 5G with the other Raspberry Pi, named "Middleware Raspberry Pi", which

is located at the rightmost point of the Industrial Lab seen in Figure 4.2.

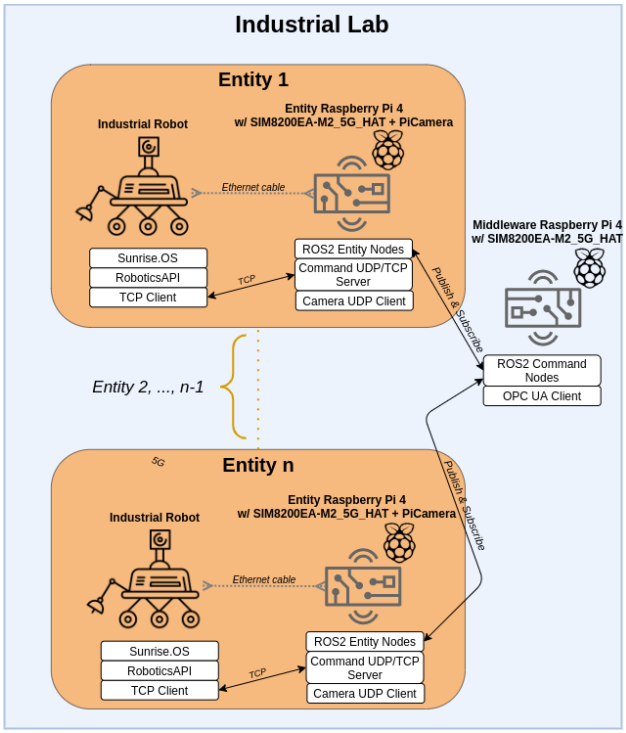


Figure 4.2: Separated look at the "Industrial Lab".

4.2.1 KMR iiwa Implementation

In order to communicate with the KMR iiwa entities, a Sunrise application responsible for handling communication and execution of robot-specific commands needs to be implemented. To fully understand this section it is suggested to read the description of Sunrise.OS, Sunrise Workbench, and KUKA Robotics API which are found in subsection 2.3.6, subsection 2.3.7, and subsection 2.3.8 respectively. The implementation described here is influenced by the code written by two previous students at NTNU, Heggem & Wahl, who also worked with the KMR iiwa [62].

Sunrise applications are written in Java with the help of Sunrise Workbench. To realize the ideas behind the architecture, a baseline for the application needs to be developed. This baseline includes a TCP Socket communication endpoint, commander nodes representing the different components of the KMR iiwa, and a main Sunrise application class.

4.2.1.1 TCP Socket in Java

In order to expose the KMR iiwa to the other components of the architecture, a socket endpoint needs to be implemented. The *ISocket*-interface shown in Listing 4.1 and developed

by Heggem & Wahl is utilized for this purpose.

Listing 4.1: ISocket developed by Heggem & Wahl [62].

```
public interface ISocket {
        public void close();
        public void send_message(String msg);
        public String receive_message();
        public byte[] encode(String string);
        public boolean isConnected();
}
```

To make use of an interface in Java, another class needs to *implement* it. The *TcpSocket* class imports the standard java.net package *Socket* and implements the *ISocket*-interface, which contains the most important methods for a socket. Both the KMP omniMove and the LBR iiwa need to run with instances of the *TcpSocket* class, as this is the client socket that connects to a TCP server socket on the attached Entity Raspberry Pi. This is what allows the Entity Raspberry Pi to sent individual commands to the two components. Details about this TCP server is found in subsubsection 4.2.2.1.

Listing 4.2: Snippets from the TcpSocket.java class which implements ISocket.

```
import java.net.Socket;
...
public class TcpSocket implements ISocket {
        ...
        public TcpSocket(String remote_ip, int port, String node_name) {
                ...
        }
        public Socket connect() {
                while(true) {
                        try{
                                String remotePC = this.remote_ip;
                                TCPConn = new Socket(remotePC, COMport);
                                TCPConn.setReuseAddress(true);
                                System.out.println(this.nodename + " connecting to
                                    ROS over TCP on port: "+ COMport);
                                break;
                        }
                        ...
                }
                ...
    }
...
}
```

4.2.1.2 Commander Nodes

In order to control the movement of the LBR iiwa and the KMP omniMove, accompanying Java classes in Sunrise Workbench need to be implemented. These include the abstract *Node* class with general node-functionality, and the two *Node* sub-classes *KmpCommander* and *LbrCommander*. Listing 4.3 displays a code snippet of the *KmpCommander*. The *LbrCommander* is similar, but with LBR-related variables and methods. Heggem & Wahl's abstract *Node* class is implemented and modified to conform to the *TcpSocket* class outlined in the previous subsection [62]. The modifications are minor and mostly pertain to the specification of the IP address and removal of irrelevant code with regard to the scope of this project. *KmpCommander* and *LbrCommander* are classes that extend, and thereby are sub-classes of, the abstract *Node* class. It is from this sub-classing that they get the ability to open a TCP connection. These classes act as entry points for the node-specific commands sent over TCP. It is with these classes that KMP- and LBR-related commands are executed using KUKA roboticsAPI.

Listing 4.3: Snippets from the KmpCommander.java class which extends Node.

```
import com.kuka.jogging.provider.api.common.ICartesianJoggingSupport;
...
import com.kuka.roboticsAPI
...
public class KmpCommander extends Node {
        KmpOmniMove kmp;
        public KmpCommander(String remote_ip, int port, KmpOmniMove robot, String
            ConnectionType ){
                super(remote_ip, port, ConnectionType, "KMP_commander");
                this.kmp = robot;
                this.kmp_jogger = new KmpJogger((ICartesianJoggingSupport)kmp,
                    jogging_period);

                if (!(isSocketConnected())) {
                        Thread monitorKMPCommandConnections = new
                            MonitorKMPCommandConnectionsThread();
                        monitorKMPCommandConnections.start();
                        } else {
                                setisKMPConnected(true);
                }
        }
        ...
        public void run() {
                ...
                while(isNodeRunning()) {
                        String Commandstr = this.socket.receive_message();
                String [] splt = Commandstr.split(" ");
                if(!getShutdown() && !closed) {
```

```
                        if ((splt[0]).equals("shutdown")) {
                                System.out.println("KMP received shutdown");
                                        setShutdown(true);
                                        break;
                                }
                        if ((splt[0]).equals("setTwist") && getEmergencyStop()) {
                                        setNewVelocity(Commandstr);
                                        System.out.println(Commandstr);
                                }
                }
                }
                System.out.println("KmpCommander no longer running");
        }
        ...
}
```

4.2.1.3 Main Sunrise Application

Section 5.4 describes the mandatory steps surrounding the creation, installation, and execution of a Sunrise application. This application needs an entry point, which is the main Java class that Sunrise.OS compiles and gives the ability to run on the KMR iiwa. The *KmrApp* class, which is shown in Listing 4.4, serves as the entry point for the Sunrise application used in this project. The class extends *RoboticsAPIApplication*, which is imported from the KUKA roboticsAPI, and this is what makes the Sunrise environment recognize it as an entry point. The purpose of the entry point is first to initialize the two command nodes, both of which then connect to a TCP server, and then run them in the application's main loop.

Listing 4.4: Snippets from the KmpApp.java class which extends RoboticsAPIApplication.

```
public class KmrApp extends RoboticsAPIApplication {
    ...
    //Setting specific variables such as IP's, ports, connection-type, etc
    //Initializing commanders
    ...
        public void initialize() {
                //Configure application
                ...
                //Configure robot
                ...
                //Create nodes for communication with corresponding TCP parameters
                kmp_commander = new KmpCommander(...);
                lbr_commander = new Lbrcommander(...);

                //Check if the commander nodes is active
                ...
        }
```

```
        public void run() {
                setAutomaticallyResumable(true);
                ...
                if(!(kmp_commander == null)){
                        if(kmp_commander.isSocketConnected()) {
                                kmp_commander.start();
                        }
                }
                if(!(lbr_commander == null)){
                        if(lbr_commander.isSocketConnected()) {
                                lbr_commander.start();
                        }
                }
                while(AppRunning) {
                        AppRunning = (!(kmp_commander.getShutdown() || lbr_commander.
                            getShutdown()));
                }
                System.out.println("Shutdown message received in main application");
                shutdown_application();
        }
        ...
        public static void main(String[] args){
                KmrApp app = new KmrApp();
                app.runApplication();
        }
}
```

4.2.2 Entity Raspberry Pi

The Raspberry Pi that is mounted on and connected to an Industrial Robot, as illustrated in Figure 4.2, works as an abstraction of the entity. This abstraction is realized through a ROS 2 program that runs multiple ROS *nodes* that act as the different components of the entity. The nodes connect to the entity components using TCP, and through this connection, they can send commands that have been received from the Middleware node described in subsubsection 4.2.3.1. In the context of the experiments presented in this thesis, the entity is the KMR iiwa, and the components are the KMP omniMove, the LBR iiwa, and the Raspberry Pi camera. Due to having only the KMR iiwa available, the implementation presented in this section provides full support only for the use of the KMR iiwa as an entity. The Entity Raspberry Pi has an attached 5G HAT and is connected to the internet with 5G.

4.2.2.1 TCP Socket in Python

In order to connect a TCP client on an entity to a ROS node on the Entity Raspberry Pi, a TCP server needs to be hosted on the ROS node. This server socket is implemented in Python, and it is run as a part of the ROS 2 project explained in subsubsection 4.2.2.2. A code snippet in Listing 4.5 shows the essential part of the implementation. Each component on an entity can connect to a corresponding instance of *TCPSocket*. When a node is executed, the socket waits for a connection from a component on the entity. After a connection is established, the socket is ready to send commands for the entity to execute and receive shutdown messages from the entity if anything happens during runtime. If a shutdown message is sent from a component, such as the KMP omniMove, the server socket will stop the execution of its corresponding ROS node and close the socket connection with the client on the KMR iiwa. After the ROS node execution has been thwarted, the node will reinitialize, and its server socket will again wait for a connection from the KMP omniMove. Explanations for the technical terminology used with regards to ROS 2 are omitted in this section for the sake of avoiding redundancy, and they are described in more detail in subsubsection 4.2.2.2.

Listing 4.5: Snippet of code from tcpSocket.py.

```
...
class TCPSocket:
    def __init__(self, ip, port, node):
        ...
        self.node, self.ip, self.port, self.isconnected = node, ip, port, False
        threading.Thread(target=self.connect_to_socket).start()

    def connect_to_socket(self):
        ...
        try:
            self.conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server_address= (self.ip,self.port)
            self.conn.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1)
            self.conn.bind(server_address)
        ...
        self.conn.listen(3)
        while (not self.isconnected):
            try:
                self.client_socket, client_address = self.conn.accept()
                self.conn.settimeout(0.01)
                self.isconnected = True
            ...
        ...
        while self.isconnected:
            try:
                # Wait for received message from a client on KMR iiwa
```

```
                data = self.recvmsg()
            ...
        self.client_socket.close()
        self.shutdown()

    def shutdown(self):
        self.node.publish_status(0) # 0 = offline
        self.conn.close()
        self.isconnected = False
        self.node.tear_down() #Stop node execution and reinitialize

    def send(self, cmd): # Send commands to the entity
        self.client_socket.sendall((cmd + '\r\n').encode("utf-8"))

    def recvmsg(self): # Receive messages from the entity
        msg = self.client_socket.recv(1024).decode("utf-8")
        if msg == "shutdown":
        self.client_socket.shutdown(socket.SHUT_RDWR)
            self.isconnected = False
        return msg
```

4.2.2.2 ROS 2 - Command Nodes

As stated in subsection 2.3.1, a node in ROS 2 is an executable process with a single purpose. In Python (and C++), these nodes are represented by single files that are made executable in a ROS project environment, called a *workspace*. The easiest way to implement a node is by utilizing the ROS Client Library *rclpy* [63], which provides a Python API for interacting with ROS 2. This API includes the *Node* class from which a Python class can inherit the appropriate ROS 2 properties. Listing 4.6 provides a snippet of code with explanatory comments showing the ROS node that works as an abstraction of the LBR iiwa robot arm. It is implemented as a PubSub-node (publisher/subscriber), as it both subscribes to messages sent from the Middleware Raspberry Pi and publishes status updates for the LBR iiwa to the Middleware Raspberry Pi. Suppose there are multiple KMR iiwas connected to the system simultaneously. In that case, the Middleware Raspberry Pi needs a way to differentiate a component on one entity from a similar component on another. This is done by having a unique ID for each node instance, represented by the `self.id` field. The node communicates over four ROS topics:

- **"manipulator_vel_(ID)"**. Subscribers to this topic receive command messages for a specific LBR iiwa component.
- **"lbr_shutdown_(ID)"**. Subscribers to this topic receive messages when a specific LBR iiwa should be shut down and the TCP connection closed.

- **"status_check"**. Subscribers to this topic receive messages when the Middleware Raspberry Pi asks for information about whether or not the component node is running.
- **"lbr_status"**. A topic for publishing the status of a specific LBR iiwa component.

Listing 4.6: Snippet of code from lbr_command_node.py.

```
from rclpy.node import Node
from tcpSocket import TCPSocket
...
class LbrCommandNode(Node):
    def __init__(self, connection_type, ...):
        #Assign a name to the Node, making it identifiable within the ROS environment
        super().__init__('lbr_command_node')
        self.name, self.robot, self.status, self.id, port = ...

        if connection_type == 'TCP':
            #Create an instance of a TCP server socket and attempt to establish a
                connection with the LBR iiwa
            self.socket = TCPSocket(ip, port, self.name, self)
        ...

        # Make a listener for relevant topics
        sub_manipulator_vel = self.create_subscription(String, 'manipulator_vel_' +
            str(self.id), self.manipulator_vel_callback, 10)
        sub_shutdown = self.create_subscription(String, 'lbr_shutdown_' + str(self.id
            ), self.shutdown_callback, 10)
        sub_status_check = self.create_subscription(String, 'status_check', self.
            status_callback, 10)

        # Publishers
        self.lbr_status_publisher = self.create_publisher(String, 'lbr_status', 10)
        ...
    ...

def main(...):
    ...
    while True:
        rclpy.init(...) #Initialize ROS communications
        lbr_command_node = LbrCommandNode(...) #Declare ROS node
        rclpy.spin(lbr_command_node) #Process work that waits to be executed, i.e.
            start node and wait for callbacks

#This will be called when the executable is run
if __name__ == '__main__':
    main()
...
```

When a message is received on a topic, a *callback* function is called. For instance, the "manipulator_vel_(ID)" topic has an associated callback function `manipulator_vel_callback()`, which is shown in Listing 4.7. This function sends the message data to the TCP client on the entity that represents the LBR iiwa. The code snippet also shows the other functions related to the *LbrCommandNode* class, including `publish_status()` and `tear_down()`. When the TCP connection to the LBR iiwa client is either opened or closed, the `publish_status()` function publishes a status update on the aforementioned "lbr_status" topic. Here, the ID of the node is reflected in the message rather than the topic name. The `tear_down()` function represents an important aspect regarding the nodes running on the Entity Raspberry Pi, which is the ability to be torn down and reinitialized during runtime. The previous section briefly mentions how a TCP server socket instance can stop its corresponding ROS node execution. This is done by calling the `tear_down()` function. When a node is torn down, the running—or "spinning"—of the node stops, and the node is initialized again with a new subscription to the relevant topics and a new TCP server socket. This process is important because it makes it so that the node does not need to be executed manually after the connection with a TCP client socket is closed due to the KMR iiwa being shut down or going offline.

Listing 4.7: Snippet of code from lbr_command_node.py showing the various function definitions.

```
...
class LbrCommandNode(Node):
    ...
    def shutdown_callback(self, data):
            self.soc.send("shutdown")
            self.soc.close()

    def manipulator_vel_callback(self, data):
        msg = 'setLBRmotion ' + data.data
        self.soc.send(msg)

    def status_callback(self, data):
        print(data.data)
        self.publish_status(self.status)

    def publish_status(self, status):
        """
            'status' is either 0 (offline) or 1 (online).
        """
        msg = String()
        msg.data = self.id + ":" + self.robot + ":lbr:" + str(status)
        self.lbr_status_publisher.publish(msg)
        self.set_status(status)

    def set_status(self, status):
```

```
        self.status = status

    def tear_down(self):
        try:
            self.destroy_node()
            rclpy.shutdown()
            print(cl_green("Successfully tore down lbr node"))
        except:
            print(cl_red('Error: ') + "rclpy shutdown failed")
...
```

A similar class to that of *LbrCommandNode* is implemented for communicating with the KMP omniMove component, but the code for this is omitted for the sake of avoiding redundancy. The topic names and callback functions differ slightly to comply with the nature of the KMP omniMove component, but the overall functionality is the same. How the messages subscribed to by the ROS nodes on the Entity Raspberry Pi are published is shown and explained in subsubsection 4.2.3.1.

4.2.2.3 ROS 2 - Camera Node

In order to present a video stream from an entity to an operator of the AAS, the camera mounted on the Entity Raspberry Pi needs a way to communicate with the outside world. This is done with a ROS node represented by the file `camera_node.py` shown in Listing 4.8. This ROS node makes it so that the video that is captured by the camera module is published and that the AAS is notified by the stream's existence. Like the two command nodes described in the previous section, the camera node includes an ID field for distinguishing a camera on one entity from the camera on another. It is also implemented as a PubSub-node, and it communicates with the Middleware Raspberry Pi using the following ROS topics:

- **"handle_camera_(ID)"**. Messages published on this topic tell the specific node to either start or stop the video stream.
- **"status_check"**. Subscribers to this topic receive messages when the Middleware Raspberry Pi asks for information about whether or not the component node is running.
- **"camera_status"**. A topic for publishing the status of a specific camera component.

Like the two command nodes, the camera node can publish its status so that the Middleware Raspberry Pi can track which components are online and offline. Whenever a message containing a "start"-command is published on the "handle_camera_(ID)" topic, the node publishes its own status as online and starts a *subprocess*. A subprocess is a module in Python that runs a new program through Python code by spawning an additional thread. For the camera node, this new thread is used to start a video stream by running the bash script

`startcamera.sh`. When a message containing a "stop"-command is published, the node publishes its own status as offline, the subprocess is terminated, and the video capturing stops. The `main()` function, that handles initialization of ROS communication, declaration of the camera ROS node, and the node spinning, is omitted from the code snippet, as it is similar in nature to what is shown for the *LbrCommandNode* in Listing 4.6.

Listing 4.8: Snippet of code from camera_node.py.

```
import rclpy
from rclpy.node import Node
import subprocess
...
class CameraNode(Node):
    def __init__(self, robot):
        super().__init__('camera_node')
        self.name, self.robot, self.status, self.ip, self.proc = ...

        # Subscribers
        sub_camera = self.create_subscription(String, 'handle_camera_' + str(self.id)
            , self.handle_camera, 10)
        sub_status_check = self.create_subscription(String, 'status_check', self.
            status_callback, 10)

        # Publishers
        self.camera_status_publisher = self.create_publisher(String, 'camera_status',
             10)
        ...

    def status_callback(self, data):
        self.publish_status()

    def handle_camera(self, data):
        if data.data.lower() == "start" and self.status == 0:
            print(cl_green("Starting camera"))
            self.proc = subprocess.Popen(["/bin/bash", "kmr_communication/
                kmr_communication/script/startcamera.sh", self.ip])
            self.status = 1
        elif data.data.lower() == "stop":
            try:
                self.status = 0
                self.proc.terminate()
                self.proc = None
                print(cl_green("Stopping camera"))
            except AttributeError:
                print(cl_red("Camera was never started, therefore never stopped"))

        self.publish_status()
```

```
    def publish_status(self):
        msg = String()
        msg.data = self.id + ":" + self.robot + ":camera:" + str(self.status) + ":" +
            str(self.ip) #ip = ip:port
        self.camera_status_publisher.publish(msg)
...
```

Listing 4.9 shows the script that starts the video capturing on the camera module. When a camera node starts a subprocess for capturing video, the UDP URL associated with the node is passed to the subprocess. This URL is the IP address of the Raspberry Pi that is hosting the AAS Video Stream (see Figure 4.1) and an available port. Adding this URL as a parameter to the video capturing tool makes it so that the video is streamed to the "Control Room" using the UPD transport protocol. The video stream is the only part of the Entity Raspberry Pi that does not go through the Middleware Raspberry Pi before reaching the AAS.

Listing 4.9: startcamera.sh uses the Raspberry Pi camera module's command line tool to capture video and stream it over UDP.

```
#!/usr/bin/env bash
udp_ip=$1

while true; do sleep 2; raspivid -a 12 -t 0 -b 8000000 -fps 24 -w 640 -h 480 -o udp
    ://$udp_ip -n; done
```

4.2.2.4 ROS 2 - Building

A Python file with a node implementation is not of any use by itself. In order for the file to be regarded as an executable, it first has to be organized in a ROS 2 container called a *package*. This package then has to be built within the ROS 2 workspace for the executable to be run. ROS 2 workspaces and packages are created using the ROS 2 toolset [41], and ROS Index recommends using the build system *ament* [64] when creating packages [65]. There are no limits to how many packages can be created within a given ROS 2 workspace, and these can be built using the *Colcon* build tool [66]. This is a command-line tool, also recommended by ROS Index, for automating the process of setting up the ROS 2 environment for building, testing, and using several packages at once. For Colcon to know what to do with the different files in a package, a set of required files will be included in the package after the initial creation. One of these files is *package.xml*, which contains meta-information about the package. The others are dependent on what build type is used, which is usually Python (*ament_python*) or CMake (*ament_cmake*)—both of which have been tested in relation to this thesis. When using Python as build type, the required files are *setup.py*, which contains instructions for how to install the package, and *setup.cfg*, which tells ROS 2 how to find the executables

within the package. Listing 4.10 is a code snippet showing how the *setup.py* file establishes rules for how to install the relevant files. The equivalent with CMake is *CMakeLists.txt*, and the code snippet in Listing 4.11 shows the parts of this file that have the same functions as the parts shown for *setup.py*. Much has been left out of the code snippets that are important for the building of a ROS 2 package but irrelevant for the scope of this chapter.

Listing 4.10: Snippet of code from setup.py.

```
from setuptools import setup
package_name = 'kmr_communication'
setup(
    ...
    data_files=[
        #Define build instructions for launch script and configuration files
    ],
    #Tell ROS 2 which files to be included as modules
    scripts=[package_name + '/script/tcpSocket.py']
    ...
    #Define executables
    entry_points={
        'console_scripts': [
            'lbr=kmr_communication.nodes.lbr_command_node:main',
            'kmp=kmr_communication.nodes.kmp_command_node:main',
            'camera=kmr_communication.nodes.camera_node:main',
        ],
    },
)
```

Listing 4.11: Snippet of code from CMakeLists.txt.

```
...
project(kmr_communication)
...
#Installing the python modules for the package
ament_python_install_package(script/)
ament_python_install_package(nodes/)
...
# Install python scripts
install(
  DIRECTORY launch config script
  DESTINATION share/${PROJECT_NAME}
)

install(
  PROGRAMS
    nodes/kmp_command_node.py
    nodes/lbr_command_node.py
    nodes/camera_node.py
```

```
    script/tcpSocket.py
    DESTINATION lib/${PROJECT_NAME}
)
```

4.2.2.5 ROS 2 - Launching Multiple Nodes

The previous segment describes the necessities surrounding the creation and execution of a single ROS node. The AAS presented in this thesis is but a small representation of what a full-fledged system for remote operation of industrial robots can look like; however, it is complex enough that running each ROS node individually would be considered malpractice. In order to avoid this, ROS 2 has a launch system that allows multiple executables to be run simultaneously from one single launch script. The launch configurations are specified in a Python file that includes what nodes to run, where to run them, and what arguments and parameters to pass [67]. Listing 4.12 shows a snippet of the launch script that runs the nodes required for communication between the AAS and the KMR iiwa. The nodes run in this script are the "lbr_command_node" for operating the LBR iiwa robot arm, the "kmp_command_node" for operating the KMP omniMove, and the "camera_node" for streaming video. There is also a set of parameters that need to be sent to the nodes regarding connection-specific details, namely what IP address the KMR iiwa can connect to, a port for each component, and a UDP URL to which the camera node can stream video. The parameters are defined in a separate *YAML* file, shown in Listing 4.13, and imported to the launch script. This ensures both higher cohesion and looser coupling, making it easier to expand with more robots and components in the future.

Listing 4.12: Snippet of code from kmr.launch.py showing ROS 2 launch configurations for the Entity Raspberry Pi.

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description(argv=sys.argv[1:]):
    pkg_name = 'kmr_communication'
    connection_type_TCP='TCP'
    robot="KMR"

    param_dir = LaunchConfiguration(
        'param_dir',
        default=os.path.join(
            get_package_share_directory(pkg_name),
            'config',
            'bringup.yaml'))
```

```
    return LaunchDescription([
        DeclareLaunchArgument(
                'param_dir',
                default_value=param_dir,
                description='Full path to parameter file to load'
            ),

        Node(
            package=pkg_name,
            executable="lbr_command_node.py",
            name="lbr_command_node",
            ...
            arguments=['-c', connection_type_TCP, '-ro', robot],
            parameters=[param_dir]
        ),

        #Not shown are the nodes for the KMP commands and camera.
        ...
```

Listing 4.13: Snippet of code from bringup.yaml showing the parameters sent to each of the nodes.

```
connection_params:
  ros__parameters:
    connection_type: 'TCP'
    robot: 'KMR'

lbr_command_node:
  ros__parameters:
    id: '1' # Robot ID
    port: '30005'
    KMR/ip: '172.31.1.69'

kmp_command_node:
    # Similar to lbr_command_node

camera_node:
  ros__parameters:
    id: '1'
    udp/ip: '129.241.90.39:5000'
```

The launch script needs to be built and run inside its corresponding ROS 2 package as described in subsubsection 4.2.2.4. When it is run, the two command nodes are started and await a connection from the TCP clients on the KMR iiwa, while the camera node starts recording with the camera attached to the Entity Raspberry Pi and sends the video stream to the Middleware Raspberry Pi.

4.2.3 Middleware Raspberry Pi

Similar to how the Entity Raspberry Pi derives its name from being an abstraction of a robot entity in the "Industrial Lab", the Middleware Raspberry Pi is exactly what the name suggests, a middleware. It is also similar to the Entity Raspberry Pi in that it has a 5G HAT and is connected to the Internet with 5G. The function of this middleware is to bridge the communication between an entity and the AAS, i.e., translating messages from the AAS into something an entity can understand and vice versa. As described in subsection 4.2.2, the Entity Raspberry Pi communicates with the middleware via the ROS 2 publisher/subscriber-model. Since DDS does not support data traffic over the Internet, as explained in subsection 2.1.10, ROS 2 is not an option for communication between the Middleware Raspberry Pi and the AAS. Therefore, this connection is resolved using the machine-to-machine communication protocol OPC UA. Due to having only the KMR iiwa available, the implementation that is presented in this section provides full support only for the use of the KMR iiwa as an entity. Still, with the goal in mind of meeting the requirements of Use Case 1.5, parts of the code have been made to support deployment and operation of multiple types of robot entities and components.

4.2.3.1 Combining ROS 2 and OPC UA

The code running on the Middleware Raspberry Pi is, like for the Entity Raspberry Pi, a ROS 2 program. This program consists of a hybrid node represented by the `opcua_ros2_pubsub.py` file. It is described as a hybrid node because it combines ROS 2 and OPC UA functionality in one single running instance. Listing 4.14 is a snippet from the node file showing how the node is initiated and run with an OPC UA client. The `main()` function creates an OPC UA client that attempts to connect to the OPC UA server on the AAS Internal Interface described in subsubsection 4.3.1.1. After a connection is established, three ROS publisher node instances are created for communication with the "lbr_command_node", "kmp_command_node", and the "camera_node" that are all running on the Entity Raspberry Pi. Next, an OPC UA subscription is created for each node to handle events being published from the AAS. With this configuration, full-duplex communication is achieved between the Entity Raspberry Pi and the AAS using a combination of ROS 2 publish-subscribe messaging and OPC UA event handling. In order to spin multiple nodes instances from the context of a single ROS node, each instance needs to be run inside its own thread. This is achieved through the use of a `MultiThreadedExecutor`. The executor is initialized with the number of needed threads, and then the node instances are added to it. The executor starts by "spinning" it in the way one would do with a regular ROS node. The definition of the classes that handle communication is omitted from the listing with respect to readability, and they are instead shown separately.

Listing 4.14: Snippet of code from opcua_ros2_pubsub.py showing how the node is initialized and run.

```
import rclpy
from rclpy.executors import MultiThreadedExecutor
from opcua import ua, Client
...
# Define classes
...
def main(...):
    rclpy.init(...)

    isConnected = False
    opcua_client = Client("opc.tcp://" + args.domain + ":4841/freeopcua/server/")

    while not isConnected:
        try:
            opcua_client.connect()
            isConnected = True
            print("Successfully connected with OPC UA server on: " + args.domain + "
                :4841")
    ...
    lbr_event = root.get_child([..., "2:LBREvent"])
    kmp_event = root.get_child([..., "2:KMPEvent"])
    camera_event = root.get_child([..., "2:CameraEvent"])

    lbr_publisher = LBRPubSub(obj)
    kmp_publisher = KMPPubSub(obj)
    camera_publisher = CameraPubSub(obj)

    lbr_sub = opcua_client.create_subscription(100, lbr_publisher)
    kmp_sub = opcua_client.create_subscription(100, kmp_publisher)
    camera_sub = opcua_client.create_subscription(100, camera_publisher)
    ...

    try:
        executor = MultiThreadedExecutor(num_threads=3) # Executor needed to spin
            multiple node instances simultaneously
        executor.add_node(lbr_publisher)
        executor.add_node(kmp_publisher)
        executor.add_node(camera_publisher)
        executor.spin() # Run executor
    finally:
        # Shut down executor, destroy nodes and unsubscribe from OPC UA events

    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

The three node instances that are added to the `MultiThreadedExecutor` are all sub-classes of *PubSub*, which again is a sub-class of rclpy's *Node*. This way, all the classes inherit the appropriate ROS 2 properties. Listing 4.15 shows how the class *PubSub* works. When either of the three sub-classes is initialized, they also inherit the properties of this class. This means that each node includes a subscription that listens for status updates from their corresponding node on the Entity Raspberry Pi. For instance, when an instance of *PubSub* is initialized for communication with the LBR iiwa, the `component` string will be "lbr". This makes the topic on which the aforementioned subscription receives messages be "lbr" + "_status", which coincides with the topic to which *LbrCommandNode* from Listing 4.6 publishes messages, namely "lbr_status". When a message is received on the "_status" topic for either of the three node instances, the `status_callback` function is called. This function takes the data from the ROS message and converts it to a regular string. It then send it over OPC UA by calling the `update_status()` function, mentioned in subsubsection 4.3.1.1, on the AAS Internal Interface.

PubSub also includes a publisher on the topic "status_check", which, as mentioned in subsubsection 4.2.2.2, is used to check whether or not the component nodes on the Entity Raspberry Pi are online or not. The `self.stauts_checker_publisher` will publish one check for each node instance when they are initialized. This is useful for situations in which the Middleware Raspberry Pi has gone offline, or the ROS 2 program has stopped working for some reason. In subsubsection 4.3.1.1 it is explained how a database containing information about the various entities in the system is implemented. This database is updated each time a new status from a component is published from the Entity Raspberry Pi. Suppose the Middleware Raspberry Pi is down for any reason. In that case, the line of communication is broken between the AAS Internal Interface and the Entity Raspberry Pi, meaning that if anything changes for any component on any entity, the database will not be updated to represent this new change. So, with the `self.status_checker_publisher`, the hybrid node on the Middleware Raspberry Pi can ask for a status update when it restarts after a period of being down, which will initiate a response from the Entity Raspberry Pi. The status update will then be forwarded to the AAS, which will update the database.

Listing 4.15: Snippet of code from opcua_ros2_pubsub.py showing the PubSub super class.

```
from std_msgs.msg import String
from rclpy.node import Node
...
class PubSub(Node):
    def __init__(self, component, ua_obj):
        super().__init__(component + '_hybrid_node')
        self.component = component
        self.server_obj = ua_obj

        self.status_subscriber = self.create_subscription(String, self.component + '
            _status', self.status_callback, 10)

        self.status_checker_publisher = self.create_publisher(String, "status_check",
             10)
        self.status_checker_publisher.publish(String())

    def event_notification(self, event):
        pass

    def status_callback(self, msg):
        rid = msg.data.split(":")[0]
        print("status update callback from " + self.component + " with RID: " + rid)
        method = "update_status"
        self.server_obj.call_method("2:" + method, str(msg.data))
...
```

Listing 4.16 shows a snippet of code in which the class for communicating with the node on the Entity Raspberry Pi that abstracts the KMP omniMove is presented. As previously stated, this is a sub-class of *PubSub*, and when it is initialized, it passes a string "kmp" to the super-class for the creation of the status subscriber. As seen in Listing 4.14, the class *KMPPubSub*, like *LBRPubSub* and *CameraPubSub*, is made as part of a OPC UA client subscription. It is for this reason that the class includes an `event_notification()` function. This function is called whenever data is sent from the OPC UA server to the client as a "KMPEvent"-event. If it happens, two ROS publishers are created for sending data to the ROS node on the Entity Raspberry Pi: one for sending a movement command and one for telling the node to shut down. The movement command is sent as a *Twist*, and this lets the program control the KMP omniMove omnidirectionally—i.e. in both positive and negative x- and y-direction as well as around the z-axis.

Listing 4.16: Snippet of code from opcua_ros2_pubsub.py showing the middleware implementation for communicating with the KMP omniMove.

```
from std_msgs.msg import String
from geometry_msgs.msg import Twist
...
class KMPPubSub(PubSub):
    def __init__(self, ua_obj):
        super().__init__('kmp', ua_obj)

    def event_notification(self, event):
        action, rid = event.Message.Text.split(",")
        e = action.split("_")

        publisher = self.create_publisher(Twist, "cmd_vel_" + str(rid), 10)
        shutdown_publisher = self.create_publisher(String, self.component + '
            _shutdown_' + str(rid), 10)

        if e[0] == "shutdown":
            msg = String()
            msg.data = "shutdown"
            shutdown_publisher.publish(msg)
        else:
            speed = float(e[0])
            twist = Twist()
            twist.linear.x = float(e[1])*speed
            twist.linear.y = float(e[2])*speed
            twist.linear.z = 0.0
            twist.angular.x = 0.0
            twist.angular.y = 0.0
            twist.angular.z = float(e[3])*speed #or turn
            publisher.publish(twist)
...
```

LBRPubSub, shown in Listing 4.17, is similar in nature to that of *KMPPubSub*. The difference is that the function `event_notification()` in this class is called whenever data is sent on the "LBREvent"-event. When this happens, the data is forwarded as a ROS 2 message in the form of a *String()* to its corresponding ROS node on the Entity Raspberry Pi. This will either tell the LBR iiwa-node to shut down or to move one of its seven joints.

Listing 4.17: Snippet of code from opcua_ros2_pubsub.py showing the middleware implementation for communicating with the LBR iiwa.

```
from std_msgs.msg import String
class LBRPubSub(PubSub):
    def __init__(self, ua_obj):
        super().__init__('lbr', ua_obj)

    def event_notification(self, event):
        msg = String()
        msg.data, rid = event.Message.Text.split(",")

        publisher = self.create_publisher(String, "manipulator_vel_" + str(rid), 10)
        shutdown_publisher = self.create_publisher(String, self.component + '
            _shutdown_' + str(rid), 10)

        if msg.data == "shutdown":
            shutdown_publisher.publish(msg)
        else:
            publisher.publish(msg)
```

CameraPubSub, shown in Listing 4.18, is similar in nature to the two aforementioned classes. The function `event_notification()` in this class is called when data is published on the "CameraEvent"-event, and this forwards the data as a ROS message to the camera node on the Entity Raspberry Pi. The data transmitted from the AAS, through the hybrid node explained in this section and ultimately to the Entity Raspberry Pi, tells the camera to either start or stop. How the messages are sent to *KMPPubSub*, *LBRPubSub* and *CameraPubSub* from the OPC UA server as events is explained in subsubsection 4.3.1.1.

Listing 4.18: Snippet of code from opcua_ros2_pubsub.py middleware implementation for communicating with an entity camera.

```
from std_msgs.msg import String
...
class CameraPubSub(PubSub):
    def __init__(self, ua_obj):
        super().__init__('camera', ua_obj)

    def event_notification(self, event):
        action, rid, *kwargs = event.Message.Text.split(",")
        camera_publisher = self.create_publisher(String, 'handle_camera_' + str(rid),
            10)

        msg = String()
        msg.data = action
        camera_publisher.publish(msg)
```

It is worth mentioning that the ROS node described in this section is run in the same way as for the Entity Raspberry Pi, i.e. launching, even though there is only one active node. This means that there is a corresponding launch file, `middleware.launch.py`, and configuration file, `bringup.yaml`, that are similar to those shown in subsubsection 4.2.2.5. This is done with regards to scalability, as it might be desirable to add more running nodes on the device if or when the system is expanded. It is also desirable to run a ROS 2 program this way because it makes it easier to provide parameters to nodes before running them.

4.3 Control Room & User Devices

A zoomed-in look at the "Control Room" and "User Devices" implementation is shown in Figure 4.3. These two segments are outlined together to further express what is shown in Figure 3.1 from chapter 3, in which an operator can control the entities from both inside and outside the "Control Room". Individual solutions are not implemented for the "Control Room" and "User Devices". Instead, the "User Devices" segment is represented by the fact that it can use the services provided by the "Control Room". The "Control Room" consists of two Raspberry Pis: the AAS Raspberry Pi and the AAS-VIDEO Raspberry Pi. Both of these, with their related software, will be outlined in this section. These Raspberry Pis are not on a 5G connection and are instead connected to the NTNU network through Ethernet, as it is difficult to host services on public cellular networks without having authorization.

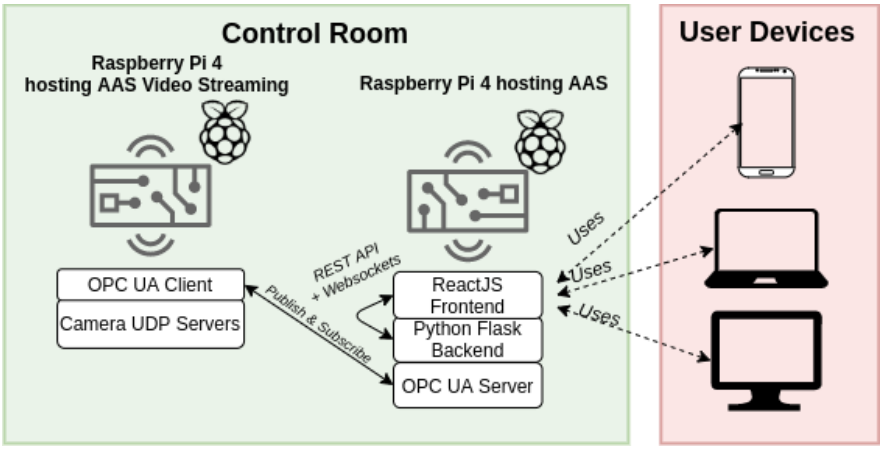


Figure 4.3: Separated look at the "Control Room" and the "User Devices".

4.3.1 Raspberry Pi Hosting AAS

The device used to host the AAS is a Raspberry Pi located in the "Control Room". As seen in Figure 4.1, this Raspberry Pi communicates with the Middleware Raspberry Pi described in subsection 4.2.3. This section is divided into five sub-sections, each of which pertains to one

of the six[1] segments introduced in subsection 2.1.6 that are part of an Industry 4.0-compliant AAS.

4.3.1.1 Internal Interface

The purpose of the Internal Interface is to handle commands from the External Interface with the use of WebSocket communication and send them to the Middleware Raspberry Pi. The Internal Interface is implemented with Flask, and it includes an OPC UA server, a REST API, and a database. A basic understanding of the core structure of Flask is recommended to follow the explanations in this section better (see subsection 2.3.2).

As explained in subsection 2.3.2, Flask is a lightweight framework for creating web services. Because of this, there are several ways to configure a Flask project environment. After Flask is installed on the system, it can be as easy as creating a regular Python file, e.g., `my_flask_project.py`, as a simple Flask application and run it as a normal Python program. For the purpose of the Internal Interface, the Flask application is represented as a directory rather than a file, and this directory contains all the application-related files. Figure 4.4 shows the directory structure of the Internal Interface, which includes the Flask application folder `aas_api`. Inside this folder is the main file, `__init__.py`. This file houses the main parts of the application, such as the Flask logic, the database implementation, the API, and the WebSocket server implementation. Another important aspect of this file is that, as an *__init__* file, it marks the `aas_api` directory as a Python package directory. This makes it so that Python is aware of the sub-modules within the directory, which is essential when running the application.

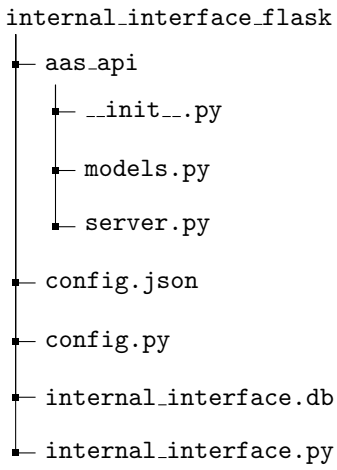


Figure 4.4: Structure of the Internal Interface directory

[1] "External interface" and "Functionality" are out of convenience explained together.

Listing 4.19 shows a snippet of code from `__init__.py`. This part is where the essential flask functions and supported third-party components are imported and implemented. The snippet includes comments with explanations for each implementation. The way objects, or sub-modules, are created in order to correctly interact with the Flask application is by passing them the application, which in this case is `aas_api`. A sub-module worthy of highlighting is `migrate`, as it is not used directly anywhere in any of the files related to the application. The reason for creating this sub-module is that it makes database operations available for the Flask command-line interface (CLI). This is the case for the other sub-modules created as well, such as `db`, but these are additionally used in the application files. Since the sub-modules are exposed to the Flask CLI, changes in the database format (tables and fields) can be realized with the following commands:

```
$ cd internal_interface_flask
$ flask db migrate -m "Added new table: User"
$ flask db upgrade
```

The astute reader will notice two configuration files named `config` imported in the snippet. The first one is for the configuration details for the database, which is described later in this section. The second one, shown in Listing 4.20, is a JSON file with network details for the Flask application, the OPC UA server that runs on it, and the domain used by the AAS Video Stream service explained in subsubsection 4.3.2.2 to host the various video streams from entities.

Listing 4.19: Snippet of code from __init__.py showing how the Flask application is configured.

```
import json
from config import Config  #Database configuration file "config.py"

from flask import Flask, Response, render_template, request, jsonify, abort
from flask_bcrypt import Bcrypt  #Used for encrypting passwords
from flask_sqlalchemy import SQLAlchemy  #Database toolkit
from flask_migrate import Migrate  #Used for handling SQLAlchemy database migration
from flask_socketio import SocketIO, emit  #Used for WebSocket communication
from flask_cors import CORS  #Needed to let browsers access the API

aas_api = Flask(__name__)  #Initiate the Flask application
aas_api.config.from_object(Config)  #Load database configuration
CORS(aas_api) #Apply CORS headers
bcrypt = Bcrypt(aas_api)  #Create encryption object for hashing user passwords
db = SQLAlchemy(aas_api)  #Create database object for querying database
migrate = Migrate(aas_api, db)  #Create object for database migration
socketio = SocketIO(aas_api, cors_allowed_origins="*")  #Create WebSocket
    communication object

from aas_api import models, server #Import application-related files
```

```
with open("config.json", "r") as f:
    ip_dict = json.load(f) #Load network configuration

opcua_instance = server.OpcuaServer(ip_dict["OPCUA_URL"], socketio, db) #Start OPC UA
    server
...
```

Listing 4.20: Snippet of code from config.json showing the network configuration.

```
{
  "OPCUA_URL": "andrcar-master.ivt.ntnu.no:4841",
  "SERVER_URL": "andrcar-master.ivt.ntnu.no:8000",
  "VIDEO_STREAM_IP": "andrcar-master-stream.ivt.ntnu.no"
}
```

The Internal Interface is run with the file `internal_interface.py` shown in Listing 4.21. The Flask service is deployed using the concurrent networking library *eventlet*. This replaces the native Flask subsystem, Werkzeug, and it makes it possible for SocketIO to use WebSocket communication. Since the `aas_api` folder is declared as a Python package directory, its sub-modules can be imported and used directly in the run-file.

Listing 4.21: Snippet of code from internal$_{i}nterface.pyshowinghowtheFlaskapplicationisrun.$

```
from eventlet import wsgi
from aas_api import aas_api, ip_dict

ip, port = ip_dict["SERVER_URL"].split(":")

wsgi.server(eventlet.listen((ip, int(port))), aas_api, log_output=True, debug=True)
```

Database

Since the AAS created in relation to this thesis is a "Proof of Concept", it is desirable to use components that are easy to use, compact, and efficient. Because of this, *SQLite* [68] is a fitting choice of database. SQLite is an embedded database, which means that it runs as part of the application it serves—not independently as a standalone process. This is favorable in smaller systems and testing environments because it requires no network configuration or administration. Listing 4.22 shows the only configuration needed to get SQLite to work with Flask and SLQAlchemy. All the data is stored with the SQLite database format in the file `internal_interface.db`, and the Flask application needs only to know the location of this file in order to interact with the database. The interaction between the Flask application and the database is explained later in this section when describing the API and the OPC UA Server.

Listing 4.22: Snippet of code from config.py showing the database configuration.

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'internal_interface.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Models

In order for the AAS database to contain any information, it needs to have tables and fields. This is achieved with the SQLAlchemy API's Model class. A model in SQLAlchemy is a single, definitive source of information about some data. These are implemented in Python as classes that inherit the properties of `db.Model`, which is stored in the SQLAlchemy instance that is created in Listing 4.19. Listing 4.23 shows the model *Robot*, which represents a robot entity that is recognized by the AAS. Each entity needs a unique ID and a name in order to distinguish them from one another, and it is necessary in order for the External Interface to display entities correctly. Each entity also needs a set of components, a unique UDP URL, and a stream port.

Listing 4.23: Snippet of code from models.py showing the Robot model used for storing information about various robot entities.

```
{
  class Robot(db.Model):
    id = db.Column(db.String(100), primary_key=True, unique=True)
    name = db.Column(db.String(100))
    components = db.Column(db.String(10000))
    udp_url = db.Column(db.String(100))
    stream_port = db.Column(db.Integer)

    def __repr__(self):
        return self.id + " " + self.name + " components:" + str(self.components)
}
```

The `components` column is used to keep track of every component's status (online/offline) on an entity, and each value inserted into this field needs to be a string. This is a technique used to support various types of entities. With a string, the components for the KMR iiwa can be represented as a serialized JSON object, such as *"kmp:true,lbr:false"*. This example means that the KMP omniMove is online, and the LBR iiwa is offline on one of the KMR iiwa entities. This helps achieve a dynamic storing of information since all the columns in the *Robot* model are ambiguous. If the model were to have a `kmp` column and an `lbr` column instead, it would only be able to support the KMR iiwa, and for each new entity

type introduced to the system, there would need to be created a new model. With the string technique, the *Robot* model supports virtually any type of entity, and new components can even be added to already established entities without affecting the database format. The `udp_url` column is used to store the IP address to which the video captured on the robot's Entity Raspberry Pi is streamed. Since this URL can be the same for multiple entities, they need different ports. This is handled by the `stream_port` column, in which every value is unique.

There are two additional models in the Internal Interface that are created similarly to *Robot* and therefore not shown with code listings. The first one in *User*, which is used for storing information about the operators of the AAS, i.e, their ID, name, password, and *role*. The role can either be *viewer*, *operator* or *admin*, each of which has a different level of access throughout the AAS. The purpose of having a role-hierarchy is explained further in subsubsection 4.3.1.5. The second model is *StreamPort*, which is used to keep track of which ports are available for the video streaming service. It makes it so that any new entity registered in the AAS is assigned a port that is not already in use, ensuring that every captured video is streamed to a unique address.

API

Flask, like any other web framework, supports the concept of serving content on a given URL. In Flask, this is done through the use of *routes* and *views*. A route refers to a URL pattern designed for the application, while a view refers to the content served at that URL. Together, they form what is known as an API *endpoint*. Listing 4.24 shows the endpoint for getting all the robot entities that are registered in the AAS. Flask routes, such as `@aas_api.route('/api/robots')`, are defined by Python decorators that Flask provides to easily assign URL patterns to functions inside the application. The view function to which this route belongs is `get_all_robots()`. This way, any time a client visits the application domain at the given route, the view function is executed. When executed, the function `get_all_robots()` queries the database for all registered robot entities and returns them to the client as an HTTP Response containing a list of JSON objects.

Listing 4.24: Snippet of code from __init__.py showing the endpoint for getting all entities in the AAS.

```
@aas_api.route('/api/robots')
def get_all_robots():
    robots = models.Robot.query.all()
    result = []
    for robot in robots:
        result.append({
            'rid': robot.id,
            'name': robot.name,
            'components': json.loads(robot.components),
            'udp_url': robot.udp_url,
```

```
            'stream_port': robot.stream_port
        })

    return {
        'robots': result
    }
```

Often it is useful to get data for a specific entry in a database. This can be handled by including a parameter in the Flask route which is unique for the desired item. Listing 4.25 shows how the Internal Interface API can get a specific robot entity based on its ID. The parameter `<rid>` in the route `@aas_api.route('/api/robots/<rid>')` means "robot id", and it is provided by a client that wishes to get information about the entity in the *Robot* table with a matching value in its `id` column. When the route is used, the view function `get_specific_robot(rid)` is called, which queries the database. If a robot entity is found with a matching ID, the result is returned to the client with an HTTP Request containing a single JSON object. If a match is not found, an HTTP 404 (Not Found) error is returned to the client.

Listing 4.25: Snippet of code from __init__.py showing the endpoint for getting a specific entity in the AAS.

```
@aas_api.route('/api/robots/<rid>')
def get_specific_robot(rid):
    robot = models.Robot.query.filter_by(id=rid).first_or_404()
    return {
        'rid': robot.id,
        'name': robot.name,
        'components': json.loads(robot.components),
        'udp_url': robot.udp_url,
        'stream_port': robot.stream_port
    }
```

The AAS Video Stream service is, as described in subsection 4.3.2, hosted on a separate Raspberry Pi, but the metadata about a video stream is provided by the Internal Interface API. This is beneficial because it lets the AAS Video Stream service be completely detached from all clients—i.e. it streams video to an IP address, but it does not know or care who visits this address—while it is the AAS itself that handles all communication with clients. Having as few components in the system as possible depend on each other helps reduce coupling, which again makes the system more scalable. Listing 4.26 shows the API endpoint that a client can use to get the video stream for the entity they wish to monitor. The client provides an `<rid>`, which the view function `get_robot_video_stream(rid)` uses to get the entity in the database with a matching ID. It then finds the IP address for the video stream in the network configuration file (see Listing 4.20) and combines this with the unique stream port for the specific entity. This URL is then returned to the client as an HTTP Request, and

the client can use the URL to display the video stream.

Listing 4.26: Snippet of code from __init__.py showing the endpoint for getting the video stream IP address for a specific entity.

```
@aas_api.route('/api/robots/<rid>/video')
def get_robot_video_stream(rid):
    robot = models.Robot.query.filter_by(id=rid).first_or_404()
    stream_ip = str(ip_dict["VIDEO_STREAM_IP"])

    return {
        'url' : "http://" + stream_ip + ":" + str(robot.stream_port)
    }
```

The API also needs a way to handle requests from operators trying to log in to the AAS. This is done with the endpoint shown in Listing 4.27. The endpoint receives an HTTP POST request with a JSON object. This object contains a username and password. The username is used to query the database for all *User* entries with that particular username, and the password is then checked against the password for every entry in order to find a match. The passwords in the database are encrypted, so they have to be decrypted using bcrypt when matching. If a matching user is found, credentials are returned as an HTTP response. If no match is found, the login is aborted. See subsection 4.4.4 for more information about the login sequence.

Listing 4.27: Snippet of code from __init__.py showing the endpoint for handling login.

```
@aas_api.route('/api/login')
def attempt_login():
    try:
        correct_user = None
        content = request.get_json()
        users = models.User.query.filter_by(username=content.get('username')).all()
        for u in users:
            if bcrypt.check_password_hash(u.password, content.get('password')):
                correct_user = u
                break
        return {
            'username': correct_user.username,
            'admin': correct_user.admin,
            'operator': correct_user.operator
        }
    except:
        abort(404)
```

OPC UA Server

The OPC UA server in the Internal Interface enables the AAS to communicate with the

Middleware Raspberry Pi and the AAS-VIDEO Raspberry Pi. Listing 4.28 shows a part of the class *OpcuaServer* used for managing server and client communication. This server is hosted as part of the Flask application, and by calling `OpcuaServer(...)`, as seen in Listing 4.19, the server is started, and communication with clients can begin. There are three parameters that are needed when starting the server:

- `opcua_url`. This is the IP address and the port on which the OPC UA server is to be hosted. Clients need this URL to connect to the server.
- `socketio`. This is the Flask-SocketIO created in the main file. It is used to enable WebSocket communication between a client and the Flask application.
- `db`. This is the database object created in the main file. It is used to interact with the SQLite database embedded in the application.

The last two parameters are not used when initializing the OPC UA server, but they are used in the functions related to the class *OpcuaServer* which are described below. When the server is initialized, it creates a server object `aas_obj`, which is used to expose the functions of the server to its clients. There is then added a *server method* to the object, which is what the clients use to send data to the server. This `aas_obj` object is then used to create *event generators* for each component with which the server communicates through the Middleware Raspberry Pi client, namely the KMP omniMove, the LBR iiwa, and the camera. These event generators are used to send data to the clients. The AAS-VIDEO Raspberry Pi described in subsubsection 4.3.2.1 also subscribes to the "CameraEvent" topic, making it so that when this event is triggered on the OPC UA server, both the Middleware Raspberry Pi and the AAS-VIDEO Raspberry Pi receives the data.

Listing 4.28: Snippet of code from server.py showing how the OPC UA server is initialized.

```
class OpcuaServer:
    def __init__(self, opcua_url, socketio, db):
        self.socketio = socketio
        self.db = db

        logging.basicConfig(level=logging.WARN)
        logger = logging.getLogger("opcua.server.internal_subscription")

        server = Server()
        server.set_endpoint("opc.tcp://" + opcua_url + "/freeopcua/server/")

        objects = server.get_objects_node()
        aas_obj = objects.add_object(..., "AasObject")

        # add server methods
        aas_obj.add_method(..., "update_status", self.update_status, [ua.VariantType.
```

```
            String], [ua.VariantType.Int64])

        lbrEvent = server.create_custom_event_type(..., 'LBREvent')
        kmpEvent = server.create_custom_event_type(..., 'KMPEvent')
        cameraEvent = server.create_custom_event_type(..., 'CameraEvent')

        self.lbrEvgen = server.get_event_generator(lbrEvent, aas_obj)
        self.kmpEvgen = server.get_event_generator(kmpEvent, aas_obj)
        self.cameraEvgen = server.get_event_generator(cameraEvent, aas_obj)

        server.start()
...
```

As mentioned, the class *OpcuaServer* includes a set of functions. These are used in combination with both the API and the WebSocket implementation in order to communicate with the External Interface and the OPC UA clients. The OPC UA clients are the Middleware Raspberry Pi and the AAS-VIDEO Raspberry Pi. Listing 4.29 shows a snippet of code from the *OpcuaServer* class with the function `update_status()`, which is the aforementioned server method. This is indicated by the Python decorator `@uamethod`. The ROS node uses this function on the Middleware Raspberry Pi to provide the AAS with status updates for the various entity components. When a status update is sent for a component, the entity's ID to which the component belongs is checked in the database. If an entry with a matching ID is not found, a new *Robot* instance is created with the information that is provided by the OPC UA client and added to the database. If a match is found, the value in the entry's `components` field, which is stored as a string, is de-serialized to a JSON object, and the component status is changed to match that of the information provided by the OPC UA client. The either new or updated information about the entity is then sent to the External Interface using WebSocket.

Listing 4.29: Snippet of code from server.py showing the server method used for updating entity status in the database and sending status to an External Interface.

```
@uamethod
def update_status(self, parent, msg):
    rid, robot, component, component_status, *kwargs = msg.split(':')

    entry = models.Robot.query.filter_by(id=rid).first()
    if not entry:
        port = models.StreamPort.query.filter_by(id="1").first()

        dump = {component: bool(int(component_status))}

        new_robot = models.Robot(
            id=rid,
            name=robot,
```

```
            components=json.dumps(dump),
            stream_port = port.available_port
        )

        if component == "camera":
            new_robot.udp_url = kwargs[0] + ":" + kwargs[1]

        port.available_port += 1
        self.db.session.add(new_robot)
    else:
        dump = json.loads(entry.components)
        dump[component] = bool(int(component_status))
        entry.components = json.dumps(dump)
        if component == "camera":
            entry.udp_url = kwargs[0] + ":" + kwargs[1]

    self.db.session.commit()

    jdata = json.dumps({
        'rid': rid,
        'robot': robot,
        'component': component,
        'component_status': bool(int(component_status))
    })
    self.socketio.emit('status', jdata, broadcast=True)
```

The two remaining functions in the *OpcuaServer* class are used for sending data to the Middleware Raspberry Pi and the AAS-VIDEO Raspberry Pi as OPC UA events. The two functions are `send_to_entity()`, shown in Listing 4.30, and `send_to_camera()`, shown in Listing 4.31. They both use the event generators created in Listing 4.28. When sending data to an entity, a component command is first received by the Flask application from the External Interface—a process that is explained in subsubsection 4.3.1.2. This command is then passed to the `send_to_entity()` function, which triggers an OPC UA event. The information about which component the command is targeted at is given in the `cmd` string, and this decides whether an "LBREvent" or a "KMPEvent" should be triggered. This implementation only supports the use of the KMR iiwa as an entity. The process for handling a camera command from the External Interface is similar, but since the camera is meant to be independent of entity type, there is no need to check `camera_cmd` before sending it.

Listing 4.30: Snippet of code from server.py showing how a command to either the KMP omniMove or the LBR iiwa is sent to the Middleware Raspberry Pi from the OPC UA server.

```
def send_to_entity(self, cmd):
    command_splt = cmd.split(":")
    if command_splt[0] == "lbr":
        self.lbrEvgen.event.Message = ua.LocalizedText(command_splt[1])
        self.lbrEvgen.trigger()
        print("LBREvent sent!")
    elif command_splt[0] == "kmp":
        self.kmpEvgen.event.Message = ua.LocalizedText(command_splt[1])
        self.kmpEvgen.trigger()
        print("KMPEvent sent!")
```

Listing 4.31: Snippet of code from server.py showing how a camera command is sent to the Middleware Raspberry Pi and the AAS-VIDEO Raspberry Pi from the OPC UA server.

```
def send_to_camera(self, camera_cmd):
    self.cameraEvgen.event.Message = ua.LocalizedText(camera_cmd)
    self.cameraEvgen.trigger()
    print("CameraEvent sent!")
```

WebSocket Communication

The communication between the Flask application and its clients needs to be as efficient as possible for the AAS to attempt to comply with the real-time requirements of Industry 4.0. This is achieved by replacing the commonly used HTTP Polling protocol with WebSockets. Both of these, as well as their differences, are explained in subsection 2.1.9. In Flask, WebSocket communication can be implemented using the SocketIO library, as shown in Listing 4.19 at the beginning of this section. How a client connects to the WebSocket server on the Flask application is explained in subsubsection 4.3.1.2. Listing 4.32 shows how SocketIO is used to create WebSocket endpoints. Like for the API endpoints described earlier in this section, the functions in the WebSocket implementation are defined with Python decorators. The decorator `@socketio.on('command')` makes it so that when data is *emitted* from the External Interface with the *event name* "command", the function that corresponds with the decorator is called. This function, `receive_command()`, receives a JSON object with a command and an entity ID and sends this to the Middleware Raspberry Pi using the OPC UA server function shown in Listing 4.30. Similarly, when data is emitted from the External Interface with the event name "camera_event", the `receive_camera_event()` function is called. This function first has to query the database in order to get additional information about the entity, i.e., the UDP URL and the stream port for the video stream for that specific entity. This information is not relevant for the Middleware Raspberry Pi, but it is necessary for the AAS-VIDEO Raspberry Pi. If for some reason, no matching entity is found with the query, an HTTP 404 error is sent back to the External Interface. If a match

is found, the data is sent to the Middleware Raspberry Pi and the AAS-VIDEO Raspberry Pi using the OPC UA server function shown in Listing 4.31. It is essential to include an ID when forwarding the data to the OPC UA clients so that they know which entities and video streams the messages are meant for.

Listing 4.32: Snippet of code from __init__.py showing the part of the WebSocket implementation that handles the handshake.

```
...
@socketio.on('command')
def receive_command(cmd):
    opcua_instance.send_to_entity(cmd['command'] + "," + cmd['rid'])

@socketio.on('camera_event')
def receive_camera_event(cmd):
    robot = models.Robot.query.filter_by(id=cmd['rid']).first_or_404()
    middleware.send_to_camera(cmd['camera_event'] + "," + cmd['rid'] + "," + robot.
        udp_url + "," + str(robot.stream_port))
...
```

4.3.1.2 External Interface and Functionality

The External Interface and Functionality are two separate segments of an industrial AAS, but since the functionality of the AAS described in this thesis is closely related to the components on the External interface, it is easier to understand both parts if they are explained in relation to each other. As opposed to the description of the Internal Interface, this section will not contain many code listings that explain what goes on behind the scenes. This reduction is an attempt to highlight the functionality that operators of the AAS would have at their disposal in an industrial environment. Additionally, the amount of code, coupled with the fact that much of the application data is shared between several components over multiple different files, would make it difficult for the reader to understand what each component does and how they work in relation to other components.

The External Interface is what the end-user of the AAS sees and uses to control the industrial robot entities available in the system. The implementation of the External Interface includes a login page and a homepage on which the available entities are displayed. The user can then click on the entity they wish to inspect, and they will then see a popup page specifically for that entity. This functionality is described and explained in more detail throughout this section. The External Interface is developed using the web framework React. This web framework is a good fit for this project because of its innate modularity with regards to reusable and loosely coupled components. It is easy to expand with more entities if necessary, as there is no need to write the same code multiple times for similar functionality. The framework is also widely supported, which means that third-party software such as

user interface (UI) frameworks can be quickly and easily implemented. The design for the External Interface is developed using the UI framework *Material-UI*, which allows for quick creation of a simple and pleasant user interface.

Login page

After starting the External Interface, the operator will be met with the login page shown in Figure 4.5. Here, the operator can enter their assigned username and password, and if they have a registered account, they will be redirected to the homepage. There is no way for operators to create an account from the login page—a feature that is deliberately omitted because it is assumed that the AAS account would be the same as the one the operator is already registered with at the institution for which they work.

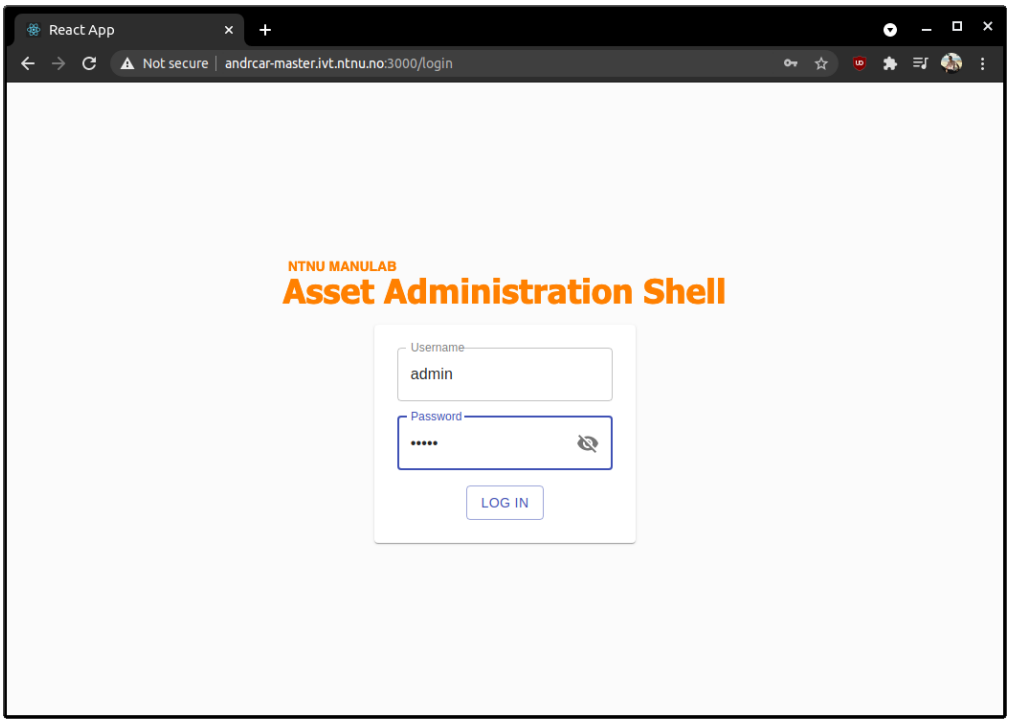


Figure 4.5: The login page.

After an operator has entered their username and password, they can click the button labeled "LOG IN". This will trigger the function shown in Listing 4.33. This function sends an HTTP POST request to the API login endpoint on the Internal Interface. If the username and password match a registered account, the operator's credentials are stored in the browser's cache before redirection. If no match is found, the operator will not be logged in or redirected. A complete overview of the flow of the data occurring during login is shown in subsection 4.4.4.

Listing 4.33: Snippet of code from login.js showing what happens when the "LOG IN" button is clicked.

```
const handleLoginButtonClicked = () => {
    axios.post(configs.API_URL + "api/login", {
        username: values.username,
        password: values.password
    }).then(resp => {
        let success_msg = "Successfully logged in as " + values.username
        dispatch({type: 'click', message: success_msg})
        localStorage.setItem('username', resp.data.username)
        localStorage.setItem('admin', resp.data.admin)
        localStorage.setItem('operator', resp.data.operator)
        history.push('/dashboard')
    }).catch(error => {
        let fail_msg = "Failed to log in. Wrong password or no user named " + values.
            username
        dispatch({type: 'click', message: fail_msg})
        console.log(error.response.data)
    })
}
```

The credentials that are returned and stored include the operator's role, which, as stated when describing the *User* database model, can either be that of a *viewer*, an *operator*[2] or an *admin*. This is represented with the two boolean values `operator` and `admin`:

- *Viewer*: `operator = false` and `admin = false`
- *Operator*: `operator = true` and `admin = false`
- *Admin*: `operator = true` and `admin = true`

Homepage

After a successful login, the operator will be greeted with the homepage shown in Figure 4.6. This page displays all the entities that are registered in the Internal Interface database. On each entity *card*, the operator will be able to see the name, ID, and status of the various entities.

[2]The *operator* role should not be confused with an AAS operator. An operator in the context of the AAS is anyone that uses the External Interface to interact with robot entities.

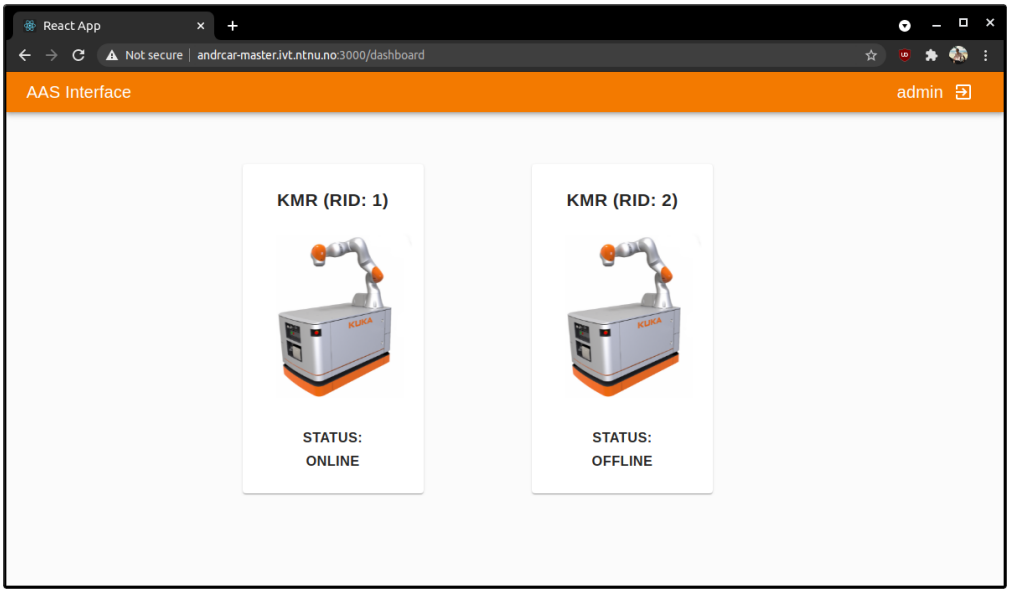


Figure 4.6: The Homepage showing two KMR iiwa entities, one is online while the other is offline.

The data for the entities shown here is retrieved from the Internal Interface with the function shown in Listing 4.34. As explained in the description for the Internal Interface API, the endpoint returns a list of JSON objects with data about each entity. This data is stored in the External Interface's *state* and used to create the card for each entity.

Listing 4.34: Snippet of code from App.js showing how the External Interface retrieves data for all entities in the system.

```
const fetchAndSetState = () => {
    axios.get(configs.API_URL + "api/robots").then(resp => {
      dispatch({type: 'setState', robots: resp.data.robots})
    })
  }
```

The cards are clickable, and when a card is clicked, the operator will be presented with a popup page containing the functionality for that specific entity.

Entity Popup Page

The entity popup page is what an operator can use to interact with an entity. The page, shown in Figure 4.7, is constructed as a grid that is populated by the various components related to the entity. The idea is that the page's structure should be similar for all entities but that the components in the grid change based on what functionalities each entity provides. Because of this, the page is created with flexibility in mind and with as little hard-coding with respect to specific entities as possible. For a KMR iiwa entity, this page includes the following components:

- **KMP Controller**. This is used to control the KMP omniMove component and is specific for the KMR iiwa.
- **LBR Controler**: This is used to control the LBR iiwa component and is specific for the KMR iiwa.
- **KMR General Commands**: This is used to send custom commands to the KMR iiwa. This requires that the operator knows what a command looks like "under the hood", and should only be used for testing purposes.
- **Video Stream Component**: This is used to start and display a video stream from the camera on the KMR iiwa. This component can be used for any type of entity.

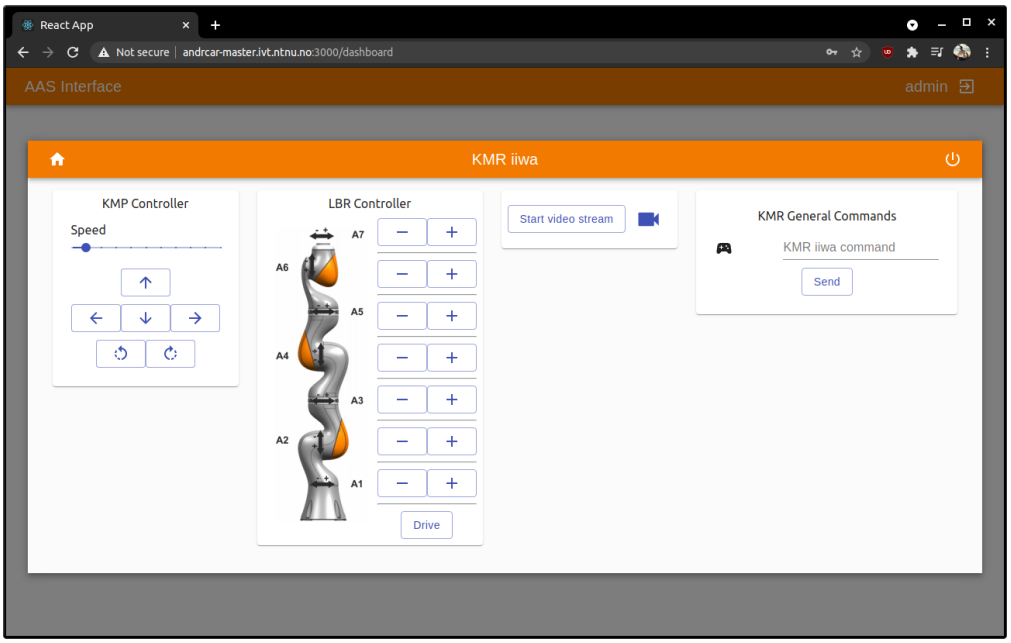


Figure 4.7: The entity popup page with KMR iiwa functionality.

When an operator is logged in as either an *admin* or an *operator*, all the components are enabled and ready for use if the entity is online. As a *viewer*, the command components will be disabled to prevent any interaction with the entity, as that role is not authorized to control anything in the "Industrial Lab". A *viewer* can still start and view a video stream, as this is not regarded as a potentially hazardous operation. How components are disabled depending on roles is the same as when the components on an entity are offline, which can be seen in Figure 4.8.

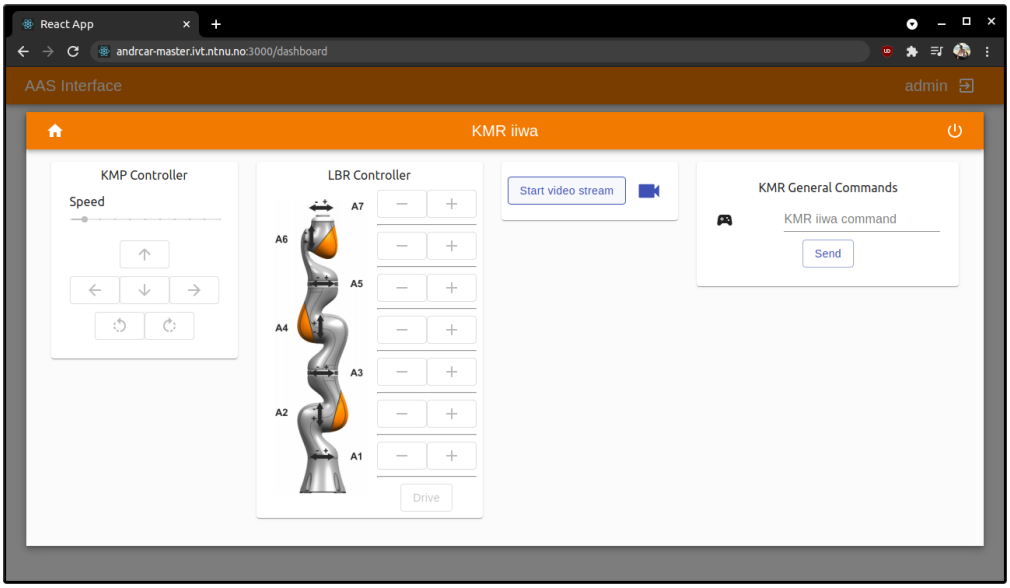


Figure 4.8: The entity popup page with disabled components.

The rest of this section will use the entity popup page for the KMR iiwa entity as an example for demonstrating the functionality. When the components are enabled, an operator with *operator* privileges—i.e. logged in as either *operator* or *admin*—can control the KMP omniMove by either pressing the buttons in the controller or by using the arrow keys on a keyboard connected to the computer on which the External Interface is used. Figure 4.9 shows how to move the entity in the positive y-direction (forward). There is a slider above the arrow buttons that can be used to change the speed at which the KMP omniMove moves. When a movement key is pressed, a function is called that emits the appropriate command for the specific entity to the Internal Interface using WebSocket communication, as seen in Listing 4.35.

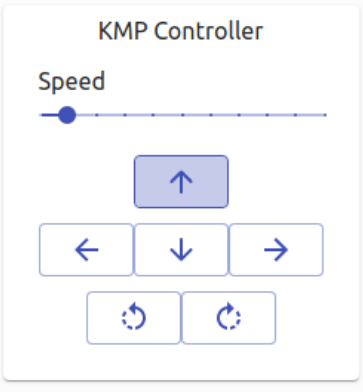


Figure 4.9: The forward button is activated by either clicking on it on the page or using the up-arrow key on a connected keyboard. The "Speed" slider is set to 0.1 which is 10 % of the maximum speed.

Listing 4.35: Snippet of code from kmpController.js showing the function for emitting KMP omniMove movement commands to the Internal Interface.

```
const moveKMP = (direction) => {
    var vector = null

    if(direction === "up"){
        vector = " 1 0 0"
    }
    if(direction === "down"){
        vector = " -1 0 0"
    }
    ... // left, right, clockwise, counter-clockwise are not shown
    if(direction === "stop"){
        vector = " 0 0 0"
    }
    props.ws.emit('command', { "command" : "kmp:" + speed + vector, 'rid': props.rid
        })
    // To move KMP forward:
    // "command": "kmp:0.1 1 0 0", "rid": "1"
}
```

The "LBR Controller" works similarly, but there are no keyboard shortcuts for this component, so an operator has to click on the "+" or "-" buttons for the joint they want to move in order to send a command. The operator is free to move any of the seven joints in both directions, as well as to set the LBR iiwa in the "Drive" position. Listing 4.36 shows the function that emits a command to the Internal Interface when a movement key is pressed on the "LBR Controller". This command contains which joint should be moved, in which direction, and to which entity.

Listing 4.36: Snippet of code from lbrController.js showing the function for emitting LBR iiwa movement commands to the Internal Interface.

```
const moveLBR = (joint, direction) => {
    props.ws.emit('command', { "command" : "lbr:" + joint + " " + direction, 'rid':
        props.rid})
    // To move bottom joint on LBR in positive direction:
    // "command": "lbr:A1 1", "rid": "1"
}
```

An operator of any role can start a video stream on a specific entity and view it. This is done by clicking the button labeled "Start video stream". When this happens, the function shown in Listing 4.37 is called. This emits a message to the Internal Interface saying whether to start or stop the camera on a specific entity. A sequence diagram of what happens throughout the pipeline when a camera event is emitted is shown in subsection 4.4.3. This operation requires some loading time, and this is presented to the operator as shown in Figure 4.10.

Listing 4.37: Snippet of code from robot.js showing the function for emitting camera start/stop commands to the Internal Interface.

```
const sendCameraEvent = () => {
    props.ws.emit("camera_event", {'camera_event': robotState.cameraButton, 'rid':
        props.robot.rid})
    // To start camera: "camera_event": "start", "rid": "1"
}
```

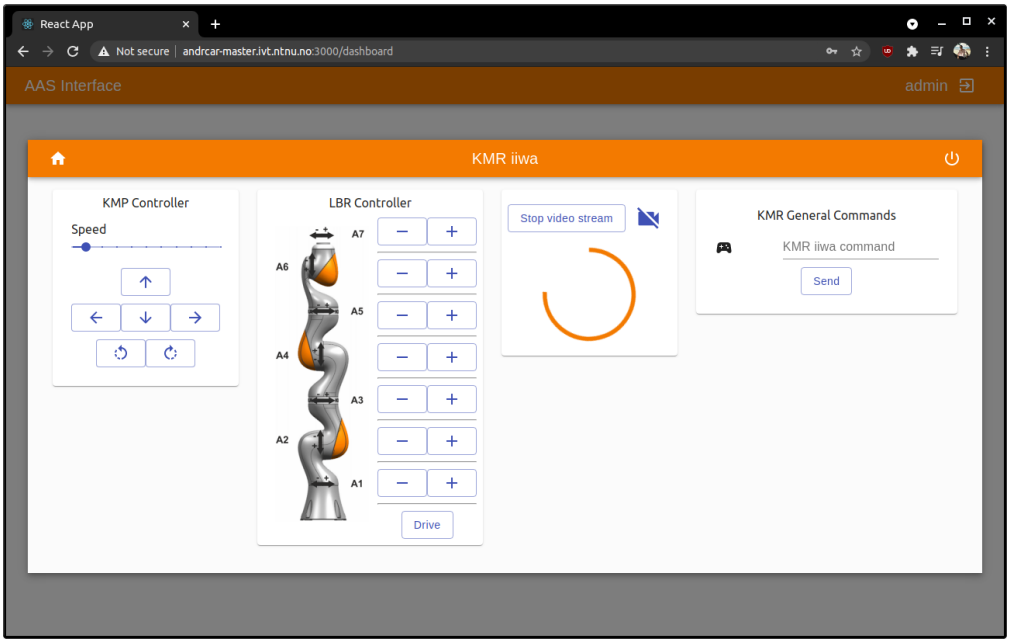


Figure 4.10: While the video service is starting, the entity popup page displays a loading animation.

When the loading is done, the video from the entity is being streamed to a known IP address. The stream can then be displayed in the External Interface using an HTML component, as shown in Listing 4.38. While the video stream is loading, this component will try to fetch it and consequently fail. When this happens, the `setSrc()` function is called, which waits 1.5 seconds before it initiates another fetch of the stream. This happens continuously until the stream is done loading, which is when `handleLoad()` is called. This function sets the `loading` boolean to `false`, making it so that the loading animation stops and the video stream is displayed.

Listing 4.38: Snippet of code from robot.js showing the function for emitting camera start/stop commands to the Internal Interface.

```
<div style={{display: loading ? "none" : "block"}}>
    <img src={props.state.streamURL || localStorage.getItem('camera_stream_url_' +
        props.rid)}  onError={(e)=>{setSrc(e)}} onLoad={(e)=>{handleLoad(e)}}/>
</div>
```

When a video stream is started, it will be displayed in the same grid as the components, as shown in Figure 4.11. After the button for starting the video stream has been clicked, the button label is changed to "Stop video stream". Clicking the button now will make an alert pop up on the screen, as shown in Figure 4.12. This warns the operator that stopping the stream for an entity will also stop the stream for any other operator controlling or viewing that specific entity. Since this is a decision that has a system-wide effect with potentially hazardous consequences, it is reserved only for operators with the *admin* role.

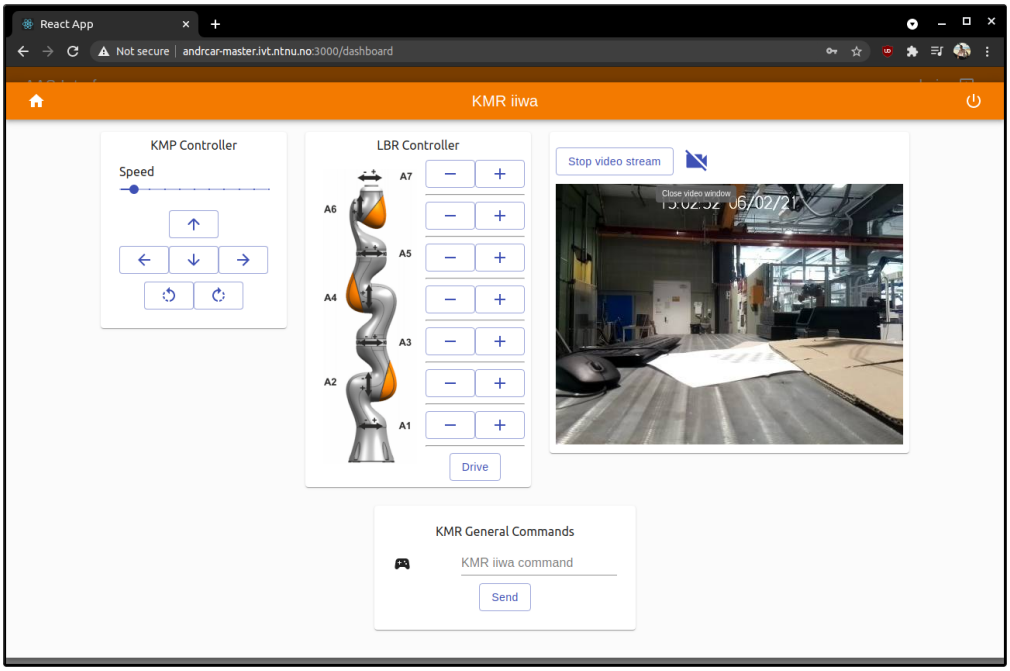


Figure 4.11: When the video service is done loading, a video stream is displayed for the operator.

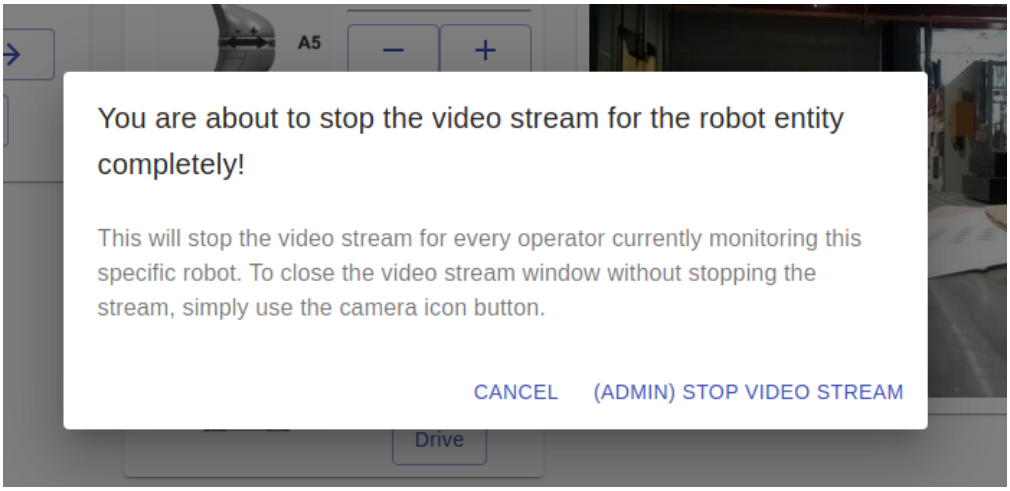


Figure 4.12: Only an operator with administrator-privileges can stop an entity video stream.

An operator might not always want the video stream to be displayed, and if the operator is not an *admin*, they will not be able to stop it. It is therefore implemented an icon button to the right of the start/stop button, which closes the video stream window without stopping the stream. When a video stream is still up, but the window is closed, an operator will see what is shown in Figure 4.13.

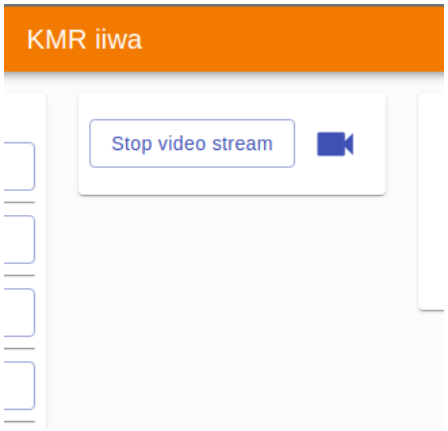


Figure 4.13: An operator can close the video stream window while a video stream is running.

An operator with the *admin* role can shut down the entire entity. This is done by clicking the button with the power symbol icon in the top right-hand corner of the entity popup page. If this is done, a prompt will open in which the operator must confirm their action. Figure 4.14 shows what an operator sees when they try to shut down an entity. This is another decision that has a system-wide effect with potentially hazardous consequences, and it is therefore reserved only for operators with the *admin* role.

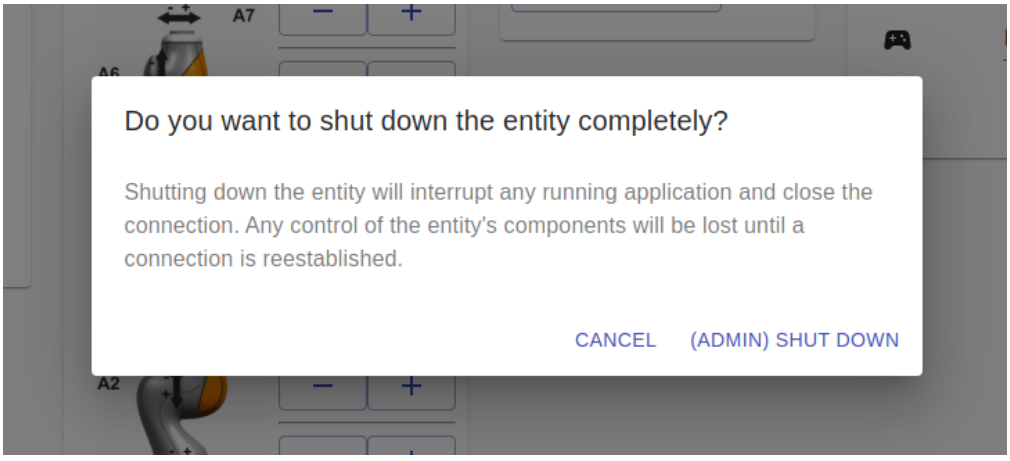


Figure 4.14: Only an operator with administrator-privileges can shut down an entity.

4.3.1.3 Administration

In an AAS, the Administration segment concerns the data flow between the other segments. The AAS presented in this thesis includes many different components that are run on separate platforms, and it can therefore be overwhelming to keep track of exactly where and how the data travels. Shown in Figure 4.15 is an attempt to boil the AAS data flow down to its most essential parts. This is supposed to give a more clear overview of which parts of the AAS handle what data. An entity is included in the figure in order to show where the data from the AAS ends up and where the status updates come from.

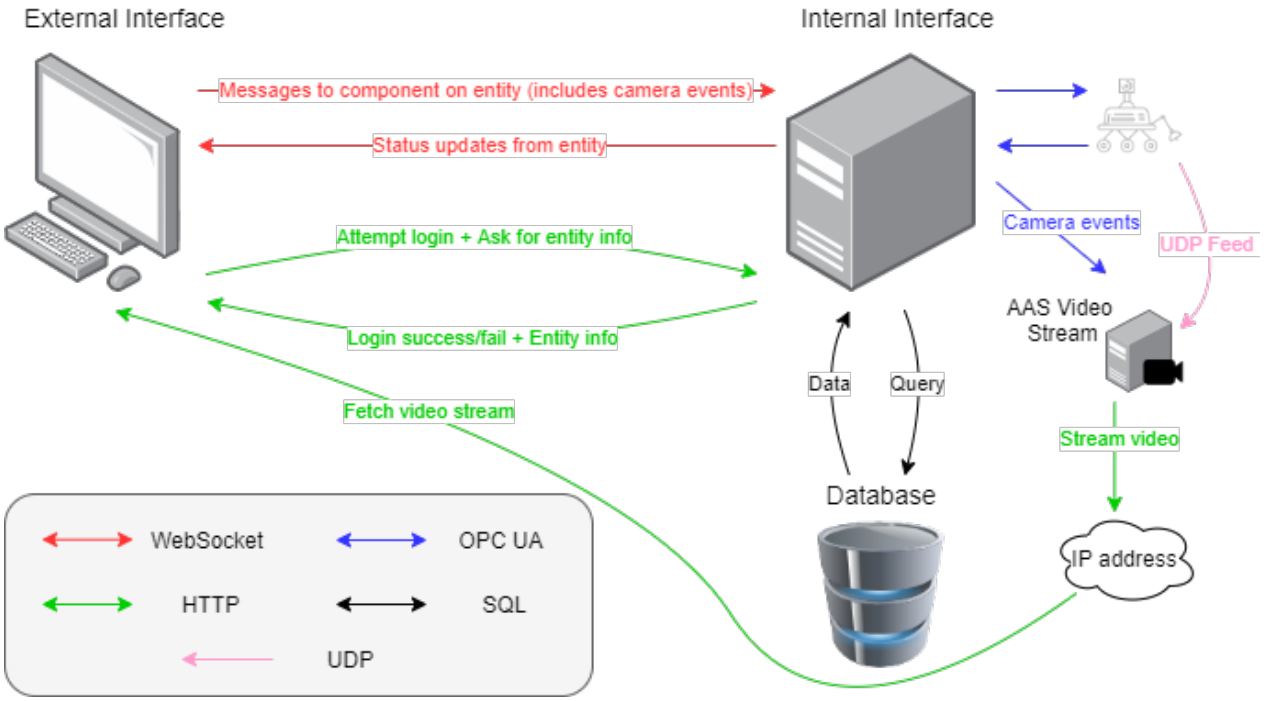


Figure 4.15: A simplified look at the data flow in the AAS.

4.3.1.4 Data Management

The AAS implemented and described using an Internal and an External Interface portrays a real-time system with many variables that need to be handled and stored. As explained earlier in this section, the Internal Interface manages data with an SQLite database and the Flask-SQLAlchemy database toolkit. This data is persistent until a change occurs on an entity. This change can either be that a status update is sent or that a new component is introduced. This way, the database always reflects the environment in the "Industrial Lab", i.e. every change in the "Industrial Lab" also happens in the database. Using the persistent data in the database, the External Interface can always display accurate information about entities to operators of the AAS. The External Interface handles storing of data in two ways: *state* and

localStorage. In React, the state is where the property values of the various React components are stored. A component in the External Interface would be the Homepage or the Entity Popup Page. These components also have a *lifecycle*. The lifecycle of a component begins when it is *mounted*, which means outputting the visual representation of the component in the user interface. There is then an *updating* period, which is when the state of the component can be used in the user interface. The lifecycle of the component ends when it is *unmounted*, i.e., when it is no longer part of the user interface representation. When a component is unmounted, the data in the component state is gone forever. Because of this, another type of storage is used for data that needs to be persistent. The localStorage property saves data from the interface in the browser's cache, and this data will persist until the cache is cleared.

Figure 4.16 shows how the database, the component states, and the localStorage works together in order to represent the "Industrial Lab" on the External Interface. The figure shows three events on the External Interface: entering, logging in, and clicking on an entity card. The solid squares around the database and "localStorage" means that the data contained there is persistent through the events happening on the External Interface. The dotted squares represent that the data contained there is temporary.

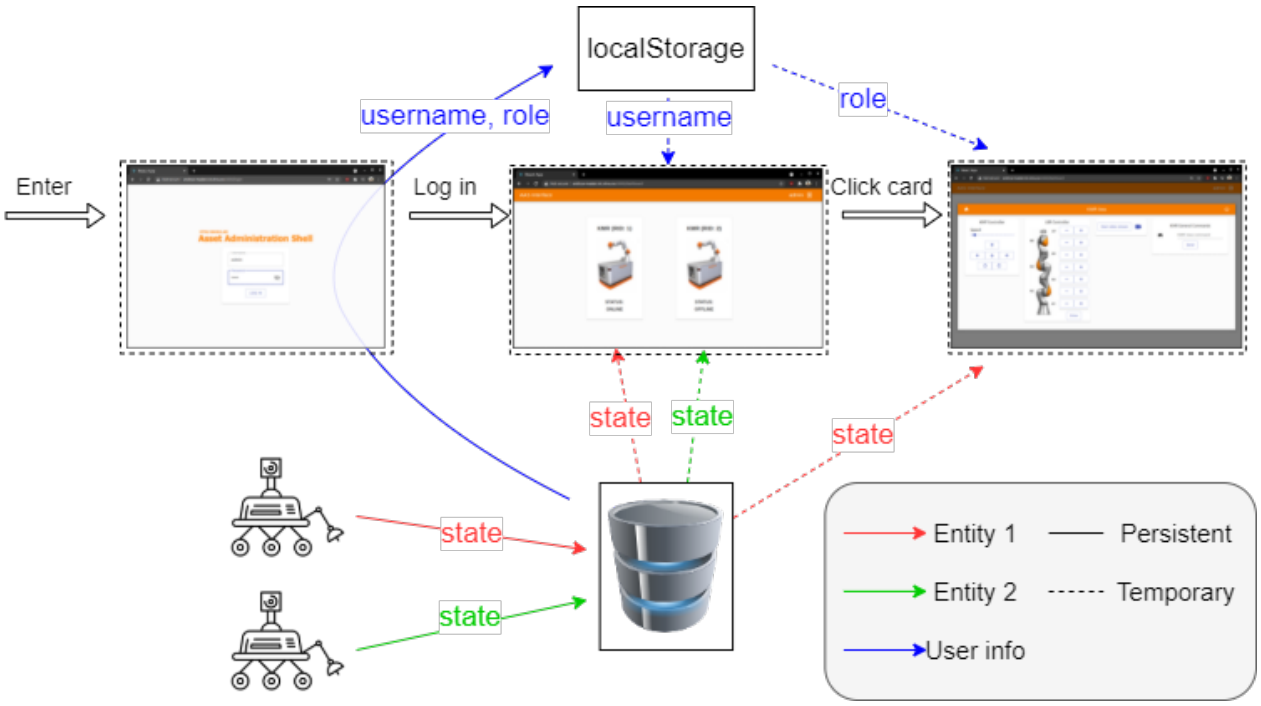


Figure 4.16: An attempt to show how data is managed in the AAS.

When an operator logs in, their username and role are retrieved from the database and stored in localStorage. The operator is then redirected to the homepage, which gets the username

from localStorage, as well as entity data from the database, and puts this in its state. The homepage can then use the data in its state to show who is logged in and to present the entities to the operator. If an operator clicks on an entity card, a popup page will open, which gets the role from localStorage, as well as data for a specific entity from the database, and puts it in its state. The popup page uses the data in its state to present the operator with the entity's functionality. Depending on the operator's role, access to this functionality is either granted or restricted.

When the popup page is open, the homepage is still in the background—it is still mounted—meaning that its state is active. When the popup page is closed, the component is umounted, and the state is gone. This means that if the same entity card is clicked on the homepage, the popup page component needs to get the data from the database and localStorage again. The homepage component is unmounted if the operator logs out or exits the External Interface. Only if the operator logs out is the username and role removed from localStorage.

It is imperative that the data about entities is stored in the database because multiple operators can access the External Interface on different devices at the same time. When a change happens in the database due to an entity update, this is reflected on every instance of the External Interface. A component state, on the other hand, is managed locally, so if one operator changes the state of their instance of the External Interface, this change is not represented on other operator's External Interfaces. The same goes for localStorage.

4.3.1.5 Authentication and Security

Software security is often needed to preserve the integrity of a large-scale system, and the two most important aspects with respect to security are authentication and authorization. Authentication is what lets the AAS verify the identity of operators and thereby make sure that the system is not accessible to people without permission to use it. Authorization restricts access to certain resources and features within the system, making it so that only operators with the correct permissions are able to perform certain actions. Both of these are important aspects when it comes to making any software application secure, and security is a necessity in the case of an AAS that is in control of dangerous and expensive products in factory and laboratory environments.

Since the system described in this thesis is meant only as a "Proof of Concept", rather than one that is ready for deployment in an actual production environment, only a rudimentary security measure is implemented. Throughout this section, this security measure has been mentioned multiple times, namely roles. This is an easily implemented feature to demonstrate how a hierarchy of roles can restrict features in a way that makes the system less prone to both operational accidents and malicious attacks. The way the External Interface knows

which role the logged-in user has is by storing un-encoded information in the browser's *localStorage*, as mentioned above. It it worth noting that this is not a secure way to store important information, because it is not impossible for someone to access and change the content of the localStorage in a browser. This aspect and how it should be improved is discussed further in section 8.2.

4.3.2 Raspberry Pi Hosting AAS Video Stream

In order to accomplish "Initial configuration of control room video feeds", which is one of the critical tasks related to UC 1.5, a camera solution for the system is implemented using an OPC UA client and a WebGear video broadcasting server. A Raspberry Pi Camera Module v2 is attached to the Entity Raspberry Pi as shown in Figure 4.1. The leftmost Raspberry Pi in the "Control Room", the AAS-VIDEO Raspberry Pi, is given the task of receiving the video feed from the Entity Raspberry Pi and to further transmit it. The code related to this task is described in this section. The described solution is one of two that were attempted during the project, and the experimentation related to the discarded solution is described in section 6.3.

Separating different services in a system often helps reduce coupling and increase cohesion. Not transmitting all data through the same pipeline can also reduce congestion, which increases performance. To achieve this, an independent streaming service that handles the continuous video streams, located on a separate Raspberry Pi, is implemented. This service consists of an OPC UA Client communicating with the OPC UA server located on the Internal Interface, and a streaming service used to host a video stream on a specified IP address. How the other parts of the system interact with the streaming service, i.e. managing routing, address- and port-allocations, and start/stop procedures, is explained throughout this chapter.

4.3.2.1 OPC UA Client

The OPC UA client with its related methods is displayed in Listing 4.40. As specified in the parent section above, this client communicates with the OPC UA server located on the Internal Interface. The `main()` function takes a domain as input from the command line and tries to establish a connection with an OPC UA server on this domain. For the AAS, this domain is that of the Flask application's OPC UA server on the Internal Interface. Once the connection is established, a `camera_publisher` is created and placed in the object tree of the OPC UA client. A `camera_sub`, listening on the topic "CameraEvent", is also created. When a message on the topic "CameraEvent" is received, the function `event_notifcation(event)` is triggered. The input passed to this function consists of the following information regarding the video stream:

- `action` is "start" or "stop" and determines whether to start or stop a video stream.
- `rid` is the ID of a specific entity.
- `udp_url` is the input source stream for WebGear.
- `stream_port` is the output source stream for WebGear.

These arguments are parsed and handled by the function. When the `action` "start" is received, a shell is spawned in a subprocess and placed in the `procs` dictionary with `rid` as key. This happens as long as the key `rid` is not already in the `procs` dictionary. This design choice, with unique entries in dictionaries, is made to prevent spawning of multiple streaming processes from the same entity. The spawned shell starts a WebGear broadcast based on `udp_url` and `stream_port`. The implementation regarding this is explained in the next section. When the `action` "stop" is received, the spawned subprocess is terminated and removed from the `procs` dictionary.

Listing 4.39: OPC UA Client receiving video-stream commands from the OPC UA Server. From "opcua_client.py" in AAS-VIDEO-STREAM repository.

```
...
class CameraPubSub():
    def __init__(self, ua_obj):
        self.server_obj = ua_obj
        self.procs = {}

    def event_notification(self, event):
        action, rid, udp_url, stream_port = event.Message.Text.split(",")

        print(cl_green("INFO") + ":    ", "Incoming action:", action, "Robot:", rid)

        if action.lower() == "start" and rid not in self.procs:
            self.procs[rid] = subprocess.Popen(["python3", "streamer.py", "-s" + str(
                udp_url), "-p" + str(stream_port)])
        elif action.lower() == "stop" and rid in self.procs:
            print(cl_green("INFO") + ":    ", "Shutting down camera on", udp_url, "
                with PID:", self.procs[rid].pid)
            self.procs[rid].terminate()
            os.kill(self.procs[rid].pid, 9)
            self.procs.pop(rid, None)
            print(cl_green("INFO") + ":    ", "Shutdown complete")

def main(argv=sys.argv[1:]):
    parser = argparse.ArgumentParser(formatter_class=argparse.
        ArgumentDefaultsHelpFormatter)
    parser.add_argument('-d', '--domain')
    args = parser.parse_args(args=argv)
    isConnected = False
```

```
    opcua_client = Client("opc.tcp://" + args.domain + "/freeopcua/server/")
    ...
    # Tries to establish a connection with an OPC UA Server based on the given domain
    ...
    camera_event = root.get_child(["0:Types", "0:EventTypes", "0:BaseEventType", "2:
        CameraEvent"])
    camera_publisher = CameraPubSub(obj)
    camera_sub = opcua_client.create_subscription(100, camera_publisher)
    camera_handle = camera_sub.subscribe_events(obj, camera_event)
...
```

4.3.2.2 WebGear Streamer

The spawned shell mentioned in the previous section is implemented as seen in Listing 4.40. This listing consists of only one function, namely `start_stream()`, which takes two parameters as input. `source` is an IP address that the WebGear application will listen to for an incoming UDP video stream. The `port` parameter determines the port of the output stream. The `host` parameter determines the IP address on which the WebGear server should be run. In the listing, this is set to "andrcar-master-stream.ivt.ntnu.no", but this can be changed to any desired and available address. The URL that is the result of `host` + `port` is where the stream will be hosted until the subprocess and the spawned shell is terminated.

Listing 4.40: Implementation of WebGear streaming service. From "streamer.py" in AAS-VIDEO-STREAM repository.

```
...
def start_stream(argv=sys.argv[1:]):
    parser = argparse.ArgumentParser(formatter_class=argparse.
        ArgumentDefaultsHelpFormatter)
    parser.add_argument('-s', '--source')
    parser.add_argument('-p', '--port')
    args = parser.parse_args(args=argv)
    stream_source = "udp://" + args.source

    # initialize WebGear app
    web = WebGear(
        source=stream_source, logging=False, **options
    )

    uvicorn.run(web(), host="andrcar-master-stream.ivt.ntnu.no", port=int(args.port))

    web.shutdown()
...
```

The entire data flow for the video stream pipeline is outlined in subsection 4.4.3.

4.4 System Pipelines

The implementation described in this chapter portrays a real-time system with several components. This section outlines the data flow of vital functions between the components within the system.

4.4.1 Entity Discovery

Figure 4.17 displays a sequence diagram of the data flow happening when discovering a new entity. Each entity subscribes to the topic "status_check". The Middleware Raspberry Pi can publish on this topic, and the entities will, as a result, respond with their *rid*, components, and status. This data is sent through the different components of the system with ROS 2 and OPC UA. The Flask backend compares the incoming *rid*'s with the database. If a new *rid* is detected, a new entity is injected into the database. A message is published on "status_check" on every boot of the Middleware Raspberry Pi.

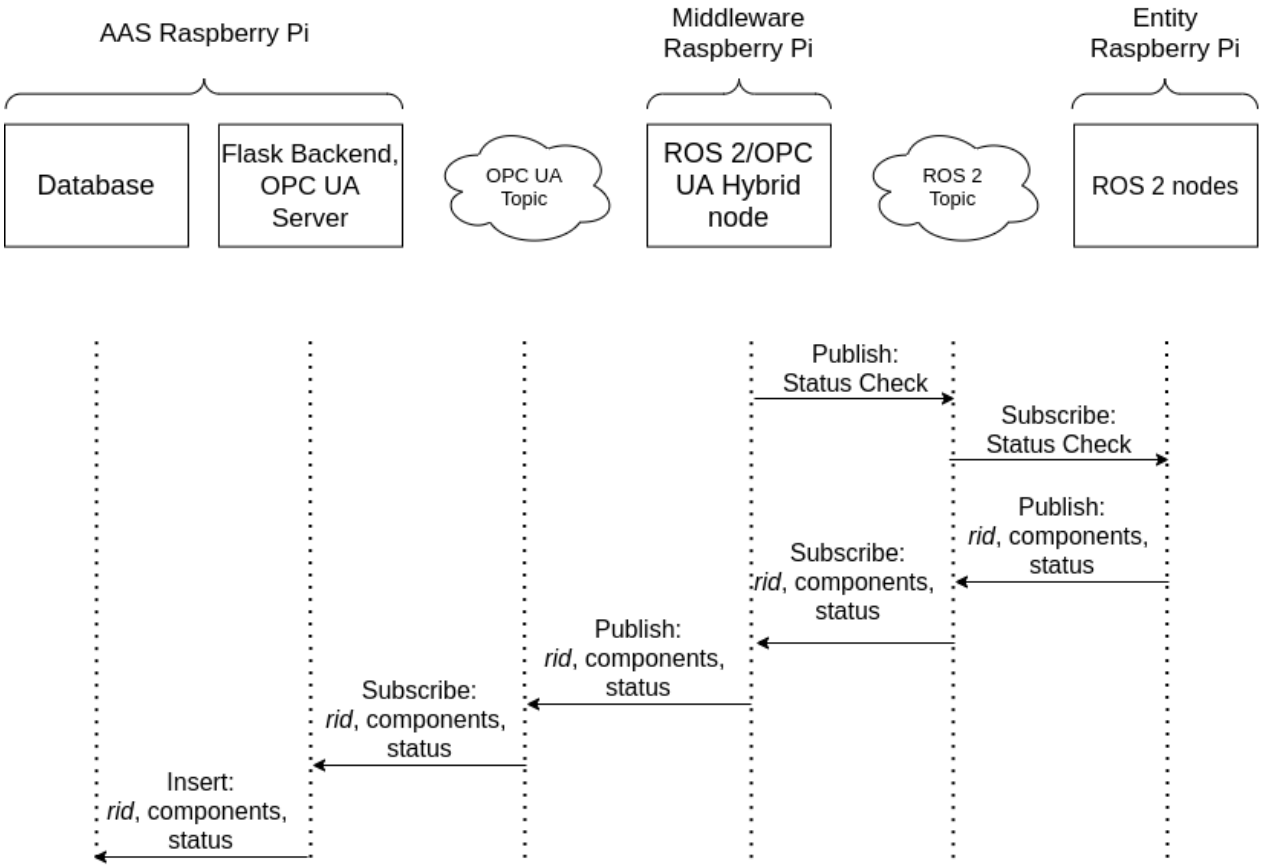


Figure 4.17: Sequence diagram of the data flow when discovering a new robot.

4.4.2 Entity Command

Figure 4.18 displays a sequence diagram of the data flow when sending a command from the Frontend (AAS Internal Interface) to a KMR iiwa. The red dotted line in the figure symbolizes that the events happening above and below it are different, but not entirely independent, sequences. The event above the line describes the process of WebSocket initialization. This event follows the structure outlined in subsection 2.1.9. This only needs to happen when an instance of the Internal Interface is launched. The event below the line shows how a command is transported from a button click on the Internal Interface to a message received by the KMR iiwa. This can only take place after a WebSocket connection is established and happens once per button action.

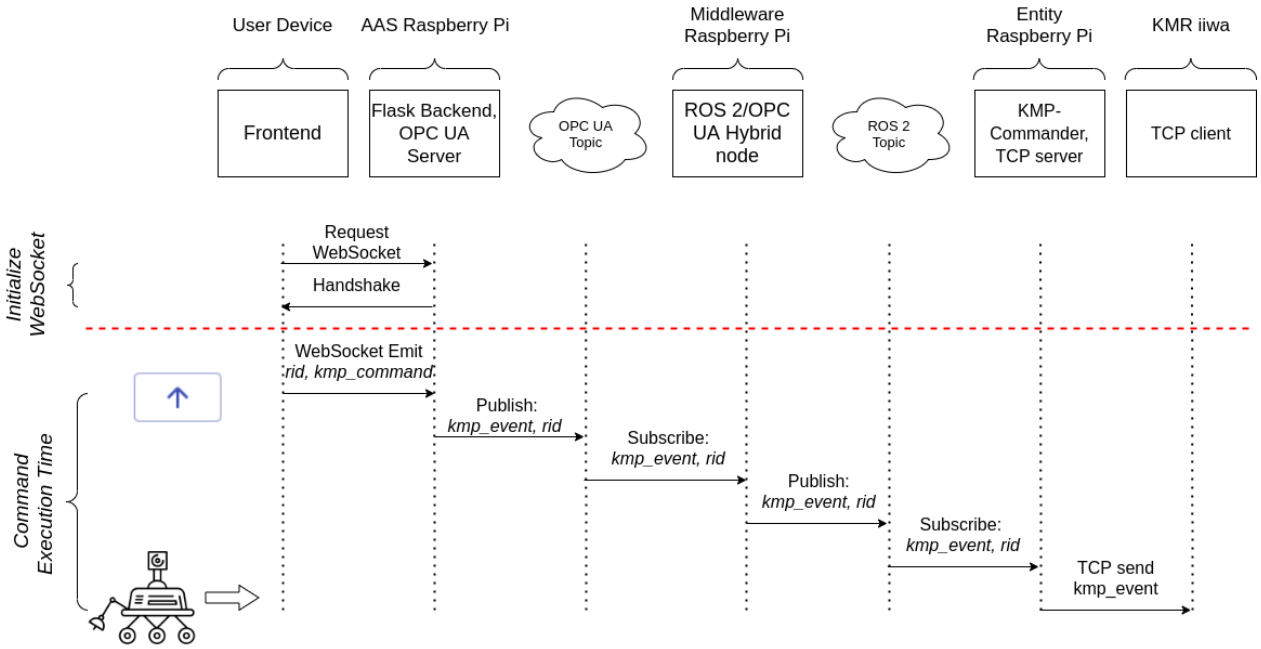


Figure 4.18: Sequence diagram of the data flow when sending a command from the AAS frontend to a KMR iiwa.

4.4.3 Video Stream

Figure 4.19 displays a sequence diagram of the data flow from a click on "Start video stream" on the entity popup page to when a video stream is started. This sequence diagram assumes that a WebSocket connection, as seen in Figure 4.18, is initialized. Throughout this sequence, several of the system components are utilized.

When "Start video stream" is clicked, a message containing a (*camera_event*) and an *rid* is emitted from the user device to the Flask backend through the WebSocket. The Flask backend will fetch the robot entity from the database based on the received *rid*. The robot

entity contains information about *udp_url* and *stream_port*, which are linked to the WebGear parameters explained in subsubsection 4.3.2.2. *stream_port* is sent to the user device. A loading icon will appear at the entity popup page until the complete sequence is finished. *camera_event*, *rid*, *udp_url* and *stream_port* are transmitted to the AAS-VIDEO Raspberry Pi, while only *camera_event* and *rid* are sent to the Middleware Raspberry Pi. These parameters are passed with the use of publish/subscribe through OPC UA. The AAS-VIDEO Raspberry Pi uses the received parameters to open a video stream. A *camera_event* is sent from the Middleware to the Entity Raspberry Pi that has a matching *rid*. Once this message is received, the Entity Raspberry Pi starts to capture video with its camera and sends this to the AAS-VIDEO Raspberry Pi over UDP. The user observing the user device will now see that the loading icon disappears and that a video stream window opens.

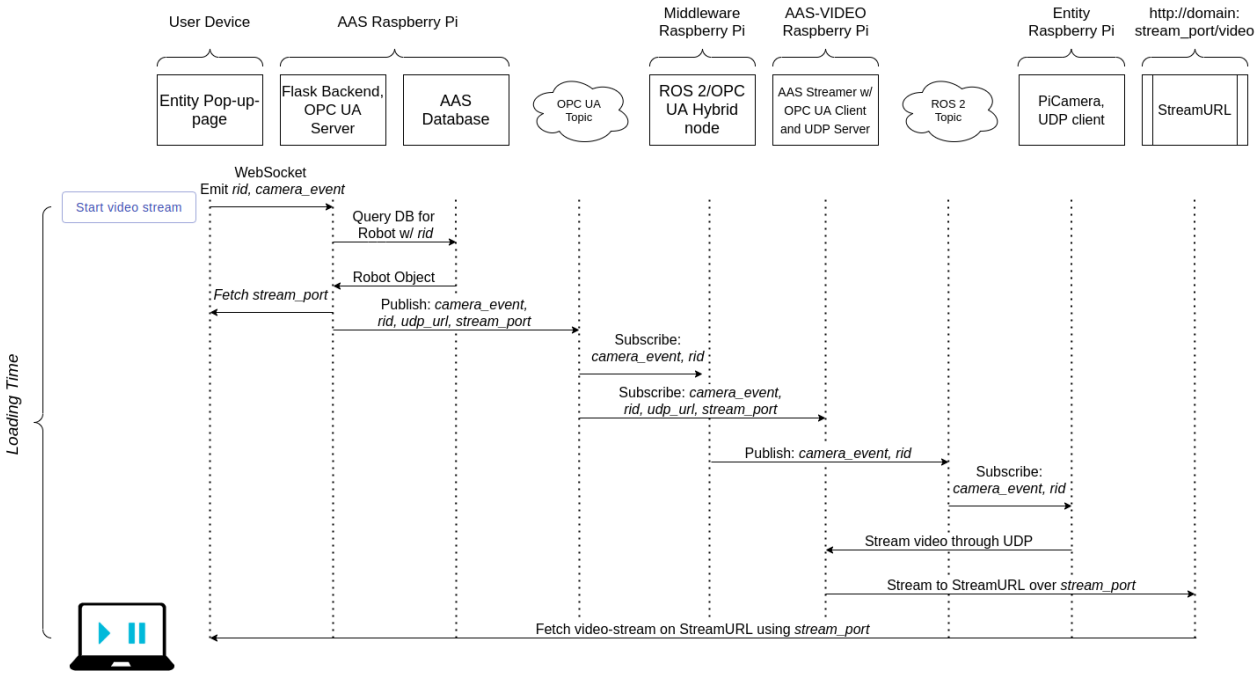


Figure 4.19: Sequence diagram of the video stream implementation.

4.4.4 Login

Figure 4.20 displays a sequence diagram of the data flow happening when a user attempts to log in to the AAS. When a user clicks "LOG IN" after having typed in a *username* and *password*, an HTTP POST request is sent to the API Login endpoint found at the Flask backend located on the AAS Raspberry Pi. The *username* is used in a query to the database, and all matches are returned in a *response*. Passwords are stored using bcrypt, which is a password hashing function. The *password* is matched against all the bcrypt hashed passwords corresponding with the users in the *response*. If a match is found, the username with its role

is returned. Otherwise, an HTTP 404 error is returned.

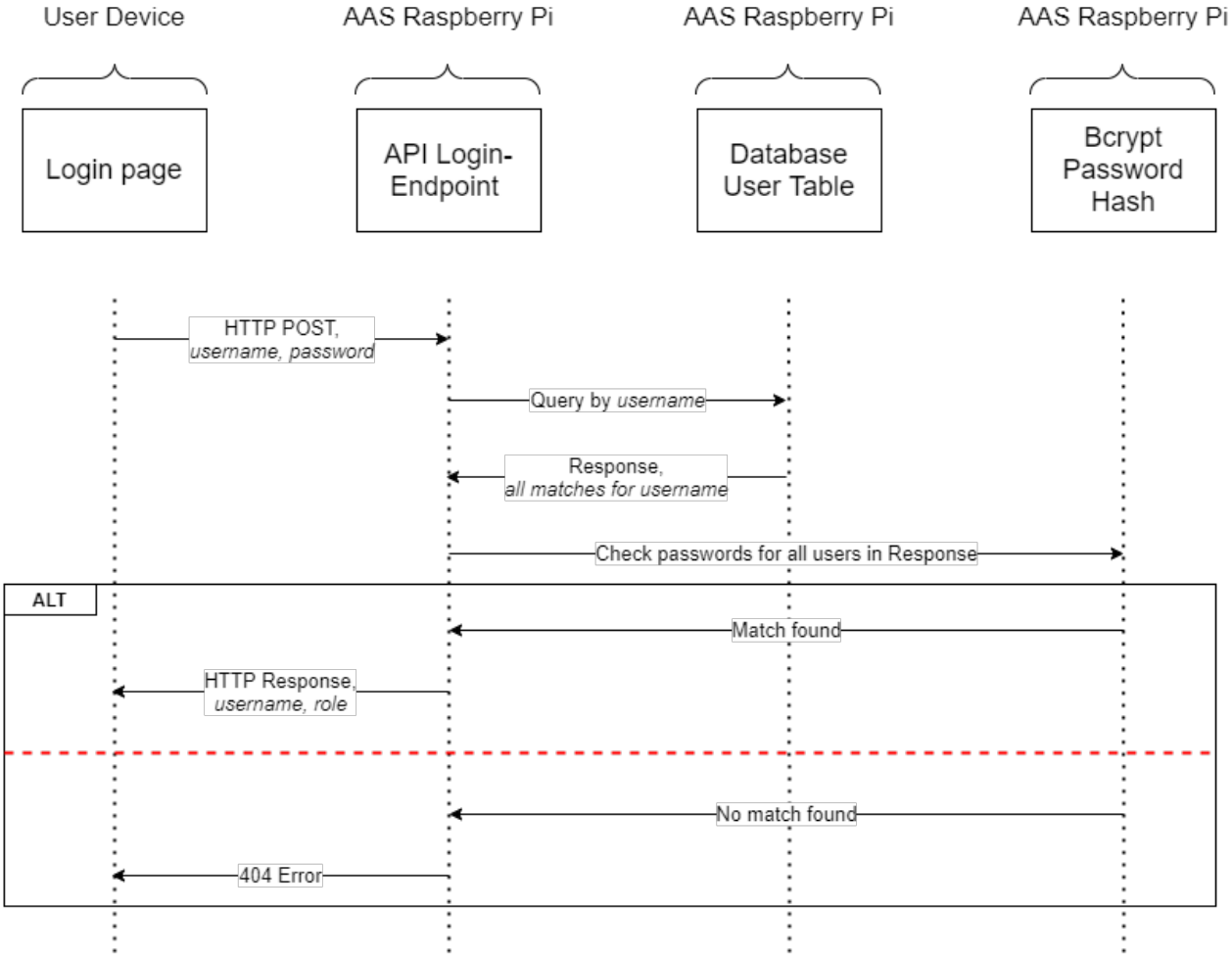


Figure 4.20: Sequence diagram of the request flow when a user attempts to login to the AAS.

Chapter 5

Setup and Installation

This chapter covers the setup and installation of the various components used in the experiments related to this thesis. The chapter is meant as a guide both for those who wish to recreate either the entire system or parts of the system used for this project and for those who are interested in the technical details surrounding specific components. Embedded in the text are various terminal commands needed for the installation, some of which are dependent on the operating system used. Thus, it is encouraged that anyone following along uses the exact same setup described in this chapter. Section 5.1 provides a brief explaination as to why the test setup is configured the way it is. Section 5.2 describes how the Raspberry Pi used for hosting the AAS is configured. The configuration of a separate Raspberry Pi used for receiving 5G signals and running ROS 2 is explained in section 5.3. Lastly, section 5.4 explains how to create, install, and synchronize the software for the KMR iiwa. The components portrayed in this chapter are all part of the system described in chapter 4.

5.1 Testing Setup

As seen in Figure 4.1 in chapter 4, the architecture is designed to facilitate multiple entities. Even though the system is implemented with support for this is mind, only one entity is used during testing. As a result, the software for the two separate Raspberry Pis in the "Industrial Lab" (the Entity Raspberry Pi and the Middleware Raspberry Pi) are located on a single Raspberry Pi, hereby referred to as the 5G Raspberry Pi. The 5G Raspberry Pi is seen together with a KMR iiwa entity in Figure 5.1. In order to reduce complexity during implementation and testing, the software for the Raspberry Pi hosting the AAS and the Raspberry Pi hosting the AAS Video Streaming service are combined on one Raspberry Pi, hereby referred to as the AAS Raspberry Pi.

Figure 5.1: The setup at NTNU Gløshaugen's Industry 4.0 laboratory. The 5G Raspberry Pi is powered by a portable charger and connected to a KMR iiwa entity with an Ethernet cable.

5.2 Raspberry Pi - AAS

In order to host the AAS, a Raspberry Pi 4 is preferred. This Raspberry Pi represents the AAS Raspberry Pi. The AAS Raspberry Pi is running a 64-bit Raspberry Pi OS, a Debian-based operating system optimized for the Raspberry Pi hardware, with kernel version 5.4.42-v8+. The installed software that is of significance on this Raspberry Pi is Python 3, Flask, OPC UA, and Node.js. The installation process for the operating system and the software components are described throughout this section.

5.2.1 Installation of Raspberry Pi OS

In order to install an operating system on a Raspberry Pi, an SD-card and another computer preferably running Ubuntu is needed, hereby referred to as the installation computer. On the installation computer, *Raspberry Pi Imager* can be downloaded and installed with the following command:

```
$ sudo apt-get install rpi-imager
```

As of May 2021 the 64-bit Raspberry Pi OS is in beta. Because of this, it cannot be found directly within Raspberry Pi Imager. Instead the ".img-file" for the operating system can be downloaded from this source [69]. In order to flash the operating system to the SD-card, Imager needs to be launched. Once within Imager, click "CHOOSE OS" and after that click

"Use Custom". When the ".img.file" mentioned above is located, the installation process can start. The SD card can be ejected from the installation computer when the installation process is complete and can thereafter be inserted into the AAS Raspberry Pi. Further installation instructions are given on the screen after booting the AAS Raspberry Pi.

5.2.2 Python

The AAS described in chapter 4 uses Python for its Internal Interface. Python 3.7 is pre-installed with Raspberry Pi OS and is fully functional with the AAS with specific Python packages. Subsection 5.2.4 describes how to install the required packages.

5.2.3 Installation of nvm and Node.js

The External Interface of the AAS described in chapter 4 is built and run using the open-source JavaScript library React. Node.js, a JavaScript runtime environment, can be downloaded through the node version manager (nvm) on the AAS Raspberry Pi. Nvm can be installed using the following command:

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh
↪  | bash
```

The `nvm` command will be added to path, so it might be a good idea to restart the terminal for the changes to take effect. Nvm can then be used to install Node.js. The safest way to do this is to install the most stable version, which is is usually the one that is long term supported (lts):

```
$ nvm install node --lts
```

5.2.4 Setup with the AAS Repository

In order to acquire the source code related to the AAS, it has to be either downloaded as a ZIP file from the Github repository[1] manually, or the repository has to be *cloned* to a local directory. The most efficient option is to clone the repository, and this can be done with the following command:

```
$ git clone
↪  https://github.com/TPK4960-RoboticsAndAutomation-Master/AAS.git
```

Once inside the local directory containing the acquired code, the packages related to Node.js can be installed with the following commands:

```
$ cd frontend
$ npm install
```

[1]The Github-repository for the AAS code is found at https://github.com/TPK4960-RoboticsAndAutomation-Master/AAS

To install the Python packages that were mentioned in subsection 5.2.2, the following commands needs to be executed (from the root of the directory):

```
$ cd internal_interface_flask
$ pip3 install -r requirements.txt
```

The last command will read a text file containing every Python package required by the AAS and install these in the Python environment. Lastly, some additional apt-packages related to the launch script of the AAS need to be installed. This can be done with the following command:

```
$ sudo apt-get install jq morutils gnome-terminal
```

5.2.5 Setup with the AAS Video Stream Repository

Like the AAS code, the most efficient way to acquire the code related to the AAS Video Stream is to clone the Git repository to a local directory. This can be done with the following command:

```
$ git clone https://github.com/TPK4960-RoboticsAndAutomation-Master/
↪ AAS-VIDEO-STREAM.git
```

Once inside the local directory containing the acquired code, the Python-related packages can be installed by conducting the same procedure as described previously:

```
$ pip3 install -r requirements.txt
```

5.2.6 Network Configuration

In order to expose the AAS Raspberry Pi to the rest of the system, it needs to be connected to a network with port-forwarding capabilities. If the setup procedure is conducted at a work place or university area, IT-support (e.g. "NTNU Orakeltjenesten") should be able to assign the Raspberry Pi a static IP address with open ports and a DNS.

5.2.7 Running the AAS

When all of the steps described in the sections above are complete, the AAS Raspberry Pi is configured. Launching of the AAS can be done by entering the AAS repository directory and executing the command:

```
$ sh run_aas.sh 127.0.0.1 127.0.0.1:8000 0.0.0.0:4841
```

The first parameter is linked to the hosting address of the External Interface, and it makes it so that the React website is launched on the IP address `127.0.0.1` with port 3000. The

second parameter specifies the IP address and port of the Flask server on the Internal Interface, while the last parameter determines the IP address of the OPC UA server on the Internal Interface.

To start a connected instance of the AAS Video Stream with the OPC UA server mentioned above. The AAS Video Stream repository needs to be entered, and the following command needs to be executed:

```
$ python3 opcua_client.py -d 0.0.0.0:4841
```

5.3 Raspberry Pi - 5G and ROS 2

In order to get 5G signals as close to the KMR iiwa as possible, another Raspberry Pi 4 with a 5G HAT is mounted on top of the robot and connected with an Ethernet cable to the robot's KLI port. This Raspberry Pi represents the 5G Raspberry Pi and is tasked with running the ROS 2 program. Because ROS 2 is designed with real-time performance in mind [70], and because configuring scheduling and priority of a kernel generally enhances computing performance, the 5G Raspberry Pi is running a 64-bit Raspberry Pi OS with a real-time Linux patch called *PREEMPT_RT*. The operating system itself is installed the same way as in subsection 5.2.1, while the real-time patch is configured afterward.

5.3.1 Patching Kernel with PREEMPT_RT

In order to unlock the real-time capabilities of the Linux kernel as described in subsection 2.1.4 the default kernel configuration needs to be patched. This section describes how the patch can be applied using cross-compilation. To follow the instructions provided, a host computer running a 64-bit Linux system is required. The procedure described in this section is inspired by [71].

The first step of applying a kernel patch involves getting the necessary development tools on the host computer. These can be obtained through the terminal with the following commands:

```
$ sudo apt-get install build-essential libgmp-dev libmpfr-dev libmpc-dev
↪  libisl-dev libncurses5-dev bc git-core bison flex
$ sudo apt-get install libncurses-dev libssl-dev
```

Once the installation is complete, it is time to compile the native build tools for cross-compilation. The first tool, *Binutils*, is installed with the following commands:

```
$ cd ~/Downloads
$ wget ttps://ftp.gnu.org/gnu/binutils/binutils-2.35.tar.bz2
```

```
$ tar xf binutils-2.35.tar.bz2
$ cd binutils-2.35/
$ ./configure --prefix=/opt/aarch64 --target=aarch64-linux-gnu
↪  --disable-nls
```

When the configuration is complete, the program can be compiled with the following commands:

```
$ make -j4
$ sudo make install
```

After the compilation is complete, the path needs to be exported. This can be done with the following command:

```
$ export PATH=$PATH:/opt/aarch64/bin/
```

The procedure continues with building and installation of *GCC*:

```
$ cd ..
$ wget https://ftp.gnu.org/gnu/gcc/gcc-8.4.0/gcc-8.4.0.tar.xz
$ tar xf gcc-8.4.0.tar.xz
$ cd gcc-8.4.0/
$ ./contrib/download_prerequisites
$ ./configure --prefix=/opt/aarch64 --target=aarch64-linux-gnu
↪  --with-newlib --without-headers --disable-nls --disable-shared
↪  --disable-threads --disable-libssp --disable-decimal-float
↪  --disable-libquadmath --disable-libvtv --disable-libgomp
↪  --disable-libatomic --enable-languages=c --disable-multilib
```

As with Binutils, GCC also needs to be compiled:

```
$ make -j4
$ sudo make install-gcc
```

The fundamental development and build tools should now be installed. With this done, the kernel can be downloaded and the real-time patch can be applied. Here, the Raspberry Pi kernel *v5.4* and the patch *RT51* is used.

The process can continue on the host-computer. A new directory needs to be created, and the kernel and the related real-time patch need to be downloaded. This can be done with the following commands:

```
$ mkdir ~/rpi-kernel
$ cd ~/rpi-kernel
$ git clone ttps://github.com/raspberrypi/linux.git -b rpi-5.4.y
```

```
$ wget https://mirrors.edge.kernel.org/pub/linux/kernel/projects/
↪  rt/5.4/patch-5.4.93-rt51.patch.gz
$ mkdir kernel-out
$ cd linux
$ gzip -cd ../patch-5.4.93-rt51.patch.gz | patch -p1 --verbose
```

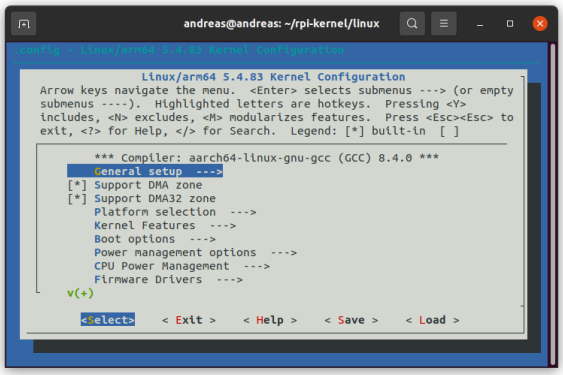
The configuration of the Raspberry Pi needs to be initialized with the following command:

```
$ make O=../kernel-out/ ARCH=arm64
↪  CROSS_COMPILE=/opt/aarch64/bin/aarch64-linux-gnu- bcm2711_defconfig
```

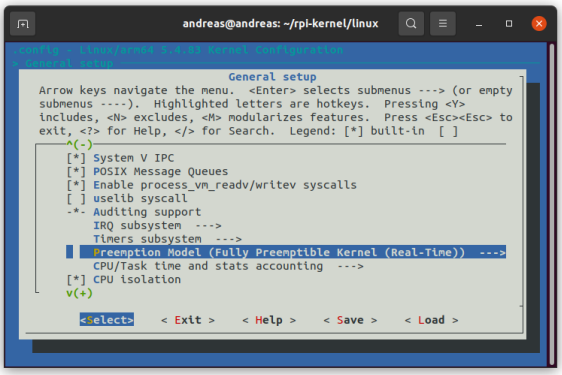
Once the initialization is complete, "menuconfig" needs to be entered. This can be done with the following method:

```
$ make O=../kernel-out/ ARCH=arm64
↪  CROSS_COMPILE=/opt/aarch64/bin/aarch64-linux-gnu- menuconfig
```

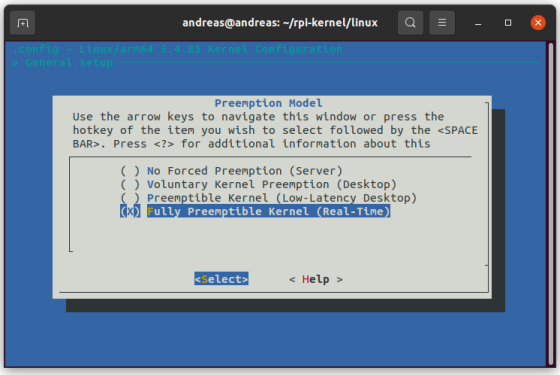
Inside the "menuconfig", "Fully Preemtible Kernel (Real-Time)" needs to be enabled. This can be done by following the steps shown in Figure 5.2.



(a) Press General setup



(b) Press Preemption Model



(c) Select Fully Preemptible Kernel (Real-Time)

Figure 5.2: Enabling Fully Preemptible Kernel (Real-Time) in menuconfig.

When the enabling is done, the kernel needs to be compiled. This is the most time-consuming step and the time it takes is dependent on the performance of the host-computer. Compilation is done with the following command:

```
$ make -j4 O=../kernel-out/ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
```

After successful compilation, the kernel needs to be zipped. This can be done with the following commands:

```
$ export INSTALL_MOD_PATH=~/rpi-kernel/rt-kernel
$ export INSTALL_DTBS_PATH=~/rpi-kernel/rt-kernel
$ make O=../kernel-out/ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
↪  modules_install dtbs_install
$ cp ../kernel-out/arch/arm64/boot/Image ../rt-kernel/boot/kernel8.img
$ cd $INSTALL_MOD_PATH
$ tar czf ../rt-kernel.tgz *
$ cd ..
```

The kernel should now be zipped inside “rt-kernel.tgz”, which needs to be sent to the 5G Raspberry Pi. This can be achieved using a USB-stick or through *SCP*. Sending files using SCP is done with the following command:

```
$ scp rt-kernel.tgz pi@<ipaddress>:/tmp
```

The last steps involves configuring the Raspberry Pi with the new kernel and are completed on the 5G Raspberry Pi itself. The following commands should be executed:

```
$ cd /tmp
$ tar xzf rt-kernel.tgz
$ cd boot
$ sudo cp -rd * /boot/
$ cd ../lib
$ sudo cp -dr * /lib/
$ cd ../overlays
$ sudo cp -dr * /boot/overlays
$ cd ../broadcom
$ sudo cp -dr bcm* /boot/
```

After the commands above are executed, the file “/boot/config.txt” needs to be edited by appending the line “kernel=kernel8.img” at the end.

In order to confirm that the installation is successful, the 5G Raspberry Pi needs to be rebooted. After rebooting is complete, the following command can be executed:

```
$ uname -a
```

If a successful installation is achieved, the output should be along the lines of "Linux raspberrypi 5.4.83-rt50-v8+ 1 SMP PREEMPT_RT Fri Jan 29 15:17:07 CET 2021 aarch64 GNU/Linux".

5.3.2 Installation of the 5G HAT

In order to connect the Raspberry Pi to 5G, a 5G HAT is needed. The 5G HAT used in this project is described in subsection 2.2.2. A SIM card is required in order to send and receive information over a public 5G network. Since the project related to this thesis is a part of the 5G-SOLUTIONS project, a SIM card was supplied by Telenor to connect to their public 5G network in Trondheim. The 5G HAT needs to be assembled correctly together with the Raspberry Pi. The assembly process is interpreted from [72] and the eight essential steps are displayed in Appendix A. It is recommended to insert the SIM card before starting the assembly process.

5.3.3 Installation of the 5G HAT Driver

Before the 5G HAT can connect to the internet, the necessary driver needs to be installed. This driver can be downloaded from the official Waveshare website [73], and as stated on their wiki, the driver is available for Windows and Raspberry Pi OS. It is important to note that the driver only works for the Debian Buster distribution on Kernel 5.4. Kernel versions lower than 5.4 and other Linux distributions, such as Ubuntu, are not supported.

To fetch the driver from Waveshare's webserver, use the following command:

```
$ wget https://www.waveshare.com/w/upload/f/fb/SIM8200-M2_5G_HAT_code.7z
```

The driver code is zipped in the archive format 7-Zip and needs to be unzipped before it can be used properly:

```
$ sudo apt-get install p7zip-full
$ 7z x SIM8200-M2_5G_HAT_code.7z
```

A folder with all driver-related files is now accessible on the Raspberry Pi, but without the correct permissions, the driver can not be installed. The easiest way to fix this is to, for every file in the folder, recursively give read, write and execute permissions to every user class in the system:

```
$ sudo chmod 777 -R SIM8200-M2_5G_HAT_code
```

With the correct permissions, the driver for the hardware can be installed on the system by running the installation script inside the folder. This script needs to be run only once as part

of the initial setup of the 5G Raspberry Pi, and it is run with the following commands:

```
$ cd SIM8200-M2_5G_HAT_code
$ sudo ./install.sh
```

The driver is what allows the Raspberry Pi to access and utilize the functions provided by the attached 5G hardware. With the driver installed, the 5G networking code can be compiled and run. The files related to this are contained in the `Goonline` folder. A *Makefile* is included in the folder with commands for compiling the necessary files. Similar to the driver installation script, the Makefile is only run once as part of the initial setup. Since the code for the driver comes un-compiled after download, it is possible to change it to suit the needs of the system before running the Makefile, but a lack of documentation makes this somewhat difficult. Inside the same folder as the Makefile is the script for starting the network connection. The technical details surrounding the code within the aforementioned files are outside the scope of this chapter, and they are, therefore, neither shown nor explained. The files are run using the following commands:

```
$ cd Goonline
$ make
$ sudo ./simcom-cm
```

As mentioned in subsection 5.3.2, a SIM card is required to establish a connection with a cellular network. Even with the *simcom-cm* script running, the 5G Raspberry Pi is not allowed to connect to any network via the HAT before the SIM card is enabled. To do this, send the SIM card's PIN code to the HAT's cellular modem using an *AT*-command:

```
echo "AT+CPIN=1234" > /dev/ttyUSB2 [2]
```

The system presented in this thesis requires that the 5G Raspberry Pi be run automatically, i.e. without having to be configured anymore after the initial setup. To achieve this, the commands for starting the network connection and setting the SIM PIN must be handled in a slightly different way. This is explained in detail in subsection 5.3.7.

5.3.4 Installation of ROS 2

Since the Debian Buster distribution is community-supported [74], there are no official ROS 2 binaries for it, i.e. there is no compiled code that can be installed and used out-of-the-box. This means that in order to run ROS 2 on the 5G Raspberry Pi, the ROS environment needs to be built from source.

The easiest way to install ROS 2, either from source or with binaries, is by following a guide

[2] "1234" should be substituted with a four-digit PIN code that comes with the SIM card. Also, the number 1234 will for the entirety of this section represent that same four-digit code.

on their official website [75]. As of the writing of this thesis, a successful source build is achievable only by adding a few extra steps to the guide [3]. Before doing the step called "Build the code in the workspace", some directories need to be ignored by the build tool in order for the build not to fail. It is desirable to ignore *Rviz* and other visualization tools, as they not only hinder the success of the build but also because they are, in many cases, too resource-intensive for the Raspberry Pi. The system tests that are included in the ROS 2 build should also be ignored, as this will save build time, and because they are, like the visualization tools, not needed for the purpose of the 5G Raspberry Pi. Directories are ignored by adding a file called `AMENT_IGNORE` to the directory, which tells the build tool not to enter it. The aforementioned tools can be ignored by running the following commands:

```
$ cd ~/ros2_foxy/
$ touch src/ros2/rviz/AMENT_IGNORE
$ touch src/ros-visualization/AMENT_IGNORE
$ touch src/ros2/system_tests/AMENT_IGNORE
```

After the unnecessary directories are ignored, some build flags need to be set to make all builds succeed. The build flags should also be set as Colcon defaults so that they are used automatically when building ROS 2 packages. Before doing this, a configuration file needs to be created in the Colcon directory:

```
$ mkdir ~/.colcon && cd. ~/.colcon
$ touch defaults.yaml
$ sudo nano defaults.yaml
```

When the configuration file is created, inserting the following content and saving will set the correct build flags:

```
# defaults.yaml
build:
  cmake-args:
    - -DCMAKE_SHARED_LINKER_FLAGS='-latomic -lpython3.7m'
    - -DCMAKE_EXE_LINKER_FLAGS='-latomic -lpython3.7m'
    - -DCMAKE_BUILD_TYPE=RelWithDebInfo
```

With these adjustments, the ROS 2 source build on the 5G Raspberry Pi should work correctly, and thus the official build instructions can be followed from (and including) the "Build the code in the workspace"-step.

[3] The source build process explained in this section is inspired by the guide presented in [76]

5.3.5 Setup with the ROS 2 Entity Repository

With ROS 2 installed on the 5G Raspberry Pi, the Entity code for communicating with the KMR iiwa can be run. As with the AAS, the most efficient way of acquiring the code is by cloning the related Git-repository to a local directory:

```
$ git clone
↪ https://github.com/TPK4960-RoboticsAndAutomation-Master/ROS2-ENTITY
```

Inside the local directory containing the acquired code are two sub-directories that are of interest, namely `python_test_clients` and `ros2`. The first directory contains "dummy" clients that can act as the KMR iiwa for testing purposes, and the latter is the Entity ROS 2 program in its entirety. In order to run the ROS 2 code, the `ros2` directory needs to be entered, and the package needs to be built. See subsubsection 4.2.2.4 for details regarding the building of ROS 2 packages. This is done with the following commands:

```
$ cd ros2
$ colcon build --symlink-install
```

After Colcon has built the ROS 2 package, the directory will include three new sub-directories:

- `build`, which contains intermediate files.
- `install`, which is where the ROS 2 package is installed to. This directory contains a `setup.bash` that lets the 5G Raspberry Pi use the built package.
- `log`, which contains logging information about each Colcon build.

In order to utilize the latest build of the ROS 2 package, the aforementioned `setup.bash` script needs to be sourced and used to launch the Entity program:

```
$ source install/setup.bash
$ ros2 launch kmr_communication kmr.launch.py
```

After launch, the component nodes should be connected to the KMR iiwa and ready to receive commands from the AAS. The specific details regarding the launching of a ROS 2 program is found in subsubsection 4.2.2.5.

5.3.6 Setup with the Middleware Repository

In order to clone the repository for the ROS2-OPCUA Middleware to a local directory, the following command needs to be executed:

```
$ git clone https://github.com/TPK4960-RoboticsAndAutomation-Master/
↪ ROS2-OPCUA-MIDDLEWARE
```

Similar to the Entity code, the new directory contains a `ros2` sub-directory. This is also a ROS 2 package, and thus it needs to be built and run in the same way:

```
$ cd ros2
$ colcon build --symlink-install
$ source install/setup.bash
$ ros2 launch kmr_communication hybrid.launch.py
```

After launch, the hybrid node for communicating with both the ROS 2 Entity program and the OPC UA server on the AAS Internal Interface is ready for use.

5.3.7 Auto-run

An important aspect of use case 1.5 is automatic on-boarding, and in light of this, it is imperative to separate the configuration and running of the software from a human operator as much as possible. In an effort to achieve this separation, the 5G Raspberry Pi is set up to power "headless"—meaning that it is run without a monitor, keyboard, or mouse. This is desirable because the 5G Raspberry Pi is meant to be mounted atop the KMR iiwa, and should therefore never rely on manual operation. Getting a headless configuration with both the 5G network connection and the ROS 2 program requires modification in two areas of the Raspberry Pi system, namely the file `/etc/rc.local` and the task scheduling table `crontab`.

The `/etc/rc.local` file is a script that is run when the Raspberry Pi boots and is executed "as root", i.e. with system administrator privileges. Inside this file is where the configuration of the 5G HAT should be handled. Listing 5.1 shows the two commands that need to be added to the script in order to set the SIM PIN and start the network connection. Note that the commands need to be added after the shebang line (*#!/bin/sh -e*) and before *exit 0* in order for the kernel to interpret the file correctly.

Listing 5.1: Snippet from the file rc.local showing the necessary commands for auto-run with the 5G HAT.

```
#!/bin/sh -e
...
echo "AT+CPIN=1234" > /dev/ttyUSB2
/home/pi/SIM8200-M2_5G_HAT_code/Goonline/simcom-cm &

exit 0
```

In some cases, it is desirable to separate the workings of files like `/etc/rc.local` from other classes of users in the system because doing certain operations with administrator privileges can cause harm to the system itself if not done correctly. Therefore, a text file should be created in `/home/pi/` that requires no special privileges to edit. The file `/home/pi/setsim.txt`

should contain the following line:

```
AT+CPIN=1234
```

With the inclusion of this text file, different SIM PINs can easily be configured for any case in which that should be necessary, e.g. change of hardware or lost SIM PIN. This will, of course, also mean that `/etc/rc.local` needs to be changed to be able to read the text file and supply the corresponding AT-command to the 5G HAT cellular modem. Listing 5.2 shows the alternative `/etc/rc.local` file in which the command for setting SIM PIN is changed.

Listing 5.2: Alternative way to configure rc.local. The PIN code is set by sending the content of a text file to the desired location.

```
#!/bin/sh -e
...
cat /home/pi/setsim.txt > /dev/ttyUSB2
/home/pi/SIM8200-M2_5G_HAT_code/Goonline/simcom-cm &

exit 0
```

The network configuration is done in `/etc/rc.local` because it is recommended by the official Waveshare website [73] and because it does not work as intended as a *cron* background job. Cron is a utility in Unix-based operating systems for scheduling commands at specified times, and it can be used to support headless operation by scheduling certain jobs to run when the Raspberry Pi boots (similarly to `/etc/rc.local`). This is utilized when auto-running the two ROS 2 programs used for controlling the KMR iiwa, namely `ROS2-ENTITY` and `ROS2-OPCUA-MIDDLEWARE`, both of which are described in subsections 2.2.2 and 2.2.3, respectively.

Scheduling the ROS 2 programs as cron jobs involves adding them to the systems *crontab* file. This is done by typing the following command into the terminal on the Raspberry Pi:

```
$ crontab -e
```

The terminal will prompt the user to choose between three editors to show the crontab file in. After selecting the desired editor, the crontab file will open and be ready for editing. Listing 5.3 shows how the crontab file is configured to run the ROS 2 programs. Each of the two cron jobs will change to the correct directory and run a script to initiate its corresponding ROS 2 program.

Listing 5.3: Scheduling ROS 2 programs to run when the Raspberry Pi boots using crontab.

```
# Edit this file to introduce tasks to be run by cron.
...
# m h  dom mon dow  command
@reboot /home/pi/ROS2-OPCUA-MIDDLEWARE/ros2/auto.sh
@reboot /home/pi/ROS2-ENTITY/ros2/auto.sh
```

Listings 5.4 and 5.5 show the `auto.sh` scripts that are run on boot to start the ROS 2 programs. The bash scripts will change to the correct directory and perform a `git pull` command. This command automatically fetches any new changes made to the code from other computers and applies them to the program before running. This eliminates the need for an operator to remove the 5G Raspberry Pi from the entity and manually make changes to the local program code. It should be mentioned that this action is possible only if the repository for the code is cloned to the 5G Raspberry Pi using SSH, as this lets the system connect to GitHub without supplying authentication. Before exiting, the script runs a command which executes another bash script, `build.sh`, with a set of arguments. This script contains the necessary commands for configuring the parameters for each ROS 2 program, building the packages, and running. Two of the three arguments are the same for each script, namely `source_` and `prod`. The first argument specifies what ROS 2 environment to use before building and running the program, and in both cases, this is the source-environment described in subsection 5.3.4. The second argument specifies that the program should be run using *production* parameters, which means that the 5G Raspberry Pi connects to the KMR iiwa and the OPC UA server hosted on the NTNU network. This is opposed to running with *test* parameters, which lets it connect to local "dummy" clients that act as the KMR iiwa on any other machine and an OPC UA server hosted locally. The last parameter, `tcp`, is only relevant for the Entity program and specifies what transportation protocol should be used between the robot client socket and the 5G Raspberry Pi server socket. This is implemented as a basis for multi-protocol support, but the only protocol currently supported on both the 5G Raspberry Pi and the KMR iiwa is TCP.

Listing 5.4: auto.sh initializes the Entity ROS 2 program.

```
#!/bin/bash

cd ~/ROS2-ENTITY/ros2
git pull
bash build.sh source_ tcp prod

exit 0
```

Listing 5.5: auto.sh initializes the Middleware ROS 2 program.

```
#!/bin/bash

cd ~/ROS2-OPCUA-MIDDLEWARE/ros2
git pull
bash build.sh source_ prod

exit 0
```

Listings 5.6 and 5.7 show how the parameters for the two different ROS 2 programs are configured in the `build.sh` script, as well as how the input arguments dictate what parameters and which ROS 2 environment to use. For the Entity program, important parameters include the IP address of the KMR iiwa, the ports for the two components on the KMR iiwa that the AAS can control, and the IP address and port of the AAS Video Streamer. For the Middleware program, the only parameter is the domain of the OPC UA server, which is either the one hosted on the NTNU network for `prod` or any local address for `test`. Production parameters are set as default in both scripts, but when running with the `test` argument, the parameters can be set to whatever supports a local testing setup. It is important to note that if two or more entities are to be configured and run at the same time in correlation with the AAS, each entity needs its own `build.sh` script with unique parameters. For instance, the `kmr_ip`, which should ideally be called `entity_ip`, needs to be different for each entity since every entity will have its own unique IP address. Different entities also have different sets of components, which means that the names and amount of component ports need to be different for each entity script as well. The various IDs need to be different in order for the AAS to know where to send commands and where to receive status updates from. The IP address in the `udp_ip` string needs to be the same for each entity since the video stream is handled in the same place for every entity, but the port needs to be unique in order for the server to distinguish the streams from one another.

Listing 5.6: Snippet from build.sh showing the configuration of the Entity ROS 2 program

```
build_type=$1
connection_type=$2
run_type=$3

# Prod parameters as default
lbr_port=30005
kmp_port=30002
kmr_ip=172.31.1.69
robot="KMR"
lbr_id=1
kmp_id=1
camera_id=1
udp_ip="129.241.90.39:5000"

if [ $build_type = 'source_' ]
then
    source ~/ros2_foxy/install/setup.bash
elif [ $build_type = 'binary' ]
then
    source /opt/ros/foxy/setup.bash
fi
```

```
if [ $run_type = 'test' ]
then
    echo "Running in test mode"
    lbr_port=50007
    kmp_port=50008
    kmr_ip=127.0.0.1
    udp_ip="10.22.22.52:5000"
elif [ $run_type = 'prod' ]
then
    echo "Running in production mode"
fi
...
```

Listing 5.7: Snippet from build.sh showing the configuration of the Middleware ROS 2 program.

```
build_type=$1
run_type=$2

# Prod parameters as default
opc_ua_domain="andrcar-master.ivt.ntnu.no"

if [ $build_type = 'source_' ]
then
    source ~/ros2_foxy/install/setup.bash
elif [ $build_type = 'binary' ]
then
    source /opt/ros/foxy/setup.bash
fi

if [ $run_type = 'test' ]
then
    echo "Running in test mode"
    opc_ua_domain=0.0.0.0
elif [ $run_type = 'prod' ]
then
    echo "Running in production mode"
fi
...
```

After the parameters are set, they are injected into the configuration file `bringup.yaml`, which is mentioned in subsubsection 4.2.2.5. With respect to readability, the code for this is omitted. After the parameters have been injected into `bringup.yaml`, the ROS 2 packages are ready to be built and run. Listings 5.8 and 5.9 show how the Colcon build tool, mentioned in subsubsection 4.2.2.2, is used to build the ROS 2 packages, and how the launch file for each program is executed when everything is set up. The specifics surrounding the building and running a ROS 2 program are described in subsection 4.2.2. It is worth mentioning that,

since Colcon needs to build ROS 2 packages in their respective directories, cron was preferred to `/etc/rc.local` for ROS 2-usage. In short, building and running ROS 2 programs from `/etc/rc.local` requires significantly more bash scripting compared to cron for changing to the correct directories and making sure that build files end up in their desired places.

Listing 5.8: Snippet from build.sh showing the building and running of the Entity ROS 2 program

```
...
colcon build --symlink-install
source install/setup.bash
ros2 launch kmr_communication kmr.
    launch.py

exit 0
```

Listing 5.9: Snippet from build.sh showing the building and running of the Middleware ROS 2 program.

```
...
colcon build --symlink-install
source install/setup.bash
ros2 launch kmr_communication hybrid.
    launch.py

exit 0
```

5.4 Industrial Robot - KMR iiwa

The subsections in this section describes how to create, install and run Sunrise applications which are used by KMR iiwa entities and other KUKA robots. In order to follow the instructions described in this section Sunrise Workbench needs to be installed on a workspace computer.

5.4.1 Creating a Sunrise Project

In order to create a sunrise project, Sunrise Workbench needs to be launched. Inside Sunrise Workbench, perform the following procedure:

"File → New → SunriseProject".

On this page, the IP address of the robot entity can be filled in. Press "Create new project" once filled in. The desired project name needs to be entered, and the appropriate model for the topology template and media flange needs to be chosen on the next page. The creation can be completed once all the desired details are specified by pressing "Create Application".

5.4.2 Installing a Sunrise Application

In order to install a new Sunrise project from the workspace computer to the Sunrise Cabinet on the KMR iiwa, the workspace computer has to be connected to the cabinet, either through an Ethernet or Wi-Fi connection. The Sunrise Project will contain a file called

"StationSetup.cat" that contains the station configuration for the station (controller) selected when the project was created. When double-clicking this file, four tabs will be visible, one of which is named "Installation". After a connection has been established between the workspace computer and the Sunrise Cabinet, clicking the "Install" button in the "Installation" tab will install the system software on the robot.

After installing the software initially, the safety configuration needs to be activated again. The safety maintenance password is required to do this, and it needs to be activated in order to move the robot.

Before a Sunrise application is available on the SmartPAD, the Sunrise project needs to be synchronized. This is done while the workspace computer is connected to the Sunrise Cabinet. The build path for the project needs to be correctly configured in order for the project to be synchronized. The necessary KUKA libraries should all be available in the Sunrise Workbench environment. To synchronise, right-click on "StationSetup.cat" and press "File → Sunrise → Synchronise Project"".

5.4.3 Running a Sunrise application

After synchronization, the application should be available on the SmartPAD under "Applications". The application has the same name as the main Java-class file created in Sunrise Workbench. Select the correct application and press the green play button. If the KMR iiwa is running with either mode T1 or T2, one of the enabling switches on the SmartPAD needs to be held at the "center position", i.e. pressed halfway in.

Chapter 6

Feasibility Experiments

This chapter explains the experiments that were conducted in order to ascertain the feasibility of some of the design choices that were made during the project. They were done with the idea in mind that each design choice should help the system reach a state in which it is able to meet the requirements of Industry 4.0. The chapter is intended for those who might want to build similar systems or to improve the one presented. Each experiment includes a description of scenarios and the results. The results are also discussed here to make it easier for the reader to grasp the importance of each experiment. Firstly, section 6.1 describes an experiment with the 5G HAT. This was done to determine if the 5G HAT could actually receive and transmit data over 5G. In section 6.2, an experiment in which the AAS is used to control the KMR iiwa remotely is outlined. This experiment was done with an early version of the AAS without the camera implementation, and the aim was to determine the system's capability. Section 6.3 describes an experiment conducted with an alternative video streaming solution. Lastly, section 6.4 outlines how the system was tested with regard to autonomy.

6.1 5G HAT Experiment

As described in section 1.5, due to delays from Telenor with regard to the installation of a local 5G node, experiments are performed on the public 5G network provided by Telenor. As detailed in subsection 2.2.2, the 5G HAT can transmit data over sub-5G networks as well as 5G networks. Because of this, an experiment is conducted in order to determine what type of connection is established and which cellular node is being used.

6.1.1 Scenarios

There are three scenarios defined for this experiment, each with an individual objective.

Scenario 1: Determine Specifications of Cellular Node in Use
The AT-command "AT+CPSI?" is used on the 5G Raspberry Pi in order to acquire the UE system information from the SIM8200-M2 5G HAT. This information is then used in combination with "Cellmapper.net" to obtain information about the cellular node that the HAT is connected to.

Scenario 2: Determine Connection Type
The 5G HAT has an LED that indicates which type of network connection it is registering. The time interval between each LED blink is measured, and, based on the information given in subsection 2.2.2, the connection type in use is determined.

Scenario 3: Bandwidth Speedtest
The terminal command "speedtest-cli" is used at two key locations, which are seen in Figure 6.1, in order to determine if there are any differences with regard to bandwidth at these locations.

Sub-scenario 3.1 - speedtest-cli used at Office (Verkstedteknisk 5th floor at NTNU Gløshaugen)

Sub-scenario 3.2 - speedtest-cli used at Lab (Verkstedteknisk 1st floor at NTNU Gløshaugen)

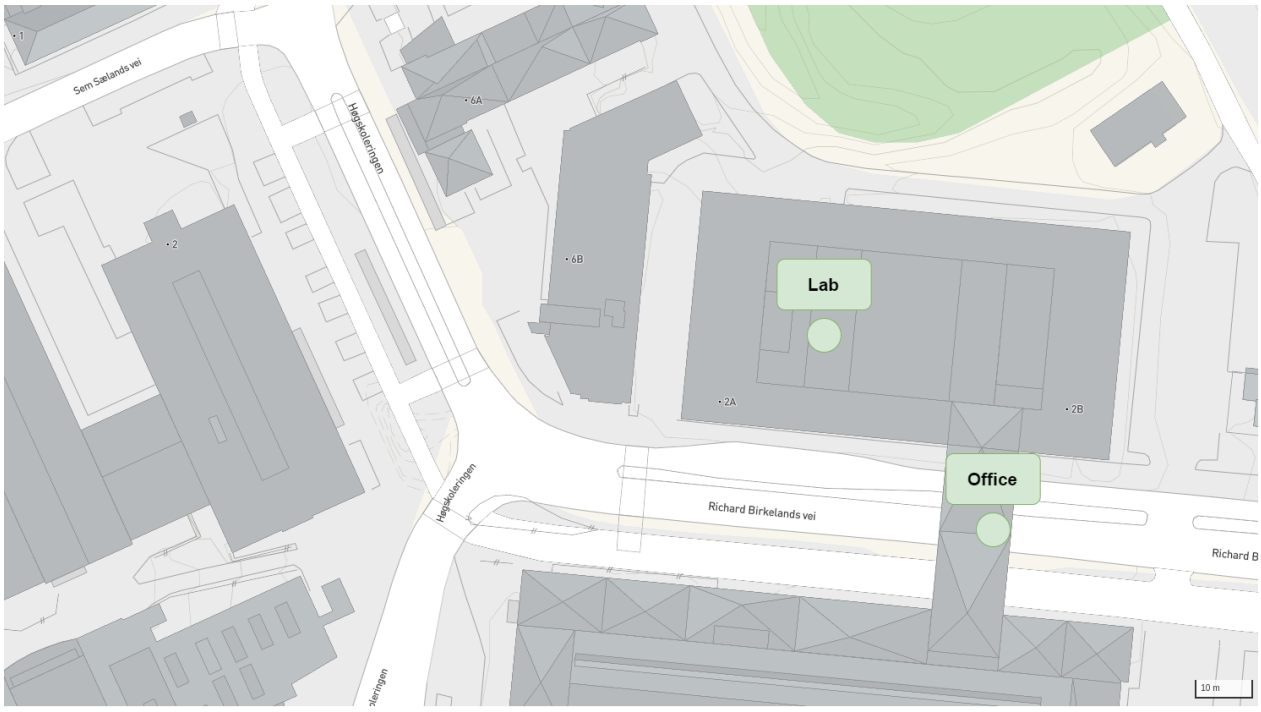


Figure 6.1: Location of Lab and Office at NTNU Gløshaugen [77].

6.1.2 Results

6.1.2.1 Scenario 1

The results after performing the "AT+CPSI?" command on the 5G Raspberry Pi are displayed in Figure 6.2. According to the "At Command Manual", this output means that the 5G HAT's modem is "camping on EN-DC connected mode". EN-DC stands for E-UTRAN New Radio – Dual Connectivity, which in short means that the modem is able to connect to both a 4G LTE master node and a 5G NR secondary node. This is possible because communication on both nodes goes through a 4G core network [78]. This makes sense, as Telenor's public 5G network is non-standalone, i.e. it depends on the existing 4G LTE core network.

```
AT+CPSI?
+CPSI: LTE,Online,242-01,0x76C1,51275781,214,EUTRAN-BAND7,3050,5,5,-89,-902,-611
+CPSI: NR5G,0,0,-14,-98,1.0
```

Figure 6.2: Results after performing the "AT+CPSI"-command.

The description of the "AT+CPSI?" command in the "At Command Manual" explains where in the output to find the "Cell Identifier". In Figure 6.2 this was found to be "51275781". By searching after the cell identifier on "Cellmapper.net" the cellular node was found. The result of this search is displayed in Figure 6.3. The blue dot on the map is the cellular node in use, and the specifications of the cell are on the leftmost side of the figure. The darkened marked area on the map outlines the coverage of the cellular node. This node is the one closest to the site that was used for testing the system, which meant that other experiments could be conducted with the best possible latency under the given circumstances.

6.1.2.2 Scenario 2

The LED-blinking on the 5G HAT was recorded at 60 frames per second using an iPhone. By looking at the resulting video frame by frame, it is apparent that there are exactly six frames between each blink. Equation 6.1 shows that this means that there is 100ms between each blink, proving that the 5G HAT is registering a 5G network according to the information provided in subsection 2.2.2.

$$\frac{60\,frames}{1000\,ms} = \frac{6\,frames}{t} \implies t = 6\,frames \cdot \frac{1000\,ms}{60\,frames} = 100\,ms \tag{6.1}$$

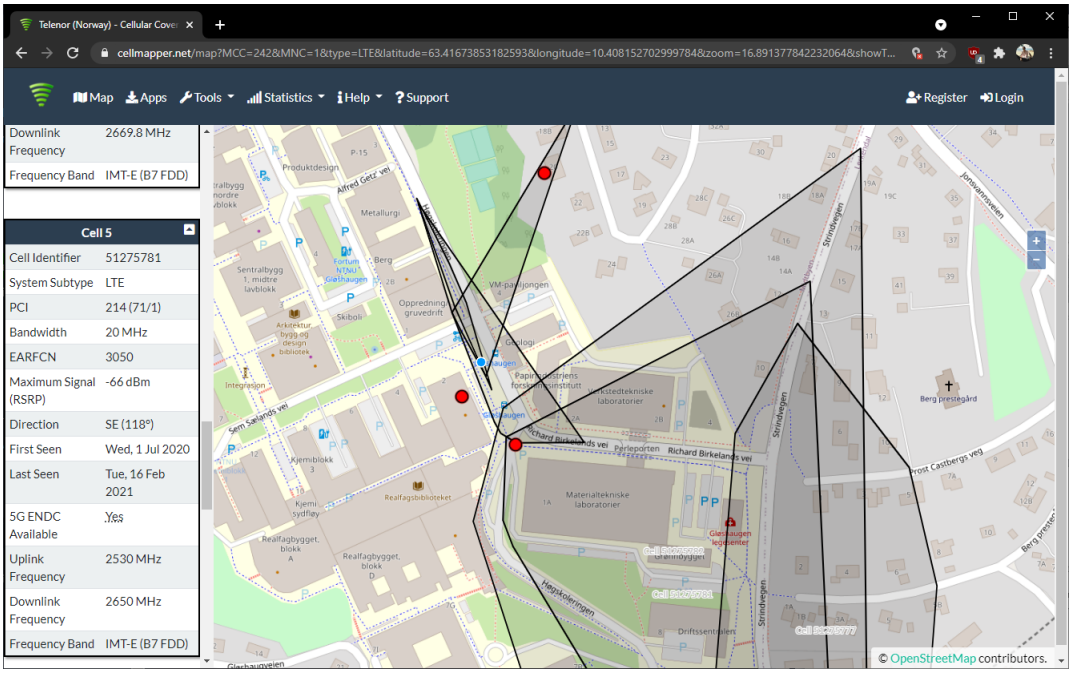


Figure 6.3: Snippet from "cellmapper.net".

6.1.2.3 Scenario 3

Combining the results from *Scenario 1* and Figure 6.1 yields Figure 6.4. This figure displays the points of interest mapped against the cellular node in use. The results of a run of each of the two sub-scenarios are shown in Figures 6.5 and 6.6.

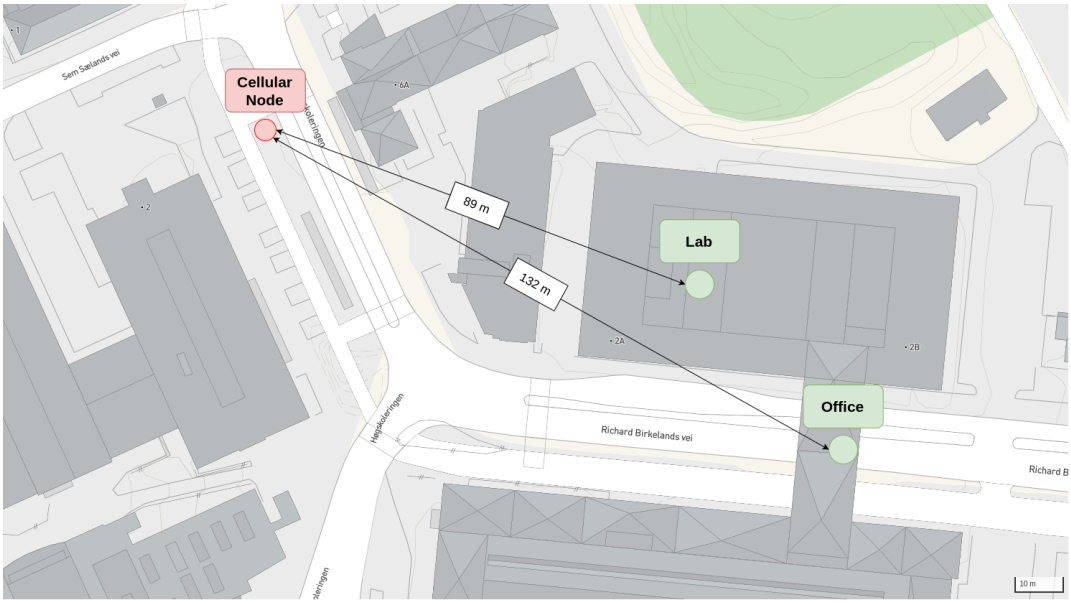


Figure 6.4: Points of interest mapped against the cellular node in use. The Lab is 89 meters from cellular node, while the Office is located 132 meters from the cellular node.

```
pi@raspberrypi-5G:~ $ speedtest-cli
Retrieving speedtest.net configuration...
Testing from Telenor Norge (77.18.56.158)...
Retrieving speedtest.net server list...
Selecting best server based on ping..
Hosted by Telenor Norge AS (Oslo) [5.75 km]: 45.297 ms
Testing download speed................................................................
.....................
Download: 69.71 Mbit/s
Testing upload speed..................................................................
.....................
Upload: 18.70 Mbit/s
```

Figure 6.5: Results of a running of *Sub-scenario 3.1* - speedtest-cli used at Office (Verkstedteknisk 5th floor at NTNU Gløshaugen).

```
pi@raspberrypi-5G:~ $ speedtest-cli
Retrieving speedtest.net configuration...
Testing from Telenor Norge (77.18.56.158)...
Retrieving speedtest.net server list...
Selecting best server based on ping..
Hosted by Telenor Norge AS (Oslo) [5.75 km]: 33.626 ms
Testing download speed................................................................
.....................
Download: 250.92 Mbit/s
Testing upload speed..................................................................
.....................
Upload: 60.24 Mbit/s
```

Figure 6.6: Results of a running of *Sub-scenario 3.2* - speedtest-cli used at Lab (Verkstedteknisk 1st floor at NTNU Gløshaugen).

The latency at the lab was measured to be consistently lower, and the overall bandwidth was significantly higher compared to at the office. Several elements can affect these results. It can be factors such as distance, number of obstacles blocking the signal, or number of devices connected to the cellular node. Each sub-scenario was tested multiple times, but the overall trends of the results were the same, with the results of the two scenarios being consistently different. The signal strength of the cellular node is believed to be what causes these differences. As seen in Figure 6.4, the lab is located closer to the cellular node compared to the office, and the number of obstacles needed to penetrate also seems to be lower. Telenor's public 5G network operates on a 3.6 GHz frequency band, which is high enough to affect the signal's ability with regard to object penetration and overall area coverage. These factors

are sufficient in causing the differences seen in the results from the two sub-scenarios.

The three scenarios and their results helped show that the 5G HAT was capable of communicating with the desired cellular node and that this communication could be done over both 4G and 5G. It is worth noting that throughout the testing with the 5G HAT, the blinking of the network indicating LED has been observed slower than 100ms intervals. With respect to the results found in this experiment, this slow blinking is most likely with 200ms intervals, meaning that the modem is connected to the 4G LTE master node during this time.

6.2 Remote Control with 5G

In order to determine if the system is capable of controlling the KMR iiwa remotely using 5G, a simple test experiment is set up. The results of the experiment are documented in a short video and sent to Telenor, a tele-company taking part in the 5G-SOLUTIONS project.

6.2.1 Configuration

The configuration for the experiment includes the following:

- A Raspberry Pi hosting the AAS and connected with Ethernet to the NTNU network. The AAS includes a Flask application with an OPC UA server and a React application.
- A Raspberry Pi with a 5G HAT connected to the KMR iiwa over Ethernet. It transmits data to the AAS using Telenor's public 5G network. On the Raspberry Pi are two ROS 2 programs implemented with the publish/subscribe communication model.

6.2.2 Scenarios

A single scenario is defined.

Scenario: Control the robot using an external video call program
This scenario requires two operators: one in the lab and one in the control room. The operator in the control room starts the AAS. The operator in the lab starts the ROS 2 Entity program and the ROS 2 Middleware program on the 5G Raspberry Pi, allowing communication between the KMR iiwa and the AAS. The robot application on the KMR iiwa is started, and a connection is established. The two operators start a live video call, and the operator in the control room attempts to control the KMR iiwa while observing it through the video stream.

6.2.3 Results

A screenshot from the resulting video is displayed in Figure 6.7. Since the feasibility of the system could be determined based on whether or not the operator in the control room was able to control the KMR iiwa remotely, no additional data was required.

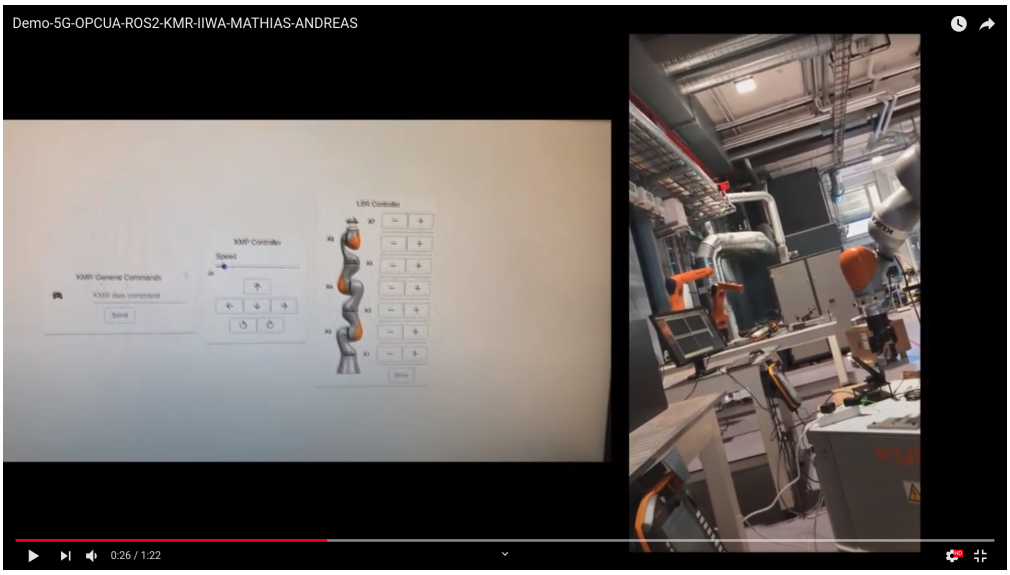


Figure 6.7: Snippet from the resulting video [79].

By conducting this simple experiment, it was proven that the system had potential. As displayed in the resulting video, the experiment was successful [79], and remote control of the KMR iiwa was achieved without any major difficulties. There was some input delay seen through the video call, but it is likely that the primary contributor to this delay was the video-call program itself, namely Facebook Messenger. The 5G Raspberry Pi was connected to Telenor's public 5G network through the 5G HAT, while the AAS Raspberry Pi was connected to the NTNU network through Ethernet. Since data transmission between these two devices involves long distances and multiple routing points, it is assumed that this setup is a source of significant latency. A more detailed experiment of the system latency is found in section 7.1. The ROS 2 programs on the 5G Raspberry Pi worked as expected, and the ROS nodes were able to connect to and communicate with the robot application. With an ideal setup, the ROS 2 Entity program would be running on the entity itself, but this is not possible due to the nature of the KMR iiwa. However, since the 5G Raspberry Pi and the KMR iiwa are connected using Ethernet, the delay between the TCP server on the ROS 2 Entity program and the TCP client on the entity is assumed low enough that the ROS nodes can act as the entity's components.

6.3 AAS Video Stream Through Robot Command Pipeline

In order to determine if the pipeline used for handling robot commands is also capable of handling the AAS Video Stream, a simple prototype implementation of a video streaming service is created.

6.3.1 Configuration

A Raspberry Pi Camera Module v2 is attached to the Entity Raspberry Pi and used to capture video. Since the system pipeline is comprised of several components that transmit data differently, the image feed needs to be converted. This needs to be done several times and to several different formats as it passes through the pipeline, see Figure 6.8. The programs running on both the Entity Raspberry Pi and the 5G Raspberry Pi are implemented using Python, so the Python library OpenCV is used to handle real-time image processing. New video-related ROS 2 topics and OPC UA events are added to the system in order to support the image feed being sent through the pipeline. The modifiable nature of ROS 2 and OPC UA makes this addition easy to implement. A WebSocket solution is added to the OPC UA server implementation on the Flask application in order to handle sending the image feed to the frontend, i.e. the External Interface. The External Interface includes a function for sending a command through the pipeline asking the Entity Raspberry Pi to start the camera.

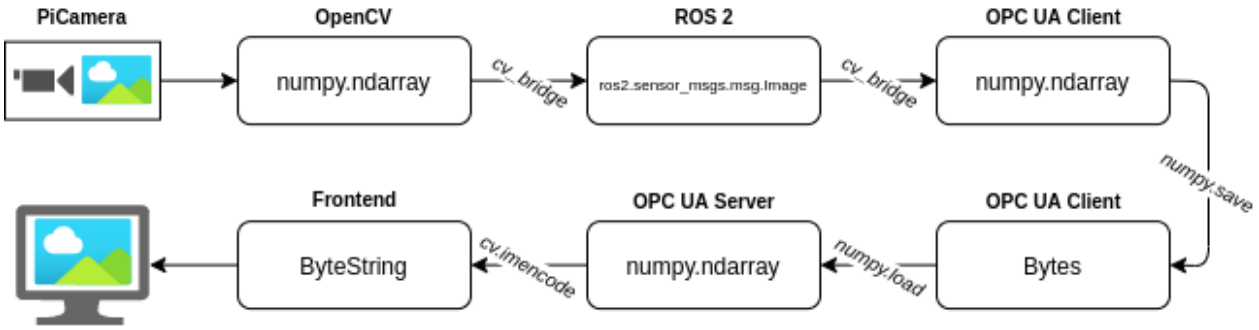


Figure 6.8: Illustration of the AAS Video Stream pipeline when the video stream is sent through the pipeline designed for robot commands.

6.3.2 Scenarios

A single scenario is defined.

Scenario: Prototype implementation
The AAS is started, and an operator uses the External Interface to view the specific entity that they wish to see video from. The operator clicks a button to start a video stream, and

the Raspberry Pi Camera Module v2 starts to capture video. The image feed is sent through each component and converted several times as shown in Figure 6.8. The video is ultimately displayed to the operator. The video quality is evaluated based on the lag that is experienced by the viewing operator.

6.3.3 Results

Sending the video stream through a pipeline in which many of the components require different data types proved to be undesirable. Each conversion meant that the system had to sacrifice both quality and latency, and the total end result was not satisfactory. The high latency of the video stream was apparent, and there was a significant delay in sending of robot commands. Additionally, since the pipeline includes OPC UA, which is not designed to transport streams of data, it best serves single data transmissions, such as commands. This solution was, as a result, discarded since real-time operation and low latency is of utmost importance with regard to the use cases described in section 1.3.

6.4 Rapid Deployment and Auto-Configuration Experiment

As mentioned in subsection 3.1.2, one important aspect of the equipment in the "Industrial Lab" is that it should be autonomous, i.e. it should typically not require manual operation. A part of the implementation that aims to help reach the goal of autonomy is the auto-run feature described in subsection 5.3.7. In order to determine if the system is made more autonomous with the auto-run feature, an experiment is conducted in which a connection between the AAS and an entity is established with the least possible amount of human interaction possible. Because of the safety configuration on the KMR iiwa, the entity is not operable in AUT (autonomous) mode. This means that an operator has to manually start the robot application with the SmartPAD, as described in subsection 5.4.3.

6.4.1 Scenarios

A single scenario is defined.

Scenario: Automatic configuration with KMR iiwa
One operator is stationed in the control room, while the other is in the lab. In the control room, the AAS is running on the AAS Raspberry Pi, and the operator has a device with the External Interface open. This operator is viewing the functionality of the entity that is to be connected. The functionality is disabled in the interface because the entity is offline. In the lab, the 5G Raspberry Pi is placed on the KMR iiwa and powered using a portable charger. When the ROS 2 programs have finished initialization, the operator can start the application

on the KMR iiwa, and the entity status should be changed to online. This change should be reflected on the External Interface, and the operator in the control room should be able to interact with the entity.

6.4.2 Results

Soon after the 5G Raspberry Pi was connected to the portable charger, the LED on the 5G HAT started blinking, indicating that a network was registered. This meant that the script for automatically establishing a connection with Telenor's public 5G network was successful. After this, it took approximately 2.5 minutes before the robot application on the KMR iiwa was able to connect to the ROS 2 program. It was known from previous work with building and launching the ROS 2 programs on the 5G Raspberry Pi in non-headless mode, i.e. with the ability to see the output on a monitor, that this was the time it took for the programs to finish the initialization. When the robot application connected to the ROS 2 Entity program, the functionality on the External Interface was enabled in a matter of milliseconds, and the operator in the control room was able to control the entity. This connection lasted until the operator in the lab released the enabling switch on the SmartPAD, canceling the robot application.

The experiment could have been more compelling had the KMR iiwa been run in AUT mode. That way, the 5G Raspberry Pi could have been powered, and after 2.5 minutes, the KMR iiwa could be turned on without the SmartPAD connected. The robot application on the KMR iiwa would then have started, and a connection with the ROS 2 Entity Program would have been established automatically. This would make it so that the operator could potentially leave the area with the entity still being operable from the control room. It is worth noting that since the experiment worked with the SmartPAD, it should, in theory, work in AUT mode as well.

The success of the experiment did prove that the implemented system is able to support automatic configuration to an extent. If a new KMR iiwa were to be introduced to the lab, another 5G Raspberry Pi could be attached to it, and it would be integrated into the AAS with little friction. This would only require the IP address of the entity to be known and for the correct Sunrise application to be installed on it. Both of these limiting factors are viewed as consequences of the robot manufacturer's decisions rather than issues with the system presented in this thesis. With a standardized, exposed API on the side of the KUKA KMR iiwa, it is not impossible that the IP address of the entity could be fetched by the ROS 2 program. Also, if KUKA allowed other programs than Sunrise applications, such as ROS 2, to be run on their hardware, the initial installation step (see subsection 5.4.2) could be skipped altogether. The idea of standardized APIs could potentially help the system support

various types of robot entities in that the 5G Raspberry Pi could be made to detect the functionality of the robots when it is attached. Even though these ideas were not explored or tested in detail during this project, the AAS paves the way for exciting possibilities in the realm of industrial automation.

Chapter 7

Performance Experiments

This chapter explains the two performance experiments conducted to ascertain whether or not the system presented in this thesis is able to meet the requirements of Industry 4.0. The chapter is intended for those who might want to build similar systems or improve the one presented and those who are interested in knowing what kinds of tests they can use for comparison. The setup used to conduct these experiment is described in chapter 5. Section 7.1 describes an experiment for determining the latency in the system, while section 7.2 compares the performance of a standard kernel to that of a kernel with real-time capabilities. Like the feasibility experiments, the experiments described here will include the results and discussions with respect to readability.

7.1 Latency Experiment

In order to measure the delay between the different components in the system, the web-based active monitoring, and testing platform Hawkeye is used. The routes that are measured are those with a "direct" connection in the architecture described in chapter 3. This includes the route between the 5G Raspberry Pi and the AAS Raspberry Pi, and the route between the AAS Raspberry Pi and a "User Device". In the tests, this device is a User Laptop named "andreas". An outline of the test setup is shown in Figure 7.1.

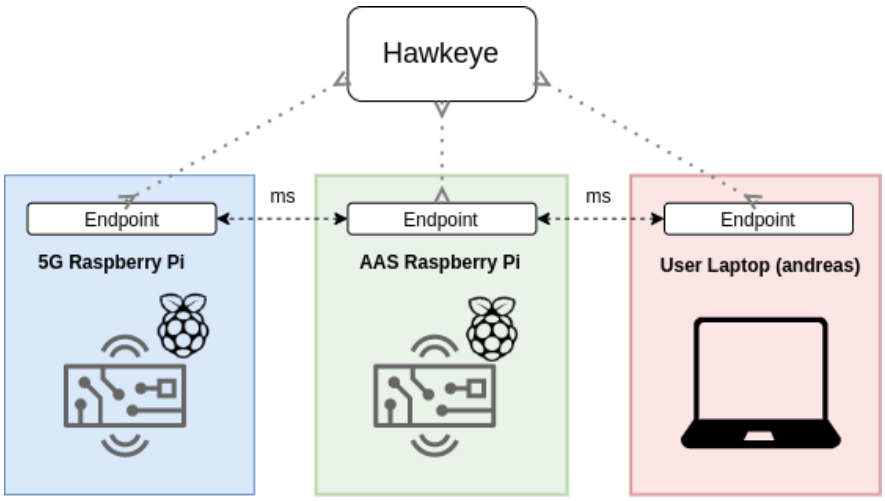


Figure 7.1: Outline of Node-to-Node test setup.

7.1.1 Hawkeye configuration

Hawkeye endpoints are installed on the three components shown in Figure 7.1. The endpoints are provided by ixia and can be found on [80]. A mesh, with the topology described in the previous paragraph, is created as displayed in Figure 7.2. A mesh can be created once Hawkeye endpoints are installed on all of the components.

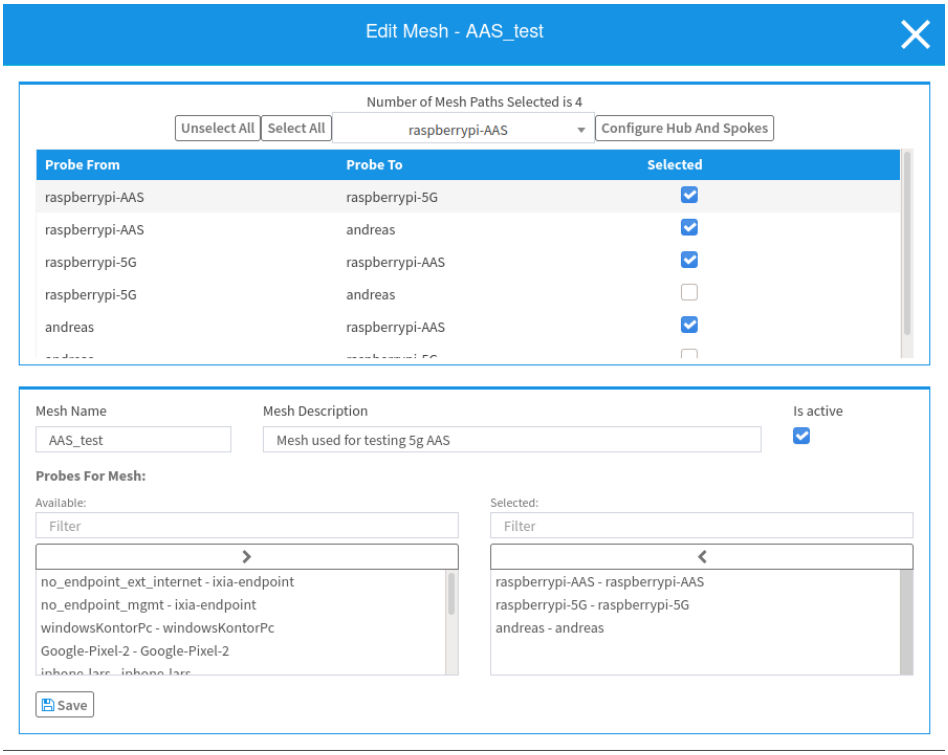


Figure 7.2: Creating a mesh in Hawkeye consisting of the relevant endpoints and routes.

Tests are then created on the mesh. An example of how to create a test that is going to run for an hour is displayed in Figure 7.3. Once the tests are complete, the results can be downloaded from Hawkeye as CSV (Comma Separated Value) files.

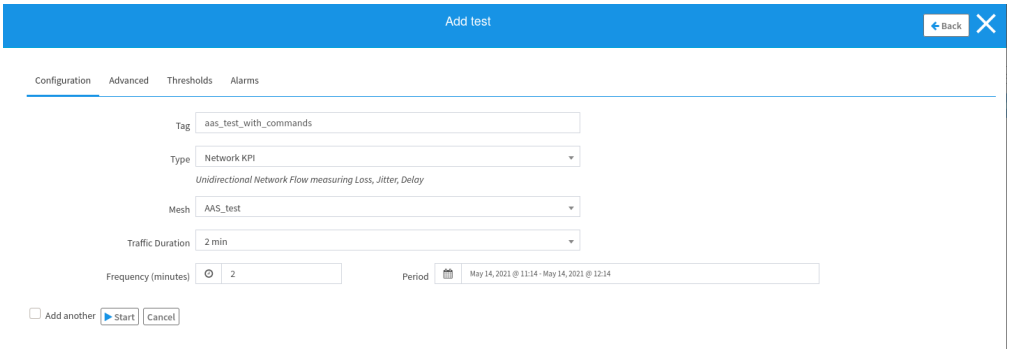


Figure 7.3: Creating a test in Hawkeye with the mesh created in Figure 7.2.

7.1.2 Tests

There are two tests created for this experiment. Each test is meant to be run for one hour. The first test simulates the AAS under load. During this time, entity commands are sent through the pipeline consistently for an hour. Every second, a command is sent from the External Interface. This is done on a rotational basis, and the commands are either for moving the KMP omniMove or the LBR iiwa. Every 50 seconds, a camera command is sent to either start or stop the camera. A script is created to automate these actions. The second test simulates the AAS while it is idling. During this time, no commands or messages are sent through the pipeline.

7.1.3 Scenarios

Three scenarios are defined, each with two sub-scenarios that corresponds with the two tests described above. The tests that are run for each of the three scenarios are the same, but the environment is different for each one. The objective is then to determine if there are any differences between the results from the scenarios, and if so, why these differences occur. Figure 7.4 displays a map with the related locations for the experiment. The AAS Raspberry Pi and User Laptop are located at the office and connected to the NTNU network in each scenario.

Scenario 1: 5G Raspberry Pi Located at the Office
The 5G Raspberry Pi is located at the office (Verkstedteknisk 5th floor at Gløshaugen), and it is connected to Telenor's public 5G network. The Hawkeye tests are started, and at the same time, the script for sending commands from the External Interface is run. After one

hour, the commands cease, and the system is idle for another hour. The tests are completed after two hours, at which point the Hawkeye results are downloaded and split into *scenario 1.1* for the first hour and *scenario 1.2* for the second hour.

Scenario 2: 5G Raspberry Pi Located at NTNU Gløshaugen's Industry 4.0 laboratory
The 5G Raspberry Pi is located at NTNU Gløshaugen's Industry 4.0 laboratory, and it is connected to Telenor's public 5G network. The Hawkeye tests are started, and at the same time, the script for sending commands from the External Interface is run. After one hour, the commands cease, and the system is idle for another hour. The tests are completed after two hours, at which point the Hawkeye results are downloaded and split into *scenario 2.1* for the first hour and *scenario 2.2* for the second hour.

Scenario 3: 5G Raspberry Pi Connected to Wi-Fi and Located at the Office
The 5G Raspberry Pi is located at the office, and it is connected to the NTNU network via a Wi-Fi router that is placed approximately 1 meter away. The Hawkeye tests are started, and at the same time, the script for sending commands from the External Interface is run. After one hour, the commands cease, and the system is idle for another hour. The tests are completed after two hours, at which point the Hawkeye results are downloaded and split into *scenario 3.1* for the first hour and *scenario 3.2* for the second hour.

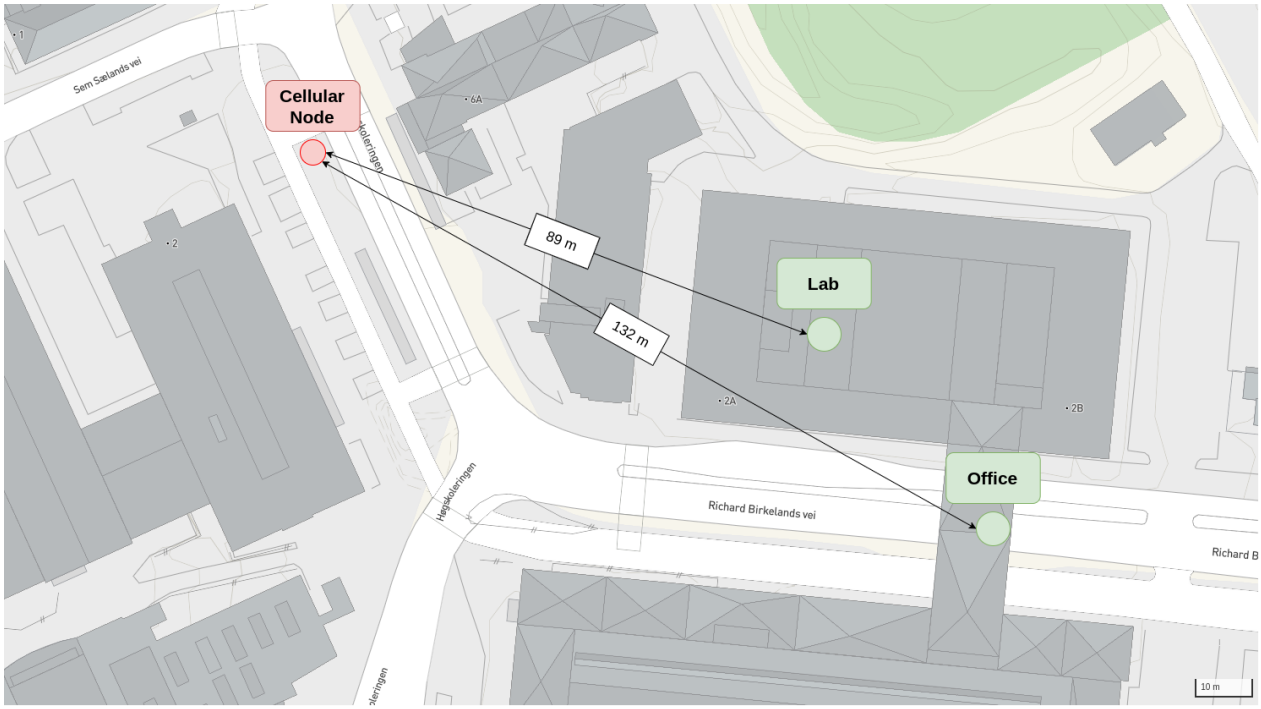


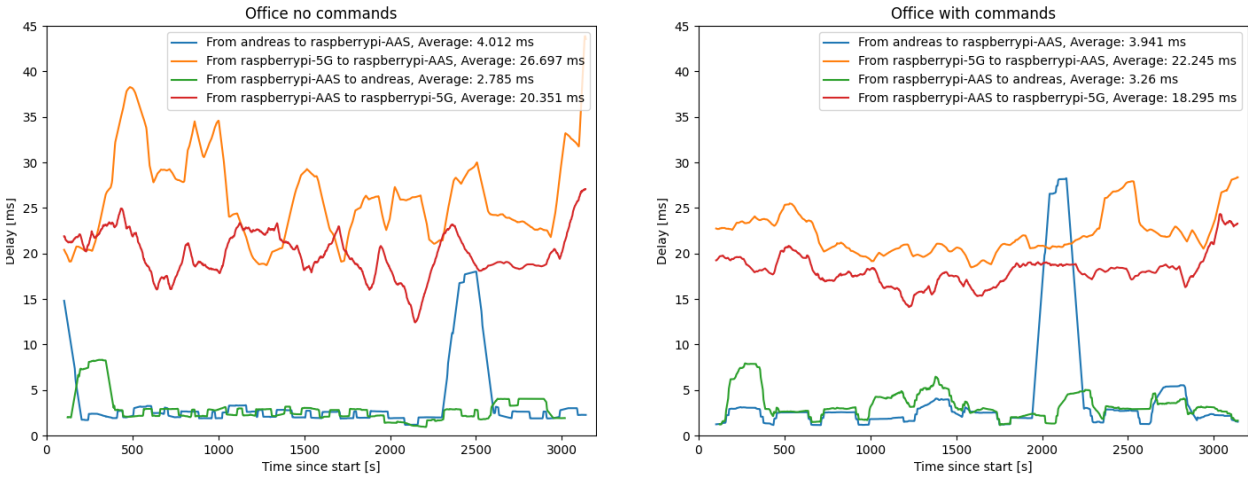
Figure 7.4: Locations for the Latency experiment [77].

7.1.4 Results

The results from the three test scenarios *1*, *2*, and *3*, each with two sub-scenarios, are displayed in Figures 7.5, 7.6 and 7.7 respectively. An interesting remark is that sending commands does not seem to affect the system's latency as there is no significant difference between the individual sub-scenarios within each scenario. These results might imply that commands do not bottleneck the overall system and indicates that the system has high capability of throughput with regard to sending commands. Another remark is that the total delay in every scenario, calculated by adding the average values, exceeds the maximum accepted latency of Use Case 1.3, which is 10 ms. See Table 7.1.

Scenario	***Total Average Delay [ms]***
Scenario 1.1	53.845
Scenario 1.2	47.741
Scenario 2.1	47.107
Scenario 2.2	48.461
Scenario 3.1	24.49
Scenario 3.2	20.407

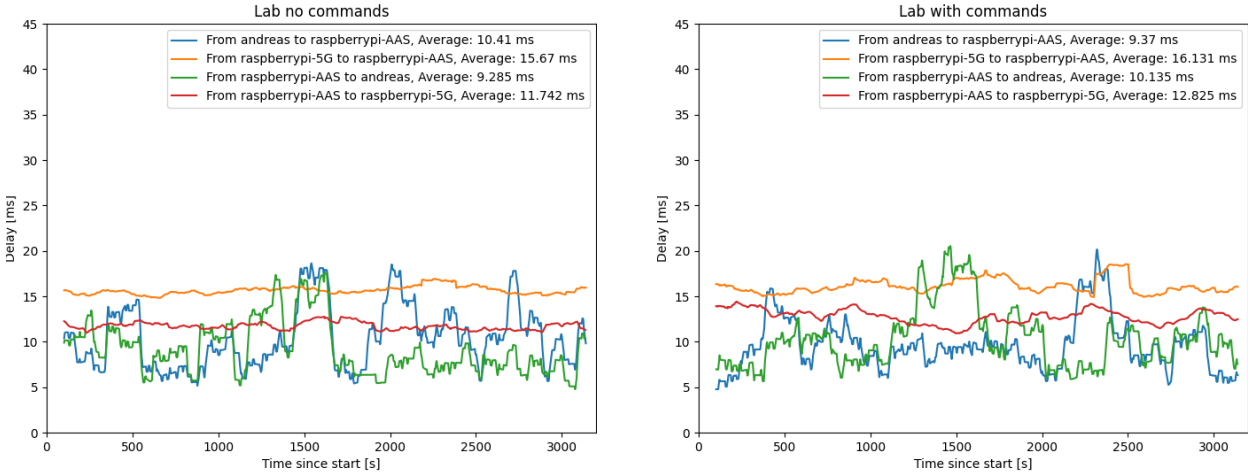
Table 7.1: Total Average Delay of all the scenarios. Calculated by adding the average values of each route.



(a) Without commands

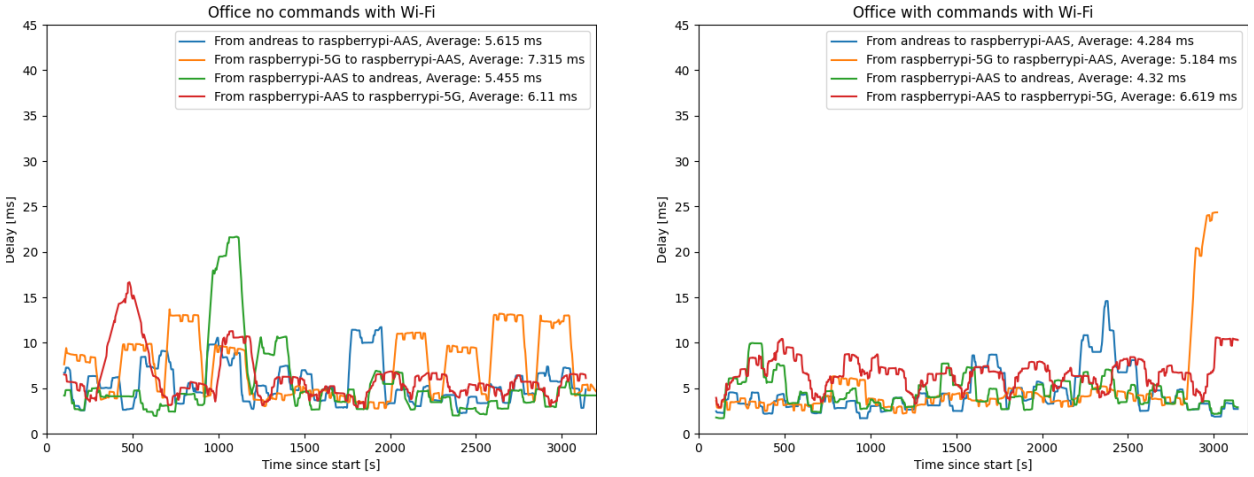
(b) With commands

Figure 7.5: *Scenario 1.1* and *1.2*. Test performed at Office.



(a) Without commands (b) With commands

Figure 7.6: *Scenario 2.1* and *2.2*. Test performed at Lab.



(a) Without commands (b) With commands

Figure 7.7: *Scenario 3.1* and *3.2*. Test performed at Office with Wi-Fi.

It can be difficult to judge the results by looking at them individually. The Figures 7.8, 7.9, 7.10 and 7.11 displays comparisons of all the scenarios based on connection routes within the mesh.

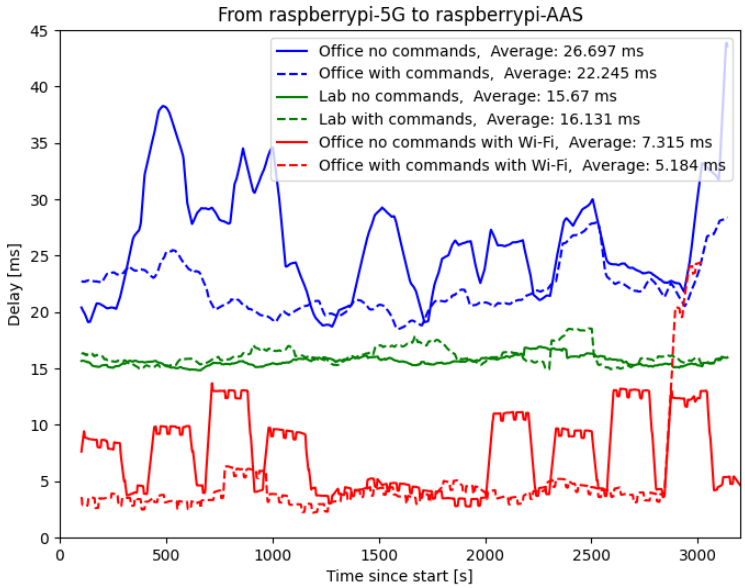


Figure 7.8: Results of all six sub-scenarios on route "raspberrypi-5G to raspberrypi-AAS"

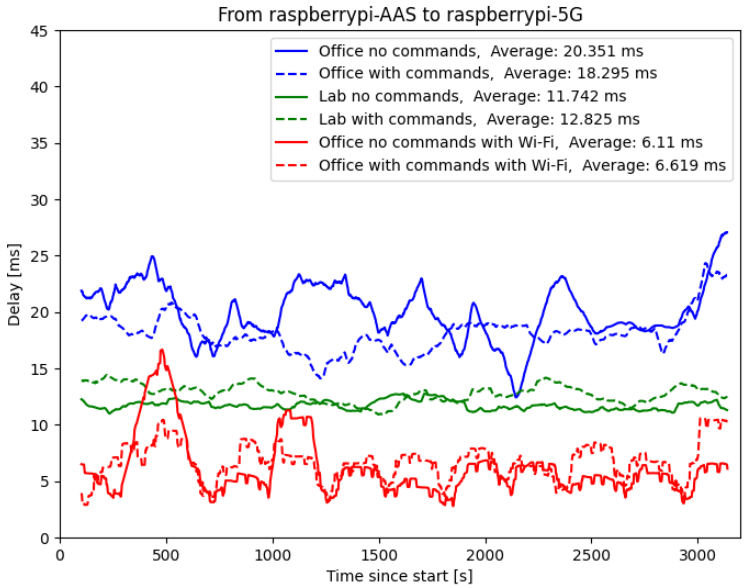


Figure 7.9: Results of all six sub-scenarios on route "raspberrypi-AAS to raspberrypi-5G"

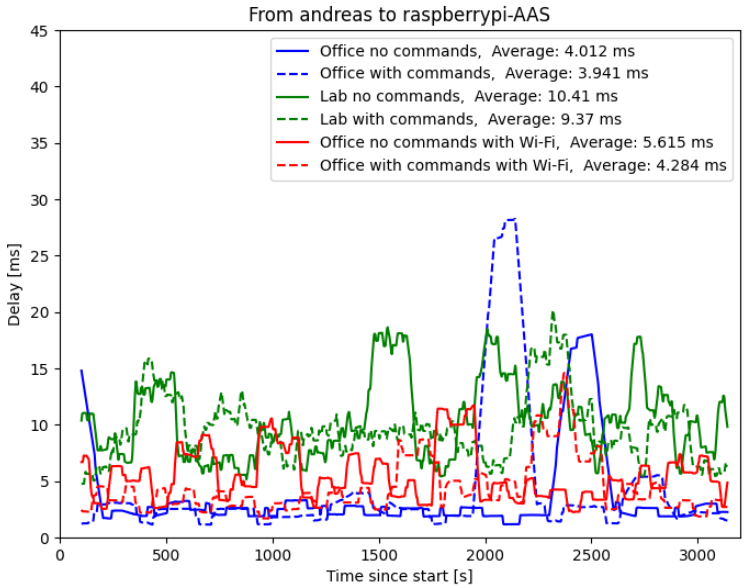


Figure 7.10: Results of all six sub-scenarios on route "andreas to raspberrypi-AAS"

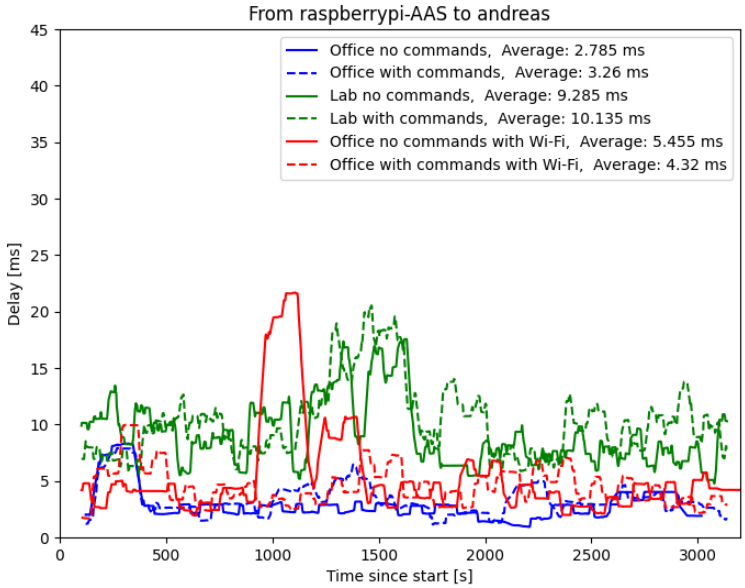
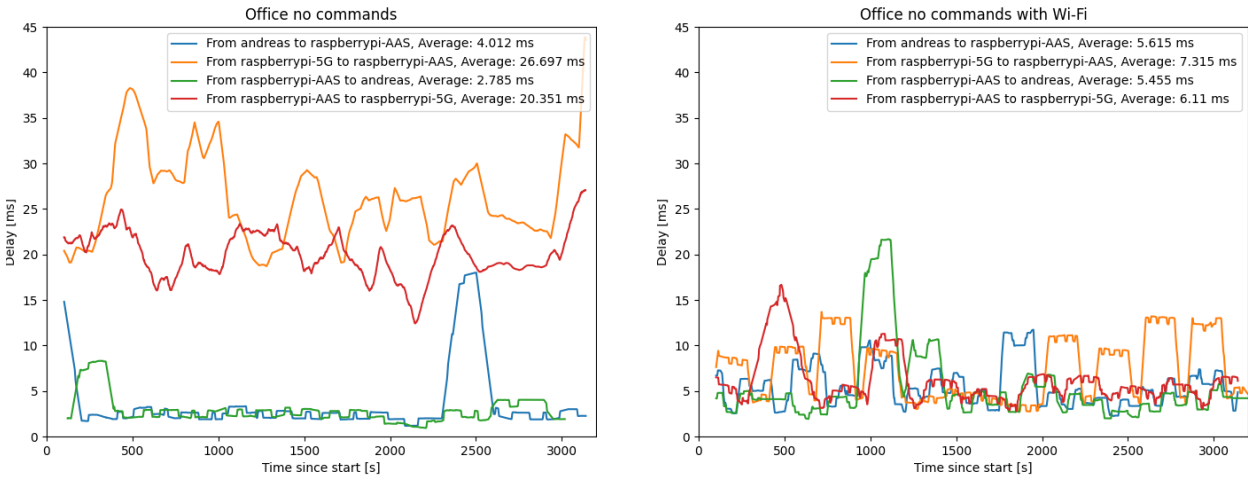


Figure 7.11: Results of all six sub-scenarios on route "raspberrypi-AAS to andreas"

From studying Figure 7.8 and Figure 7.9, some interesting observations can be made. There seems to be a correlation between the scenarios and the different end results. *Scenario 3 (Wi-Fi at Office)* has a lower latency than *Scenario 2 (5G at Lab)*, which has a lower latency

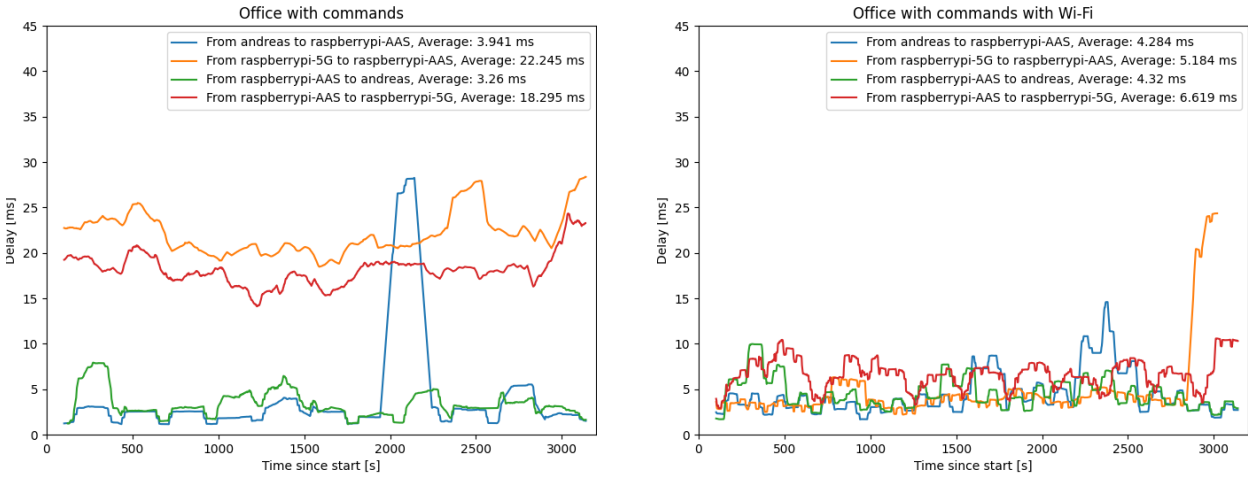
compared to *Scenario 1 (5G at Office)*. The difference between *Scenario 2* and *Scenario 1* can be derived from various factors. As seen in Figure 7.4, there are certain geographical differences in the two scenarios. Distance to the cellular node and the amount of obstacles that need to be penetrated can affect the end results. The number of user devices connected to the cellular node and the general network load are also important aspects to consider. The difference between *Scenario 3* and *Scenario 2* is detailed in the last paragraph of this section, while an overall discussion of the complete results is found in chapter 8.

There are slight differences in the test results of the non-5G routes shown in Figure 7.10 and Figure 7.11. From the setup of the experiment, these results were expected to be more similar, as the variables of these routes were not altered between the scenarios. There is a high probability that these variances come as a result of the tests being performed on the campus network on NTNU Gløshaugen on different days. Many people utilize this network, and it is difficult to predict the load on the network.



(a) 5G (b) Wi-Fi

Figure 7.12: Direct comparison of Figure 7.5a and Figure 7.7a.



(a) 5G (b) Wi-Fi

Figure 7.13: Direct comparison of Figure 7.5b and Figure 7.7b.

Direct comparisons of the 5G and Wi-Fi results are displayed in Figure 7.12 and Figure 7.13. An immediate takeaway from these results is that, on the routes compatible with 5G, the solution with a Wi-Fi connection seems to have a significantly lower overall latency compared to the solution with a 5G connection, although neither solution satisfies the Use Case 1.3 goal of a total delay of 10 ms or lower. This data is interesting to examine in relation to Use Case 1.5, which focuses on testing if 5G can replace traditional networking technology like Wi-Fi. Even though it may seem like Wi-Fi outperforms 5G when looking at the results, there are several factors to take into consideration when comparing the two directly. Firstly, there are more links of data transmission using this 5G solution. The first is from the 5G Raspberry Pi to the cellular node. From the cellular node, the data is routed to Telenor's core network, where the data is made available on the Internet. The data is then picked up by the NTNU network, which routes it to the AAS Raspberry Pi in the office via Ethernet. With the Wi-Fi solution, the data is transmitted from the 5G Raspberry Pi directly to the Wi-Fi router in the same room, which routes the data through the NTNU network via Ethernet and directly to the AAS Raspberry Pi in the office. It therefore comes as no surprise that, even with all its capabilities, 5G is put at a disadvantage with that many extra links. A more fair comparison would have been a 5G solution in which a private and local 5G network was introduced. This way, the AAS Raspberry Pi in the office, or "Control Room", could be connected with Ethernet to the local 5G node, while the 5G Raspberry Pi could communicate with this same node through 5G from the "Industrial Lab". This would result in only one wireless link, which is similar to the Wi-Fi solution.

7.2 Cyclictest: Standard Kernel vs. Kernel w/ PREEMPT_RT

Cyclictest is a high-resolution test program that benchmarks event latency in a Linux kernel by measuring the amount of time that passes between when a timer expires and when the actual thread that starts the timer runs [81]. In other words, cyclictest measures the amount of time the system uses to respond to an interrupt. The average latency will increase if there are processes with non-deterministic blocking behavior running in the system. This happens since the scheduler cannot meet the deadlines of the interrupts profiled in the program [13]. The interrupts are in cyclictest generated by a timer.

7.2.1 Cyclictest Configuration

The test scenarios, described in subsection 7.2.2, are performed using the following command:

```
$ cyclictest -l100000000 -m -Sp90 -i200 -h400 -q
```

The command is run with six arguments, and these are explained in the ubuntu manuals as follows [82]:

- **"l"** sets the number of loops. The cyclictest is stopped once the number of loops is reached.
- **"m"** locks the current and future memory allocations to prevent the test from being paged out.
- **"Sp"** sets options for standard testing on SMP systems and describes the priority of the threads. The given priority is set to the first test thread, in this case "90", and each following thread gets a lower priority according to the function $Priority(Thread_N) = max(Priority(Thread_{N-1}) - 1, 0)$.
- **"i"** sets the base interval of the threads in microseconds.
- **"h"** generates a latency histogram where the input, in this case "400", is the max latency to be tracked in microseconds.
- **"q"** forces prints only on exit.

7.2.2 Scenarios

Two scenarios are defined. The objective is to benchmark the performance of the kernel with PREEMPT_RT, and compare it with a standard (non-real-time) kernel.

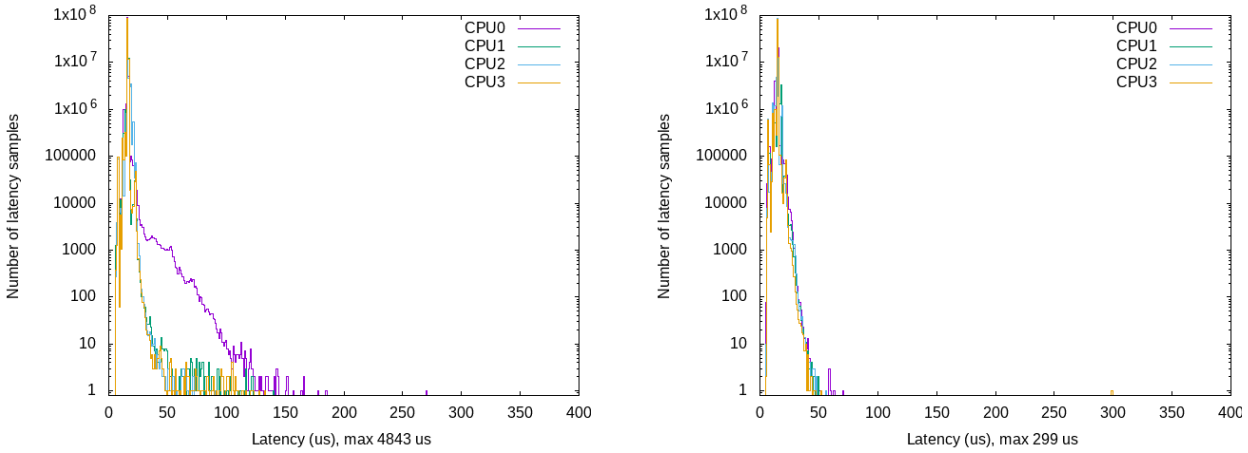
Scenario 1: Standard Kernel
To test the performance of the standard kernel, a cyclictest is performed on the 5G Raspberry

Pi before being patched with PREEMPT_RT. The kernel being used is Linux kernel 5.4.42-v8+.

Scenario 2: PREEMPT_RT Kernel
To test the performance of the kernel with PREEMPT_RT, a cyclictest is performed on the 5G Raspberry Pi after being patched with PREEMPT_RT. Detailed instructions regarding the PREEMPT_RT patch are described in subsection 5.3.1. The patched kernel being used is Linux kernel 5.4.83-rt50-v8+.

7.2.3 Results

Figure 7.14 displays the results of the two scenarios. The results from this experiment were as expected. Both of the tests were completed in approximately five hours and produced two distinctly different outputs. One key difference is the maximum latency. While the patched kernel only had a maximum latency of 299 µs, the standard kernel reached the heights of 4843 µs. This difference can make the standard kernel not able to reach certain deadlines. The overall area found under the graph (integral) is substantially larger when comparing the standard kernel to the patched kernel, indicating a consistent higher amount of latency.



(a) Standard Kernel

(b) Kernel patched with PREEMPT_RT

Figure 7.14: Cyclictest performed on the two different kernels.

Due to the low latency experience with the PREEMPT_RT kernel, there is a strong argument to be made for why it should be incorporated in the system presented in this thesis. Since there are stringent latency requirements regarding real-time operation in Industry 4.0, a system that aims to meet these requirements needs to optimize performance in as many areas as possible. Since robotic systems need to be responsive, the PREEMPT_RT kernel is a good fit for the machines running ROS 2, namely the Entity Raspberry Pi and the Middleware Raspberry Pi described in section 4.2. It should be noted that the PREEMPT_RT kernel is

not necessary for the devices in the “Control Room”, as these machines are intended to work as servers that need to take care of multiple processes at the same time.

Chapter 8

Discussion

This chapter discusses the architecture described in chapter 3 and the system implementation explained in chapter 4 and how they solve the tasks related to the objectives. The discussion surrounds the most important parts of the system, their advantages and disadvantages, and whether or not they help the system meet the requirements of Industry 4.0. This chapter is meant for those who want to understand the underlying principles of the system. For those who might want to build similar systems, this chapter also sheds light on which aspects should be improved. Section 8.1 reviews the implemented system in its entirety, while section 8.2 reviews the AAS and its functionality. Technical reviews of the components OPC UA, ROS 2, and KUKA's Sunrise.OS are provided in section 8.3.

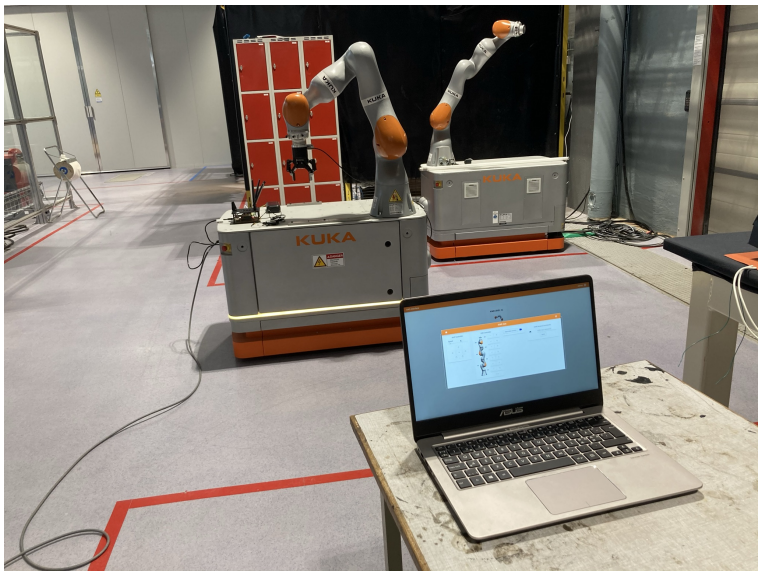


Figure 8.1: Controlling a KMR iiwa robot with the AAS at NTNU Gløshaugen's Industry 4.0 laboratory.

8.1 Review of Implemented System

To study if 5G can be used in conjunction with Industry 4.0 initiatives, an architecture was proposed as a solution in which 5G is incorporated into a modern, industrial factory environment. The realization of this architecture was a system comprised of an Asset Administration Shell, a set of Raspberry Pis with specialized communication capabilities, and a KUKA KMR iiwa industrial AGV. The system was implemented in the light of the two use cases mentioned in section 1.3, which meant that it should support rapid deployment, auto/re-configuration, and testing of new robots, as well as achieving remote control in digital factories.

Based on the theory that was researched in relation to this thesis, it was already known that 5G had been tested with positive results in industrial settings before. What the implemented system presented in this thesis showed is how easy it can be to implement a 5G connection into an "Industrial Lab" facility. Through various experiments, it was shown that this was possible with nothing but a Raspberry Pi, a SIM82000-M2 5G HAT, and a SIM card from an internet service provider.

The relatively simple solution with the Raspberry Pi and the 5G HAT was instrumental in the work towards Use Case 1.5. The Raspberry Pi is portable and easily connected to robotic systems via Ethernet, which makes rapid deployment realizable. Although it is not tested during this project, the idea behind this portable, component-based solution is that when a new entity is introduced to the lab, the content from an existing Entity Raspberry Pi is copied to a new one, a 5G HAT is installed and the new Entity Raspberry Pi is mounted on and connected to the entity. Thus, the new entity is rapidly deployed into the industrial environment without much friction.

Use Case 1.5 also includes the idea of auto/re-configuration, which was an important aspect to consider when implementing the system design. The experimentation results showed that it was possible, with the use of a Raspberry Pi and ROS 2, to automatically configure and start a connection with the KMR iiwa and introduce it to the AAS. With the use of the version control system Git combined with SSH, the code on the Raspberry Pi can be changed remotely, making re-configuration possible with no operator interaction in the lab facility other than restarting the Raspberry Pi on the entity in question. The presented solution does, however, have its limitations. Because the KMR iiwa is reliant on a Sunrise application in order to run, much of the code in the rest of the system had to be written to accommodate this. Thus, many of the features that one might expect to flexible with regard to robot type had to be "hardcoded" to suit the KMR iiwa. A way to remedy this is for robot manufacturers to develop their software with exposed APIs that allow developers to more easily integrate the robots' functionality into their system.

The AAS that was developed to control and monitor the KMR iiwa remotely was tested successfully. Once an entity is detected by the system, an operator will have full access to the functionalities that the entity provides, and they can control it in real-time with fast and reliable connectivity. WebSocket communication, OPC UA, and ROS 2 proved to be valuable assets when developing a system for remote control in digital factories. The AAS partly satisfies the goal of Use Case 1.3, which is mainly remote control, but it does not fulfill the performance requirements. A real-time system for controlling robots in industrial settings needs to be ultra-fast in order to maintain reliability. The stringent requirement of 10 ms delay was not met by the system, either with a 5G connection or a Wi-Fi connection between the Entity Raspberry Pi and the AAS. It should be noted that this result is not believed to be a representation of the 5G technology's capabilities but rather an unfortunate consequence of external factors.

As shown from the system latency experiment, Wi-Fi outperformed 5G under the specific conditions that were available during testing. While this data is interesting to evaluate, it must be emphasized that the test does not paint an accurate picture of the differences between 5G signals and Wi-Fi signals since the two were not tested under the same conditions. Based on previous 5G performance testing, some of which is described in subsubsection 2.1.1.2, it is safe to assume that 5G outperforms previous generations of cellular networking in areas that are important for robotics and automation. With respect to Industry 4.0 and IoT, one can also make a strong argument for why 5G would be superior to Wi-Fi. Wi-Fi networks, while more common and accessible than the new cellular competitor, rely on an unlicensed spectrum that is free and used nearly everywhere. This is beneficial for home and office environments, but having many devices on the same frequency band increases the chance for interference. This can quickly become problematic in an IoT environment in which hundreds of separate entities need fast and reliable transfer of data. This, coupled with the fact that 5G is made to handle a massive amount of highly interactive devices with low latency and high throughput, supports the idea that 5G can, in many cases, replace Wi-Fi in industrial IoT environments.

With access to only a company-owned, public 5G network, it seems to be unrealistic to use 5G instead of Wi-Fi in an industrial setting with stringent performance requirements. Even with support for massive amounts of devices, the latency that is experienced because of the excess routing and data travel makes it too slow to be reliable enough to complement real-time operation. There is one way to remedy this that is posed as the ideal system solution. As mentioned in the result discussion of the latency experiment, a local 5G network on the NTNU Gløshaugen campus would allow for a direct 5G connection between entities and the AAS. The idea is that a 5G node is stationed near the "Control Room", so that the stationary AAS Raspberry Pi can be connected to it via Ethernet. The Entity Raspberry Pis that are

roaming around with the entities in the "Industrial Lab" would communicate wirelessly with the Middleware Raspberry Pi, which again would communicate wirelessly with the 5G node. With a local solution, the system could potentially avoid all routing, which makes the only sources of latency be pure 5G signals and the components themselves. Since 5G has been tested to perform to the standards of Industry 4.0, it is highly likely that this solution would make the system Industry 4.0-compatible.

8.2 Review of AAS

The Asset Administration Shell described and outlined in subsection 4.3.1 satisfies the essential requirements of an AAS proposed by Erdal Tantik and Reiner Anderl [20]. The combination of the Internal Interface created with Flask and the External Interface developed with React generates a well-defined base that reduces complexity and allows for high scalability. The lightweight nature of Flask makes it easy to implement third-party components that are essential for real-time applications, such as WebSocket communication. It also provides a fast and flexible way to implement database and REST API integration, which is perfect for a test environment that requires agile development and quick response to external changes.

The External Interface is easy to learn and use, and it provides quick response times for robot control. Visual effects, such as blurring buttons for components that are offline, make it easy for an operator to see what goes on in a factory or laboratory without a direct view. By implementing the External Interface with React, it makes it highly expandable with respect to introducing and testing new robots. Components in React are reusable, which means that if a new entity is introduced to the system with similar functionality to another established entity, the time required to implement support for the new entity is lowered significantly. An example of this is the video stream component, which is meant to be similar for every entity in the "Industrial Lab". Even with a hundred different entities in the AAS database, the amount of code related to the video stream component would be nearly the same as if there were one entity. There is also low coupling between React components, making it so that implementation of new functionality can be done without significant change in the other parts of the code. A downside of the External Interface is that it currently only supports the KMR iiwa in terms of operational functionality. This does not include the video stream feature, which is intended to be the same for every entity. For multiple types of entities, a smart solution for versatile and dynamic components would have to be implemented.

The Internal Interface reliable and fast, and Flask makes it so that the system is easily expandable with new software components that might be needed for constantly evolving industrial environments. The Internal Interface is the backbone of the AAS, and it is what exposes the "Industrial Lab" to operators working remotely. With the Internal Interface

keeping track of every change that happens with the entities registered in the system, and with data about new entities and entity components being dynamically entered into the database once discovered, the AAS can be seen as a digital representation of the "Industrial Lab". Using Flask also allows for seamless integration with thr real-time communication protocol WebSocket. Even though the solution with Flask and SQLite works well for the purpose of quickly developing a system for testing purposes, it might not be the best fit for a full-fledged system. Flask itself can, in many cases, work well for bigger projects, but the implementation needs to be a lot more robust. A robust backend includes type safety, thorough error handling, more encryption, component tests, and much more in order to be reliable enough for real scenarios.

While it is essential that the AAS is robust and secure in terms of error handling and encryption, it is just as important that the integrity of the industrial environment that the AAS is a gateway into is upheld. This is done by authenticating users and restricting access to functionality through authorization. In the current version of the AAS, this is done by assigning certain roles to operators, which decide their level of access throughout the AAS. This is an easily implemented feature to demonstrate how a hierarchy of roles can restrict features in a way that makes the system less prone to both operational accidents and malicious attacks. The downside of this solution is that, once an operator is logged in, the personal login information is stored un-encoded in the browser. This is not a secure way to store important information because it is not impossible for someone to access and change the content of the local storage in a browser. A more robust security implementation will normally be token-based, i.e., users who log in to the system get a secure access token generated from the API. Then, each time they try to access a certain feature, their request would be sent together with their encoded token to the API. Here, the token would be validated, the token permissions would be verified, and the appropriate access would be returned to the user. The main difference between these two solutions is that with tokens, the security happens in the Internal Interface, while with roles, it happens in the External Interface. Since the Internal Interface is not inherently exposed to the outside world, the token solution would provide significantly more security.

8.3 Technical Reviews

The different technical components used within the system have both advantages and disadvantages. The sub-sections in this section discuss the technical components individually.

8.3.1 OPC UA

In a system that combines a multitude of different platforms and components, OPC UA is a perfect fit. It provides flexibility in that it is easy to implement on even a lightweight backend server, and the publish/subscribe functionality not only accentuates the loosely coupled nature of the entire system but also smoothly integrates with ROS 2. Without OPC UA, a solution can be to use TCP sockets to connect the "Industrial Lab" and the "Control Room", but this requires a more complex code structure and even more connections to keep track of. Having to connect a server and a client can be cumbersome because if one part drops out, the connection needs to be reestablished. The benefits of the publish/subscribe model are discussed throughout this report, but it is difficult to over-emphasize how much easier it is to deal with two components communicating through a message-oriented middleware with which none of the "interactees" need to know about each other. OPC UA also handles encryption and decryption of messages between two applications, which leads to an overall more secure system.

8.3.2 ROS 2

ROS 2 consists of a powerful set of tools for creating robot applications. Using ROS 2's powerful publish/subscribe pattern, a dynamic discovery method has been created in order to support the rapid deployment and automatic configuration aspects that are part of Use Case 1.5. The data between an Entity Raspberry Pi and the Middleware Raspberry Pi is transmitted with the use of ROS publisher and subscriber nodes that operate independent of each other. The Middleware will "ping" the entire subnet in the "Industrial Lab" when it starts in order to detect new entities, and if there are any, they will be added to the database of the AAS. This is only possible since ROS 2 utilizes RTPS in DDS, which is explained in subsection 2.1.10. In order to have RTPS work over a wider area than LAN, multicast routing is required. Since the Internet does not support UDP multicasting, the current implementation of RTPS in DDS does not allow ROS 2 to work over the Internet and is, therefore, a limitation for the proposed system. It is believed that this issue is solved by a local 5G network that operates similarly to a traditional LAN.

For ROS 2 to capture the needs of a real-time robotics application and the standards of Industry 4.0, the core software components of the system must not interfere with the requirements of real-time computing. To adhere to this, the Linux Kernel located on the Middleware and Entity Raspberry were patched with PREEMPT_RT. Introducing a real-time kernel is beneficial because it enhances the capabilities of ROS 2. Some strengths of PREEMPT_RT are described in section 7.2. Only patching the kernel and not installing an entirely new real-time operating system is advantageous because, while there exist special-

ized operating systems written from the ground up intended to serve real-time applications, many of them do not support the other software used for this project, like ROS 2. A choice can be made to entirely dismiss the use of ROS 2 and PREEMPT_RT and build something from the ground up. However, since the focus of this thesis is directed toward exploring the possibilities of 5G, and not maximizing real-time performance, using established software is a more efficient solution.

8.3.3 Sunrise.OS

KUKA's robot application software, Sunrise.OS, has some influence on the overall result of this project. There is no doubt about the fact that Sunrise.OS makes the programming of KUKA-specific robots easier, but as a result, the process of installing, synchronizing, and starting the software is cumbersome, to say the least. As mentioned, Sunrise.OS is technically a software run on top of Windows 7, which is located on the Sunrise Cabinet. These layers of abstraction feel more like barriers that need to be worked around, and with everything being tailored for KUKA-specific solutions, they make it difficult to effectively expand with other types of software. The ideal communication flow would include direct communication with the Programmable logic controller (PLC) on the KMR iiwa, skipping the use of Sunrise.OS. This would require a higher degree of low-level development but overall increase the control of the robot and potentially skip much of the heavy on-boarding process. All in all, Sunrise.OS provides a useful set of tools for controlling the KMR iiwa, but it leaves little room for customized solutions, especially with regard to automatic configuration.

The Sunrise.OS software is manufacturer-dependent and therefore specifically tailored to KUKA entities like the KMR iiwa. As a result of this, its degree of impact substantially falls off as the scope of the project widens to other robots. The global scope of Industry 4.0 spans a large pool of entities, with KUKA only being a percentile. Taking these considerations into account, it should be emphasized that the focus of the proposed system is more directed towards the other aspects and components.

Chapter 9

Conclusion

9.1 Summary

This thesis explores whether 5G can complement and be used as part of or in conjunction with Industry 4.0 initiatives such as Asset Administration Shells by providing a system that attempts to support two use cases proposed by the 5G-SOLUTIONS project. Both use cases involve implementing innovative strategies in digital factories. A system has been made with an accompanying AAS, and it utilizes 5G technology, OPC UA, and ROS 2. The functionality of the system revolves around auto/re-configuration and remote control of the KMR iiwa AGV.

The first part of the thesis describes the relevant background theory, where the concepts and technologies used within the system are introduced. The two subsequent chapters explain the proposed architecture used to answer the problem description and the implemented system based on this architecture. What follows is a chapter that can be viewed as a guide on how to utilize this system for testing purposes. This chapter is followed by two chapters describing the experiments that were conducted. The basis for certain design choices are outlined here, and performance benchmarks of the system are also portrayed. Finally, there is a discussion chapter. This chapter looks at the implemented system with regard to the objectives of the thesis. The various parts of the system are reviewed individually in an attempt to ascertain their relevance both in the overall system and in relation to Industry 4.0 standards.

9.2 Conclusion

Incorporating 5G networking technology in order to make an Industry 4.0-compliant real-time system for operating industrial robots is a daunting task. Based on previous studies, 5G shows great potential in terms of performance, but it is still relatively new. New technologies can

be cumbersome to work with, as it often takes time to establish widespread support within the industry. This fact was made apparent by the delays associated with the installation of a local 5G node at NTNU Gløshaugen's Industry 4.0 laboratory, which made it difficult to achieve substantial performance results. It was found that, with the use of a public 5G network, it was not possible to meet the latency requirements of Use Case 1.3. A conclusion that can be drawn from this is that a local solution must be facilitated in order for 5G communication to be acceptable for industrial applications.

Even though the performance requirements of Use Case 1.3 were not satisfied, other use case objectives were tested and found plausible. With the AAS, remote control of a digital factory was achieved to some extent. The KMR iiwa was controlled remotely, over the internet, and with the use of 5G. The system was only tested with one entity, but it was developed with support for multiple entity types in mind. A part of Use Case 1.5 is rapid deployment, which was attempted by making the processes in the "Industrial Lab" start automatically. This was achieved with the exception of the KMR iiwa, which required an operator in order to start the robot application. Utilizing Raspberry Pis, which are easily mounted on devices like the KMR iiwa, enables efficient configuration and deployment of entities that do not allow custom software to be installed directly. This again makes it so that new robots can be introduced to factories and laboratories and tested within a short amount of time. The Raspberry Pis not only start automatically once they are powered, but their software is also partly configurable remotely. This ties in with another important part of Use Case 1.5, namely auto/re-configuration. In relation to these two aspects, it can be concluded that Use Case 1.5 is only realizable if the software that comes with industrial robots is manufacturer-independent and includes well-documented and exposed APIs. This way, it is possible for components such as the Raspberry Pi to obtain the information it needs from the entity in order to configure it and introduce it to the system automatically.

There are still several enhancements that can be made to the system for it to be applicable in real-world industrial applications. This includes adding full, dynamic support for different types of entities and implementing more security and robustness. Additionally, a local 5G network opens possibilities for enhancements of the system architecture. Since a local solution would not need to be connected to the Internet, it is believed that ROS 2 can be used throughout more of the system pipeline. With pure ROS 2 communication between the AAS and the entities, it is believed that the system would experience less latency as the entire OPC UA "Middleware" link would be omitted. Decoupling more components through the ROS 2 publish/subscribe pattern also allows for even more flexibility and scalability.

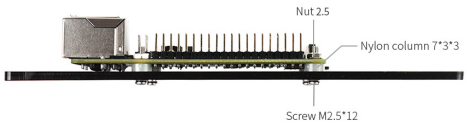
Even with a public network, 5G is proven to have potential within the realm of Industry 4.0, and the implemented system with its AAS certainly demonstrates feasibility. ROS 2 and

OPC UA offer essential functionality in factories of the future, such as flexibility, scalability, and manufacturer independence. Having a system comprised of small, portable Raspberry Pis provides significant versatility, and the solution can be extended and reorganized to fit many types of environments. The work has also shown that nothing more than a small 5G HAT and a SIM card from an internet service provider is required to incorporate the new generation of cellular networking.

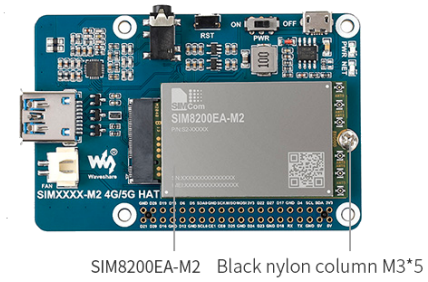
The hope is that these contributions can potentially pave the way for new innovative strategies and help revolutionize the way software systems are implemented in digital factories. For any further development of the system presented in this thesis, the most conclusive recommendation is using a local 5G network. Without this, it is believed that the system simply cannot meet the requirements of real-time robotic control. Although the development towards fast and reliable factories of the future is progressing at a tremendous speed, one of the last pieces of the puzzle is utilizing 5G to its fullest potential!

Appendix A

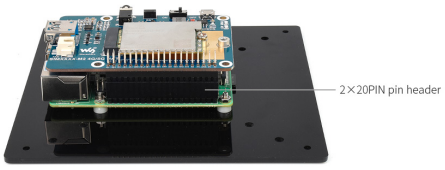
5G HAT Assembly Process



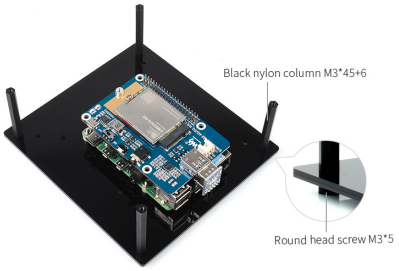
(a) Step 1: Install the Raspberry Pi into the base.



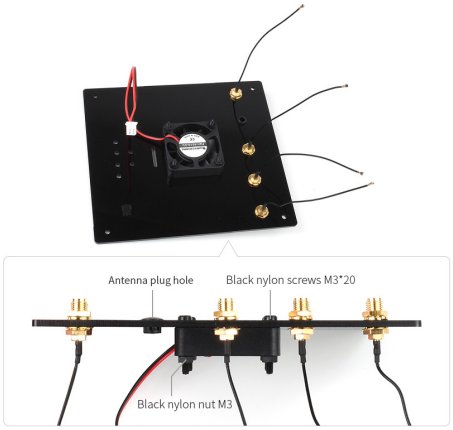
(b) Step 2: Install the SIM8200-m2 mainboard into the SIM82000-M2 5G HAT base board.



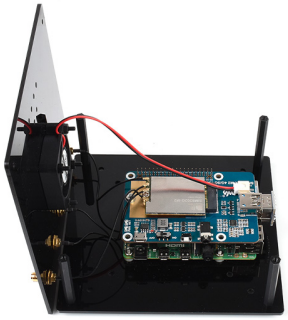
(c) Step 3: Install SIM8200-M2 5G HAT base into the Raspberry Pi.



(d) Step 4: Install nylon columns in the base board.

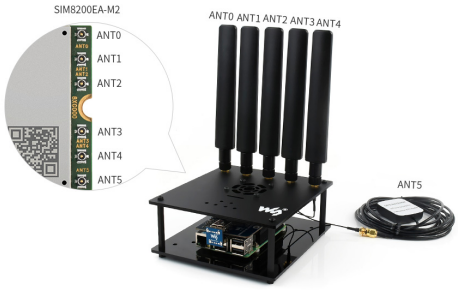


(e) Step 5: Install cooling fan and antenna adapter cables into upper cover board.



(f) Step 6: Connect the antenna adapter cables to the SIM82000-M2 main board and connect the cooling fan to the 5G HAT base board.

(g) Step 7: Assemble the top cover and fix with screws.



(h) Step 8: Install the external antennas.

Figure A.1: 5G HAT assembly process [72].

Bibliography

[1] 5G-SOLUTIONS. *5G - SOLUTIONS FOR EUROPEAN CITIZENS*. 2021. URL: https://5gsolutionsproject.eu.

[2] 5G-SOLUTIONS. *Living Labs*. 2021. URL: https://5gsolutionsproject.eu/living-labs/.

[3] 5g PPP. *5G-VINNI: 5G Verticals INNovation Infrastructure*. 2017. URL: https://5g-ppp.eu/5g-vinni/.

[4] Telenor. *Telenor Group to coordinate pan-European 5G project*. 2018. URL: https://www.telenor.com/media/press-release/telenor-group-to-coordinate-pan-european-5g-project.

[5] The Research Council of Norway. *MANULAB: Norwegian Manufacturing Research Laboratory*. 2017. URL: https://prosjektbanken.forskningsradet.no/en/project/FORISS/269898?Kilde=FORISS&distribution=Organisasjon&chart=bar&calcType=funding&Sprak=no&sortBy=date&sortOrder=desc&resultCount=30&offset=30&ProgAkt.3=FORINFRA-Nasj.sats.+forskn.infrastrukt&source=FORISS&projectId=269870.

[6] M. Shafi et al. '5G: A Tutorial Overview of Standards, Trials, Challenges, Deployment, and Practice'. In: *IEEE Journal on Selected Areas in Communications* 35.6 (2017), pp. 1201–1221. DOI: 10.1109/JSAC.2017.2692307.

[7] Q. Pham et al. 'A Survey of Multi-Access Edge Computing in 5G and Beyond: Fundamentals, Technology Integration, and State-of-the-Art'. In: *IEEE Access* 8 (2020), pp. 116974–117017. DOI: 10.1109/ACCESS.2020.3001277.

[8] Meriem Mhedhbi et al. 'Performance Evaluation of 5G Radio Configurations for Industry 4.0'. In: *2019 International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 2019, pp. 1–6. DOI: 10.1109/WiMOB.2019.8923609.

[9] Plattform Industrie 4.0. 'Network-based communication for Industrie 4.0'. In: (2016). URL: https://www.plattform-i40.de/PI40/Redaktion/EN/Downloads/Publikation/network-based-communication-for-i40.pdf?__blob=publicationFile&v=4.

[10] Joachim Sachs et al. 'Adaptive 5G Low-Latency Communication for Tactile InternEt Services'. In: *Proceedings of the IEEE* 107.2 (2019), pp. 325–349. DOI: 10.1109/JPROC.2018.2864587.

[11] James Martin. *Programming real-time computer systems*. 1965. URL: https://archive.org/details/programmingrealt0000mart/page/n1/mode/2up.

[12] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer US, 2011. ISBN: 9781441982377. URL: https://books.google.no/books?id=oJZsvEawlAMC.

[13] Inc. Open Source Robotics Foundation. *Introduction to Real-time Systems*. 2021. URL: https://design.ros2.org/articles/realtime_background.html.

[14] The Linux Foundation. *What Is Linux?* 2021. URL: https://www.linux.com/what-is-linux/.

[15] Amit Kumar Das. *Working with the real-time kernel for Red Hat Enterprise Linux*. 2020. URL: https://www.redhat.com/sysadmin/real-time-kernel.

[16] The Linux Foundation. *Intro to Real-Time Linux for Embedded Developers*. 2021. URL: https://linuxfoundation.org/blog/intro-to-real-time-linux-for-embedded-developers/.

[17] Graig Hunt. *TCP/IP Network Administration, Third Edition*. O'Reilly Media, Inc., 2002.

[18] Santosh Kumar and Sonam Rai. 'Survey on transport layer protocols: TCP & UDP'. In: *International Journal of Computer Applications* 46.7 (2012), pp. 20–25.

[19] Valenthin De Leeuw. *Concepts and Applications of the I4.0 Asset Administration Shell*. 2019. URL: https://www.arcweb.com/blog/concepts-applications-i40-asset-administration-shell.

[20] Erdal Tantik and Reiner Anderl. 'Integrated data model and structure for the asset administration shell in Industrie 4.0'. In: (2017).

[21] OPC Foundation. *Unified Architecture*. 2020. URL: https://opcfoundation.org/about/opc-technologies/opc-ua/.

[22] Eclipse Foundation. *Milo*. 2016. URL: https://projects.eclipse.org/proposals/milo.

[23] codecademy. *What is REST?* 2020. URL: https://www.codecademy.com/articles/what-is-rest.

[24] Sagar Mane. *Understanding REST*. 2017. URL: https://medium.com/@sagar.mane006/understanding-rest-representational-state-transfer-85256b9424aa.

[25] Victoria Pimentel and Bradford G. Nickerson. 'Communicating and Displaying Real-Time Data with WebSocket'. In: *IEEE Internet Computing* 16.4 (2012), pp. 45–53. DOI: 10.1109/MIC.2012.64.

[26] Object Management Group. *What is DDS?* 2020. URL: https://www.dds-foundation.org/what-is-dds-3/.

[27] Object Management Group (OMG). *The Real-time Publish-Subscribe Protocol DDS Interoperability Wire Protocol (DDSI-RTPSTM) Specification*. Document formal/2019-04-03. Object Management Group (OMG), 2019.

[28] Stephen E. Deering and David R. Cheriton. 'Multicast Routing in Datagram Internetworks and Extended LANs'. In: *ACM Trans. Comput. Syst.* 8.2 (May 1990), 85–110. ISSN: 0734-2071. DOI: 10.1145/78952.78953. URL: https://doi.org/10.1145/78952.78953.

[29] Jean-Jacques Pansiot and Dominique Grad. 'On Routes and Multicast Trees in the Internet'. In: *SIGCOMM Comput. Commun. Rev.* 28.1 (Jan. 1998), 41–50. ISSN: 0146-4833. DOI: 10.1145/280549.280555. URL: https://doi.org/10.1145/280549.280555.

[30] The Raspberry Pi Foundation. *Raspberry Pi 4 Computer Model B*. 2021. URL: https://datasheets.raspberrypi.org/rpi4/raspberry-pi-4-product-brief.pdf.

[31] The Raspberry Pi Foundation. *Raspberry Pi 4 Tech Specs*. 2021. URL: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/.

[32] Gareth Halfacree. *Benchmarking the Raspberry Pi 4*. 2019. URL: https://medium.com/@ghalfacree/benchmarking-the-raspberry-pi-4-73e5afbcd54b.

[33] Simcom. *SIM8200EA-M2 Hardware Design*. 2020. URL: https://www.waveshare.com/w/upload/d/de/SIM8200EA-M2_Hardware_Design_V1.03.pdf.

[34] waveshare. *SIM8200EA-M2 5G HAT for Raspberry Pi, 5G/4G/3G Support, Snapdragon X55, Multi Mode Multi Band*. 2021. URL: https://www.waveshare.com/product/sim8200ea-m2-5g-hat.htm.

[35] VMAC. *Understanding AT commands*. 2011. URL: https://sites.google.com/site/vmacgpsgsm/understanding-at-commands.

[36] Kuka. *KMR iiwa*. 2020. URL: https://www.kuka.com/en-ch/products/mobility/mobile-robots/kmr-iiwa.

[37] KUKA Deutschland GmbH. 'Mobile Robots - KMR iiwa omniMove - Mobile Industrial Robot System - Assembly and Operating Instructions'. In: (2020).

[38] Charlotte Heggem and Nina Marie Wahl. 'Mobile Navigation and Manipulation - Configuration and Control of the KMR iiwa with ROS2'. In: (2020).

[39] Michal gurgul. *WIRELESS TEACH PENDANTS FOR ROBOTS*. 2019. URL: https://roboticsbook.com/wireless-teach-pendants-for-robots/.

[40] ROS Index. *ROS 2 Documentation*. 2020. URL: https://index.ros.org/doc/ros2.

[41] ROS Index. *ROS 2 Foxy Fitzroy (codename 'foxy'; June 5th, 2020)*. 2020. URL: https://index.ros.org/doc/ros2/Releases/Release-Foxy-Fitzroy/.

[42] Inc. Open Source Robotics Foundation. *Why ROS 2?* 2020. URL: https://design.ros2.org/articles/why_ros2.html.

[43] ADLINK Technology Inc. *ADLINK Partners with FFG for 5G- and ROS2-based Factories of the Future*. 2020. URL: https://www.adlinktech.com/en/News_18051803244567414.

[44] Yuya Maruyama, Shinpei Kato and Takuya Azumi. 'Exploring the performance of ROS2'. In: Oct. 2016, pp. 1–10. DOI: 10.1145/2968478.2968502.

[45] ROS Index. *Understanding ROS 2 services*. 2020. URL: https://index.ros.org/doc/ros2/Tutorials/Services/Understanding-ROS2-Services/.

[46] ROS Index. *Understanding ROS 2 nodes*. 2020. URL: https://index.ros.org/doc/ros2/Tutorials/Understanding-ROS2-Nodes/.

[47] ROS Index. *Understanding ROS 2 actions*. 2020. URL: https://index.ros.org/doc/ros2/Tutorials/Understanding-ROS2-Actions/.

[48] Pallets. *Werkzeug*. 2007. URL: https://werkzeug.palletsprojects.com/en/1.0.x/.

[49] Pallets. *Jinja*. 2021. URL: https://palletsprojects.com/p/jinja/.

[50] Pallets. *Click*. 2014. URL: https://click.palletsprojects.com/en/7.x/.

[51] M. Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, 2018. ISBN: 9781491991695. URL: https://books.google.no/books?id=cVlPDwAAQBAJ.

[52] Netflix. *The Netflix Tech Blog*. 2021. URL: http://techblog.netflix.com/2015/09/introducing-lemur.html?m=1.

[53] Ryan Horn. *Introducing Flask-RESTful*. 2012. URL: https://www.twilio.com/blog/introducing-flask-restful.

[54] Matthew Bryant and Collin Greene. *Uber Engineering Bug Bounty: The Treasure Map*. 2012. URL: https://eng.uber.com/bug-bounty-map/.

[55] Facebook inc. *React*. 2020. URL: https://reactjs.org/.

[56] Abhishek Thakur. *WebGear API*. 2021. URL: https://abhitronix.github.io/vidgear/v0.2.1-dev/bonus/reference/webgear/.

[57] Keysight Technologies. *Hawkeye: Active Network Monitoring*. 2021. URL: https://www.keysight.com/zz/en/products/network-test/performance-monitoring/hawkeye.html.

[58] Keysight Technologies. *Hawkeye – Active Network Monitoring Platform*. 2020. URL: https://www.keysight.com/zz/en/assets/3119-1140/data-sheets/Hawkeye-Active-Network-Assessment-and-Monitoring-Platform.pdf.

[59] KUKA. *KUKA Sunrise.OS*. 2015. URL: https://www.kuka.com/en-de/products/robot-systems/software/system-software/sunriseos.

[60] KUKA. *KUKA Sunrise.OS 1.11, KUKA Sunrise.Workbench 1.11, Operating and Programming Instructions for System Integrators*. 2016. URL: http://www.oir.caltech.edu/twiki_oir/pub/Palomar/ZTF/KUKARoboticArmMaterial/KUKA_SunriseOS_111_SI_en.pdf.

[61] Andreas Angerer et al. 'Robotics API: object-oriented software development for industrial robots'. In: (Jan. 2013).

[62] Charlotte Heggem and Nina Marie Wahl. *kmriiwa_ws*. 2020. URL: https://github.com/ninamwa/kmriiwa_ws.

[63] ROS Index. *rclpy*. 2020. URL: https://index.ros.org/r/rclpy/.

[64] Open Source Robotics Foundation. *The build system "ament$_c$make" and the meta build tool \ament$_t$ools"*. 2020. URL: https://design.ros2.org/articles/ament.html.

[65] ROS Index. *Creating your first ROS 2 package*. 2020. URL: https://index.ros.org/doc/ros2/Tutorials/Creating-Your-First-ROS2-Package/.

[66] Dirk Thomas. *colcon - collective construction*. 2018. URL: https://colcon.readthedocs.io/en/released/.

[67] ROS Index. *Launching/monitoring multiple nodes with Launch*. 2020. URL: https://index.ros.org/doc/ros2/Tutorials/Launch-system/.

[68] M. Owens. *The Definitive Guide to SQLite*. Expert's voice in open source. Apress, 2006. ISBN: 9781430201724. URL: https://books.google.no/books?id=VsZ5bUh0XAkC.

[69] RASPBERRY PI FOUNDATION. *Raspberry Pi OS 64-bit*. 2021. URL: https://downloads.raspberrypi.org/raspios_arm64/images/raspios_arm64-2020-05-28/2020-05-27-raspios-buster-arm64.zip.

[70] ROS Index. *Real-time programming in ROS 2*. 2021. URL: https://index.ros.org/doc/ros2/Tutorials/Real-Time-Programming/.

[71] Menubis in Circuits. *64bit RT Kernel Compilation for Raspberry Pi 4B* . 2021. URL: https://www.instructables.com/64bit-RT-Kernel-Compilation-for-Raspberry-Pi-4B-/.

[72] waveshare. *SIM8202G-M2-5G-HAT-Assembly*. 2021. URL: https://www.waveshare.com/w/upload/d/d8/SIM8202G-M2-5G-HAT-Assembly-en.jpg.

[73] waveshare. *SIM8200EA-M2 5G HAT*. 2021. URL: https://www.waveshare.com/wiki/SIM8200EA-M2_5G_HAT.

[74] Software in the Public Interest. *Support*. 2021. URL: https://www.debian.org/support.en.html.

[75] Open Robotics. *Building ROS 2 on Ubuntu Linux*. 2021. URL: https://docs.ros.org/en/foxy/Installation/Ubuntu-Development-Setup.html.

[76] Sander van Dijk. *Raspberry Pi + ROS 2 + Camera*. 2020. URL: https://medium.com/swlh/raspberry-pi-ros-2-camera-eef8f8b94304.

[77] NASA Meti Norkart AS Geovekst. *Kommunekart*. 2021. URL: https://kommunekart.com/?urlid=2ca84c8a-163e-47a8-93d7-342b8fc42c93.

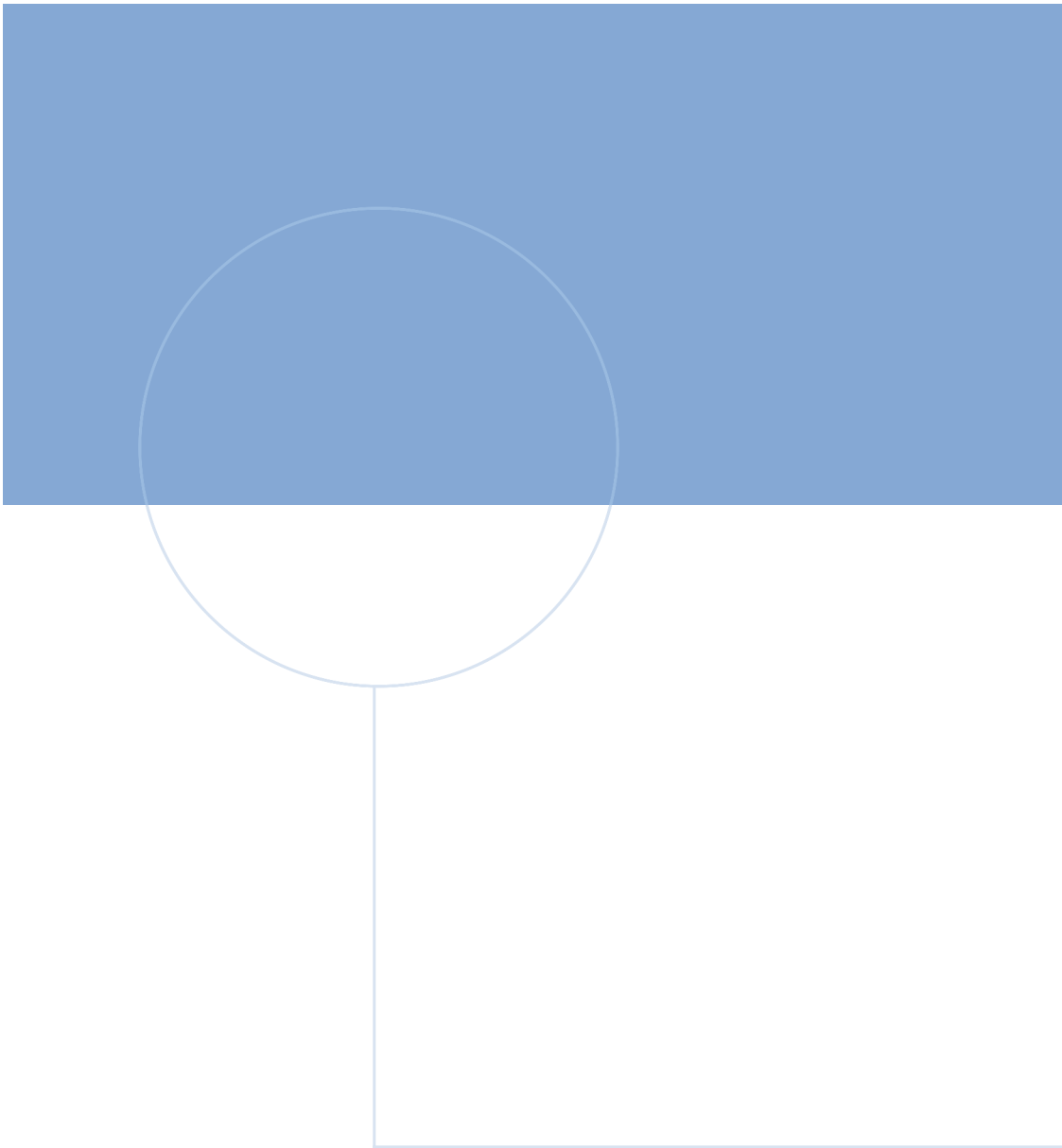
[78] Oumer T Osman Y. *LTE-NR tight-interworking and the first steps to 5G*. Ed. by Ericsson. 2017. URL: https://www.ericsson.com/en/blog/2017/11/lte-nr-tight-interworking-and-the-first-steps-to-5g.

[79] Andreas Arnholm. *Demo-5G-OPCUA-ROS2-KMR-IIWA-MATHIAS-ANDREAS*. 2021. URL: https://www.youtube.com/watch?v=YXchGNEquUo.

[80] ixia. *IXCHARIOT/HAWKEYE PLATFORM ENDPOINTS*. 2021. URL: https://support.ixiacom.com/support-links/ixchariot/endpoint-library/platform-endpoints.

[81] Linaro Ltd. *automated/linux/cyclictest/cyclictest*. 2021. URL: https://test-definitions.readthedocs.io/en/latest/automated/linux/cyclictest/cyclictest.html.

[82] Canonical Ltd. Ubuntu and Canonical are registered trademarks of Canonical Ltd. *Cyclictest*. 2019. URL: http://manpages.ubuntu.com/manpages/cosmic/man8/cyclictest.8.html.



NTNU

Norwegian University of
Science and Technology