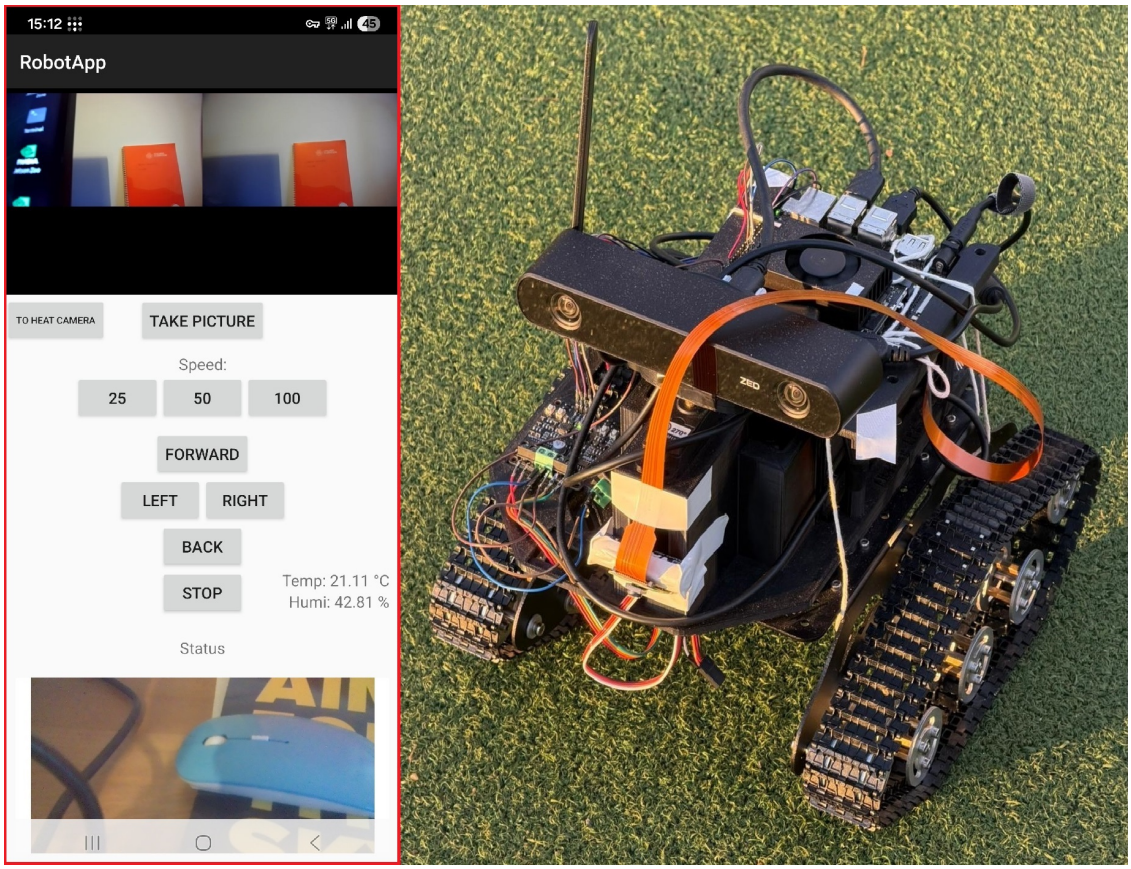
CHALMERS
UNIVERSITY OF TECHNOLOGY



App-Operated Robot with Live Streaming and environmental sensor

A Hardware and software project

Degree Project in Computer Engineering

Ahmad-Reza Firuzian

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Combining hardware and software to an app-operated robot.
A study about the necessary harware and software needed to build a robot that is controlled via an app.

Ahmad-Reza Firuzian

Supervisor: Sakib Sistek, Computer Science and Engineering
Examiner: John J. Camilleri, Computer Science and Engineering

Degree project report 2026
Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Abstract

Being in inaccessible or dangerous environments requires remote controlled observation and control systems. This report describes the steps necessary for connecting various hardware components to each other and the software required to combine the hardware and software into a functioning system. This project presents the construction and implementation of an app-operated mobile robot capable of navigating different terrains, providing live video and image capture, and measuring ambient temperature.

The system is built around a Jetson Orin Nano and integrates multiple hardware components, including a motor driver, motors, cameras, an environmental sensor and a 4G modem for remote communication. A server running on the robot handles video streaming, motor control commands, sensor data transmission, while a custom-developed mobile application enables real-time monitoring and remote control.

The robot was developed in a modular manner, allowing individual features to be implemented and tested incrementally. The results show successful remote control, live video streaming, and temperature monitoring, although communication instability due to the 4G modem affected latency and control responsiveness. Overall, the project demonstrates the feasibility of integrating hardware and software into a functional remotely operated robotic platform and provides a foundation for future enhancements.

Acknowledgement

This degree project has been carried out as part of the Bachelor of Science in Computer Engineering program at Chalmers University of Technology and has given me a chance to apply theoretical knowledge in practice, in the areas of digital learning, hardware connectivity, software and app development. I thank Sakib Sistek who supervised me during the course of the work.

Ahmad-Reza Firuzian, Gothenburg, 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

API	Application Programming Interface
APN	Access Point Name
AT	Attention
CG	Carrier-Grade
COM	Communication Port
CSI	Camera Serial Interface
CSV	Comma-Separated Values
DASH	Dynamic Adaptive Streaming over HTTP
ECM	Ethernet Control Model
FFmpeg	Fast Forward Moving Pictures Experts Group
FPS	Frames Per Second
GPIO	General Purpose Input/Output
GPS	Global Positioning System
HLS	HTTP Live Streaming
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
IPv	Internet Protocol version
IR	InfraRed
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
MCU	Microcontroller Unit
NAT	Network Address Translation
ND	Non-Dormant
OOP	Object-oriented programming
OS	Operating System
PD	Power Delivery
PPP	Point-to-Point Protocol
PWM	Pulse Width Modulation
QMI	Qualcomm MSM Interface

RTMP	Real-Time Messaging Protocol
RTSP	Real-Time Streaming Protocol
SBC	Single Board Computer
SDK	Software Development Kit
SIM	Subscriber Identity Module
SRT	Secure Reliable Transport
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
USB	Universal Serial Bus
UVC	USB Video Class
VPN	Virtual Private Network
WebRTC	Web Real-Time Communication
WSL	Windows Subsystem for Linux
XML	eXtensible Markup Language

Contents

1

Introduction

In this chapter, the background, purpose, goal and the demarcations of the project are described.

1.1 Background

It is dangerous or impossible for humans to be in certain environments, such as smoke-filled spaces, fires, unstable buildings after an accident, or inaccessible places in nature. In such situations, a way is needed to remotely see what is happening, get information about the temperature, and navigate safely – without putting humans at risk.

When there are areas that needs to be investigated, but it is not possible for human beings to be there and see with their own eyes there should be a way to see and know how the place/situation is like. E.g. a building where chemicals have leaked, or a radioactively contaminated site, or active geological zones, or war zones and areas with mines, or in buildings where there is a fire. In those situations, the robot is very useful, and no human lives need to be in danger. There are also other areas of use of the robot, such as examining under vehicles, exploring narrow spaces in nature and examining inside buildings or rubble (after accidents).

1.2 Purpose

The purpose is to find out how to build/create a robot that can move on different roads (both asphalt roads and less accessible paths), show live video, take pictures, show ambient temperature and how the robot can be controlled from a long distance. To find out what hardware you need, what software you need to develop and how to connect hardware with software into a functioning system.

1.3 Goal

The goal is to create an app-controlled robot that can be in places where humans cannot be and observe on site. The robot will drive forward, backward or turn via

the app. The robot is equipped with a camera that can take pictures and send live video to the app. It is possible for the user to see close up in front of the robot via an additional camera. It is possible for the user to see the temperature where the robot is in the app. Since the robot will have a modem with a SIM card, the robot can be controlled remotely. The maximum distance will be determined by the telecom operator's mobile network. Here are the subgoals:

- Connect all parts (Jetson Orin Nano, rechargeable battery, cameras, motor driver (MDD3A), motors, modem, temperature sensor) physically and mount all parts on the chassis.
- Develop a server that can send live video, images, and ambient temperature data from the robot's location, and receive commands for the motors.
- Ensure the modem has an internet connection so that the Jetson can access the internet via the modem.
- Develop an app that connects to the server (via the modem).
- Implement motor control via the app.
- Enable live video and image capture in the app.
- Display the temperature in the app.

If time permits, the robot will also have some of the following functionalities: Autonomous navigation with obstacle recognition, thermal camera, display of battery percentage in the app and solar cell charging via solar cells.

1.4 Limitations / Demarcations

Although the original idea is that the robot should operate in "disaster situations", the focus of this project is on how to build/create a robot with the aforementioned features. That is, how to connect hardware with software to build a robot with the desired functions. In this project, the robot is not designed for rough terrain, rain or extreme temperatures. The electronics are not protected from rain/snow or dust. The robot can only operate outdoors when it is not raining/snowing. The robot will also not be operated in environments with a lot of dust particles. If you are going to use the robot in "disaster situations", you need to add protection for all electronic components, you need to make sure that the camera is not obscured by dust or other things. The cameras are for live video and image capture, the video is mainly for viewing, not interpretation. Communication between the server (on Jetson) and the app is via 4G with SIM card, which means that the project does not focus on Wi-Fi connections or local network control. The app shows live video, temperature and gives control commands – no complex user management, no history logging or cloud storage. So, no advanced backend. Server security is not a priority. Only

if time is available, some type of secure connection to the server will be implemented.

The following features are only included if time is available (These are not part of the basic requirements and will not be prioritized):

- Autonomous navigation with obstacle recognition
- Thermal camera
- Battery percentage (remaining)
- Solar cell charging via solar cells

2

Methods

This chapter describes the approach from a "before" perspective: the perspective that was in mind when entering the project and planning the work.

2.1 Preliminary Work

Before starting the work, an initial plan will be made that describes the time-line of the whole process (the plan is shown in figure A.1 in Appendix 1). At first, an investigation and a research will be done about how to start and proceed about building a robot with the specific features and functionalities. What devices and hardware are needed will be identified, the challenges for the project will be determined, and potential solutions may be explored. When that is done, the components will be ordered and the work will proceed according to the plan.

2.2 System Setup and Hardware Integration

When the initial material is received, the work with the SBC (Single-Board Computer) (henceforth referred to as the “main unit”) will begin. If necessary, an installation/re-installation of the OS on the main unit will be carried out (so that add/install, edit, change the settings in the operating system can be done).

At the beginning and throughout the project, it is necessary to consider and ensure that the components (hardware devices) can be accommodated and mounted on the chassis. It must also be ensured that all components can be connected to each other once they are mounted on the chassis.

2.3 System Implementation - Basic Features

Once the OS is in place, the construction will start by connecting the main camera (a ZED2i camera) to the main unit. Then, a program in python will be written on the main unit that can connect to the camera and display live video. When that works, a server on the main unit that sends live video and has the ability to take pictures will be written. Then, a small Android app (in java) that receives live video

from the server and displays the video in the app will be written. The app should also have a button for taking pictures. In the first stage, the server and app will be connected via local Wi-Fi (not via the mobile network at first). When that works, the work-process will move on to the next step.

In the next step, motor control is implemented. At first, it is necessary to find out how the motors work, which cables to connect, which components are needed and how to connect the motors to the main unit. When all this is known, the motors will be connected to the main unit and a simple program that sends signals to the motors and makes them rotate is written. When the test goes well, a server is created on the main unit that can receive commands to drive forward, backward, turn and change the speed of the robot. Then an app that can send signals (forward, backward, turn, etc.) to the server and thus control the motors is created.

When motor control is complete, a modified server is created where live video, image capture is combined with motor control. Then, the app is modified so that it has the combined functionalities. When this is complete, the battery, camera and the motors (plus any necessary devices) are connected to the main unit and a test that drives and tests the robot (so that the functionalities work) is carried out.

When live video, image capture and motor control work, the work continues with connecting the modem. A server that is accessible via mobile network is developed. As before, the server should be able to send live video, take pictures and be able to receive commands for motor control. When the server is complete, the app is modified and updated. The app should be able to connect to the server via mobile network, receive video streaming, take pictures and send commands to the server for motor control. When the server and app are done with this part, an investigation about how to optimize server-client will be carried out. This is done so that live video in the app has low latency and good quality. After that, the functionalities (so far) is tested from distance. If it works properly, then the work continues with the next part.

The next part is about the temperature sensor. First, information about its functioning should be studied. Then, the sensor is connected to the main unit and a small program that connects to the sensor and reads the data is written. When it works, the working code is put in the server code and the app is modified so that it gets the temperature and shows it in the app.

The next step is about a small camera (the second camera) that is supposed to be there to observe close to the front of the robot, an area where the big and main camera do not see. First, some reading and studying will be carried out to understand how the small camera works. Then, a program that can show live video from the camera is written. When it works, the code is added to the server code and some additional code will be added to the app to show live video from the second camera.

2.4 Extra Features

If time permits, the following functionalities will be added:

- Autonomous navigation with obstacle recognition
- Thermal camera
- Display of battery percentage in the app
- Solar charging via solar cells

For autonomous navigation with obstacle recognition, a research is needed to find out how it works. It is necessary to find out if the modem being used is sufficient and if there is a need for a GPS module. Also, to know if an algorithm or something else is needed.

After some reading about the thermal camera, it will be connected to the main unit. After that, a small program that shows what the camera sees is written. When it works, the code will be added to the server and the app is modified so that the thermal video is shown in the app.

For battery percentage in the app, an investigation is needed to find out if it is possible for the battery to send data about percentage and other things to the server. If possible, the code is added to the server so that it sends data to the app. If not, other batteries should be investigated. Further investigation will also be done to find out if there is another way to find out the battery percentage (e.g. via an additional component).

For solar cells and solar cell charging, a research has to be done to find out and gain knowledge about how it works. After ordering and receiving the material, the solar cells will be connected to the battery. If the current battery does not support solar charging, it maybe necessary to think of another battery. When the solar cells are connected to the battery, a test will be carried out outdoors (on a sunny day) to see that the battery is being charged.

3

Technical Background

In this chapter, the technical background of the project is explained.

3.1 Nvidia Jetson Orin Nano

Nvidias Jetson Orin Nano is a kind of computer (more specifically a SBC) that is compact and powerful. A picture of Jetson Orin Nano is seen in figure 3.1. It is mainly used for generative AI and provides access to a broad ecosystem for AI software [1]. However, it can be used for many purposes. It delivers performance for both embedded systems and portable systems. It can also advantageously be used for robotics. It is used by researchers, educators and students to get access to cutting-edge technology and access to robotics and AI frameworks [2].

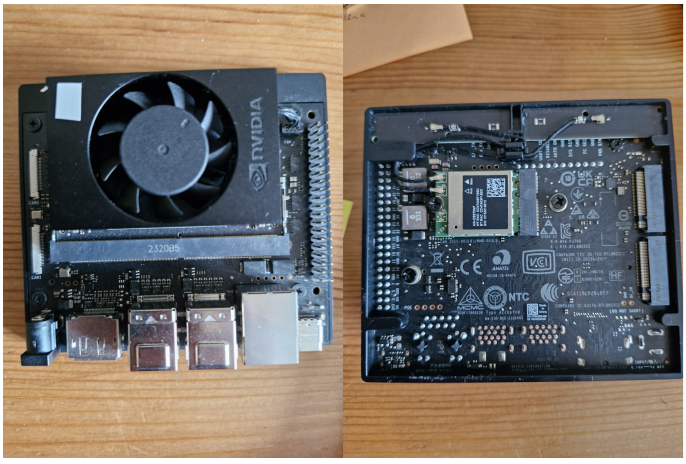


Figure 3.1: Picture of Jetson Orin Nano, front and back.

The Jetson has a 6core Arm Cortex 64-bit CPU. Ampere architecture with 1024 core with 32 tensorcores [3]. The Jetson's size (103 mm × 90.5 mm × 35 mm) is approximately the size of a small pocket notebook (A7 size). However, it has many possibilities for connections. It has 4 USB-A-ports, 1 USB-C port, 40 GPIO-pins, displayPort connector, Gigabit Ethernet, DC barrel power-jack, Wi-Fi connection and 2 CSI-connectors. If you connect a keyboard, a mouse and a computer screen to the Jetson , it can be used as any Linux computer.

Jetson uses Ubuntu as the OS. JetPack SDK is provided by Nvidia. JetPack SDK includes Linux which is Ubuntu-based. Jetson uses by default a microSD card for storage, but it can use other types of storage that are high-performance (e.g. USB 3 SSD, eMMC or NVMe SSD) [4].

3.2 Robot control

The robot has 2 DC-motors. A picture of the chassis is seen in figure 3.2 and a picture of the 2 motors are seen in figure 3.3. The user that uses the app (the client), clicks on buttons (forward, backward, turn left, right, set speed) to control the robot. The server receives the command and calls the corresponding function to execute the command. Every DC-motor has 2 wires that are connected to the motor driver, one pin is for direction of the motor and the other one is for the speed of the motor. The direction pin is set to HIGH or LOW (for forward or backward) and the PWM-pin is set to a percentage (0-100 for speed). Depending on which command the server receives, the GPIO pins on the Jetson are set, and thus signals are sent to the motors and the robot runs.

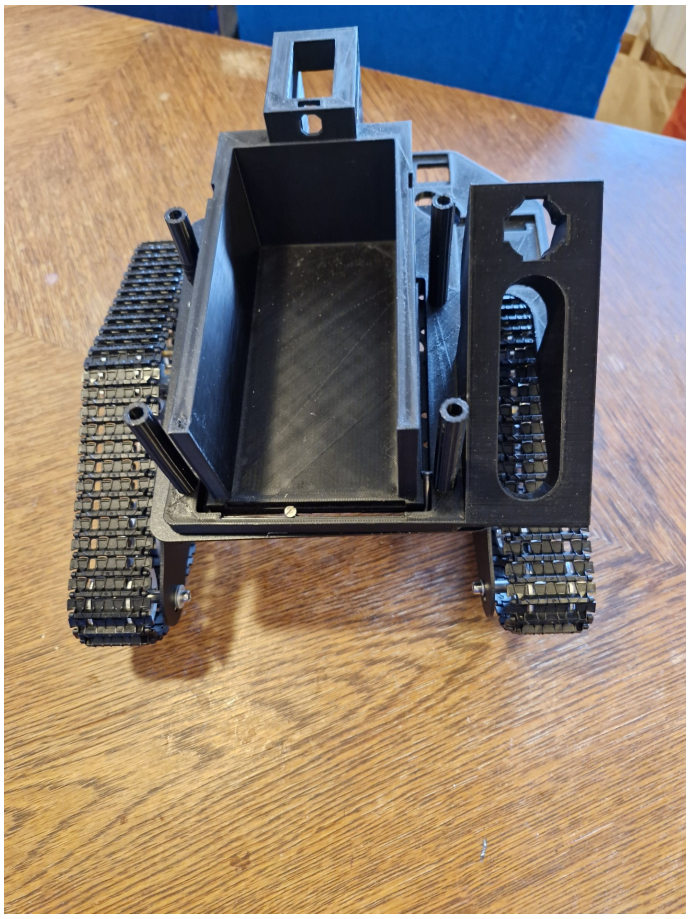


Figure 3.2: Picture of the robot's chassis with caterpillar tracks.

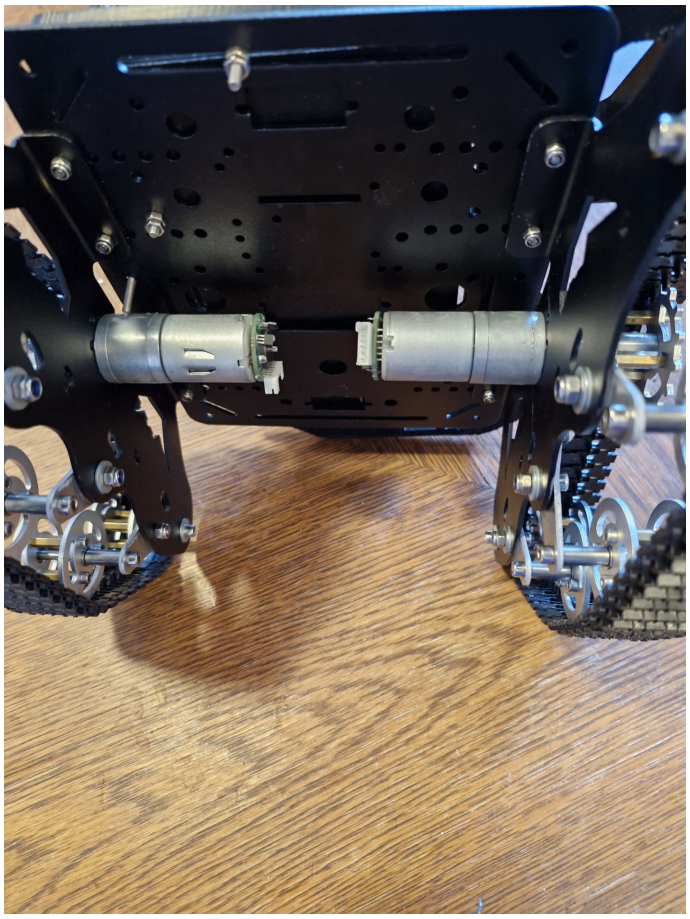


Figure 3.3: Picture of the 2 motors on the underside of the robot.

3.3 Motor driver (MDD3A)

The robot has 2 DC-motors. Since the Jetson cannot deliver enough electricity to the motors, a motor driver is used. So, Jetson is connected to the motor driver and the motors are connected to the motor driver. A picture of the motor driver can be seen in figure 3.4. Cytron's MDD3A can control 2 DC motors. The supply voltage is 4V to 16V. The controller can work with logical values of 12 V (PWM and DIR), 5V, 3.3V and 1.8V. That means that it can be controlled by many microcontrollers (e.g. Raspberry, Jetson etc.). MDD3A has 6 inputpins for receiving signals from Jetson (for the direction and speed of the motors), and 4 other pins for connecting to the motors (for motor direction and speed) and 2 other pins for power supply [5].

The MDD3A has testbuttons on the board for every motor. It has also LED's for every motor. The testbuttons are very convenient for testing, checking and debugging instead of the need for complicated wiring setup. The MDD3A has reverse polarity protection, that is very good for all types of users [6].

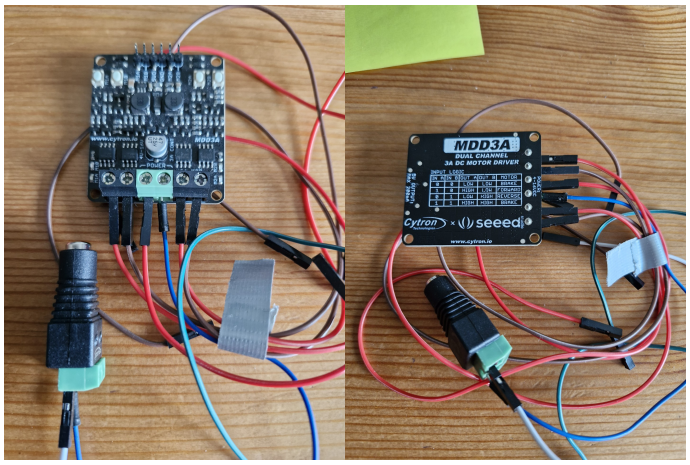


Figure 3.4: Picture of the motor driver, front and back.

3.4 Mobile communication

Mobile communication use radio frequency for sending signals via an antenna. This enables mobile devices to connect to networks that are mobile [7]. To be able to control the robot from long distance (not only within WiFi distance), a modem is neccessary. In this project a SIM7600E-H 4G DONGLE modem is used. To solve the problem of public IP, NAT, port forwarding, complex network setup, firewall, security and access control, Tailscale is used.

3.4.1 SIM7600E-H 4G DONGLE modem

The SIM7600E-H 4G DONGLE modem is an industrial-grade type of dongle. It is used for remote mode of the robot [8]. A picture of the modem can be seen in figure 3.5. The modem has up to 150 Mbps speed for download and up to 50 Mbps for upload. As its name says, it uses 4G as its primary communication technology. It supports operating systems such as Linux, Android and Windows. It gets connected to a device via USB-A port. It has a nano SIM card slot. With a sim card that has an active subscription from a mobile phone operator, it connects to mobile network. By connecting it to a device such as a Windows-PC, Mac-computer, Linux-machine, Raspberry Pi, IoT or an industrial computer, the device automatically connects to internet and the device gets internet access (when the settings of the modem are set correctly) [9].

3.4.2 TailScale

The app that controls the robot and receives live video from the server (on the Jetson) uses the IP address of the server to get and send data. With a regular subscription for the sim card, the internet provider doesn't provide a public IP address (it is though possible to pay extra and get a public IP). So, without a public IP, it is not possible to connect to the server that is on the Jetson. But, there are ways to solve this, e.g. using a VPN, a tunneling service, relay or a cloud service.

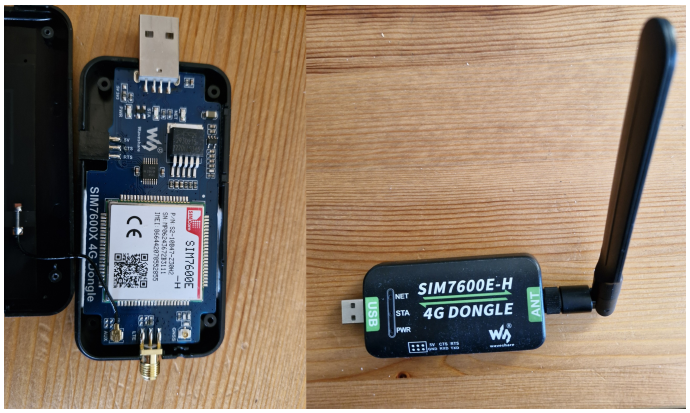


Figure 3.5: Picture of the modem.

This is where Tailscale comes in. Tailscale is like an intermediary between two or more devices and makes it easy to connect devices that are initially not on the same network. For up to 3 users and up to 100 devices, TailScale is free [10].

Originally, TailScale was designed for users collaborating across different platforms. It has a very user-friendly device connectivity. TailScale uses WireGuard VPN to establish direct and fast connection between devices with low latency [11].

TailScale works with Windows, MacOS, Linux, Android, iOS, Arm and more [12]. For connecting devices, NAT traversal is used by TailScale. This works even if the devices are behind NATs or firewalls. It works regardless of the geographical position and what infrastructure is being used. Every device gets assigned a static IP address by TailScale [13]. To use TailScale, you have to install it in all used devices with a private identifier. In the TailScale app, you can see all devices and their assigned IP addresses. Then, you can use that IP address in the code as if the two devices were in the same network.

3.5 Cameras

In this project, two cameras have been used. The Zed2i camera is used for live video and image capture, and the RaspberryPi camera module2 is used for live video from just in front of the robot (to see possible obstacles).

3.5.1 Zed2i

Stereolabs Zed2i camera is an advanced camera that combines 120 degree depth sensing with AI perception. It can be used for spatial analytics and robotics. It connects to a device via USB-A cable. This makes it compatible with many devices such as desktop PC's, embedded devices and Jetson devices. It is ideal for autonomous navigation, has many other features and can be used for many implementations [14]. However, in this project it is only used for live video and image capture.

3.5.2 Raspberry Pi Camera Module 2

The Raspberry Pi Camera Module 2 can be used to take pictures or sending high-resolution video. It is simple to use (for beginners) and has advanced features for more experienced users. It can be used for slowmotion, time-lapse or other smart video processing features [15].

The Raspberry Pi Camera Module 2 features 8-megapixels resolution, uses one channel and is connected to a device via a CSI connector. Its max frame rate capture is 30 fps. It can be connected to many devices such as Raspberry Pi, Jetson etc. It can be connected to Jetson's CAM0-port or CAM1-port. By using Sony's image sensor IMX219, it offers high sensitivity and high-speed video imaging. It supports Full HD video (1080p) at standard frame rates, HD video (720p) at 60 fps and VGA video (640 * 480) at 90 fps [16]. In this project, it is used to send live-video to the user-app (so that the user can see infront of the robot).

3.6 The Server

The server in this project is a multifunctional, multi-process control server for a robotic edge system. It integrates video streaming, computer vision, network services, low level motor control and process management. The entire server is written in Python. In theory, it is a hybrid of different domains:

- Embedded systems engineering (Jetson GPIO + PWM motor control)
- Concurrent server architecture (multi-threading and process orchestration)
- Networked multimedia systems (RTSP, MJPEG, HTTP streaming)
- Edge computing / robotics (real-time feedback and control over local network)

3.6.1 Flask

Flask is a web framework that is used to develop web applications. It is written in Python. It is lightweight and simple. It is built so that developers can develop an application quickly without complexity. Its API is simple so that web routes that handles requests can be created simply and quickly. It can be used for HTTP-requests [17].

In this project Flask is used for image capture from camera1 and live-streaming from camera2. The user requests gets routed. When the user sends a HTTP-request to the server, the server calls the corresponding function and returns a response to the client. For the streaming of camera2, MJPEG-streaming over HTTP is used. MJPEG (Motion JPEG) is a format of video compression that each frame of the sequence is a JPEG image [18].

3.6.2 Socket Server

Communication that are between different processes requires an interface (or gate). Socket is a such an interface. Network sockets can be described as connection handlers. A socket can be called a connection endpoint, that is like a traffic destination. Sockets are bound to an IP-address and a port number on the host machine. In the model of client-server, the server starts a socket on a specific port and waits for a client. When a client connects, the server accepts the client and waits for requests. The server responds with a response to any request from the client. There are different protocols that can be used for the communication e.g. TCP, UDP [19].

In this project, the socket server listens on port 9999 and accepts a client via TCP connection for commands that are about the motor control. Each client runs in its own thread for concurrency. Commands for motor control are processed asynchronously.

3.6.3 GPIO and PWM Motor Control

As shown in figure 3.6, the Jetson Orin Nano has 40 GPIO-pins. The pins include, voltage level pins (3.3V and 5V), GND, UART-pins, SPI-pins, GPIO-pins, I2C-pins, and I2S-pins. All of the pins are connected directly to the Jetson module, except for the I2C pins. There are 2 ways to address the pins. Either via BCM-mode or BOARD-mode. BCM is the way the Nvidia chip refers to each pin and BOARD is the actual pin-number. You can use the GPIO-pins to control the motor driver (MDD3A) that runs the motors. This by setting/triggering a signal via the GPIO-pins. The GPIO pins only have two PWM channels that are connected to hardware PWM controllers, and software-generated PWM is not supported. However, it is possible to configure certain pins as PWM-signals [21].

In this project, 2 GPIO-pins are used for every motor. One high/low signal for the direction of the motor and one PWM-signal for the speed of the motor. Two additional ground-signals are used and a 5V-pin for the motor driver.

3.7 Real-Time Video Streaming

The process of video (and audio) transmission via internet in real-time or nearly real-time is called live streaming. To deliver low latency video, it contains several processes. The events starts with video capture by a source (e.g. a camera). The next step is about encoding the raw video signals to digital format, so that other devices can handle the data. Examples of standards for encoding are H.265, H.264, MP3, VP9, and AAC. After the encoding, the digital data is compressed so that it is well suited for transmission via internet. Live video data contains enormous amount of data. Therefore, the streaming tools split the data into segments. The user (that receives the data) uses some kind of player (e.g. Media Player) to decode and decompress the data-segments, enabling continous playing of the live stream. Many media-players has adjustable bitrate setting, that can be used to set video

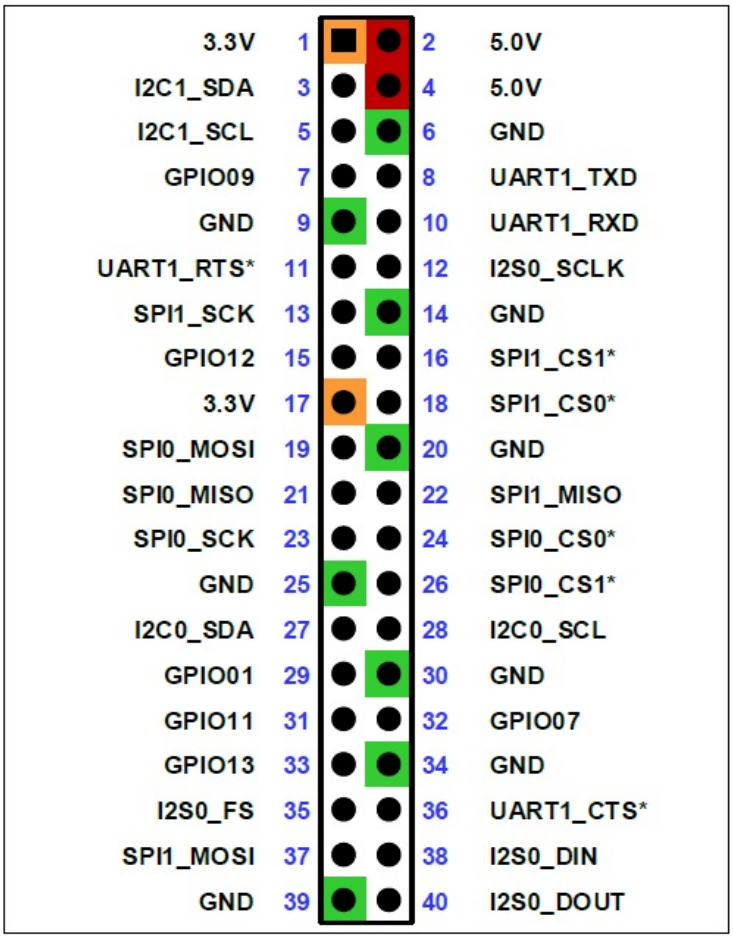


Figure 3.6: Expansion Header Connections. Jetson Orin Nano Developer Kit carrier board overview. (from page 21, Figure 3-1 in [20]).

quality depending on the internet speed. There are different protocols for delivering video data via internet. Some examples are RTSP, SRT, WebRTC, RTMP, HLS and DASH. Every protocol has its own application [22]. One major challenge for live streaming is the latency, that is the time between video capture until it is shown to the user. As one might conclude, the speed of internet, the speed of processing the data, the kind of network connectivity (e.g. 4G, 5G), amount of buffering and some other properties, affect the latency and the user experience.

3.7.1 Real-Time Streaming Protocol (RTSP)

RTSP is designed to control media streams over a network, allowing playback in real-time without requiring the full media file to be downloaded locally. RTSP provides lower latency and is optimized for streaming video (especially over networks). It is used in surveillance systems, drones and much more. RTSP supports request commands such as "setup", "play", "stop", and can be played by media players such as VLC Media Player and QuickTime Player [23].

RTSP establishes and controls streams of time synchronized media like video or audio. In multimedia servers, RTSP acts as the remote control of networks. When the server starts, it setups a connection between the server and the camera. After the setup, video can be sent using RTP [24].

RTSP uses TCP for control, but the media that is controlled by RTSP can use TCP or UDP. RTSP has not support for encryption of data or retransmission of not received media packets. It does not directly stream to browsers over HTTP. This is due to the fact that RTSP connects to a specific server for the streaming and uses RTP for transmission of media [25].

For streaming a RTSP-stream (e.g. in VLC player), you can write the following as the stream address:

```
rtsp://serverip:port/mystream
```

I.e. instead of a HTTP stream, a RTSP-stream is used.

3.7.2 FFmpeg process

FFmpeg is a media converter. It reads different kinds of inputs and can convert to many kinds of outputs. FFmpeg is a multimedia framework. It can encode, decode, transcode, stream, play, mux, demux and filter many different types of data. FFmpeg supports many formats, from old formats to contemporary cutting-edge formats. An usal FFmpeg that comes with a Linux machine supports 370 formats and 460 codecs. FFmpeg can open streams, read a socket or a pipe and read from a file. It can also output to a stream, to a file, to a socket or a pipe [26].

The process of FFmpeg begins by opening the source (e.g. a file or a stream). Then, it demuxes the source into encoded data packets. Then, the packets are decoded into decoded frames. The decoded frames are pixel format that is called "RGB24" and contains blue, green and red values. After that, the decoded frames are sent to the encoder which encodes into data packets. Finally, the encoded data packets are muxed to an output stream or file [26].

3.7.3 MediaMTX

MediaMTX (formerly known as rtsp-simple-server) can be used as a live media server or as a media proxy. MediaMTX has been developed and used as a media router. It can route a media stream from some end to another end. It can publish and read streams that are live with media stream protocols/transport formats such as RTSP, SRT, HLS or RTMP. When using MediaMTX, streams can automatically be converted from one protocol to another one. It features reloading of the configuration without disconnection of connected clients ("hot reloading"). MediaMTX is compatible with Windows, macOS and Linux. It can also be used for recording streams, authentication, forwarding streams, proxy requests and much more [27].

3.7.4 OpenCV

OpenCV (Open Source Computer Vision Library) is a big computer vision library. It contains more than 2500 algorithms. It can be used for computer vision and in machine learning software. The vast amount of algorithms can be used for live-video, image capture, finding similar images in an image database, tracking camera movements, identifying objects, face recognition, tracking moving objects, extracting 3D models of objects and much more. OpenCV can be used by Python, C++, MATLAB and Java. It supports Linux, MacOS, Windows and Android. It is mostly used by applications that use real-time vision [28]. Examples of usage is in smartphones, big tech-companies and even NASA. Other fields that use OpenCV is in medicine, self-driving vehicles, biotech, sport and metaverse [29].

One of OpenCV's features is video processing. Since videos consists of several frames in sequence, thorough handling of frame rates and smooth processing are required. This demands dynamic frame management and efficient computation. OpenCV's `VideoCapture()` function allows users to load video and access camera streams in real-time. The `VideoWriter()` function assembles the sequence of frames and encodes them to a desired format. The frame rate, resolution and encoding options are customizable [30].

3.7.5 GStreamer

Gstreamer is a multimedia framework that is based on pipeline. GStreamer works in a way that it processes some media by connecting some processing elements to a pipeline. It can link a media processing system to a complex workflow. It can be used in a system where it reads data (e.g. a file, a stream) in some format, processes the data and outputs the data in another format. GStreamer supports many types of components that handle media, e.g. streaming, recording and video and audio playback. The pipeline design of GStreamer makes it possible to use it in applications like transcoders, video editors, streaming broadcasters and media players. It works in Windows, Linux, macOS, Android and iOS. It has bindings for Python, C++, C# and many more languages [31].

3.8 Android app development

Android studio is the official IDE for developing Android apps. It has all neccessary tools for developement of an Android app. It has the tools for code developement, deployment, testing, version control and much more [32]. Besides a code editor, it has UI designer and posibility for one or more emulators. Android studio can be run on Linux, macOS or Windows. While the main code can be in Java or Kotlin, the code for UI elements is in XML. There are hundreds of different phones from many different manufacturers that use Android. When developing Android apps, one has to consider adapting the app so that it can run on most of the devices [33].

Android studio has a build system that compiles the source code and the app re-

sources and packages them to Android app bundles or APK's that can be tested, signed or deployed. Gradle is the central build system of Android Studio. While Gradle is used in every app that is made in Android Studio, it has many different customizable parameters for different projects. Android has different API levels from level 1 to 36 (until now, 2026). When developing an app in Android Studio, one have to set the compileSdk, the minSdk and targetSdk [34]. The reason why everyone doesn't set the sdk's to the latest is that 100% of all Android users doesn't use the latest Android version, and by setting a lower sdk might mean that the app can run on more Android devices.

3.9 Power supply

The correct power supply for the Jetson is crucial for correct setup and operation. Incorrect power supply may cause damage to the board or would lead to performance that is unstable [35]. Input voltage from the DC barrel jack is 9-20V. A safe power supply that can deliver 3.5 A is required [36].

The battery that is used is a Sandberg Powerbank USB-C PD 130W 50000 mAh. A picture of the ports and connections of the battery can be seen in figure 3.7. It is rechargeable and has different outputs. The input of Jetson should be connected to the correct output of the battery. The GPIO-pins of the Jetson has both 3.3V pins and 5V pins. The input voltage of the motor driver should be connected to the correct output of the battery and the logic input of the motor driver should be connected to a 5V pin of the Jetson. The cables that connect the Jetson and the motor driver to the battery should be able to provide the correct amount of voltage and current (i.e. using any kind of cable won't work and may cause damage). In addition, the Jetson and the motor driver should have common ground.

3.10 Environmental sensor

The Adafruit SHT45 Trinkey environmental sensor is half USB stick and half temperature and humidity sensor. It is designed for easy integration into any computer with a USB-A port. It is powered by the ATSAMD21 microcontroller, that ensures seamless operation with minimal circuit drift. It has a reset button that enables entering bootloader mode whenever neccessary [37]. It can output data as float or CSV via a serial/COM port.

Its operating range is 0 to 100 % for humidity and from -40 to +125 °C. The SHT45 has a ±1.0% humidity accuracy from 25 to 75% and ±0.1°C accuracy from 0 to 75 °C. However, due to self-heating, the measured temperature may be some degrees higher than the actual ambient temperature [38]. Other nearby components' heat may also affect the measured temperature.

To use the sensor, one has to connect the sensor to the device, put it on bootloader mode, download and transfer the needed libraries to the sensor, write a send-code

Figure 3.7: Picture of the battery's ports and connections.

that makes the sensor send data at an interval and then restart. So you write code for Trinkey that runs CircuitPython and reads the SHT45 via I^2C on the Trinkey itself. It then sends data over USB-serial to the device. After that, one has to write a code that can retrieve the continous data from the sensor.

3.11 Thermal camera

The Thermal-90 USB Camera is an infrared (IR) thermal imaging camera that operates in the long-wave spectrum. It uses technology of thermopile pixel and microbolometer and features a resolution of 62 × 80 pixels. It detects infrared distribution of objects and by calculation turns the data to surface temperature, and by this it can generate thermal images. It is used for online temperature monitoring, for safety and security, for motion detection, as an IR thermometer, in smart homes and more [39]. Thermal cameras are also used by firefighters. By rendering the IR-radiation, firefighters can see heat through darkness, through smoke or other barriers. Thermal cameras can see body heat and are therefore used in situations where humans are trapped in a building and the rescuers tries to find those people. A thermal camera has an optic system, an amplifier, a detector and a signal processor. The combined work of the components of a thermal camera enables rendering of

infrared radiation. The image produced by a thermal camera is based on infrared differentials, meaning that two things with equal temperature appears with the same colour [40].

The Thermal-90 USB Camera can be used by any Android device, any PC or any other device that has an USB port. The Thermal-90 USB Camera uses a MI48x3 with Senxor Bus processor. Its maximum video-stream rate is 25 frames per second (fps). It operates on 5V and 61 mA. Its operating temperature is from -20 to 85 °C. But its target temperature is from -20 to 400 °C. Its accuracy is ±2 °C (temp. 10 to 70 °C) [41].

4

Construction and developement

The work began with an initial stage, where some research was made about building a robot. Some reading was done about how a robot is built, what hardware is needed, what software is needed and what tools are needed. Also, some research was done about the challenges and the possible solutions of building a robot. The work was carried out in a modular manner, adding one feature/module in every step.

In the following sections, the detailed construction and development process are described, beginning with a full system overview.

4.1 System overview

A picture of the robot and the app is shown in the cover page and a picture of the app in thermal camera mode is shown in section 4.15. In this section, a full overview of the final hardware and software is presented.

4.1.1 Hardware

The hardware and components used are the following:

- Jetson Orin Nano
- Rechargeable battery
- Zed2i camera
- Chassis with caterpillar tracks
- 2 DC-motors
- Motor driver
- Modem
- Raspberry Pi camera

- Environmental sensor
- Thermal camera

A hardware component diagram is shown in figure 4.1.

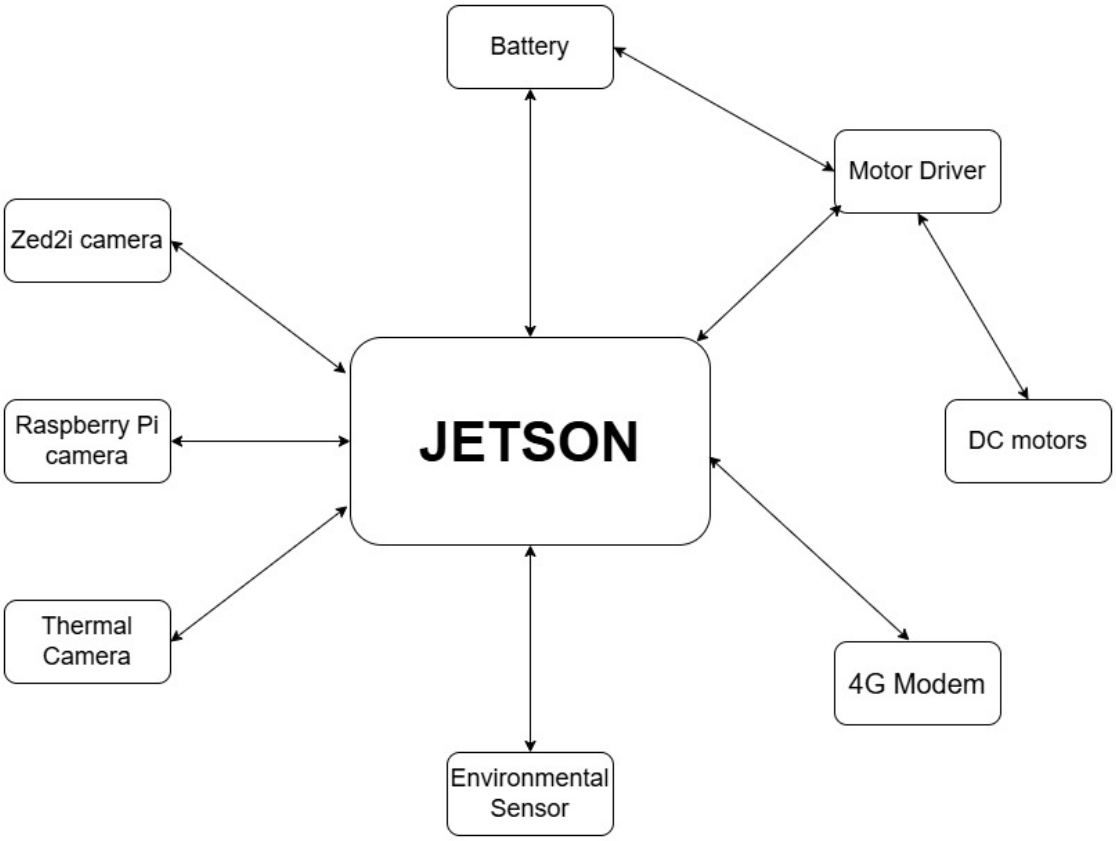


Figure 4.1: Overview of the hardware components and how they are connected.

The components were mounted and connected to the chassis. Some of the components had USB-connection and some a more complex connection. As explained in section 4.3, the Zed2i camera was connected to the Jetson. The Jetson and the motor driver were connected to the battery according to section 4.4. The DC-motors were connected to the motor driver and the motor driver was connected to the Jetson, which is described in section 4.5. As described in section 4.9, the modem was connected to the Jetson. The second camera was connected to the Jetson and that is described in section 4.13. The temperature sensor was connected to the Jetson, which is described in section 4.14. As written in section 4.15 the thermal camera was connected to the Jetson.

4.1.2 Software

The software architecture used is the client-server architecture. The software used can be divided into the following:

- Server code
- App code

The complete source code for the server and the app is publicly available on GitHub.
The server implementation is available at:
https://github.com/rezamars/RobotServer
and the mobile application implementation at:
https://github.com/rezamars/RobotApp

A high-level system architecture diagram is shown in figure 4.2.

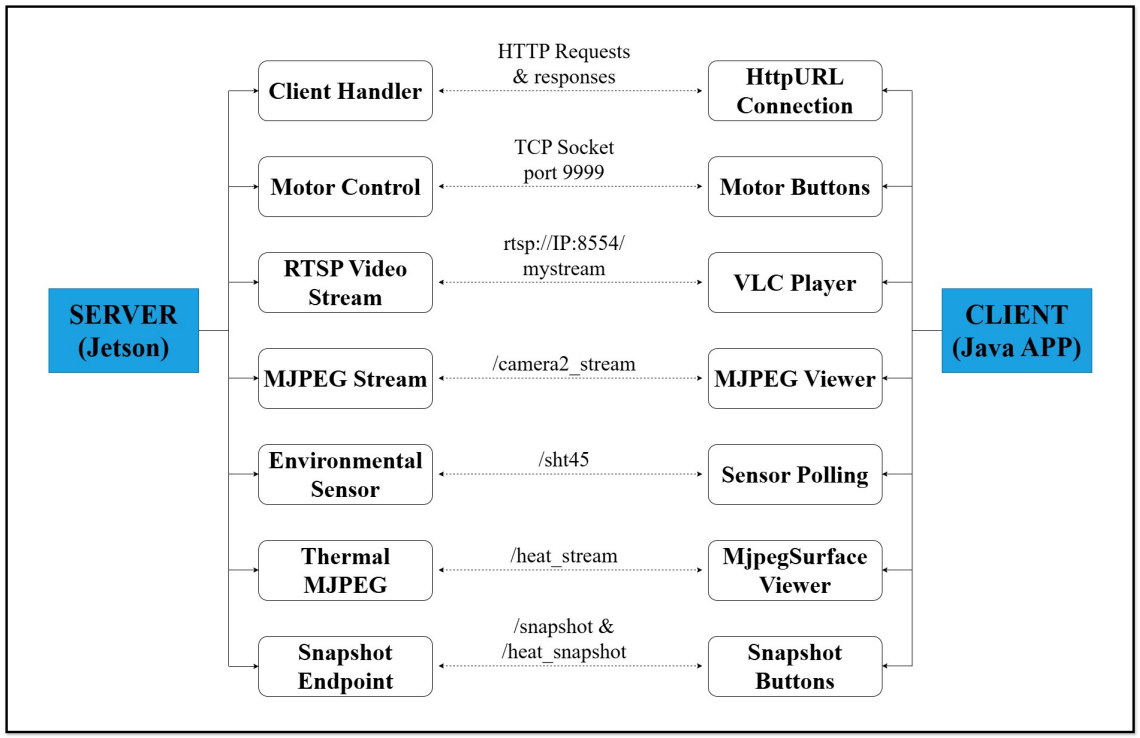


Figure 4.2: Overview of Jetson-Based Server and Java Client Communication flows.

4.1.2.1 Server

The server is written entirely in Python. Although the server code is one file, it contains many different functionalities.

Server high-level code overview:
The server code is a full robotic control and video-streaming server running on an NVIDIA Jetson Orin Nano.

The server code integrates:

1. MediaMTX RTSP video streaming

- Streams video from a USB camera using FFmpeg → MediaMTX RTSP server.
- Provides robust reconnection, watchdog, restart logic.
- Flask endpoint provide snapshot.

2. Jetson IMX219 camera support
   - A second camera uses GStreamer through `nvarguscamerasrc`.
   - Flask endpoint provide MJPEG stream.
3. Motor control over GPIO
   - Uses Jetson.GPIO to drive two DC motors with PWM speed control and direction pins.
   - Includes command queuing, emergency stop, internet watchdog.
4. Socket command server
   - A TCP server listens for movement commands: forward, backward, left, right, stop, speed commands, etc.
5. Environmental sensor functionality (SHT45)
   - A dedicated thread continuously reads temperature and humidity over a serial connection.
   - Automatically reconnects if the USB/serial link is unplugged or unavailable.
   - Stores the latest readings in protected shared variables using thread-safe locking.
   - Flask endpoint `/sht45` exposes the latest sensor values as JSON.
   - Provides fallback values and robust parsing to prevent crashes on corrupted sensor data.
6. Thermal camera system (SenXor MI48)
   - Connects via serial/USB using the SenXor MI48 SDK.
   - Streams raw thermal sensor matrices at ~15 FPS.

- Applies noise filtering, histogram equalization, and color mapping (`OpenCV COLORMAP_JET`).
- Automatically reconnects if the thermal camera cable is unplugged or communication fails.
- Flask endpoints:
  - `/heat_snapshot` – returns a JPEG thermal image
  - `/heat_stream` – continuous MJPEG thermal video stream
- Uses rolling-average temperature filtering for stable min/max thermal scaling.
- Fault-tolerant frame acquisition with full auto-recovery on read errors.

7. Watchdogs and stability systems
   - Internet watchdog (stops robot if connection lost).
   - RTSP watchdog (restart FFmpeg if video freezes).
   - MediaMTX restart logic.
   - GPIO lockfile handling.
   - Capture retry loops.
8. Flask web server
   - `/snapshot` returns latest RTSP frame as JPEG.
   - `/camera2_stream` streams IMX219 camera.
   - `/sht45` exposes environmental sensor data.
9. Multithreading everywhere
   - Motor control thread
   - RTSP video capture thread
   - IMX219 camera capture thread
   - Thermal camera capture thread

- Environmental sensor reading thread
- Socket command server thread
- Flask web server thread
- MediaMTX logging thread
- RTSP watchdog thread
- Internet watchdog thread

The architecture ensures high responsiveness, real-time control, and automatic recovery from hardware or network faults.

4.1.2.2 App

The app is written in Java alongside XML for the UI elements. The app code has two activities containing many different functionalities, two XML-files and one manifest file. The app provides live video feeds, robot control, thermal imaging, snapshots, sensor data, and robust connection handling.

High-level overview of the robot control app:

1. Video streaming (live camera feeds)
   Provides multiple real-time video perspectives from the robot.

   A: RTSP Stream (Main Camera)
   - Uses VLC (LibVLC) to display a live RTSP video stream from the robot.
   - This is the primary low-latency camera feed.
   - VLC is chosen because Android does not natively support RTSP well.

   B. MJPEG Stream (Secondary Camera)
   - Uses manual parsing of MJPEG image data.
   - Displays frames in an ImageView.

   C. Thermal Camera Stream (Heat View)
   - Uses the `niqdev/MJPEG` library to display the robot's thermal camera feed.
   - Connects to the robot server's thermal endpoint: `/heat_stream`
   - Supports:
     - Automatic reconnection if the stream drops

  - Timeout handling
  - `BEST_FIT` scaling and optional FPS overlay
- Runs in its own activity (`HeatActivity`), giving a dedicated thermal-camera interface.

2. Robot control (movement & speed)
   The app sends text commands to the robot's server using a simple TCP socket connection, allowing the user to drive and control the robot remotely. Examples:
   - Forward, backward, stop.
   - Left, right.
   - set25, set50, set100 (speed control).
3. Snapshot capture
   The app can take a still image from the robot's cameras.
   A. Main camera snapshot
   - Downloads a still JPEG from the robot's `/snapshot` endpoint.
   - Saves the file to the phone's gallery via Android's MediaStore.

   B. Thermal camera snapshot
   - Retrieves a thermal frame from `/heat_snapshot`
   - Automatically saves it to the gallery with a timestamped filename.
   - Uses Volley's `ImageRequestfor` efficient background downloading.

   Both snapshot systems run off the main thread to ensure smooth UI interaction.
4. Environmental sensor data (temperature & humidity)
   The app periodically fetches environmental readings from the robot's SHT45 sensor.
   - Retrieves JSON data from the server's `/sht45` endpoint.
   - Parses temperature and humidity values.
   - Updates on-screen TextViews every 5 seconds.
   - Runs entirely in a background thread to avoid blocking the UI.
5. Connection management
   Ensures robust video connectivity even under unstable network conditions. The app handles:
   - RTSP video interruptions.
   - Automatic reconnection attempts.
   - Retry logic for MJPEG and thermal MJPEG streams.

- UI notifications when reconnection is in progress.

6. User interface
   Everything is arranged vertically to create a simple control console. The UI consists of:
   - Main video window (RTSP)
   - Snapshot button
   - Movement control
   - Speed controls
   - Status text
   - Secondary video window (MJPEG)
   - Live temperature and humidity readouts
   - Thermal camera access button / separate thermal view (MJPEG)
7. Non-blocking operations
   All networking (video, control commands, snapshots, sensor fetches) runs in background threads to keep the UI responsive.

4.2 Installing OS

Before connecting anything, it was necessary to make sure that there was access to the settings and the features of the OS on the Jetson. So, the microSD-card (containing OS) that was attached to the Jetson was formatted. According to many clips and sites on the internet, one had to download a Jetpack image and then flash it on the microSD-card in order to install Ubuntu on the microSD-card. Flashing can be done with a software such as BalenaEtcher. Dozens of different images from Nvidia and other places on the internet was tested, without any success. After additional tests, readings and trouble shootings, a notice was made, that Nvidia recommended that one should use a native Ubuntu machine for the flashing. Then, an attempt was made to flash via WSL (a program where the user has access to Ubuntu in Windows). After writing a post on Nvidia's forum about this, a recommendation was received to use Nvidia's SDKmanager for the flashing. An additional advice was that one need to put the Jetson in `Recovery mode` before flashing the microSD card. One should put a jumper between pin9 (GND) and pin10 (FC REC) on the 12-pin part (not the 40 pins) between the card and the fan. Without having the jumper, pin9 and pin10 was connected with a regular cable. When the pins were connected, the power was turned on, and after 2-3 seconds pin9 and pin10 was disconnected. Then, the Jetson was in `Recovery mode`. After that, the flashing could continue. According to the advice from Nvidia, it should work via WSL and an additional link with instructions about some extra things that needed to be done to make it work with WSL was provided. Apt, WSLU, winget, usbipd-win, lots of extensions for WSL, linux-tools-vitual, SDKmanager and APX driver was installed. Via a USB-C to USB-A cable the Jetson was then connected to the computer where the WSL was

installed on. The device (Jetson) in Recovery mode was visible on the computer. Then, the following commands were executed (with the computer's BUSID):

```
usbipd bind --busid <BUSID> --force
usbipd attach --wsl --busid=<BUSID> --auto-attach
```

When running SDKmanager (via WSL), the program should auto-detect the Jetson device. And it did. When running SDKmanager, it automatically found the Jetson. After running SDKmanager via WSL, all downloads and installations went well except for the "Flash Jetson Linux"-part. And the program ended with "installation failed". After that, installing and flashing was tried in several different ways. By changing Jetpack version, changing USB port, entering name and password, not entering name and password, target hardware (you could choose between 2), installing different components, not installing components, disabling the firewall via:

```
sudo ufw disable
```

Then, formatting the microSD card according to different formats (ext4, FAT32, exFAT ...), reinstalling `nfs-kernel-server`, deleting the `/etc/exports` folder. Still, it didn't work.

After that, the initial advice was followed and tested, namely flashing via a native Ubuntu machine. Ubuntu 20.04 was installed on another computer. SDKmanager and some additional software so that the computer could detect the Jetson was installed. While Jetson was connected to the Ubuntu-computer, it was put in `Recovery Mode` and the SDKmanager and the process was started. The process went well, the downloading and the flashing part worked as intended (on the first try on the native Ubuntu). Then, the Jetson was restarted and the OS installation was completed. Now, the microSD-card had an OS on it. When the OS installation was done, the Jetson was mounted on the upper side of the chassis.

4.3 Connecting camera1

The Zed2i camera has a USB-A cable. The camera was placed on the chassis and connected to a USB-A port on the Jetson. The ZED SDK and the Qt5 libraries was installed on the Jetson. By opening ZED Explorer on Jetson, the live-video of the camera could be seen locally. Before beginning writing the server code, it was necessary to install Python3 libraries, OpenCV libraries and Flask. After installing these, a simple server code (in Python) that sends live-video and that can take a picture was written. In the code it was necessary to refer to the camera's video interface `/dev/video0` or `/dev/video1`. From time to time, sometimes the camera got the reference `video0` and sometimes `video1`. So, it was necessary to check which reference was the actual one. When running the server code and by writing the following on some other device's browser (in the same WiFi), live-video and picture was shown.
Live-video:

http://server-ip:port/video_feed
Picture:
http://server-ip:port/snapshot

When the live-video and image capture worked with a browser, the work continued with developing the app (in Java) that could receive live-video and that has a take picture button. After writing the code, the app was installed on a phone (a Samsung galaxy) via Android Studio. The live-video starts when the app starts and when clicking on the take picture button, the picture is saved on the phone. The app code consisted (until this point) of two Java files, one XML file and a Manifest file. To allow connection via the internet and allow connection to the HTTP endpoint for live video and snapshot, it was necessary to include the following in the Manifest file:

```
<uses-permission android:name="android.permission.INTERNET" />
```

At this point, the take picture functionality worked, but not the live-video. In the app code, an own class for `MjpegView` was written. That class handled the creation of an `ImageView` and the display of the video and the images. That class used an `ImageView` from `android.widget`. which is a package from the Android standard library. It turned out that that class was not sufficient. To use `MjpegView` the following command was used:

```
import com.longdo.mjpegviewer.MjpegView;
```

After using `MjpegView` which is a class included in an external Android library, both taking pictures and live video did work as they should. It was necessary to add the following to the `build.gradle.kts(:app)`-file, so that the program has access to the external library:

```
implementation("com.perthcpe23.dev:android-mjpeg-view:1.1.3")
```

4.4 Battery connection

Until this step, the Jetson had been connected to a power outlet via an adapter. To later handle motor control, it is necessary to connect the motor driver to the battery, connect the DC motors to the motor driver, and also later connect the Jetson to the battery.

After some research, it was found that one must first and foremost make sure that the voltage and power are compatible. The battery has 3 USB-C and 3 USB-A ports. Since only USB-C to barrel-jack cables were available, the three USB-C ports on the battery were investigated. One of the ports are OUT, one is IN and the third one is IN/OUT. After some research, it was found that the recommended port for the Jetson battery is the USB-C IN/OUT 3, as it supports up to 100W (20V/5A) and can deliver enough power for the Jetson Orin Nano. Jetson requires

Minimum 5V DC, Minimum 4A current (depending on load), power connector that is barrel jack (typically 5.5 mm/2.1 mm center-positive) and a cable/adapter that must request 5V or 12V from power bank via PD. PD stands for Power Delivery, a communication protocol used by USB-C devices to negotiate voltage (V) – for example 5V, 9V, 12V, 15V, 20V, and for Current (A) – for example 1A, 2A, 3A, up to 5A. A research was made to find out which power cable should be used (12V 5A, 20V 5A or something else). The cable should be a USB-C to 5.5 mm/2.1 mm DC adapter cable that must support PD. The conclusion was that the 12V 5A cable is the best choice. The Jetson requires 5V, but many use step-down modules or run directly on 12V (e.g. via a regulator on the board). The 20V 5A cable may be too much for the Jetson Orin Nano – there is a risk of damaging the board if it does not have protection or a regulator for 20V. Based on the specifications of the Jetson, one can conclude that: Jetson Orin Nano operates at 19V, which is a clear sign that it has a built-in regulator that can handle voltages up to at least 19V. Then, 12V 5A cable is a safe choice. The battery was placed on the chassis and the battery (port3 IN/OUT, 100W) was connected via the 12V 5A cable to the Jetson. Now, the Jetson gets power from the battery and it works as it should.

Next part is about how to make sure that the motors get power. A research was made about how to connect the motors to the Jetson. It was found out that, the Jetson's GPIOs are just little logic signals—not power outputs. A DC motor is an inductive, current-hungry load. Connecting it directly risks burning the Jetson. After realizing that one can't connect the motors directly to the Jetson, it was concluded that, it is necessary to use a motor driver. The motor driver should be connected to the battery, the Jetson to the motor driver, and the motors to the motor driver. This meant that the Jetson and the motor driver has separate power supplies and that the motors get their power from the motor driver. The motor driver was a Cytron MDD3A type, it can handle up to 3A per channel and works with 3.3V or 5V logic, which is suitable for the Jetson Orin Nano. It was necessary to make sure that the ground (GND) between the Jetson and the motor driver is common. After some research, it was found out that the motors are 9 V DC gear motors with built-in encoders. The motor driver can handle 4–16 V and 3 A continuously $\rightarrow$ the 9 V motors are a perfect fit, and you can also run them on 12 V if you want more speed (as long as the current is kept below the limit). The MDD3A's VB+ and VB- was connected via barrel jacks with a barrel to USB-C 12V 5A cable to the battery's OUT 1 30W. At this point, the Jetson, the MDD3A and the motors got the required power.

During the project, there was a problem with the battery going into standby-mode. With the motor driver connected to the battery, if one didn't use motor control for more than 15 seconds, the battery went to standby-mode. In standby-mode, motor control didn't work and one had to reconnect the motor driver to the battery. After some search for how to solve the problem, different kind of suggestions was found. Like e.g. using a different battery, adding a permanent small load across the port/output that feeds the MDD3A so that it draws little power but keeps the port awake. Other suggestions were using a proper USB-C PD sink/adapter

that negotiates 12V/PD with the power bank, so that if the MDD3A receives 12V via a PD negotiated channel, the power bank can keep that port active. Another suggestion was building a load resistor 240 Ω 1 W (or 120 Ω 2 W for safer wake-up) and connecting it between MDD3A's VB+ and VB- so that this keeps the gate awake and the MDD3A on all the time. While considering all suggestions, an attempt was made to program the Jetson to pulse motors at regular intervals (every 15 seconds) and it didn't work. Also, a test was carried out with different times for the pulsation (in the heartbeat method) and different percentages for changedutycycle. It was found out that it doesn't work with time less than 1 second and the percentage didn't work because it always became HIGH or LOW. One specific pulsing code worked, but in that code PWM is not used at all and the pin was set to HIGH for at least 1 second. But, that's not how the code on the server works, since PWM is used, 1 second is too much as it will rotate the motor and in drive mode the robot will go forward/backward. However, at one time when testing the battery's ON-button, it was discovered that if the ON-button is held down for a few seconds, the battery goes into a state where it is active all the time. Later, it was discovered that when it is in the active-state, the current strength is affected (reduced) and when driving a robot, the speed is halved. Since speed is not the main concern in this project, half speed is acceptable.

4.5 Motor control

To implement motor control, first the motor driver was placed on the chassis. After that, the Jetson and the motor driver were connected to the battery and the motors were connected to the motor driver. Every motor has 6 pins. Only pin1 and pin2 are used in this project. Pin1 is for PWM (motor speed) and pin2 is for direction. Then, the outputs of the motor driver were connected to the motors. That is connecting M1A to pin1 of motor1, M1B to pin2 of motor1. M2A to pin1 of motor2 and M2B to pin2 of motor2. Every motor has two test-buttons, one for forward and one for backward. A test was done with the test-buttons on the motor driver to see if the connection between motor driver and the motors were correct, and it was visible that the motors ran as they should.

To be able to implement control signals from the Jetson to the motor driver (and thereby the motors), an additional test was done. The test was based on sending a signal that made one motor rotate continuously. The motor driver has 6 pins (on the top), 2 pins for every motor, a ground-pin and a 5V logic pin. The M1A was connected to the 5V pin and M1B to GND. As the Jetson connected to the battery, the power of the motor driver was connected. It was evident that motor1 rotated continuously (until disconnection).

Next step was about how to connect Jetson to MDD3A so that an app can be used to make the robot go forward, backward, turn and set speed. At first, a correct connection was essential. Connecting Jetson to the motor driver was one of the major challenges in this project, i.e. what pin on Jetson should be connected to which pin on the motor driver. For motor control via app, it can be divided into 3

parts:

1- Hardware connection Jetson Orin Nano → MDD3A
MDD3A can be controlled with PWM + DIR signals (or serially, but PWM is easiest from Jetson `GPIO`).
On Jetson Orin Nano `GPIO`:
- 2 PWM outputs (one per motor)
- 2 digital direction pins
- Ground
- 5V logic signal

2-Jetson Orin Nano – software:
Python server code to control the motors via `GPIO`.
On Jetson, the `Jetson.GPIO` library is used.

3-Android app communication:
An app that sends control signals.

At first, it was necessary to confirm that Jetson can set the four control signals and that MDD3A responds. A simple test was done, that is just setting a pin to HIGH and then measure (with a multimeter) if the pin gets the signal. The following code snippet was executed to set pin17 of Jetson to HIGH and LOW:

```
sudo gpioset gpiochip0 17=1
sudo gpioset gpiochip0 17=0
```

At first, it didn't work. After some reading, it was found out that `GPIO` has two modes, `BCM`-mode and `BOARD`-mode. In `BOARD`-mode the pin-number refers to the actual pin-number. But in `BCM`-mode, the number refers to the manufacturers number and pin17 maps to pin number 11 (physical number). So, the code snippet was executed again. When the pin11 was measured, the voltage was around 3.4 V after setting it to 1 and 0 V after setting it to 0. That meant that the signals went as expected.

After many tests, problems and different connections, the work continued with a specific setting for the pins. Before connecting the 6 `GPIO` pins to the motor driver, it was necessary to check which of the 40 `GPIO` pins of Jetson could be used for PWM-signals (all pins cannot be used for PWM) and which for direction signals (HIGH/LOW). A hardware configuration of the pins was also necessary. After some reading and research, the work continued with the setup shown in Table 4.1. The table shows which Jetson pins were connected to which pins on the motor driver (including cable color to keep track of the cables):

For hardware configuration of the pins, it was necessary to add a mark for the pins used after running the following command in a terminal:

Jetson GPIO-number	Cable Colour	MDD3A	Function
15	Blue	M1A	Motor1 PWM
22	Darkbrown	M1B	Motor1 DIR
6	Lightbrown	GND	Common ground
33	Red	M2A	Motor2 PWM
18	Gray	M2B	Motor2 DIR
2	Violett	5V0	MDD3A logic input

Table 4.1: Jetson to MDD3A signal connection.

```
sudo /opt/nvidia/jetson-io/jetson-io.py
```

Figure 4.3 shows a picture of the page where the configuration was done.

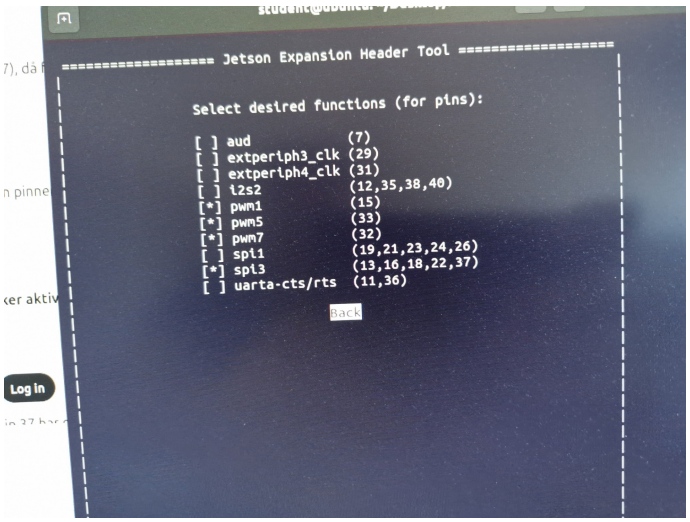


Figure 4.3: Configuration of the pins used.

At this point, the motors could be controlled by signals (forward and backward) from the server code. Then, another problem was encountered, namely that when the battery was connected to the MDD3A, the motors started rotating (without any signals being sent). This meant that the signals are HIGH from start. Then, there was a need to run a code at start that sets them to LOW. To solve this, 2 code files, `gpio_idle.py` and then `motor_control.py` was written, where the first one should set all pins to LOW and the second one should wait for a signal and drive the motors. The next problem was that the two files couldn't run at the same time. A "device or resource busy" error-message was out-printed. After some searching, it was found out that one can create a "lock file" when `gpio_idle.py` runs. When `motor_control.py` runs, `gpio_idle` is terminated, the "lock file" is deleted and the `GPIO` is released. It was necessary to install `psutil` to make it work.

The next step was about the functionality for turning (right, left) and about how to change speed. These two functionalities are related since turning for a robot with caterpillar tracks happens when the speed of one motor is less than the other one. To implement speed change it was required to set percentage level in the

`ChangeDutyCycle`-function for the PWM pins. Functionality to the server was added so that the speed of the motors (for forward) could be set to 25, 50 or 100%. At first, there was a problem with the motors rotating in the opposite direction, but when changing the cable from Jetson to MDD3A (pin 33 and pin 18) the motors now rotated in the correct direction (swapped the gray and red cables on Jetson). Changing the cable is a valid and common way to solve it. By this, the motor was turned electrically. For backwards, setting the speed did not work. After some reading it seemed that, 4 PWM pins was needed, but on this Jetson there are only 3 PWM pins. A test with 3 PWM pins was conducted, without any success. At the end, it was sufficient being able to drive forward at three speeds and backwards at only one speed. For turning, the PWM pins was set at different percentages, more specifically setting 100% speed for the opposite motor and 50% speed for the motor of the turning side. E.g. for turning right, the PWM-signal for the left motor is set to 100% and the PWM-signal for the right motor is set to 50%.

When writing the server code for motor control, a time property was set for all the different driving commands. For forward, the time was set to 10 seconds. I.e. when a drive forward signal is received, the robot drives forward for 10 seconds, and then stops and waits for a new command. The time for backwards, turn left and turn right is 2 seconds. But if the robot is driving forward and a right turn signal is received, it turns right for 2 seconds and then continues forward. These time properties is mostly for safety reasons (if for some reason the motor control is disconnected or out of control, the robot won't crash with some obstacle).

After testing all signals (forward, backward, turn left, turn right, stop, set25, set50, set100) locally on the Jetson and making sure that all signals work, the work continued with the server-code for motor control, that should wait for signals that control the motors. First, a TCP server was developed that starts a server on 192.168.1.100 port 9999 and waits for commands. The server listens on the port for TCP connections. When a client connects, it can send text commands ("forward", "backward", "left", "right", "set25" ...). The server runs the corresponding motor function and sends back status messages. The exit command ends the client session (but the server continues to run). Then, another version of the app (that is supposed to control the robot) was developed. The app had 8 buttons (for all the signals). When the server was running and the app started (with the phone on the same Wi-Fi as the server), the app connected to the server via a socket-connection using the server's IP and port. After that the connection was established, the server waited for commands. Then all signals were tested, and all worked as they should. But, there was a limitation, namely that while a command is running you can't send a new command. Then, the server code was changed so that when a command is received, it cancels the previous command and implements the new one. This meant that you can send a new command at any time and it involves updating `current_command` instead of directly controlling the motors. Also, a modification was made so that while running a command (e.g. forward), you can change the speed.

4.6 Combined video/image and motor control

Until this stage, live-video, image capture and motor control had been implemented separately. So, it was necessary to develop a server that can handle the combination, and an app that has the combined features. It was necessary to make sure that the different functions work side by side. Until now there was a video streaming and image capture server via Flask and another one for motor control via sockets and `Jetson.GPIO`.

A new server code was developed, where:
- Flask is responsible for web routes (`/video_feed`, `/snapshot`,).
- Flask runs in a separate thread (so it doesn't block motor control loops).
- The motor system (`GPIO + socket`) runs as before, in parallel in another thread.
- A socket server is used for motor control, but both are integrated into the same program, with Flask running in a separate thread.

The code was also fixed, so that when a client (running the app) closes the app or disconnects for some reason, the server should not stop and throw an exception, but simply disconnect the client and continue running, waiting for a new client and commands.

When the new server code was done, a new app version was developed where the live-video, image capture and motor control was combined. The methods for motor control, start-stream and take snapshot was added to the Java code and the UI-elements was added to the XML-file. After adding the functionalities to the code, all features (until this stage) was tested by connecting the app to the server. It could be observed that these functionalities worked as expected.

4.7 Server auto-start

Until this stage, for starting the server it was required to start the Jetson (connected to a computer screen), open a terminal and run the `server.py` code. What if one takes the whole robot and put it on some road and want to run it? Then, an automatic server start is required. To start the server at system start, one need to create a systemd-service file. Write some commands in that file, enable it, start the service, check that it is running and then restart the Jetson. Now the service/file should start automatically on startup. If something goes wrong you can look at the logs. If you want to edit the server code, just edit the server code, when the service is started next time it will run the updated code. If you want to stop the service, just disable the service (through 2 commands). The auto-start service can be simplified in the following steps:

- Create a systemd service file to manage your script:

```
sudo nano /etc/systemd/system/my\_service.service
```

- Add the following content, replacing `user` with your username and adjusting paths as needed:

```
\left[Unit\\\right]\\
Description = My custom Autostart Service\\
After = network.target\\
\left[Service\right]\\
User = user\\
WorkingDirectory = /path/to/working/directory\\
ExecStart = /usr/local/bin/startup.sh\\
Restart = on.failure\\
\left[install\right]\\
WantedBy = multi-user.target
```

Save and exit the file.

Enable the service to start on boot:

```
sudo systemctl enable my\_service.service
```

After this, the file that is stated in the service-file will run at system boot. After creating the start service and stating the name and directory of the server, the server runs every time the Jetson is started. After every update of the server code, one just have to restart the Jetson.

4.8 Increasing video quality

Until this stage, Flask was used for live-video. The video quality wasn't optimal and had latency very much depending on the network connectivity. It was learned that there are many small details that one can reduce and use to reduce latency, such as reducing camera buffer, lowering resolution and FPS, (less data = faster transfer). Different tests was carried out to see if reducing different parameters would result in improved quality and reduced latency. Since, no noticeable improvement was observed, the implementation of another method was considered. One thing was for sure, that Flask is not optimal for video streaming. There are several ways for video streaming. After some research, RTSP streaming was chosen. But to use RTSP, many changes of the current code was required to make it work. First, installation of the necessary packages. Then, adding code for RTSP and making sure that the server sends video via the RTSP protocol. Then, on the client, open VLC or another RTSP player had to be used. With RTSP, the Android app cannot fetch video via `/video_feed` as a JPEG stream (as in the Flask/MJPEG setup). The Android app must instead use an RTSP client to play the video stream from :

```
rtsp://<ip>:8554/video
```

In the next step, an RTSP server with Gstreamer was set up. All the necessary

packages had to be installed. During this step, it was found that you can't run RTSP in browsers. Browsers are designed for HTTP(S) traffic, not for RTSP, which is a specialized video streaming protocol. The `v4l-utils` was also installed, which can be used for viewing connected cameras.

The server code structure was as follows:
- The RTSP server streams low latency video via RTSP. (streaming with Gstreamer) on port 8554.
- The HTTP server receives commands, runs motor control and has the ability to take snapshots via HTTP (`/snapshot`). Flask for HTTP API (snapshot, on port 5000), plus motor control (socket server on socket port 9999).
The RTSP-server is started from inside main-server as a background thread.

Then, the app was modified so that it could show live-video via RTSP, take picture and control the motors. For the RTSP-streaming, Android's `LibVLC` was used, which is a stable and easy way for RTSP streaming. A `VLCVideoLayout` was used in the XML-file. For motor control and image capture, the previous setup worked as it should.

After the modification of the server and the app, another problem came up. Namely, the live video was not showing in the app. At first, it seemed to be because of that 2 different processes (stream and snapshot) were trying to use the camera at the same time. After reviewing the server code again, it could be confirmed that the problem of 2 different processes fighting over camera access is completely avoided by not allowing any other process to directly access the ZED2i camera. Only the FFmpeg process opens `/dev/videoX`, and all other parts of the system (Flask, snapshot, streaming, internal threads) use the RTSP stream from MediaMTX instead of opening the camera directly. This meant that only one process reads the camera. All other consumers read a network stream (RTSP) that can be shared. This means no conflict over `/dev/videoX`. However, after many tests and checks, lots of changes was made to video settings/properties in the code, and then it worked. The problem was about specific features of the ZED camera connected to the Jetson. The video has lots of settings such as tune, speed-preset, bitrate, etc.

4.9 Mobile network connection

Until this stage, the Jetson and the phone was connected to a WiFi in the same building, I.e. local WiFi. To be able to control the robot in various outdoor locations and remotely, the Jetson should connect to a mobile network. The phone where the app is installed on was already connected to a mobile network at the same time as it was connected to the WiFi.

The work continued by looking at how to connect the modem (SIM7600E-H). At first, some research was made about how to connect and use the modem. Then, the sim card was inserted into the modem, the antenna was attached, the modem was

placed on the chassis and the modem was connected to Jetson via an USB-A cable. Then, the command `lsusb` was executed in terminal to see if it was recognized, and it was. ModemManager, ppp and pppconfig was installed. The 'raspberry' configuration was created using `sudo pppconfig`, and a connection attempt was subsequently made with `sudo pon raspberry`. The attempt was unsuccessful. It was learned that mobile networks (rarely) do not have public IP addresses, which can be a problem when connecting to the server. However, one can use tunneling (e.g. Ngrok or a VPN bridge or static IP from the operator) to solve the problem. Various AT commands were executed where the pin code for the sim card had to be entered. It was found out that the Jetson Orin Nano kernel version lacks the PPP module (ppp_generic). The QMI-mode was also tested. It seemed like the status of the sim card was failing, that it could not find the sim card. After inserting the sim card into a working Samsung Galaxy phone, it could be concluded that both mobile data and calls work, so there was nothing wrong with the sim card. Another test was conducted to see whether it is possible to connect a Windows PC to the internet via the modem. After some searching about this, a link on the Waveshare website was found, where instructions were written. The specific driver was downloaded and installed. After the installation, the different parts for the modem were shown in the device manager, which meant that the installation was successful. In the putty app, one could write the serial port COM12 (changed every time after reconnection of the modem) and the baud rate 115200. After specifying the port and the baudrate, one could write AT-commands in the putty app. Several commands were executed and everything seemed to work. However, after running a ND-mode command, everything stopped working. When the modem was connected via USB to Jetson or a Windows PC, the device was not recognized. The LEDs of the modem were lit, indicating that the modem was receiving power, but the device was not found or recognized by the OS (neither in the device manager nor `lsusb`). The device was restarted and the drivers were uninstalled and reinstalled. Different USB ports and different computers were tested, but nothing worked with the modem. A search was made for how to restore (reset) the modem to factory settings, nothing specific was found. The modem device was opened to find a reset button, but there was no reset button. It was thought that the modem might have become defective.

The modem has 6 serial pins, 5V, CTS, RTS, GND, RXD and TXD. A search was made about how to connect some pins of the modem to the Jetson to accomplish/-carry through a reset on the modem. After searching, it was learned that it is absolutely possible, but that it is important to do it correctly, otherwise there is a risk of damaging the Jetson or the modem. One should never connect 5V on the modem to the Jetson, Jetson is not tolerant to 5V signals. It was also found that one can connect the modem to the Jetson's GPIO/UART only if the modem runs 3.3V logic. In the modem's datasheet, it says:
Power supply: 5V
Logic level: 3.3V

Since the modem's logic level is 3.3V, it is fine to connect the modem to the Jetson's GPIO pins. Table 4.2 describes how the modem was connected to the Jetson.

Jetson, pin number	GPIO-pin	Modem port	Cable colour
pin 8	UART_Tx	RXD	green
pin 10	UART_Rx	TXD	yellow
pin 9	GND	GND	white

Table 4.2: Modem to Jetson serial connection.

Rx and Tx for the modem and jetson are reversed, because one sends and the other receives and vice versa. According to this connection, the modem can get power (5V) via USB and signals (3.3V) from and to Jetson via serial cables. This meant that no level converter was needed. Then, `ls -l /dev/ttyTHS*` was executed to see which UART device is used. The output was "THS1" and "THS2". Then, minicom was run, and when the AT command `AT` was executed, an "OK" was received. Now the functionality of writing AT commands worked. Then, the command `AT+CUSBPIDSWITCH=9011,1,1` was executed, this was to switch USB profile to standard profile (9011) with modem, AT port, GPS and network interface (ECM), and it also re-enables USB mode and can re-enable broken USB communication. Then, the modem was restarted and plugged back in. Then, by running `lsusb`, the modem was found again with the name "Qualcomm / Option Simtech". The modem was also connected to a windows PC to test if it is recognized. By certainty, it could be seen that the modem was again recognized in device manager and that the problem had been fixed.

Next step was about how to configure the modem so that every time the modem is connected to the Jetson, it should automatically connect to internet and make sure that Jetson has internet connection via the modem. When the command `dmesg | grep ttyUSB` was executed in terminal, to see the "ttyUSBx"-devices, some printouts were observed, indicating that the modem is not automatically connected to internet and that when it is connected to internet, it constantly disconnects and connects. This was a sign that something was unstable. This could be due to insufficient power, incorrect or incomplete driver, incorrect USB mode on the modem, or incorrect cable or port. At that time, the guess was that it might have to do with something about the USB mode, because when minicom was running and some AT commands were written, the commands were executed and the response was "OK" most of the time. There was an attempt to use a USB hub with its own power, to rule out cable or power problems. Also, different ports on the Jetson were tested along with or without the use of an extension cable. Then, it was suspected that there was some problem with ModemManager. After some troubleshooting, it turned out to be ModemManager that was sabotaging the communication. ModemManager was restarted, stopped, and temporarily turned off. Since only running AT-commands, PPP (Point-to-Point Protocol) or scripts was desired, ModemManager was not needed. After that, the modem turned out to be automatically connected to internet and was registered in the network and had a signal. When ModemManager was stopped with `sudo systemctl stop ModemManager`, the modem was released from being controlled by ModemManager, which allowed talking directly to the modem via Minicom without ModemManager "taking over" the ports. ModemManager

can block the port or the modem can become "uncontactable". By stopping ModemManager it was possible to send AT commands directly without interference. When running Minicom on `/dev/ttyUSB2` (which is the one used for network connection) and typing "AT", an "OK" was received. After running several other AT commands, many "OK" were received, indicating that the commands were executed correctly. At the end of running different commands, a "CONNECT 115200" was outprinted meaning that the modem was now connected to the internet via PPP data call, and a data port was now active. This meant that the SIM7600 was working properly, the modem had been initialized correctly, the SIM card was okay and the APN (data.tre.se) was working. After pinging `8.8.8.8`, a response was received, indicating that the internet connection was correct. Then, the Jetson was disconnected from WiFi and the Jetson was turned off. After that, the modem was connected to Jetson again and the Jetson was turned on. Now, Jetson had internet connection via the modem. Another test was done by pulling out the modem cable, waiting some seconds and reconnect the modem again while the Jetson was on, and it worked. After all this, every time the modem is connected to Jetson, it automatically connects to internet and the Jetson has access to internet. But, one problem remained, sometimes it happens that the modem disconnects from the internet for a few seconds and then reconnects again. The reason for this problem was not found, but it may be due to that the modem is an older modem or maybe because of local mobile network reception. However, The control of Jetson was tested, both video stream and motor control from some distance, and these functionalities worked fine.

4.10 Connecting to Tailscale

There are several things that need to work for it to function 'straight out of the box'—simply by plugging a SIM7600E-H 4G dongle into a Jetson Orin Nano, starting a server, and connecting to it via an Android app over the cellular network. When you plug in the 4G modem, the modem connects to the mobile operator's network. You usually get a private IP address (e.g. 10.x.x.x or 100.x.x.x), which makes the Jetson not directly accessible from the internet. The app tries to connect to the IP of the server, but the Jetson is behind a CG-NAT (Carrier-Grade NAT), which means you can't reach it from the outside. There is no port forwarding on the carrier's network. The server listens on a local port (e.g. 5000), and is reachable within the network. However, the Android app is not on the same network.

This problem can be solved in several ways:
1- Jetson dials up to an external server (Reverse Tunnel / MQTT / Websocket)
2- Getting a SIM card with public IP (static or dynamic)
3- Using a VPN (e.g. WireGuard or Tailscale)

Since it was not known whether a public IP was assigned, it was checked whether a public IP address was available for the modem (with a Hallon SIM card). The server on Jetson was started and a test was done to see if a response can be received from the server on the phone's browser (from the IP address which is written when you type `curl ifconfig.me`, which is an IPv6 address that the Jetson has with

the modem connected). No response was received, which means that the modem doesn't have a public IP address. Many operators do not allow devices to receive traffic from outside, even if the device has a global IPv6 address. There was also a test to ping the IPv6 address from a windows PC, where a "failure" was received.

So, another approach was needed to be able to connect the app to the server via mobile network. After some research, Tailscale was chosen and the implementing of Tailscale began. After registering on TailScale's website, the TailScale software was installed on Jetson and on a windows PC. The TailScale app was installed on the Android phone. Then, a login was made in all devices and the devices were connected on TailScale's website. Then, an attempt to connect to the server via the browser on the phone was made, and it went successfully. Tailscale is a quick and easy way to make Jetson, Android and other computers "see" each other directly, as if they were on the same local network - even if they are connected via 4G, WiFi or different networks. After registering the Jetson in Tailscale, the reachable IP of the Jetson could be observed in the Tailscale app. That IP was used in the app to connect to the server. After that, when the app was run, a "Cleartext HTTP traffic to 100.92.223.77 not permitted" error was printed. After some debugging, it turned out that after a certain update to Android, they made the connection to be via HTTPS. HTTPS is encrypted for security reasons (HTTP is not). It is possible to allow the server to use HTTPS, but it is complicated. Instead, a file `network_security_config.xml` was created, where `cleartextTrafficPermitted='true'` is allowed for the specific IP address 100.92.223.77. The IP for the server was dedicated and then the file was specified in the AndroidManifest (it could also be allowed globally, but it's a bit insecure). To use HTTPS you need to get a trusted certificate (or a local one) and then configure the server, but this requires some extra things. After adding `network_security_config.xml` and the server's Tailscale IP in the Manifest, the app now worked via the modem. The app shows live video, can take pictures and sends motor commands to the server and the server receives all commands. So, by connecting the server and the app via Tailscale, the server and the app are now connected as if they are on the same network. A picture of the Tailscale app is shown in figure 4.4.

4.11 Fail-safe motor stop

After implementing internet connection via the modem, a new problem had emerged. Sometimes when the robot was driven forward and then the stop button was pressed, it didn't stop, it continued to drive forward. At first, it was investigated if there was a problem with the command handler on the server, some changes were made and then some tests. But, the same problem still persisted. Then it was thought that there might be some problem with the network connection of the modem, since the same code worked without problems when the Jetson and the app were connected to the same WiFi. So it was concluded that due to the problem of sometimes disconnection of the modem, there was a big problem with the motor control. If the forward-button was pressed and exactly after that moment, the modem got disconnected, the robot would continue driving forward. As the robot driving forward to a wall, no matter

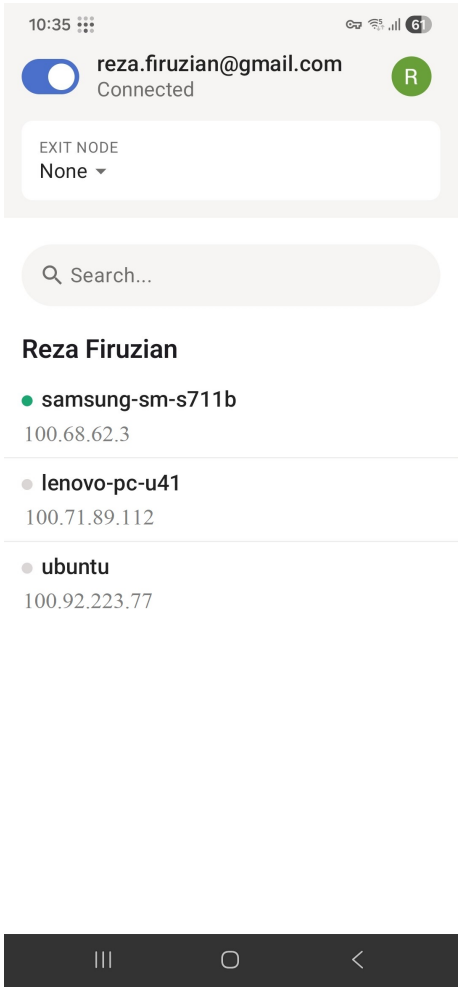


Figure 4.4: The Tailscale app showing all registered devices with their assigned IP addresses.

how many times one clicks on the stop-button, the command won't reach the server and the robot drives forward. As stated in the robot control section, the drive forward has a command time of 10 seconds, but if the wall is close, then the robot would collide with the wall. So, another safety mechanism was essential.

An "internet monitor" was added to the server, which checks every 250 milisecond to see if Jetson is connected to the internet or not. At first, the downtime limit was set to 2 seconds. I.e. if Jetson was disconnected for 2 seconds or more, it should stop the motors. After some testing, it was realized that the time was too much. In one test, the forward button was clicked, the robot started driving forward. At some point, the Jetson was disconnected. The stop button was pressed, but the signal didn't reach the server, robot continued forward. After 1.5 seconds, Jetson was connected again, but it wasn't known when it was connected. Shortly after, it was disconnected again. The robot continued and ran into a wall. So, it was concluded that the downtime limit should be reduced. So, the time was set to 0.5 seconds. I.e. if it is not connected to the internet for 0.5 second or more, it puts the motors in stop mode. With this, the motor control now worked with little latency

and without the problem of running into something.

4.12 App-Server reconnection handling

With the robot sometimes disconnecting from the internet, obviously there was an issue with the video stream and motor control: the app stopped receiving the stream and lost motor control. When running the server, a printout could be seen in the terminal that MediaMTX/RTSP opens and then closes. This was due to modem connection and disconnection. When the app starts, it connects to the stream that the server sends and on an another thread the motor control is started. After disconnection, the sending and receiving just stops and the motor control just disconnects. So, a reconnection handler was essential.

So, the server was modified in a way that if it is disconnected, it will restart the sending of stream as soon it gets back the internet connection. I.e. automatic reconnection of RTSP (FFmpeg+MediaMTX). The motor control on the server was already in a way that as soon it lost connection with the client (the app) it would close that connection and waits for a new motor control connection. An update on the app was also necessary. The app was modified so that if the receiving stream is disconnected it should automatically restart the connection. If the motor control thread is disconnected, a new socket connection is established. The new reconnection handler was tested and it worked fine. After a disconnection, the server continued and as soon as it got back internet connection, it started sending video and waited for motor commands. When the app lost connection to the server, it continuously tried connecting to the server and the stream. This was a stable handling of temporary network disconnection.

4.13 Connecting camera2

For the connection of Camera 2 (Raspberry Pi Camera Module 2), the 15-pin to 22-pin cable was connected to the `CAM0` port on the Jetson. Then, the lens was taped on the front side of the chassis. Then, it was necessary to configure the hardware configuration for the camera, via:

```
sudo /opt/nvidia/jetson-io/jetson-io.py
```

On the configuration page, the Jetson 24-pin CSI connector was configured for compatible hardware, and `Camera IMX219-A` was selected for CAM0.

After the hardware configuration and reboot, a `gst-launch`-command was tested to see if the camera works. And it worked. When running `V4l2-ctl -list-devices`, the following was printed:
IMX219 (camera2) : video0
ZED 2i (camera1) : video1, video2

This meant that it was required to change the code in the server that sets the video parameters. So, now the new camera would be "video0" and the old camera would be "video1" or "video2".

Then a small server code, just for the new camera was tested. And it worked. Then, the new camera code was added to the server code. After that, the app code was modified so that the app shows live video for the new camera as well. Camera2 was configured so that live video is accessible via the link to the server and by `/camera2_stream`. At first, when trying to run the server and connecting via a browser, a "green screen" was received. Then, when adding `GST_PIPELINE` in the code, it worked. The new camera uses Gstreamer pipeline, Flask + MJPEG streaming and OpenCV. Now camera2 worked but with a delay that depends on the network connection.

After connecting camera2 and adding the new code, a new test concerning all (so far) functionalities was conducted. The cameras worked, the modem worked, but not the motor control. Some troubleshooting was done, but the problem was not found. It was then suspected that the issue might be related to the hardware configuration. In this part of the work, the hardware configuration for "CAM0" had been changed, but nothing else. After some time, it turned out that it was necessary to configure both CAM0 for IMX219-A and GPIO-40 pins for the pins to the motor driver (MDD3A) at the same time. The configuration only for CAM0 had been changed and then saved. It turned out that, if the configuration of GPIO-40 pins is not done at the same time, the system resets the configuration for the GPIO-40 pins. After configuring both the CAM0 and the GPIO-40 pins, the configuration was saved and the Jetson was restarted. Then, all functionalities were tested again and all worked as they should.

4.14 Connecting envrionmental sensor

For the connection of the Adafruit Trinkey SHT45 sensor to the Jetson, all necessary libraries were first downloaded. Then, the sensor was put into 'bootloader' mode by double-clicking the reset button. When the device was opened in the device map, all libraries were copied to the device. After that, the firmware (UF2 file) was downloaded and copied to the Trinkey. Then, the required `CircuitPython` library was installed on the Trinkey. Then, a file (`code.py`) was created and the sender code was written in that file. The file was saved on the Trinkey. Worth mentioning is that in that code the interval at which data is sent from the sensor was specified. The interval was set to 1 second, meaning that measured temperature and humidity is sent over USB serial every second. It was also required to install `pyserial`, `circup` and `adafruit_sht4x` on the Jetson. Then, it was necessary to find the Trinkey serial port (`/dev/ttyACM*`) . In this case, it was `/dev/ttyACM0` .

After initial setup, a simple code that prints the temperature/humidity received from the sensor was written. The code worked as intended, it wrote the temperature and humidity every second. One problem was that it seemed like the temperature was

always 3-4 degrees higher than the actual temperature. The fixing of this problem was postponed. Then, the code was modified and put in the server code so that the server receives the data from the sensor and makes it possible for the app to get the data. Then, the server was run to verify whether the measured data were received and printed. I was required to modify the code so that if the sensor is not connected (whatever reason) it should not crash or throw exception, but just output 0.00. This was tested by removing the cable of the sensor while the server was running. It could be seen that it printed 0.00. Initially, there was a problem that one had to run the server with `sudo` so that the server could receive data from the sensor, but after some changes in the settings, this problem was fixed.

Then, the server code had to be modified so that data from the sensor could be provided to the app. Code was added to the server, so that it can send the data to an endpoint (which the app can then read from). The program was modified so that it reads temperature and humidity from the "SHT45" sensor constantly, saves them in global variables and exposes them via a Flask endpoint. The temperature reading function runs on its own thread. The function reads line by line and converts string to float values. If the sensor reading is interrupted (for any reason), the old value is saved and the server and the app do not have to crash. If opening the sensor data does not work, it tries again (after a certain time, a reconnection is carried out). The endpoint (`/sht45`), first locks the sensor data mutex, reads temperature/humidity and returns a JSON object.

Then the app was modified so that temperature and humidity (as JSON objects) are retrieved every 5 seconds via the `/sht45` endpoint and the following URL:

```
http://serverip:5000/sht45
```

The app opens a regular HTTP connection to the Jetson and downloads the data. It then reads the response line by line and builds a JSON string. Once the JSON string is received, two double variables are extracted. Then, the two TextViews on the UI thread are updated.

After the modification of the server and the app code, the sensor was tested, first indoors where it worked. Data is downloaded and updated every 5 seconds. When the internet connection was disconnected, the old values were retained. One problem was that when the robot was put outdoors, it took a few minutes before the real temperature was displayed (it went down very slowly), so it took about 7 minutes for it to go from 19.5 (room temperature) to 2. For 2 to 19.5, it went faster. Not perfect, but acceptable.

Then, the problem was investigated, where the temperature sent by the sensor appeared to be 3–4 degrees higher than the actual value. For the humidity, there was no indication that the measured value was incorrect. Also because the fact that humidity does not get affected by nearby heat sources. After some research and reading, it was found out that this is a common problem. The reasons can be

several, it could be due to heat from Jetson, self-heating of the sensor, if you read data too often the USB card around it gets hot, or poor air circulation around the sensor. Adafruit themselves write that this type of sensor can be 2-5 degrees too high. It was also found that the sensor has “Precision mode”. If the sensor is in high precision mode, it consumes more power and gets hot. It was tested to change to low precision mode, but an error (always printed 0.00) was received. After some reading, it was found that this particular sensor does not support precision mode. It was also found that if one uses “NOHEAT”, the sensor creates less heat. This was also tested, without any success. It was found that this sensor does not support "NOHEAT".

After some additional research and reading it was learned that one way to solve the problem of 3-4 higher temperate is to calibrate with a certain offset. So, this is how it was solved in the end. An offset of –3.5 degrees was added and subtracted from the temperature sent by the sensor.

4.15 Connecting thermal camera

First, the Thermal-90 USB Camera was connected to the Jetson’s USB-C port. The command `lsusb` was typed in the terminal and it could be seen that it was found as "Winbond Electronics Corp. USB Virtual COM". The command `v4l2-ctl -list-devices` was also written in the terminal, but the device was not printed there. That meant that the Thermal-90 is not registered as a UVC camera, but it is registered as a serial USB device (Virtual COM Port), because it is visible via `lsusb`. This meant that the camera is a cheap thermal sensor with an MCU that sends raw pixels via a serial port. The camera was identified as "ttyACM1". This meant that it has to read the raw data from "/dev/ttyACM1". Many example codes was tried, to see if the camera works, without any success. Different baudrates were tested. Another test that includes sending a start-command to the camera was tested, without any success. The problem was that the camera didn’t send anything at all and no response was received. An attempt was also made to connect the thermal camera to the USB-A port, considering that the issue might be due to the USB-C port being used only for power. However, the same problem occured with the USB-A port as the USB-C port. The camera is found via the code, but does not receive a response from the camera.

Subsequently, it was discovered that installation of the USB SDK from Waveshare’s website might be required. The `Thermal_Camera_hat.zip`-file was downloaded. The file was unzipped and in the `pysenxor-master` directory, the command `pip install -e ./` was executed to install the SDK package. Then, `sudo python3 stream_usb.py` was run, which is a test code from Waveshare in the example folder that should work after installation. When this was executed, an error message was received that it cannot find `senxor`. It was found out that it was required to install the SDK for `root/Python3` globally. After running `sudo pip3 install -e` in the `pysenxor-master` folder, and after running `python3 stream_usb.py` (in the example folder) the thermal camera worked. A video window opened and the

camera's printout in color could be observed.

So, after installing `pysenxor`, it was necessary to take some of the code in the sample code from Waveshare, import `senxor.mi48`, adapt the code and put the code in my server code, and then create the Flask endpoints `/heat_stream` and `/heat_snapshot` for video and image respectively. Then, the code was modified so that `/heat_stream` returns continuous MJPEG video and `/heat_snapshot` a single JPEG image.

Then, modifying the app began so that it shows heat_stream and can take pictures. First, an attempt was made to display heat_stream in an ImageView, but it appeared choppy. Then, the `com.github.niqdev_mjpeg` was tested and it appeared in black and white. It was thought that maybe, it had to do with the (small) size of the JPEG images. Then, the size was changed and the color was set, without any success. Then, an attempt was made by using `com.github.niqdev.mjpeg.MjpegSurfaceView`. The following was added in the `build.gradle.kts(:app)`-file:

```
implementation("com.github.niqdev:ipcam-view:2.4.1")
```

and in the `settings.gradle.kts`-file (RobotApp) under plugingmanagement and under repositories and dependencyresolutionManagement , the following was added:

```
maven {url = uri("https://jitpack.io") }
```

With the use of `MjpegSurfaceView` and `jitpack.io`, the implementation of the `heat_stream` continued. Some settings for the heat-video in the app were changed. Finally, a few lines in the server were changed to make it work so that the heat video is colored instead of black and white. It was required to specifically set `colored-heat` and change the size of the frame. Then, some code was added so that when you click take picture button, a picture of the stream is saved on the phone. Now `heat_stream` and `heat_snapshot` worked as they should. A picture of the app when it is in thermal camera mode is shown in figure 4.5. The code was also modified so that if you disconnect the thermal camera while the server and app are running, it will try to reconnect to the camera's frames after 5 seconds. This means that if you reconnect the thermal camera, it will start broadcasting live video again, which is the expected behavior.

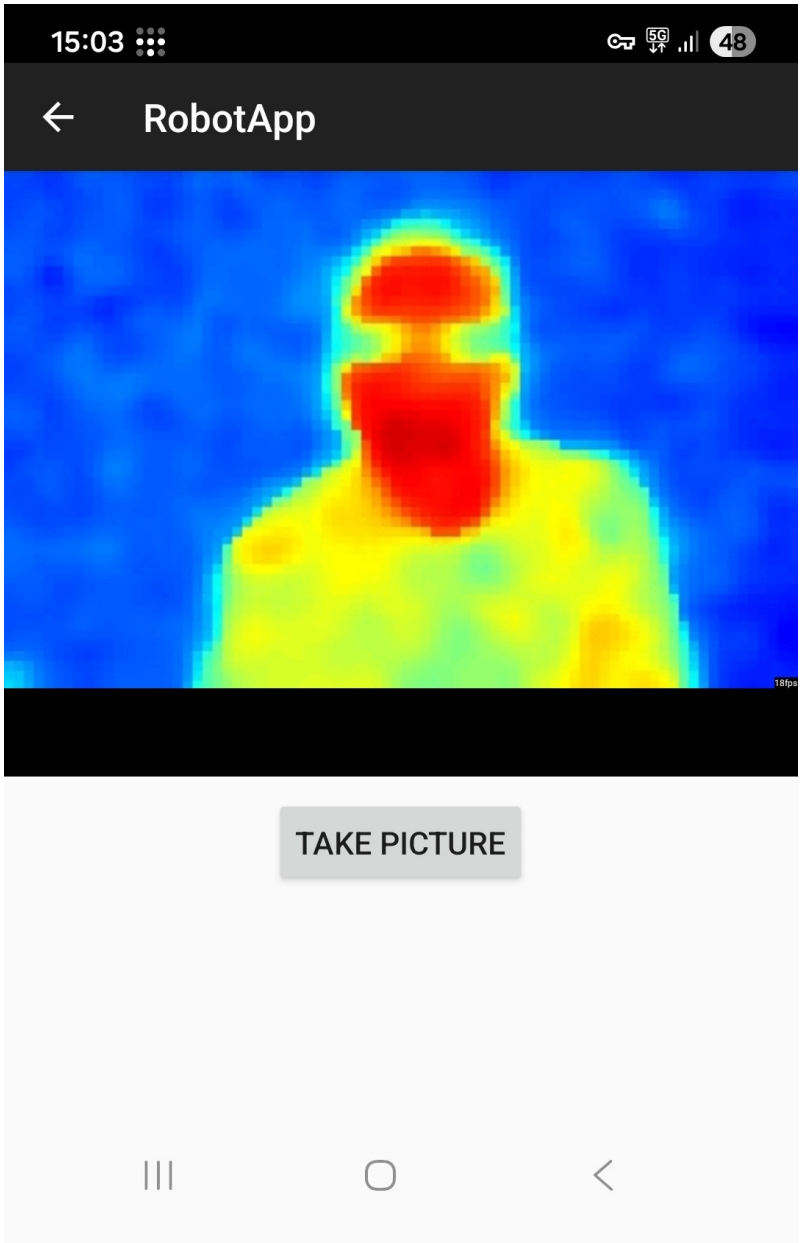


Figure 4.5: An image of the app in thermal camera mode showing a person.

5

Results

In this chapter, the results of the project are presented. Although testing has been a part of every feature/module, at the end of the work, an extensive test that evaluates all functionalities has been carried out. The robot was placed on a lawn and the different functionalities was tested. Below is an explanation of how the different tests went, divided into different functionalities.

5.1 Robot driving

The robot is able to drive forward, backward, turn left, turn right. Driving forward can be at three different speeds (25, 50 or 100%), while driving backwards is done only with one speed (100%). Robot driving has been tested on apshalt road, on grass, on carpet (indoors) and appartment floor, but not on a dusty road. The tests went well except when there was a sloping surface. The problem was specifically about uphill driving. An exact maximum slope has not been calculated, but when the slope of the surface was approximately more than 45 degrees, the robot and the chassis tipped over backwards. An illustration of this is shown in figure 5.1.

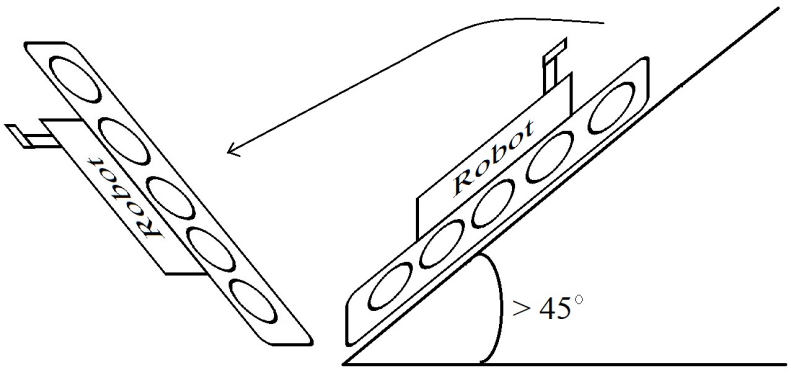


Figure 5.1: An image that represents the robot tipping over backwards when the uphill slope is more than 45 degrees.

The same problem happened when the robot tried to drive pass a threshold (a door sill) indoors. This problem was mainly because of the type of caterpillar tracks and because of that the battery (which is the heaviest component) is placed at the back of

the chassis. So, if a newer type of caterpillar tracks that has movable springs was used in combination with placing the battery in the middle of the chassis this problem could be solved (or at least it would increase the angle). A similar problem exists if the road slopes to the side, but this problem seems to exist for many vehicles. There was no problem with driving the robot downhill. The safe-motor stop mechanism also worked as it should. In some situations where the robot was driving forwards and the modem got disconnected, the motors stopped after 1 second, and that is an acceptable time. All tests for robot driving has been done when there was no rain or snow. When a command-button was clicked in the app, a toast-message was shown, indicating if the command was sent or not.

5.2 Internet connection via modem

The robot got accessed to internet via the modem. Due to this the robot could be controlled remotely and the maximum distance is determined by the mobile operator. Tests have been done to control the robot from a few hundreds of meters. But if it works from hundreds of meters, there is no reason it won't work from 1 km. However, there was a major problem with the modem. During certain times, the modem was disconnected from the internet and then reconnected again. This problem affected the whole system. When there was no internet connectivity, the commands didn't reach the robot and the live-streams didn't continue. The exact reason to this disconnectivity has not been concluded, but it maybe due to that the modem is a cheap and older model. The testing of the modem also showed that in some areas, the connection was good and stable. The reconnection mechanism made sure that when the modem got disconnected, it automatically got connected again after some seconds.

5.3 Live-video streaming

The main camera showed live-video streaming. The quality of the video was good and "smooth", mainly due to RTSP. However, the latency was very much dependent on the network connectivity. In areas where the robot had good connectivity, the latency was reduced to below 1 second. In areas where connectivity was bad, the latency was increased to between 3 and 4 seconds. When the internet connectivity was completely disconnected, then the stream just froze. The reconnection mechanism, made sure that when the stream was interrupted, then it tried again to connect to the stream from the server. In some moments when the internet connection was disconnected, it was visible that after some seconds, the internet connection was connected again and the live streaming continued. This could be observed due to the fact that when the internet connection was down, a toast-message was shown indicating that the app tried to reconnect to the stream. The given explanation also applied to camera2, with the difference that camera2 used MJPEG streaming and has not the same quality as the main camera.

5.4 Temperature and humidity data

In the app, the ambient temperature and the humidity of the robot is shown. The values get updated every 5 seconds. The temperature data was affected by the heat of all components on or attached to the chassis. By using an offset the correct temperature is shown in the app. When the robot was indoors, the temperature was around 20 °C, which seemed to be correct. When outdoors, it was around the temperature shown in other apps (that shows local temperature). However, there was a problem with the changing of the temperature. When the robot was indoors, it showed 20 °C. After putting the robot outdoors, it took between 7 and 8 minutes for it to go down to 3 °C. But, on the other way around, from 3 °C to 20 °C, it went faster. The reason have not been concluded, but maybe, this is because that the sensor is a cheap variant. The data for the humidity have not been verified.

5.5 Thermal camera streaming

In the thermal camera mode of the app, a live stream containing thermal images was shown. The latency of thermal streaming was also affected by the network connectivity. When the modem had good and stable connection, the latency was below 1 second. When the network connectivity was unstable, the latency was from 2 to 3 seconds. However, the thermal video has been tested in a dark environment, I.e. indoors when all the lights are off. The thermal camera stream worked very well in darkness. When the network connectivity of the robot's modem was down, a toast-message was shown in the app, indicating that it is trying to connect to the stream again.

5.6 Snapshots

The two snapshot-buttons (image capture) for the main camera and the thermal camera worked well. Both worked by taking a frame from the stream and saved it on the phone. However, the time from clicking the button until it was saved was also affected by the network connectivity of the robot. When the connection was stable, it took less than one second. But, with an unstable network connectivity, it took between 2 to 3 seconds. After clicking on the snapshot-buttons, a toast-message was shown, saying if the picture has been saved or if there was a problem with taking or saving the image. The quality of the pictures were good.

6

Discussion

This chapter contains a discussion of some important aspects of this project and how further development could be. Beginning with a summary of accomplished objectives.

6.1 Fulfillment of project purpose and goals

The purpose of this project, written in section 1.2, was to find out how to build/create a robot that can move and turn, show live video, take pictures, show temperature and how the robot can be remotely controlled. To find out what hardware, software is necessary and how to connect various hardware with software into a functioning system. The purpose has been fulfilled and the necessary knowledge about how to build/create such a robot with the desired features has been gained. It has been learned how to connect the different components, how to configure the components and how to develop the server and the control app into a functioning system.

The sub-goals written in section 1.3 have been accomplished. It has been learned how to connect all parts. A server has been developed that can send live video, image, temperature and able to receive commands for motor control. A modem that has mobile internet connection has been configured. An app that connects to the server (via the modem) has been developed and motor control via the app has been implemented. Live video, image capture in the app has been implemented while the app also displays the temperature of the robot. In addition to all basic functionalities, live_stream for the thermal camera has been implemented.

6.2 Server-App security

Server or app security was not a priority in this project. In fact, almost no time has been spent on security. Command messages are sent and received as pure texts and the endpoints doesn't require any authentication. However, by using Tailscale (described in section 4.10), some kind of security has been implemented. By using Tailscale, only registered devices can connect to each other. The IP-address of the server cannot be reached from devices that are not registered in the Tailscale-account of the user. In this way, the user can control which devices should be able to connect

to the server. By this, Tailscale has introduced a kind of security, even though the use of Tailscale was not based on security reasons.

6.3 Ethical use of cameras

Ethical use of cameras means that live video and photography are not used in a way that violates people's privacy. Consideration should be given to privacy, consent, security and purpose. Since this robot is supposed to be used in disaster situations or at accident locations, there shouldn't be a problem. However, the user should know what ethical use of cameras means, in case of using the robot in other circumstances.

6.4 Power supply knowledge

Although this project was mainly about computer engineering and computer technology, it was quite surprising how much basic electrical knowledge was required to carry out the project. For connecting the power to the motor driver, it was necessary to connect the right type of cable (correct voltage and current), connect the cable to the correct port (correct power) on the battery and connecting a common ground to the motor driver (to the correct in-port). For connecting the pins on the Jetson to the motor driver, one has to know what pins should be used and where to connect the voltage-pin. There was some situations where a voltage measurement by a multimeter was needed, to see what voltage the pins had. There was situations where there was a need to check if two pins or cables are connected or not. No advanced electrical engineering knowledge was needed, but the basics was definitely required to make it work.

6.5 Unstable internet connection with 4G modem

The unstable modem connectivity has clearly affected the whole project. It has introduced latency in video-streaming, delay for taking picture, late or completely unreached commands. However, it has also introduced a need for reconnection. Both for the internet connection of the modem and the reconnection of streams. If the modem hadn't occasionally got disconnected, maybe all the reconnection mechanisms hadn't been implemented at all. However, the problem of the internet disconnection can probably be solved by a newer and more expensive modem, preferably with 5G.

6.6 Remaining battery percentage

A lot of research has been done about adding the feature of seeing the robot's remaining battery percentage in the app. What was concluded is that some batteries has this feature and some not. Meaning that if one wants data from the battery, the battery itself has to have the feature of sending battery data. The battery used in this project didn't have this feature. There is also another method for knowing the remaining battery percentage. It is based on measuring how much current is used.

The battery has a known capacity in mAh. The electronics continuously measure how many milliamperes are used. The system counts down from 100% when energy is used. Since this method only works when the battery is 100% at start, it is not much practically useful.

6.7 Improving maintainability through code separation

The server code for this project is written entirely in one file (with almost 1000 lines). If someone besides the code writer would look at the code, it would be difficult to understand the code. It would also be difficult to troubleshoot, debug or modify the code by someone else. It would be better to split the code into several files (according to the OOP-principle).

The app code is however split into different files. Two activities, two XML-files and one manifest. This was mainly because of that the structure of a project in Android studio is split in a way that makes the code divided into different parts.

6.8 Working according to the plan

In general, most of the work went according to the initial plan. Mainly because of that the first material was received early. More time than expected was spent on installing the OS on the Jetson. More time than expected was also spent on motor control, specifically on how to connect the motors to the motor driver and how to connect the motor driver to the Jetson. More time than expected was also spent on the disconnectivity of the modem, specifically trying to fix the problem. However, some parts was carried out faster than expected. Once the server structure was in place, adding features was fairly quick. So, adding camera2, environmental sensor and thermal camera was fairly quick.

6.9 Further development

To use this robot as a real tool in disaster situations or any situation that requires a remotely controlled device for investigation, there are some parts and properties that need enhancement/improvement. The following states the most important ones.

- **Faster and more stable modem**
  With a faster, newer and more stable modem, the risk of not arriving or late arriving commands would probably be solved. The problem of several seconds of video-latency would probably be solved.
- **Component protection**
  To be able to use the robot in rain, snow and dusty environment, all the hardware and components need protection. The components should be waterproof

and not affected by dust.

- **Obstacle Detection and Robot Stop Condition**
  For safety reason, the robot should auto-stop if there is an obstacle in near front of the robot. This would require an additional sensor.
- **Remaining battery percentage**
  In some circumstances, it is important to know the remaining percentage. E.g. if the robot is sent to a situation, it is essential to know if the battery has enough power to carry out a mission (so the robot doesn't get stopped somewhere because of power outage). For this feature, a new battery that has support for battery data is required.
- **Solar cell charging**
  With solar cell charging, you both ensure that the battery can be charged and that you move towards sustainable development.

7

Conclusion

This project demonstrated the different processes of constructing, developing and implementing an app-operated robot. It describes how to integrate a specific setup of hardware alongside the necessary software into a functioning system. The robot can drive on different roads, captures live video and images, and measures ambient temperature. The construction and development of the robot have been carried out in a modular way, adding one feature or module at a time. The main objectives of the project has been fulfilled. But the advanced features, such as battery monitoring, solar charging and autonomous navigation has not been implemented due to resource or time constrains. However, this project provides good foundation for further development.

The main issue identified is the unstable 4G modem, which affects video latency and motor control response. Using a faster and more stable modem, preferably 5G, would probably solve this problem. The work highlights the feasibility of creating a remotely operated observation robot that can function in environments unsafe or inaccessible for humans. In general, this project accomplished its main goals and provided a development example that with enhancements can be used in monitoring and exploration tasks. To use the robot in real "disaster" situations, it is essential to add protection for the electronic components. With these protections, the robot could operate safely in rain, snow, or dusty environments.

References

[1] Nvidia, "Transform Generative AI Concepts Into Reality," 2025. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/nano-super-developer-kit/ (accessed on: 2025-11-06).

[2] Nvidia, "Build the Future of Robotics With NVIDIA Jetson Orin Nano™ Super," 2025. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/nano-super-developer-kit/ (accessed on: 2025-11-06).

[3] Nvidia, "NVIDIA Jetson Orin Nano Super Developer Kit," 2025. [Online]. Available: https://nvdam.widen.net/s/zkfqjmtds2/jetson-orin-datasheet-nano-developer-kit-3575392-r2 (accessed on: 2025-11-07).

[4] Particle, "Getting Started with NVIDIA Jetson Nano Orin Nano," 2025. [Online]. Available: https://developer.particle.io/linux/devices/nvidia/devices-nvidia-nano (accessed on: 2025-11-07).

[5] Botland, "Cytron MDD3A - dual-channel 16V / 3A motor controller," 2025. [Online]. Available: https://botland.store/motor-drivers-modules/15819-cytron-mdd3a-dual-channel-16v-3a-motor-controller-5904422324841.html (accessed on: 2025-11-09).

[6] Neven 7, "MDD3A 3A 4V-16V DC Motor Controller (2 Channel)," 2025. [Online]. Available: https://www.neven7.eu/p/mdd3a-3a-4v-16v-dc-motor-controller (accessed on: 2025-11-09).

[7] GeeksforGeeks, "Technology for 4G Mobile Communications," 2025. [Online]. Available: https://www.geeksforgeeks.org/mobile-computing/technology-for-4g-mobile-communications/ (accessed on: 2025-11-09).

[8] A. Moshynska and O. Khrokalo, “Remote Vehicle Diagnostic System Development Based on the Internet of Things Technology,” *Information and Telecommunication Sciences*, vol. 15, no. 1, pp. 28-32, Jun. 2024, doi:10.20535/2411-2976.12024.28-32.

[9] Waveshare, "SIM7600X 4G DONGLE," 2025. [Online]. Available: https://www.waveshare.com/wiki/SIM7600G-H_4G_DONGLE (accessed on: 2025-11-09).

[10] TailScale, "Plans that work for everyone," 2025. [Online]. Available: https://tailscale.com/pricing?plan=personal (accessed on: 2025-11-09).

[11] D.-F. Hrițcan and D. Balan, "Using Tailscale and PfSense for Security and Anonymity of IoT Environments," in *Proceedings of the 17th International Conference on DEVELOPMENT AND APPLICATION SYSTEMS*, Suceava, Romania, 2024, pp. 91-94. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=10541192 (accessed on: 2025-11-09).

[12] TailScale, "Infrastructure-agnostic," 2025. [Online]. Available: https://tailscale.com/why-tailscale (accessed on: 2025-11-09).

[13] TailScale, "Resilient Networking," 2025. [Online]. Available: https://tailscale.com/why-tailscale (accessed on: 2025-11-09).

[14] Stereolabs, "ZED 2i Stereo Camera," 2025. [Online]. Available: https://www.stereolabs.com/en-se/store/products/zed-2i (accessed on: 2025-11-11).

[15] Raspberry Pi, "Raspberry Pi Camera Module 2," 2025. [Online]. Available: https://www.raspberrypi.com/products/camera-module-v2/ (accessed on: 2025-11-11).

[16] RS, "Raspberry Pi, Camera Module , CSI-2 with 3280 x 2464 pixels Resolution," 2025 [Online]. Available: https://se.rs-online.com/web/p/raspberry-pi-cameras/9132664 (accessed on: 2025-11-11).

[17] K. Antony, "Getting Started with Flask, a Python Microframework," *Sitepoint*. [Online]. May. 2023. Available: https://www.sitepoint.com/flask-introduction/ (accessed on: 2025-11-12).

[18] "Motion JPEG," in Wikipedia. 2025. [Online]. Available: https://en.wikipedia.org/wiki/Motion_JPEG (accessed on: 2025-11-12).

[19] Datacamp, "A Complete Guide to Socket Programming in Python," 2023. [Online]. Available: https://www.datacamp.com/tutorial/a-complete-guide-to-socket-programming-in-python/ (accessed on: 2025-11-12).

[20] NVIDIA Corporation, *Jetson Orin Nano Developer Kit Carrier Board Specification*, Version 1.3, 2023. [Online]. Available: `https://developer.nvidia.com/embedded/jetson-developer-kits`. (accessed on: 2025-11-13).

[21] A. Kumar, "How to Use GPIO Pins on Jetson Nano Developer Kit," *MakerPro*. [Online]. Jun.2020. Available: https://maker.pro/nvidia-jetson/tutorial/how-to-use-gpio-pins-on-jetson-nano-developer-kit (accessed on: 2025-11-12).

[22] C.R. China, "What is live streaming?," *IBM*. [Online]. 2025. Available: https://www.ibm.com/think/topics/live-streaming (accessed on: 2025-11-13).

[23] P. Henken, "RTSP – all you need to know about real-time streaming protocol," *Kaltura*. [Online]. Feb. 2024. Available: https://corp.kaltura.com/blog/rtsp-streaming/ (accessed on: 2025-11-13).

[24] F. Veselinovic, "RTSP: The Real-Time Streaming Protocol. What is RTSP and How Does It Work?," *Ant Media*. [Online]. Jul. 2025. Available: https://antmedia.io/rtsp-explained-what-is-rtsp-how-it-works/ (accessed on: 2025-11-13).

[25] Stream, "Real-time Streaming Protocol (RTSP)," 2025. [Online]. Available: https://getstream.io/glossary/rtsp-protocol/ (accessed on: 2025-11-13).

[26] C. Kopias, "FFmpeg - The Ultimate Guide," *Img.Ly*. [Online]. Nov. 2022. Available: https://img.ly/blog/ultimate-guide-to-ffmpeg/ (accessed on: 2025-11-13).

[27] MediaMTX, "Introduction," 2025. [Online]. Available: https://mediamtx.org/docs/kickoff/introduction (accessed on: 2025-11-13).

[28] OpenCV, "OpenCV is the world's biggest computer vision library.," 2025. [Online]. Available: https://opencv.org/about/ (accessed on: 2025-11-13).

[29] OpenCV.AI, "Computer Vision software and services," 2025. [Online]. Available: https://www.opencv.ai/ (accessed on: 2025-11-14).

[30] Moukthika, "Reading and Writing Videos using OpenCV," *OpenCV*. [Online]. Feb. 2025. Available: https://opencv.org/blog/reading-and-writing-videos-using-opencv/ (accessed on: 2025-11-14).

[31] "GStreamer," in *Wikipedia*. Oct. 2025. [Online]. Available: https://en.wikipedia.org/wiki/GStreamer (accessed on: 2025-11-14).

[32] GeeksForGeeks, "How To Make An Android App," 2025. [Online]. Available: https://www.geeksforgeeks.org/android/how-to-make-an-android-app/ (accessed on: 2025-11-14).

[33] IBM, "How do I make an Android app?," 2025. [Online]. Available: https://www.ibm.com/think/topics/android-app (accessed on: 2025-11-14).

[34] Developers, "Configure your build," 2025. [Online]. Available: https://developer.android.com/build (accessed on: 2025-11-14).

[35] Piveral, "Power Supply Requirements for Nvidia Jetson Orin Nano," 2024. [Online]. Available: https://nvidia-jetson.piveral.com/jetson-orin-nano/power-supply-requirements-for-nvidia-jetson-orin-nano/ (accessed on: 2025-11-14).

[36] *Jetson Orin Nano Developer Kit Carrier Board*, Santa Clara, USA: NVIDIA Corporation, 2023.

[37] Opencircuit, "Adafruit SHT45 Trinkey - USB temperatur- och fuktighetssensor," 2025. [Online]. Available: https://opencircuit.se/product/adafruit-sht45-trinkey-usb-temperature (accessed on: 2025-11-26).

[38] Adafruit, "Adafruit SHT45 Trinkey - USB Temperature and Humidity Sensor," 2025. [Online]. Available: https://www.adafruit.com/product/5896 (accessed on: 2025-11-26).

[39] Waveshare, "Long-wave IR Thermal Imaging Camera Module, Raspberry Pi IR Camera, 80×62 Pixels, Options for FOV and Connector," 2025. [Online]. Available: https://www.waveshare.com/thermal-camera.htm?sku=26984 (accessed on: 2025-12-03).

[40] "Thermal imaging camera," in *Wikipedia,* 2025. [Online]. Available: https://en.wikipedia.org/wiki/Thermal_imaging_camera (accessed on: 2025-12-03).

[41] D. Das, "Waveshare Thermal Imaging Camera Module – Raspberry Pi HAT or USB-C model, 80×62 resolution, dual FOV options (45°/90°)," *CNX Software*. [Online]. May. 2024. Available: https://www.cnx-software.com/2024/05/14/waveshare-thermal-imaging-camera-module-raspberry-pi-hat-or-usb-c-model-80x62-resolution-dual-fov-options-45-90/ (accessed on: 2025-12-03).

A

Appendix 1

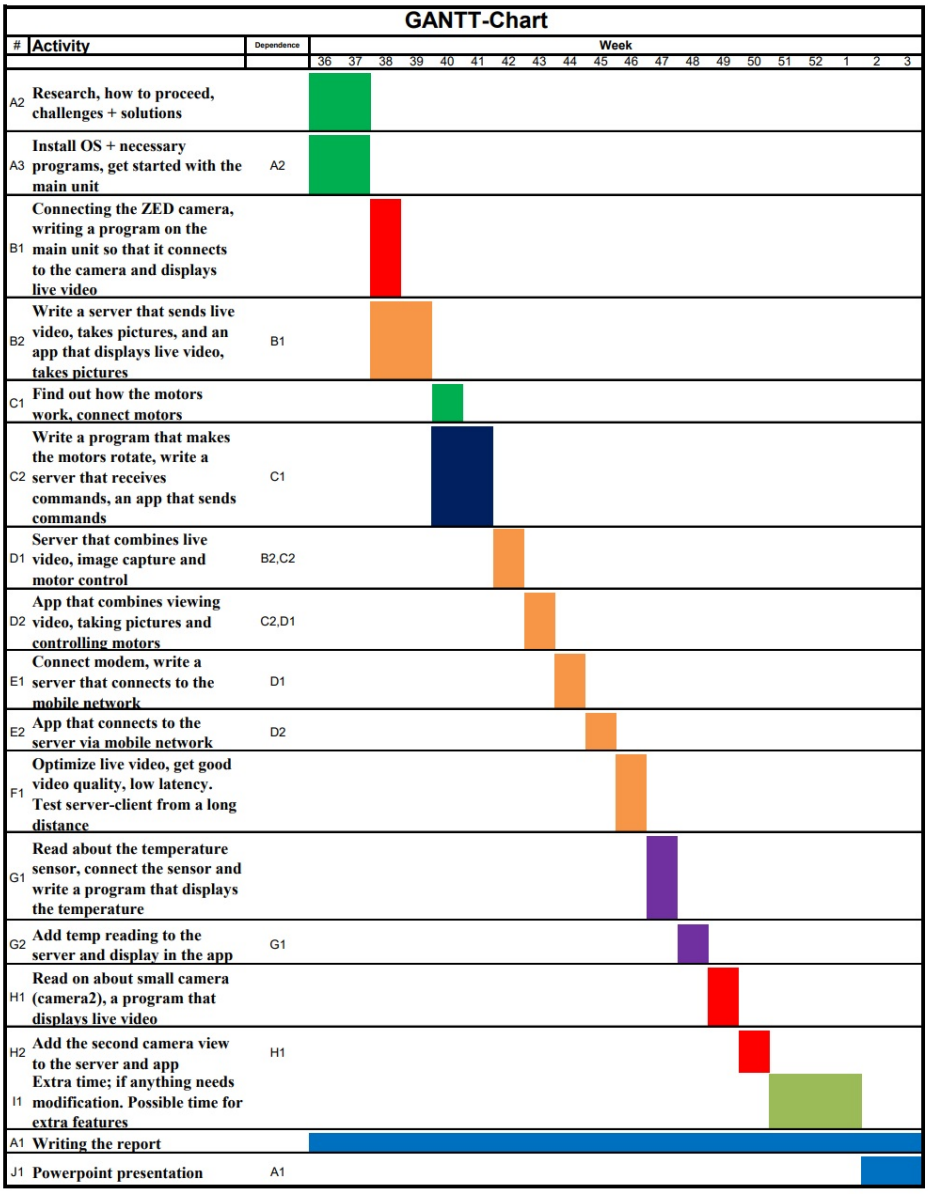


Figure A.1: The Gantt chart for the project planning.

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY