POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

The Faculty of Information Technology

Chair of Software Engineering

Software and Database Engineering

Mateusz Bugaj
25540

Integration of real-life devices with the internet software

Master Thesis written under the supervision of:

Mariusz Trzaska Ph. D.

Warszaw, July 2023

Abstract

In a world where access to the high-speed Internet grows increasingly abundant, new ways of interacting with the real world emerge. The following thesis describes a proposed solution for interacting with real-life, physical devices using streaming services available for free for everyone. The example implementation shows a cartesian manipulator with a microscope as two external devices that can be used in real-time via live-stream on Twitch and YouTube.

Keywords

Live-stream, hardware, API

POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Mateusz Bugaj
25540

Integracja urządzeń ze świata rzeczywistego z oprogramowaniem internetowym

Praca magisterska napisana pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, Lipiec 2023

Streszczenie

W świecie w którym dostęp do szybkiego internetu staje się powszechny, pojawiają się nowe sposoby interakcji ze światem rzeczywistym. Praca opisuje proponowane rozwiązanie problemu interakcji z fizycznymi urządzeniami przy użyciu serwisów streamingowych dostępnych za darmo dla każdego. Przykład implementacji pokazuje użycie manipulatora kartezjańskiego z mikroskopem jako dwóch urządzeń które mogą być obsługiwane w czasie rzeczywistym poprzez transmisje na żywo w serwisie Twitch oraz Youtube.

Słowa kluczowe

Transmisja na żywo, urządzenie, API

Contents

1. INTRODUCTION 6

1.1 Live streaming 6
1.2 Online devices 6
1.3 The goal 9

2. EXISTING SOLUTION 10

2.1. Streamlabs Chatbot 10
2.2. Nightbot 11
2.3. Streamer.bot 12
2.4. SCADA (Supervisory Control and Data Acquisition) 13

3. DESIGN OF THE COMPETITIVE SYSTEM 15

3.1. Ease of use and flexibility 15
3.1.1. For hosts 15
3.1.2. For users 17
3.2. Security 18
3.3. Scalability 20
3.4. Real-time feedback 21

4. SYSTEM DESIGN 24

4.1. Device Management 25
4.2. Chat Management 26
4.3. User Management 28
4.4. Command Interpreter 29

5. IMPLEMENTATION 31

5.1. Used technologies 31
5.2. Desktop application 32
5.3. Cartesian Manipulator 37
5.3.1. Construction 38
5.3.2. Control system 40
5.3.3. Firmware 41
5.3.4. Configuration 42
5.4. Microscope 46

6. SUMMARY AND CONCLUSION 50

6.1. Contribution 50
6.2. Possible improvements 52

7. REFERENCES 54

8. LIST OF FIGURES 56

9. LIST OF LISTINGS 57

1. Introduction

This chapter describes the context of the thesis, its potential application, and the overall goal.

1.1 Live streaming

Live streaming means streaming media simultaneously recorded and broadcast in real-time over the Internet. Livestream services encompass various topics, from video games to educational content in different forms. Streams may feature big scheduled events such as esports or concerts and content created by relatively small creators and influencers for their audience, which could be about chatting, playing games, or creating art live. A big role in the experience often plays a live chat in which viewers can interact with the host and each other. The most popular live-streaming platforms include Youtube Live [1] and Twitch [2]. It is estimated that around 2.5mln viewers are on Twitch at any given moment. [3]

1.2 Online devices

Amongst the intended use cases of live streaming already mentioned, there is a niche content category in which viewers can interact via chat with some device. For example, one such device can be a feeder the stream host places on their farm, allowing viewers to feed animals using chat commands as seen in Figure 1.

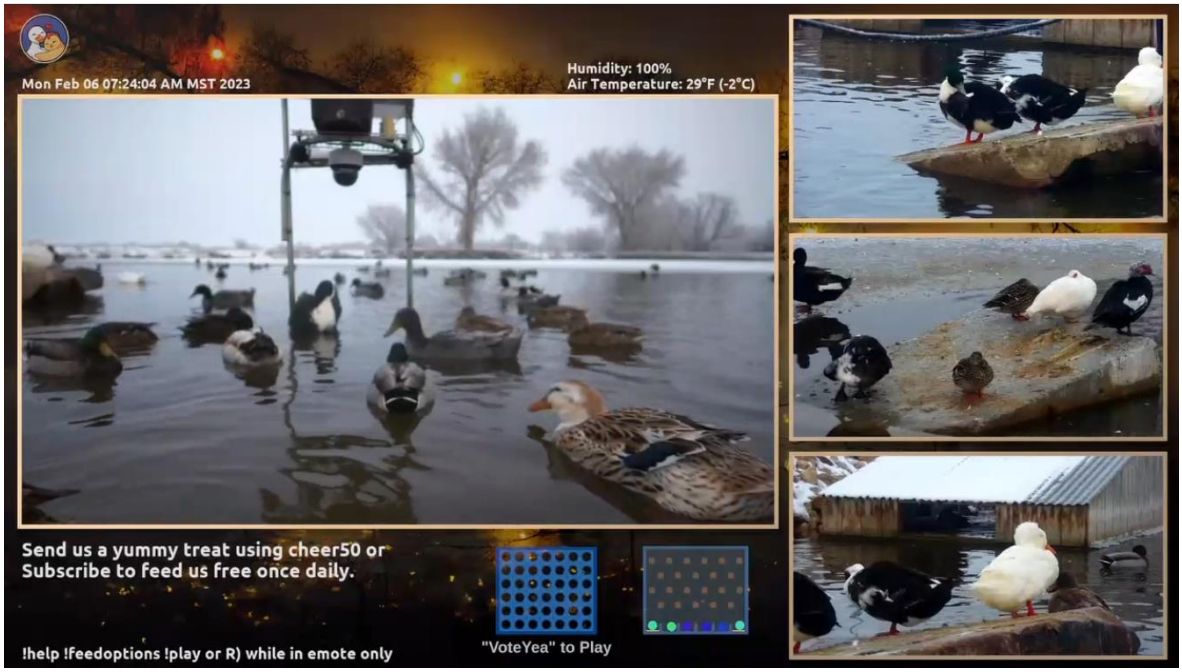

Figure 1. Example of a duck-feeding live stream.

Different content creators modified their off-the-shelf desktop pet robot Vector to make it react to chat commands. The robot seen in Figure 2 can move in a particular direction and perform other

choreographies. In this example, users have additional control over the live stream itself and can change which camera is in the main window. The host also implemented a queue of user commands.

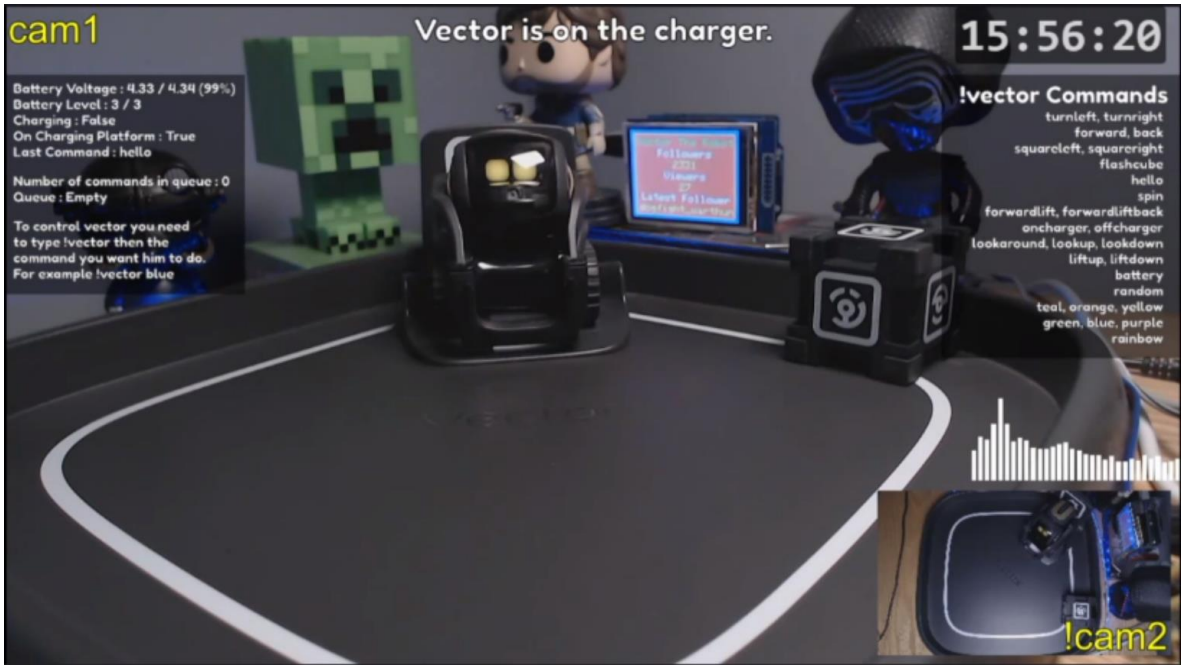

Figure 2. Live stream of a Vector robot controlled by chat.

The creators of the "Remote Reality game," Isotopium: Chernobyl [4], created their live-streaming platform to host it. The game is about controlling a physical 12-inch tracked RC car and driving on a reduced-scale 210-square-meter copy of the city of Chornobyl (Figure 3) while searching for collectibles scattered on the map. Here each user sees the camera view from their vehicle and uses a keyboard to control it as pictured in Figure 4. What's important to mention is that this game is governed by the real-world laws of physics. This means no errors or bugs often found in regular games with physics engines.

Figure 3. A person standing on a game's environment with RC tanks. Source [4]

Figure 4. A person controlling his tank and encountering another player. Source: [4]

Devices controlled via the Internet can be used for scientific and educational purposes, not just entertainment. An example is a project called Emerald Cloud Lab [5] pictured in

Figure 5, which offers researchers access to the chemical and biological laboratory online. This service improves the accessibility of science instruments and the reproducibility of experiments. One user can orchestrate multiple experiments simultaneously using a selection of available research

devices. There are also universities starting to offer similar services to their students and researchers, allowing them to increase the speed at which the experiments are conducted by queueing tasks 24 hours a day.

Figure 5. A row of research stations with scientific devices. Source: [5]

1.3 The goal

This thesis aims to design a system that allows anyone to open their devices to public service quickly and efficiently. The proposed solution will leverage existing and free-to-use live-streaming platforms to remove the complexity of setting up video-streaming-capable servers and web design. The adaptation of user and chat management will provide additional relief. Additionally, it can provide an excellent opportunity for advertisement and monetization since most big platforms have an existing user base and monetary donation service in place.

The system's design should be flexible enough that hosts can set up their devices and connect their streaming platforms without needing to know the application's inner workings with minimal effort. Hosts also need to secure their external devices against misuse or mishandling. This can be caused by uninformed or deliberated action, which is why, if possible, each command sent by the users or viewers to the device should be validated to check if it is correct or allowed in the current state of the machine. To prevent users from accessing one device simultaneously and disturbing each other work, they have to request a specified time for which they want to use a machine and join a queue.

To illustrate the system's potential, an example implementation will feature a desktop application written in Java managing multiple external devices via multiple streaming services. Implemented chat connectivity will feature Twitch and Youtube Live. As for the devices, a cartesian manipulator will be configured to operate a movable microscope with a sample selector to research given samples and view them on the live stream.

2. Existing Solution

The topic of "chatbots," or programs which are scraping the contents of a stream chat in search of commands to execute, is well known in the live-streaming industry. Chatbots have become essential for streamers to engage with their audience and manage various aspects of their live streams. Some popular usages of chatbots in streaming include:

- Chat moderation by automatically filtering and removing inappropriate messages, links, or spam. They can also enforce chat rules and issue warnings, timeouts, or bans to users who violate them.
- Viewer engagement through various interactive features, such as polls, bets, or mini-games, which viewers can participate in using chat commands.
- Custom commands that can display specific information, trigger sound effects or animations, or interact with third-party APIs. Viewers can use these custom commands to access frequently asked questions, streamer's social media links, or other interactive features.
- Timers and automated messages at regular intervals inform viewers about stream-related information, such as the stream schedule, donation links, or upcoming events. This feature helps to keep the chat active and informative.
- Integration with popular streaming tools like OBS (Open Broadcaster Software) [6] or Streamlabs [7] to control various aspects of the stream, such as scene transitions, overlays, or other visual effects.

For monitoring and management of hardware online, there are also multiple recognizable industry standards such as OPC (OLE for Process Control), MQTT (Message Queuing Telemetry Transport), or SCADA (Supervisory Control and Data Acquisition) [8]. These systems are often used in highly specific, high-security, high-reliability environments,

The next part of the chapter presents an overview of the SCADA, and a few selected popular chatbots in the context of this thesis.

2.1. Streamlabs Chatbot

The Streamlabs Chatbot is a robust and feature-rich chatbot explicitly designed for streamers on Twitch, Youtube, and Facebook Gaming platforms. Developed by Streamlabs, a leading provider of streaming tools and software, the chatbot helps automate various tasks, interact with their audience, and manage their live streams more efficiently. Streamlabs Chatbot offers several core features and unique aspects that set it apart from other chatbot solutions, such as:

- Custom commands that execute specific actions such as displaying information, triggering sound effects and animations, or interacting with third-party APIs.
- Timers and automated messages allow the chatbot to send automated messages at regular intervals to share information with viewers, such as stream schedules or upcoming events.
- Chat moderation via various moderation features to help maintain a respectful and friendly chat environment. It can automatically filter and remove inappropriate messages, links, or spam and enforce chat rules by issuing warnings or ban users.

Streamlabs Chatbot is part of the Streamlabs ecosystem, which includes various popular streaming tools like Streamlabs OBS, Streamlabs Mobile App, and more. This integration allows for seamless interaction between the chatbot and other Streamlabs tools, making it easier for streamers to manage their live streams. The Chatbot does not have a built-in feature to execute commands on the

local computer. To communicate with the external device, the user would have to write a custom Python script supported by the chatbot.

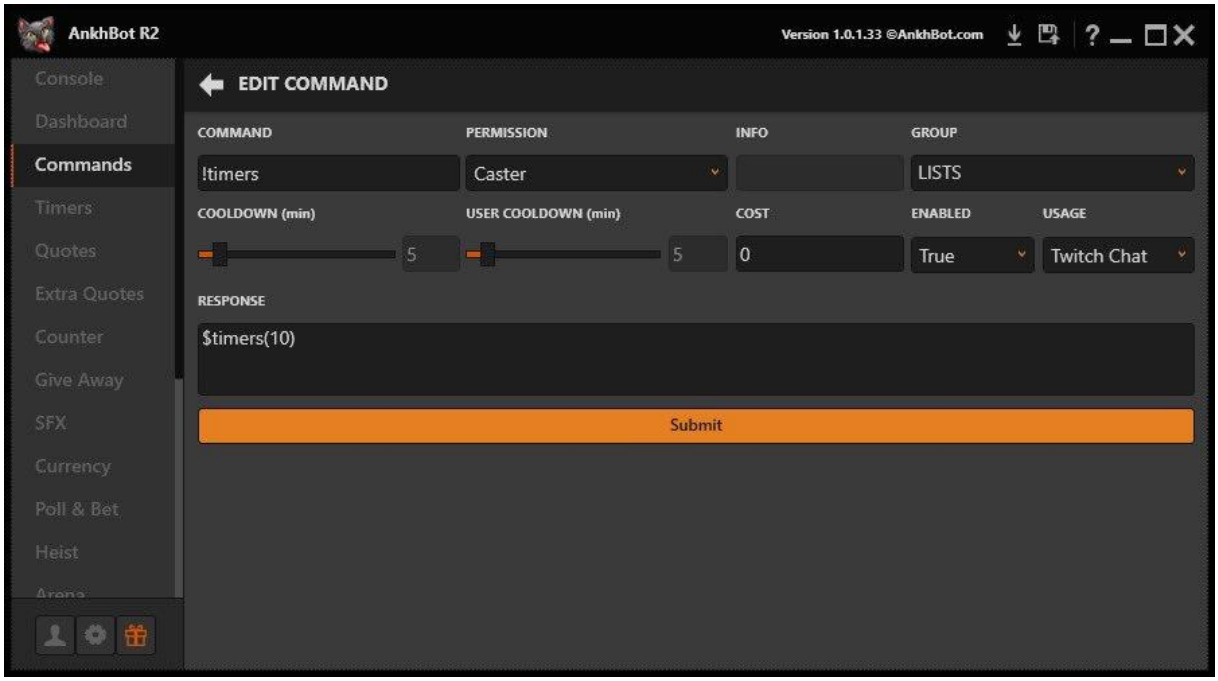

Figure 6. Example view of the Streamlabs Chatbot GUI. Source: [9]

Users of the Streamlabs chatbot who want to connect their external device would have to create a custom Python script to interact with the USB-connected device. Moreover, while Streamlabs provide some security features for chat moderation, it does not have the same level of security as the solution proposed in the thesis using commands interpreter and finite-state machines to limit the commands users can send to the device. Users need to secure the custom scripts to ensure that these do not expose devices to potential misuse.

2.2. Nightbot

The Nightbot [10] is a popular chatbot service designed to help live streamers manage their chats and interact with their viewers more effectively. Developed by NightDev, it offers similar features to the mentioned Streamlabs Chatbot, such as:

- Chat moderation,
- Custom commands,
- Timers and automated messages,
- Stream alerts

It also provides various user engagement tools, including polls, giveaways, and song requests. What differentiates it from the previous service is its ease of use via the web-based dashboard, making it accessible to those with limited technical knowledge. It is also a cloud-hosted service, which means streamers do not need to install or run any software on their local computer. This reduces system resource usage and ensures that Nightbot is always up and running. However, it makes it more challenging to integrate with external devices connected to the local computer. One of the solutions would be to use third-party API support to send user commands to the local server. This point negates

the previous advantages correlated with accessibility for non-technical people and cloud-based operation.

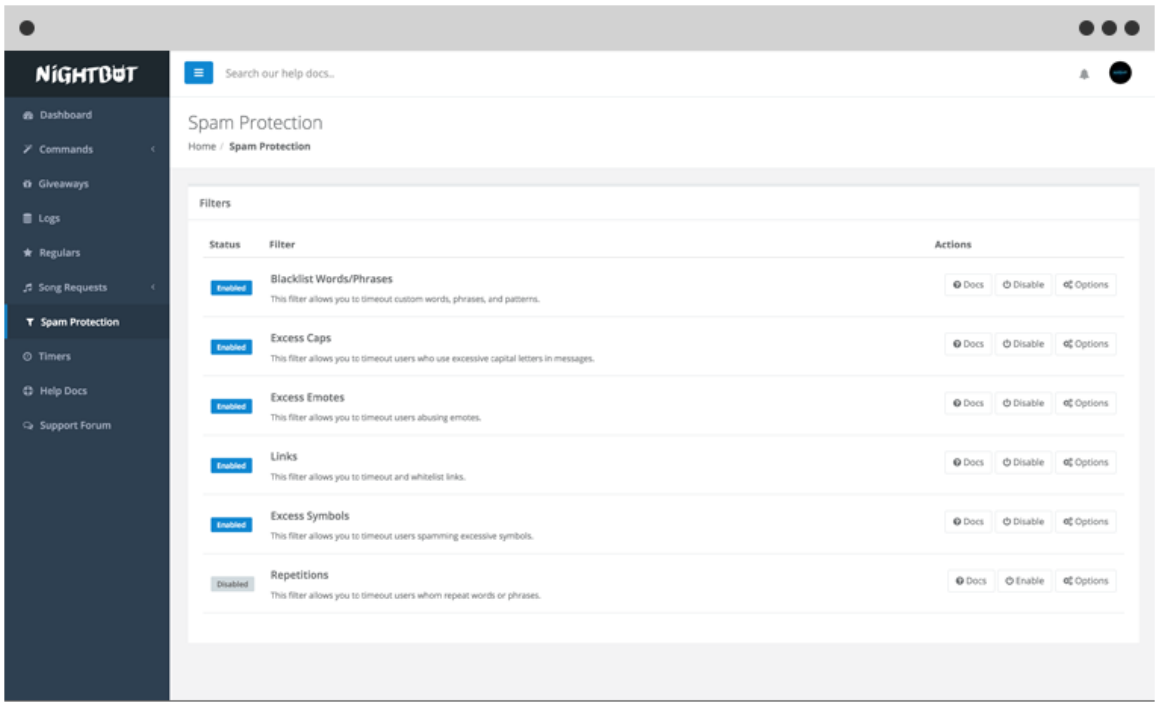

Figure 7. The example view of the NightBot dashboard. Source: [10]

2.3. Streamer.bot

The streamer.bot [11] is arguably the most powerful tool from the already mentioned. Apart from supporting all the basic features common to the Streamlabs Chatbot and Nightbot, it also provides support for custom C# scripts and methods execution. Integration with streaming tools like OBS or XSplit allows viewers to control the various aspects of the stream, such as displayed webcam, overlays, or other visual effects. It is also relatively easy to use and has comprehensive documentation, which helps users to make the most of the chatbot's features and capabilities.

Like the Streamlabs Chatbot, users who want to connect their USB-connected devices would need to write a custom script and ensure secure interaction with viewers. Controlling displayed webcams and overlays in the streaming software is a significant advantage over previously mentioned solutions. This, however, would require users to hardcode the associated webcams in the and thus limiting the systems' flexibility. Commands interpreting and usage would all have to be implemented by the users, making this solution very troublesome and time-consuming for most people.

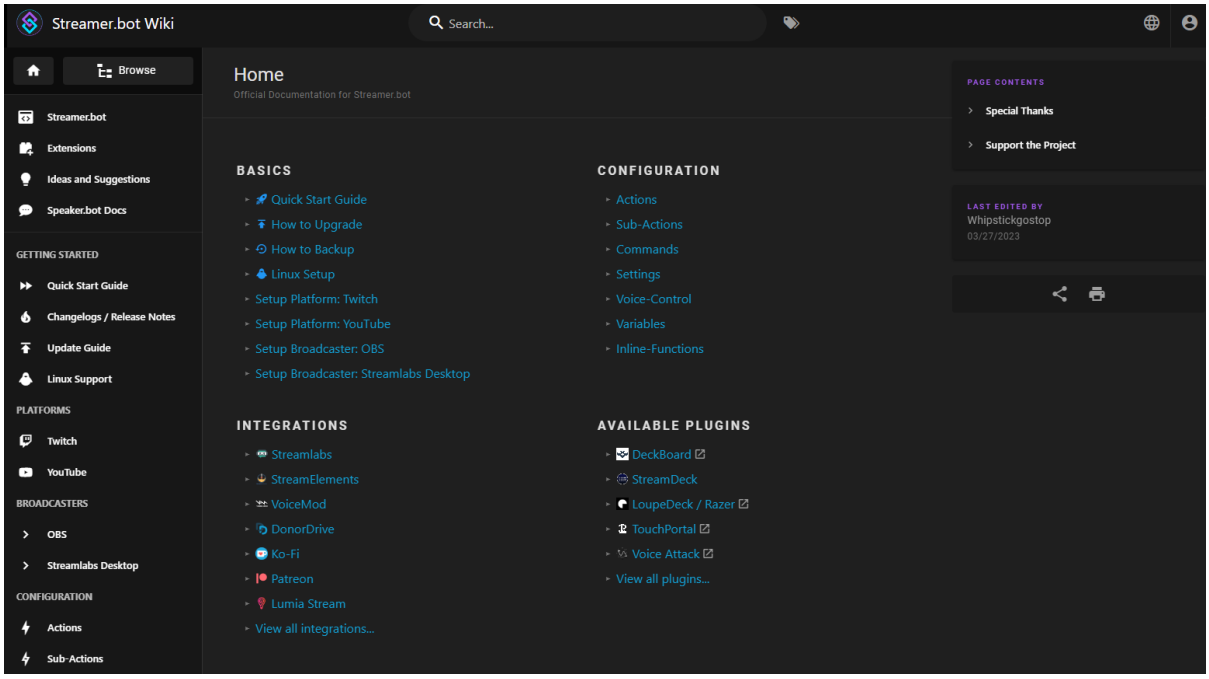

Figure 8. Image of the Streamer.bot's comprehensive documentation. Source: [12]

2.4. SCADA (Supervisory Control and Data Acquisition)

The SCADA systems are industrial control systems that monitor and control various processes in large-scale industries such as power generation, manufacturing, water treatment, and oil and gas. A SCADA system typically consists of a central control room with Human-Machine Interface (HIM) software, remote terminal units (RTUs) or programmable logic controllers (PLCs), communication infrastructure, and field devices such as sensors and actuators, as pictured in Figure 9.

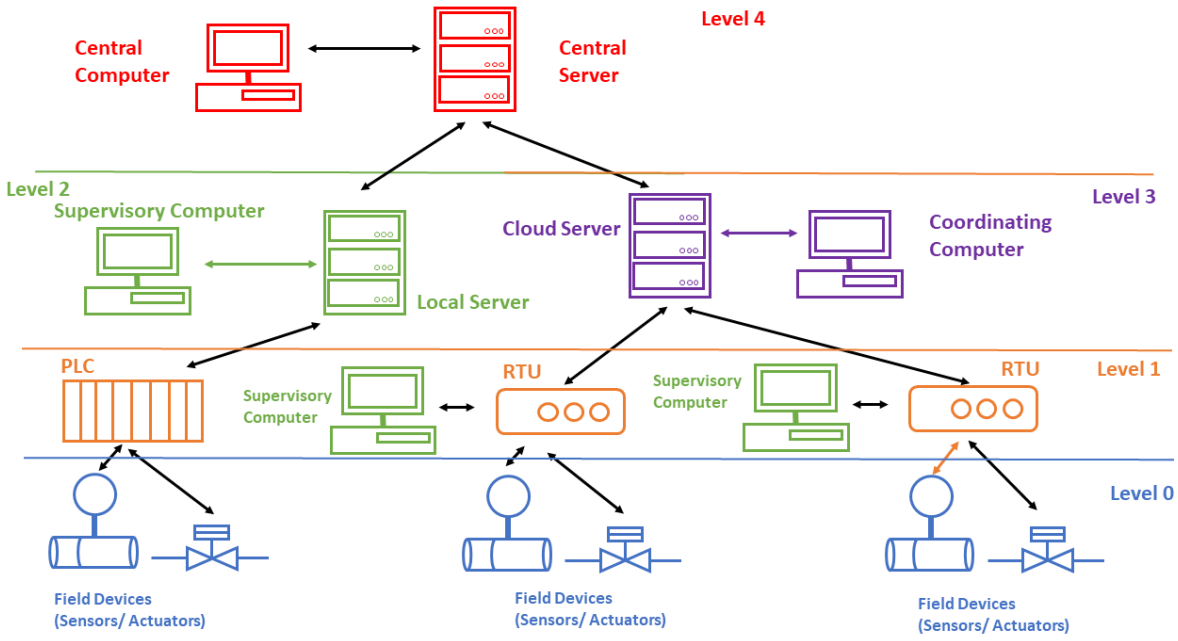

Figure 9. Architecture of SCADA System. Source: [13]

The system provides real-time monitoring of various processes, allowing operators to make adjustments and control devices as needed, ensuring optimal performance and minimizing downtime. The communication is achieved via robust and reliable protocols designed to provide nominal operation in challenging industrial environments. Remote access reduces on-site personnel and enables more efficient management of resources. Alarms generated in response to specific events or conditions can alert operators to take corrective action. These events may include equipment failure, process deviation, or other safety hazards. Because these systems are usually involved in critical infrastructure, they have many security measures to protect against unauthorized access, cyber-attacks, and other threats. These measures may include firewalls, VPNs, encryption, and other security technologies.

One of the most essential features of the system is data acquisition. The SCADA system collects data from sensors and other devices, enabling operators to analyze the performance of processes and equipment, detect anomalies and identify issues before they become critical. Collected data is stored in the SQL database allowing operators and engineers to analyze trends, identify patterns and optimize processes over time. The log collection feature is essential when working with hardware, and the solution proposed in this thesis won't be deprived of it.

One of the well-known implementations of the SCADA system is Wonderware by AVEVA. It offers extensive features for real-time monitoring and control, data acquisition, alarm management, and more. It is used in various industries in complex industrial processes.

Software like this is, however, out of reach for most users wanting to control their devices online. It involves a complex setup, often with industrial-grade hardware. Also, most of its features won't be utilized by regular users, such as excessive security or robustness, as these characteristics are primarily associated with critical infrastructure.

3. Design of the competitive system

Users who want to share their external devices with people on the Internet face several challenges when relying on off-the-shelf solutions. These solutions are often not designed with the specific requirements of remote device control. They may lack the necessary features, customization options, or security measures to enable seamless and secure sharing of external devices. Instead, there is a need for an IT system that fulfills the following requirements.

- Ease of use; The system should be designed with a user-friendly interface and simple setup process, making It accessible to a wide range of users with varying technical expertise. Users might include streamers, which should be able to add devices and configure settings with minimal hassle, and viewers, who should interact with the device through straightforward chat commands.
- Flexibility; The system needs to be versatile enough to accommodate a variety of external devices and applications. It needs to support integrating different types of hardware, allowing users to share devices ranging from animal feeders to robotic manipulators and specialized lab equipment. It should be possible to tailor the system to their specific needs by creating custom commands or configuring each device's parameters. Additionally, the system should be adaptable to various live-streaming platforms, ensuring broad compatibility and ease of integration.
- Security; Given the potential risks of granting internet users access to physical devices, the system must implement robust security measures. This includes restrictions on the types of commands that can be executed and customizable levels of user permissions. Furthermore, the system could benefit from incorporating a finite-state machine for each device, ensuring that only valid sequences of actions are allowed.
- Scalability; The solution should be designed to simultaneously accommodate multiple instances of different kinds of devices. This allows the creation of a much more complex system in which other machines performing different functions can work together or where multiple machines of the same kind can perform tasks simultaneously.
- Real-time feedback; To enhance the user experience and capabilities, the system should provide real-time feedback on the status of the devices and the outcome of the executed commands. This can be achieved through live camera feeds and chat notifications. This helps users to stay informed and engaged through the interaction.

By addressing these essential requirements, the proposed IT system offers a competitive solution for users who wish to share their external devices with the online community.

3.1. Ease of use and flexibility

Ease of use is essential when developing a system that aims to connect regular live-streaming service viewers with physical devices. To ensure a positive reception, both sides must be considered; the people who will be setting up the machines (the hosts) and the people who will be interacting with the devices remotely (the users).

3.1.1. For hosts

For hosts, the system should offer a straightforward and intuitive setup process. This includes connecting devices to the system and managing user permissions. Providing information such as name

of the device, necessary USB port information, and other configuration data, and changing the user permissions could be achieved in multiple ways.

The first and easiest way is not to give hosts any option for configuring the devices. The application itself could search for connected devices among systems' peripherals and try to connect with them using a range of preconfigured settings. The configuration related to different commands which could be executed on the device could be discarded. Instead, the device itself could decide if the provided instruction is valid and can be accepted. One of this approach's most significant downsides is that the host has little control over who can execute what on the device without changing the source code. Also, each device connected to the computer will be automatically made available to the public. All users could also have the same permissions, but this again limits the flexibility and security of the system as there could be commands that could potentially cause more damage to the system, like changing the internal configuration of the device.

Using one of many network discovery protocols like Simple Service Discovery Protocol (SSDP) could resolve some of the mentioned issues. Network discovery protocols are used by devices to identify and communicate with each other over a network. They are used to simplify the setup and management of networked devices. The SSDP is part of the Universal Plug and Play (UPnP) protocol standard. It allows the networked devices to announce their presence to other devices and provide information about their capabilities. The immediate downside of this approach is that every device must be connected to the network and have the SSDP implemented to work with the system. However, in an industrial or academic setting where dozens of devices are available, like in the remote laboratory mentioned earlier, this could be beneficial and, in some cases, necessary.

What is left requires the host to explicitly add each device to the system by providing all the necessary connection information and configuration. This can be done quickly using configuration files as these can be efficiently parsed by the program and prepared before the system starts. Data serialization languages such as YAML, JSON, or XML are good propositions because:

- They are human-readable
- They can represent hierarchical and complex data structures with nested elements, making them suitable for specifying complex configurations.

YAML seems to be a better choice for this particular application due to being less verbose than XML and does not use tags or brackets like JSON to define elements. Instead, it uses indentation and hyphens to indicate the structure of the data. It also allows for comments within the document as opposed to JSON. Because YAML is a superset of JSON, and every valid JSON file is also a valid YAML file, hosts who are used to JSON can use it, giving the system more versatility.

The example of the YAML configuration file is show in the Listing 1.

Listing 1. Basic device configuration file.

```
1 name: 'Manipulator'
2 portName: ttyUSB1
3 portBaudRate: 9600
```

Besides providing information necessary for the setup of devices, configuration files can also help hosts connect to different streaming services. By doing that, the system can parse user commands from chats of services like Twitch.tv or Youtube.com. The ability to choose a platform is essential since different creators and hosts can already have established audiences on one of these services. By connecting to multiple services at once, hosts can increase their reach.

Different platforms often require different authentication methods, so each must be implemented explicitly. Related configuration files should reflect that and provide the necessary

parameters for each platform to which hosts wish to connect. For example, the service Twitch.tv configuration file requires an *access token* from the user profile.

3.1.2. For users

For users, the system should be intuitive and straightforward to interact with. This involves both the understanding of the system and the execution of the commands to control the devices.

The system should provide adequate information about the available devices and their capabilities. This could include a list of devices, a brief description of each device, and a list of commands each device can execute. Information provided by the host in the configuration file could be presented in a user-friendly manner, such as a help command in the chat, an FAQ section on the streaming platform, or a user manual hosted on an external website. The application will have a form of GUI (Figure 10) shared through the streaming service. Some of this information can be placed on it to always be available for users.

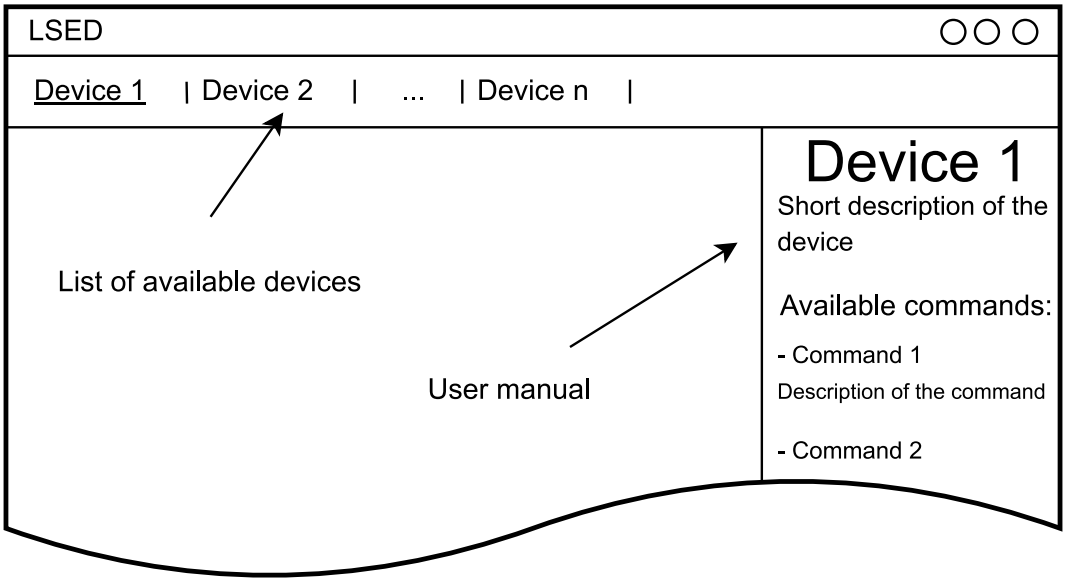

Figure 10. Mockup of GUI featuring a list of devices and user manual.

The system needs to allow users to control the chosen devices using simple and easy-to-remember commands. The command syntax should be intuitive and forgiving, with the system providing helpful feedback when an order is incorrectly entered. For instance, if a user enters an invalid command, the system could respond with a message, for example:

```
System: Command not recognized. Try again or type 'help' to see a list of
valid commands.
```

This guides the user and encourages them to continue interacting with the system.

3.2. Security

Providing access to physical devices for an unlimited audience creates a wide range of problems caused by not intentional or intentional mishandling. One way the system can help hosts limit the amount of potential damage is by allowing them to define a list of possible actions that users can perform. Specifying a command signature host can control the type of parameters appended to it. Each parameter can have an allowed range or associated list with possible keywords. This part of a YAML configuration file can be seen on Listing 2.

Listing 2. Proposition for command in a device configuration file.

```
commands:
  - name: "Move Relative"
    description: "Move device by distance of millimeters."
    params:
      - name: "XAxisDistance"
        type: Integer
        range: [-100, 100]
```

Here, the value parameter `XAxisDistance` is limited to the integer ranging from -100 to 100.

Complex machines cannot always execute all possible commands in their current state. Some attempts might lead to incorrect output or, in some cases, damage to the device. Let's imagine a scenario where user wants to check the temperature of some sample using remotely controlled laboratory equipment. In order to do this, there needs to be a temperature sensor attached to the effector of the manipulator, but after the previous operations, there is a syringe instead. Temperature readings won't be accurate, and a piece of equipment like a syringe needle or sample might brake in the process. A complex device like that might benefit from incorporating a form of Finite-State Machine (FSM) into the configuration. The host might specify what state is a result of executing a selected command and what state is requires for the execution.

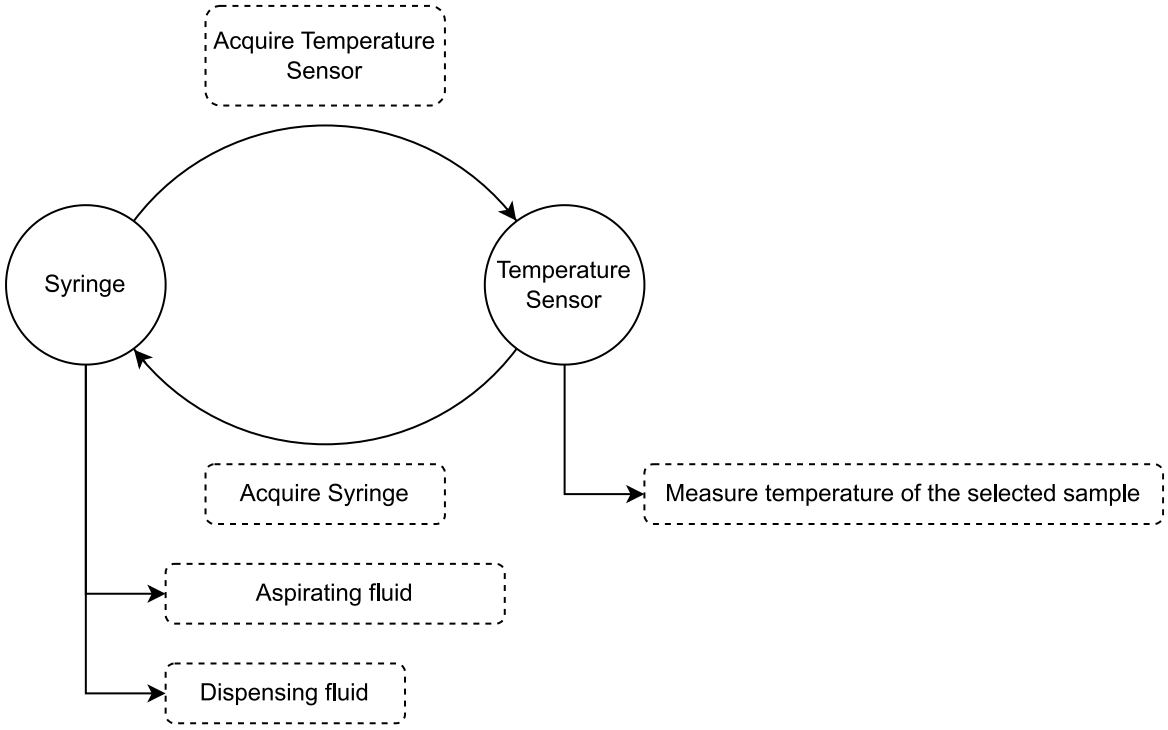

Figure 11. A graph of two possible states with available actions.

Figure 11 presents a graph with two possible states and five possible actions. The action "`Acquire Temperature Sensor`" results in the machine changing state to "`Temperature Sensor.`" The "`Temperature Sensor`" state has precisely two possible actions: "`Measure temperature of the selected sample`" and "`Acquire Syringe.`"

This FSM can be configured using commands like those pictured in Listing 3.

Listing 3. A FSM described using commands.

```
commands:
  - name: "Acquire Temperature Sensor"
    required_state: "Syringe"
    resulting_state: "Temperature Sensor"
  - name: "Measure temperature of the selected sample"
    required_state: "Temperatur Sensor"
  - name: "Acquire Syringe"
    required_state: "Temperature Sensor"
    resulting_state: Syringe
  - name: "Aspirating fluid"
    required_state: "Syringe"
  - name: "Dispensing fluid"
    required_state: "Syringe"
```

Now, users who try to perform actions unavailable for the current state of the machine can be informed by the system what is the correct state and eliminate a possible point of failure.

Hosts and administrators of the system might want to have special privileges for executing commands that are dangerous for the system, deal with finite resources, or are otherwise prone to abuse. An example of a command like this can be changing the device's internal configuration or triggering a maintenance routine to clean the machine. For that, there should be a group of users holding admin status, while commands could have additional parameters specifying if higher privileges are needed for execution. Users who cause trouble or are in some way dangerous to the system need to be banned from issuing commands. There needs to be a persistent database for storing permissions and bans, and adequate commands must be implemented so that multiple admins can ban and grant permissions online.

3.3. Scalability

Scalability is a fundamental attribute of any modern IT system, and it is especially crucial in the context of an application designed to facilitate interactions between multiple users and physical devices in a live-streaming environment. As the number of users and connected devices increases, the system must be capable of managing the interaction between them, ensuring a smooth and consistent user experience. In the context of the proposed application, scalability signifies two primary capabilities. First, the system's ability to accommodate an increasing number of devices, each with potentially different characteristics, configurations, and demands. Second, the capacity to handle a growing user base with varying permissions and interactions, often simultaneously demanding the system's attention.

A key aspect of scalability in our system lies in its capacity to accommodate multiple devices simultaneously and seamlessly. This capability allows hosts to share a wide variety of devices with users and paves the way for complex, multi-device interactions in a single setup. As discussed previously, hosts can add multiple devices through configuration files which will be automatically made available for the users, who can then select which one they want to control or execute commands on multiple devices.

Managing a diverse and growing user base is another fundamental aspect of scalability in the system. This includes managing user permissions and interactions and implementing a fair and efficient method for managing control queues (Figure 12). Users who are not banned can issue a control request for a specified amount of time. This request will be added to the line, and when it is time, the user who sends it will be able to execute commands on all devices. The queue should be on the GUI so users can see how long it takes until it is their turn.

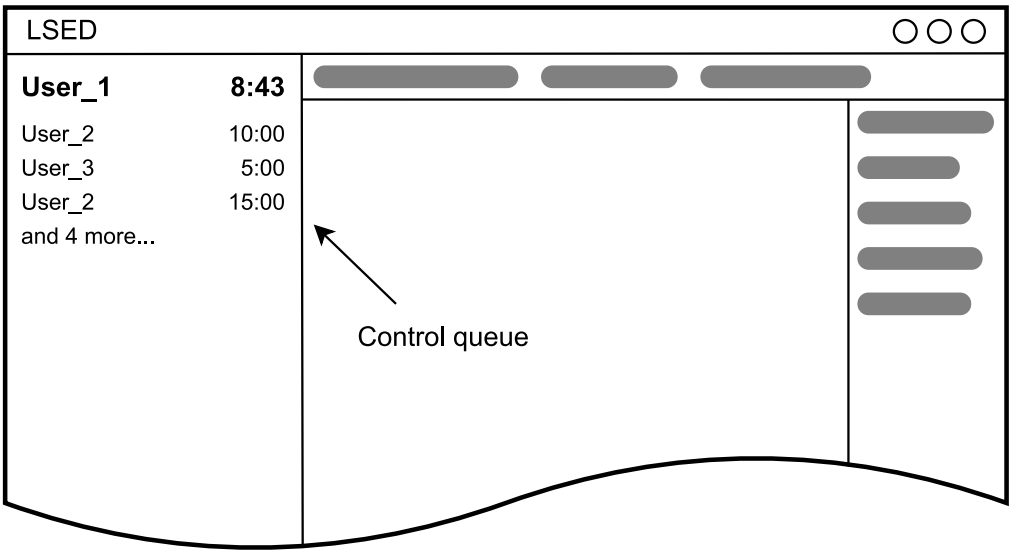

Figure 12. Mockup of GUI featuring the control queue.

3.4. Real-time feedback

In interactive systems, real-time feedback plays a crucial role. This feature allows users to understand the status and impact of their commands, enhancing their engagement and trust in the system. Real-time feedback is also vital for hosts, as it helps them monitor and control the devices effectively.

The proposed system provides immediate confirmation whenever a user issues a command. This acknowledgment presented on the GUI assures users that their input has been recognized and is being processed. Furthermore, users are informed about any mistakes they made in the command syntax or in situations when the issued command is not available in the current state of the machine. This feedback loop allows users to track their interactions with the device and adjust their orders accordingly.

In addition to the command list, the system provides a way for the devices to show their status on the log list. Each input and output of the device is tracked and displayed on the GUI for the users and hosts to examine. These logs include operational status, changes in configuration or state, and any alerts or error messages. Updates help users understand the device's current state and how their commands affect it. A mockup of the GUI with both discussed lists can be seen in Figure 13.

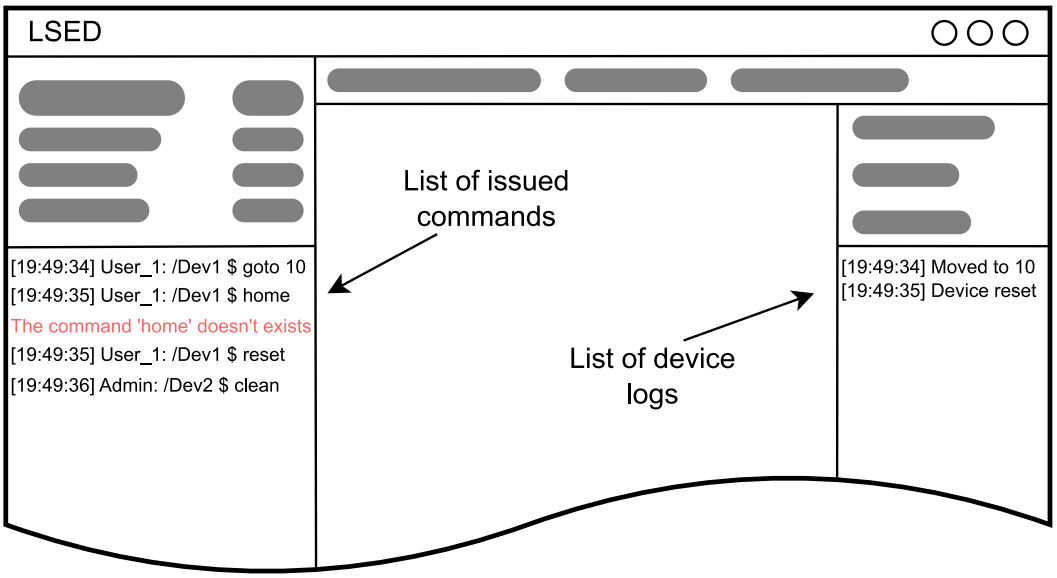

Figure 13. Mockup of GUI featuring a list of commands and logs.

Since the whole point of using a live-streaming service is to provide a visual experience, the main feedback is in the form of camera footage. While not mandatory, hosts are strongly advised to connect one or more webcams to monitor the device. The webcams can be shared between devices, or each can have its own set. Users can control which view is in focus using commands, while a preview of each camera is visible as a thumbnail. This creates a dynamic and complex scene with lots of information which maintains engagement even during the wait times as pictured in Figure 14. Transparency in the feedback also allows users to identify and correct errors in their commands or quickly react to changes in the device environment.

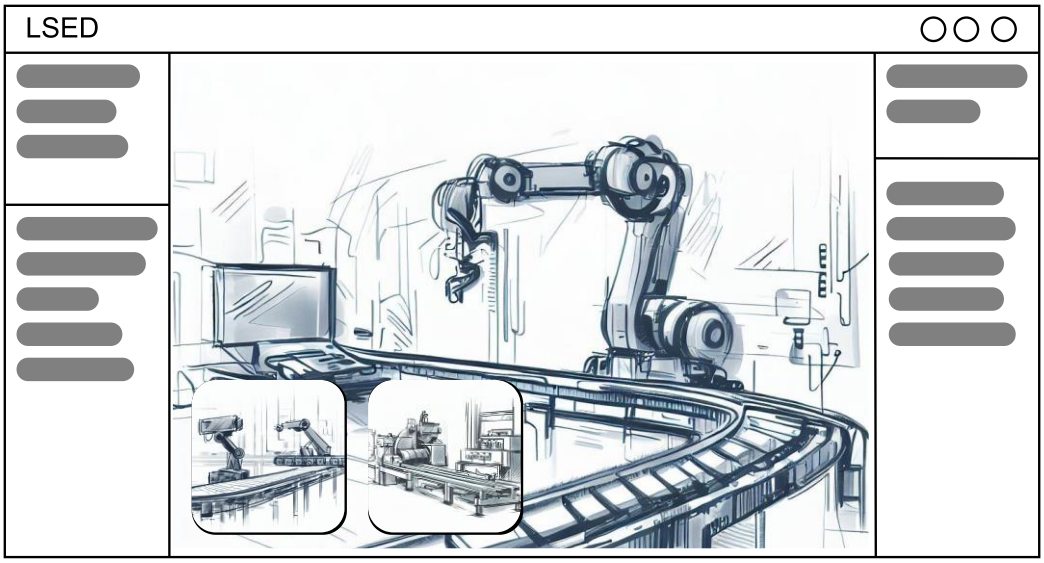

Figure 14. Mockup of GUI featuring different camera views.

A list of webcams can be added to each device configuration file, as seen in Listing 4.

Listing 4. A list of webcams added to the configuration file.

```
name: 'Manipulator'
cameras:
  - name: "Effector_View"
    portName: "/dev/video0"
  - name: "Wide_Shot"
    portName: "/dev/video1"
  - name: "Z_Axis_Zoom"
    portName: "/dev/video2"
```

Real-time feedback is a crucial part of the system. It enhances the user experience, builds engagement, and ensures effective device control. It makes the experience enjoyable for viewers and users in control. It is designed to work with multiple devices at once and provide multiple different kinds of information.

4. System Design

The LSED (Live Stream External Device) system is designed as a desktop application that runs on a host computer. The primary role of this application is to facilitate the interaction between live stream viewers and external devices connected via USB to the host's computer.

The system operates in tandem with a broadcasting tool chosen by the host. The broadcasting tool aims to share the application's window with online viewers. Hosts can select their preferred tool, ranging from browser-based options to free and open-source desktop applications. A popular choice among desktop applications is OBS (Open Broadcaster Software). OBS offers compatibility with various streaming services such as Twitch, Youtube Live, Facebook Live, and TikTok Live. Additionally, it provides hosts with a suite of tools enabling them to overlay text, icons, and other elements onto the LSED application window, enriching the viewer experience without requiring modifications to the LSED source code.

To facilitate interaction between viewers and external devices, LSED must implement the chat API of the streaming service chosen by the host. This implementation enables live-stream viewers to submit commands via the streaming service's chat feature. LSED interprets these commands based on instructions provided in configuration files. Subsequently, commands are translated into instructions transmitted to external devices for execution. Viewers can then witness the effects of their commands in real time via webcams, creating a continuous cycle of viewer interaction.

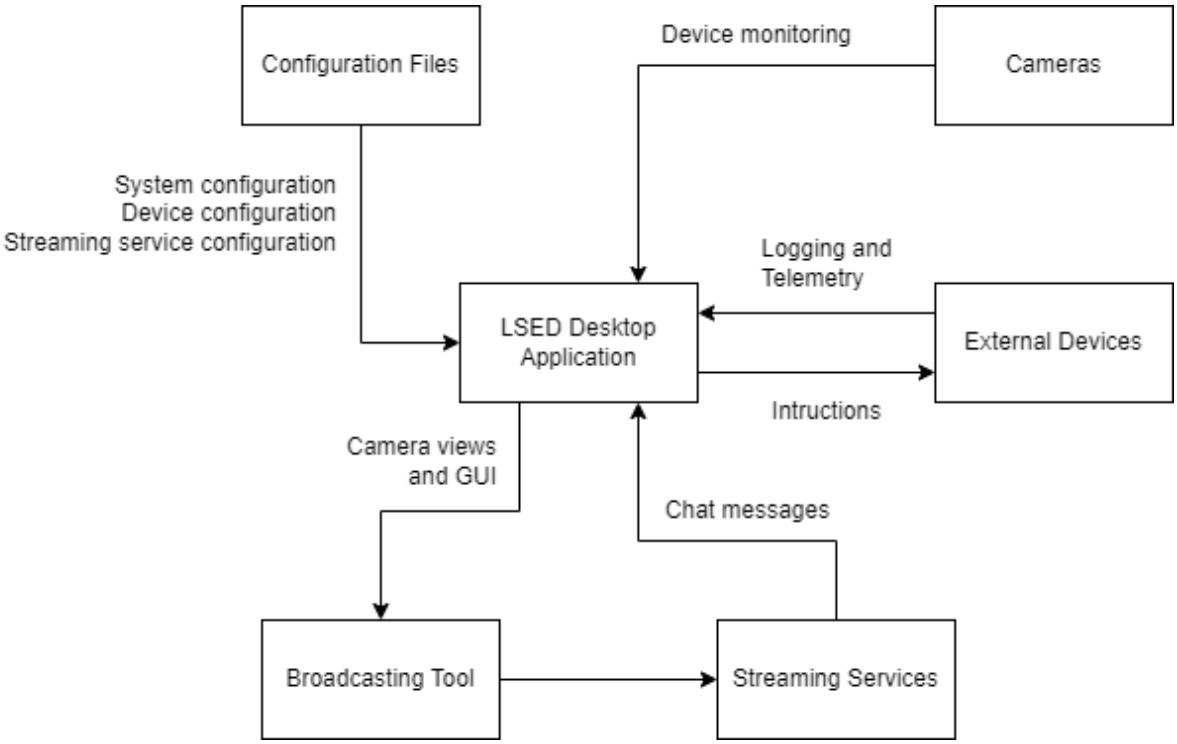

Figure 15. General Architecture View.

4.1.Device Management

Device Management forms a foundation for the LSED system as it is responsible for managing the connected devices. This involves a variety of tasks, such as:

- Establishing connections with devices
- Sending instructions
- Receiving feedback
- Managing the state
- Ensuring the proper utilization

The objective of the Device Management component is to provide a uniform interface for interacting with a diverse range of devices, regardless of their specific characteristics.

To accomplish this, it is structured around a set of key classes and interfaces, as seen in Figure 16. The classes are used to abstract the complexities of interacting with physical devices or building the representation of the devices in the system.

- `External Device`; This class provides a digital representation of the connected external devices. It serves as a reference point for various device characteristics like name, available commands, current state, and linked cameras within the system. `ExternalDevice` manages a list of instructions, transmitting them synchronously via USB to the associated physical device. Any feedback received from the device is stored as timestamped messages, ready to be displayed on the GUI.
- `Serial Com`; This class is essentially a facade over the library responsible for managing USB port communication. It initializes a connection using the serial port name and baud rate and offers methods to add serial port data listeners, facilitating real-time device feedback.
- `External Device Builder`; A part of the builder design pattern implemented to generate `External Devices` based on the YAML configuration file. `External Device Builder Director` uses it to compose the final representation. It has multiple methods for creating each external device field, taking data transfer objects (DTOs) as input. The DTOs result from paring the configuration files using the snakeyaml [14] library. Using the builder pattern helps organize the building process and spreads the responsibility across multiple classes.
- `Device Manager`; The main class of the component. Responsible for managing the devices and for delivering commands sent by users. It also has unique system commands that the users can use to change the device or camera in the main view. With the help of the `User Manager`, it decides whether the received command should be interpreted or with what priority.

By handling these responsibilities, the `Device Management` component ensures a fluid user experience, bridging the gap between digital commands and physical interactions.

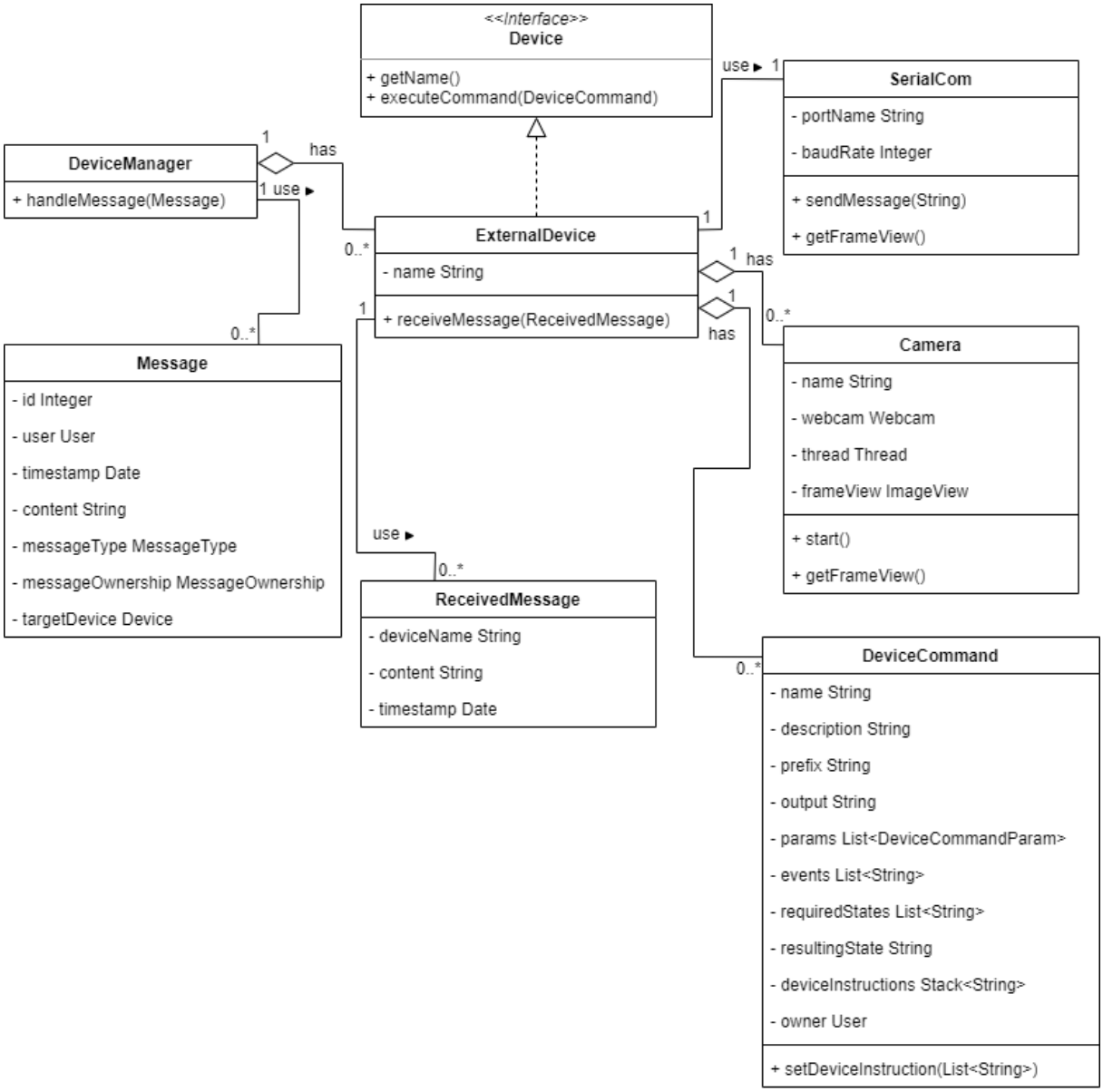

Figure 16. Device management class diagram.

4.2. Chat Management

The `Chat Management` component serves as a primary interface for handling live chat streaming services and managing the flow of messages. It is designed to interact with live streaming platforms, facilitating real-time communication. The component's primary responsibilities are:

- Connecting with different live-streaming platforms
- Managing incoming messages
- Classifying and processing messages
- Routing messages to the appropriate components
- Handling chat service configurations

The architecture of the `Chat Management` component simplifies the interaction with multiple streaming platforms and provides a standardized way of dealing with messages.

- `Chat Service`; Simple interface providing essential characteristics of the chat service like name or icon for displaying.
- `Twitch` and `Youtube`; Classes that implement the `ChatService` interface and are dedicated to handling specific live-streaming platforms. They contain the necessary logic and tools to connect and fetch chat messages from their respective platforms.
- `Chat Builder`; Utilized to create an instance of a specified chat service. It uses the provided YAML configuration file to extract the necessary authentication information.
- `Chat Manager`; The core class of this component. It handles and routes incoming messages. It has a list of `ChatServices` and is responsible for dispatching new messages to the appropriate components of the system.
- `Message`; Represents a single message in the system, encapsulating features like content, user details, target device (if specified), message type, and ownership.
- `MessageOwnership` and `MessageType`; These enums classify the messages based on ownership and type.

By handling these responsibilities, the `Chat Management` component ensures smooth and efficient communication in the system, catering to the dynamic nature of live-streaming chats. The example of interaction between various components around the message issued by the viewer can be seen in Figure 17.

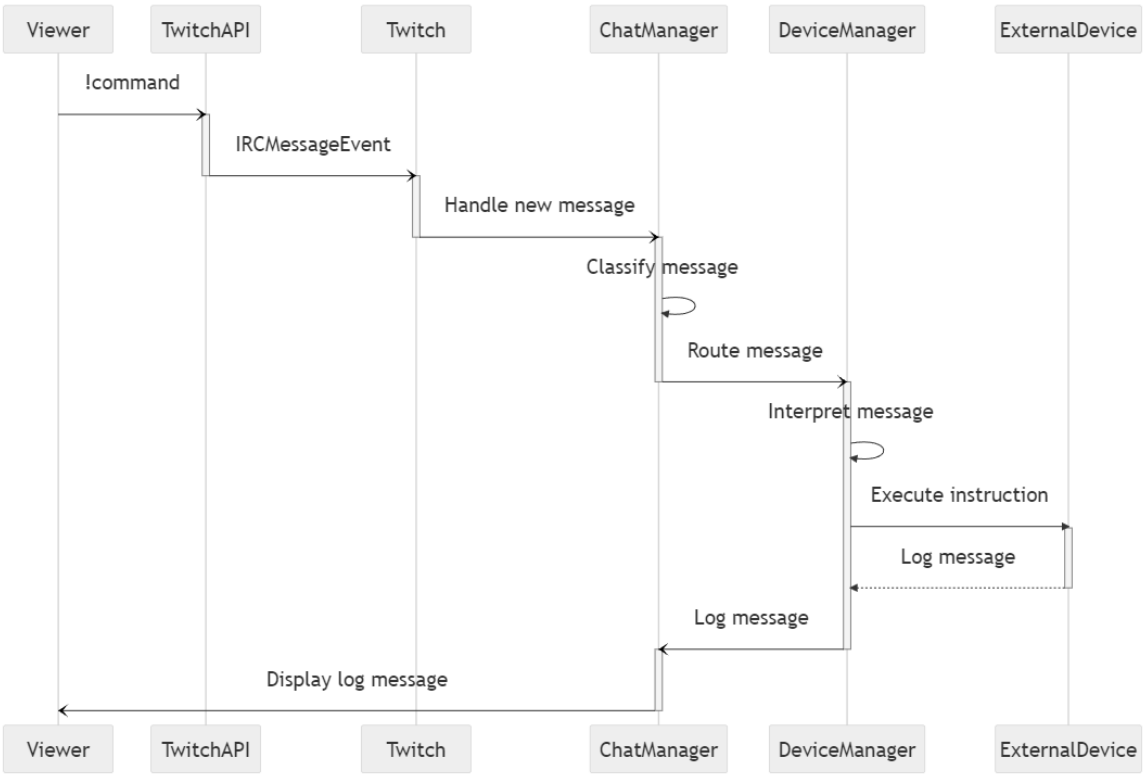

Figure 17. User command sequence diagram.

4.3. User Management

The `User Management` component is important for maintaining an orderly and fair usage of the LSED system. Its primary responsibility is to manage user interactions, ensuring the application operates smoothly even in scenarios where multiple users want to interact concurrently. This component facilitates the following:

- Management of users' control requests queue
- User control access time regulation
- Management of user permissions

These functions are provided by several classes, as can be seen in Figure 18, with an example sequence diagram of control request in Figure 19:

- `User Manager`; This is the principal class within this component. It maintains a list of current control requests and oversees user interaction with the system. It manages the state of the user currently in control and counts the time for changing the control ownership for the following user in the queue. Additionally, the `User Manager` administers a set of commands that users can employ to request control over the system for a specified period. There are also commands for admins to ban or unban users. The list of admins is obtained from the system's configuration file.
- `User`; Encapsulates the properties of an individual user. It includes the user's name and attributes indicating their status as either an admin or a banned user.
- `User Database`; Offers a simple way to persistently store `User` properties in a file. It comes into play when checking or altering a user's admin or ban status.

The `User Management` component ensures that the system is fair and secure. It regulates control access and provides essential administration tools for managing users' privileges.

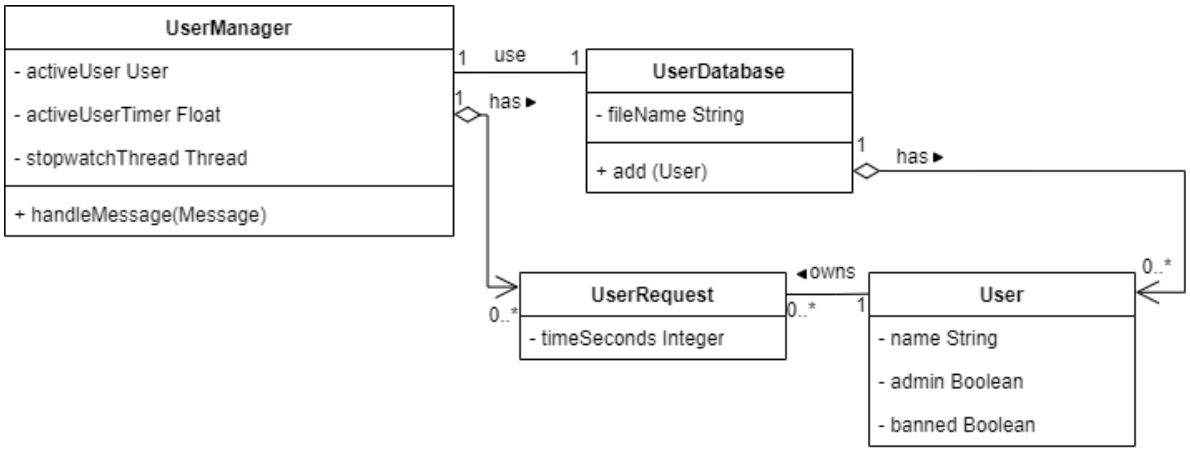

Figure 18. User management class diagram.

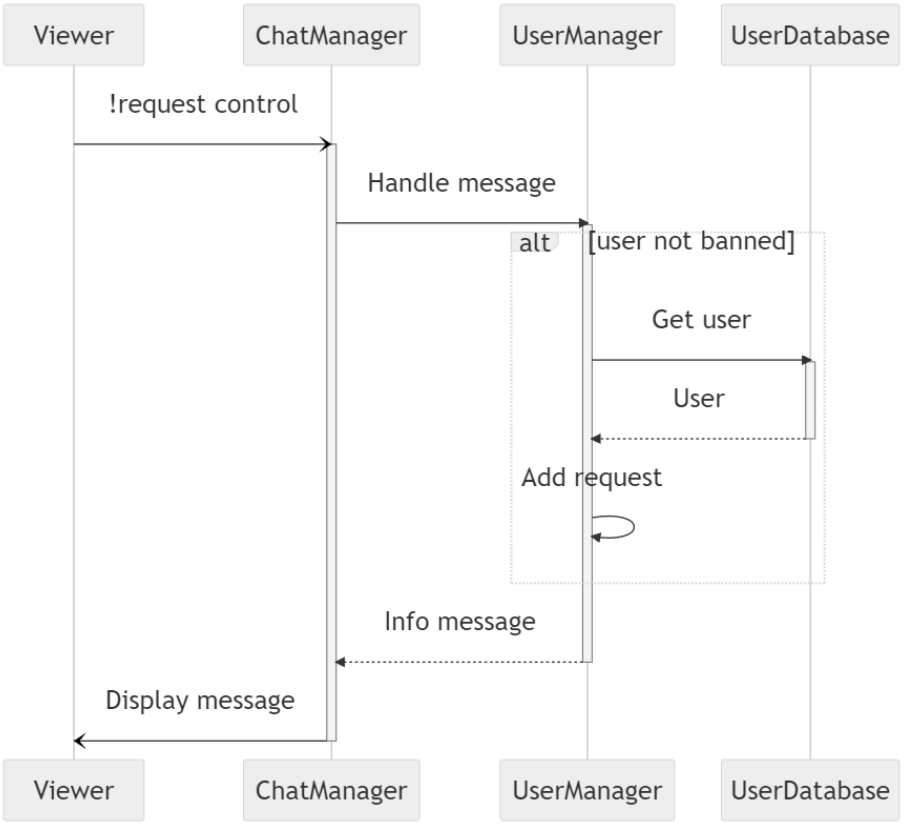

Figure 19. User control request sequence diagram.

4.4. Command Interpreter

The `Interpreter` component is integral to the LSED system, translating user commands into specific instructions that the devices can execute. It is designed to identify and build command messages and transform them into `ExternalDevice` instructions. The component's responsibilities include:

- Identifying messages that are commands
- Creating and validating `DeviceCommand` objects from `DeviceCommandDTO`s
- Translating commands into `ExternalDevice` instructions
- Handling the specifics of different device command parameters

The architecture of the Interpreter component is relatively simple and centered around a few classes:

- `Interpreter`; This Is the central component, containing methods for interpreting commands from messages and building `DeviceCommand` objects from `DeviceCommandDTO`s The command-building functionality is used when creating `ExternalDevice` representation from the YAML configuration file. It also identifies command messages, checks for matching device commands, validates their correctness and generates instructions for the target device.
- `Device Command`; Represents a single command for a device, encapsulating properties such as name and description, prefix and parameters, or required and resulting state.

The `Interpreter` uses the prefix "`!`" to identify command messages. When a message is recognized as a command, the content of the message is parsed and interpreted, generating instructions from the `DeviceCommand` instance to the corresponding external device. The component forms a bridge between user instructions and device actions by validating and interpreting command messages as seen in Figure 20.

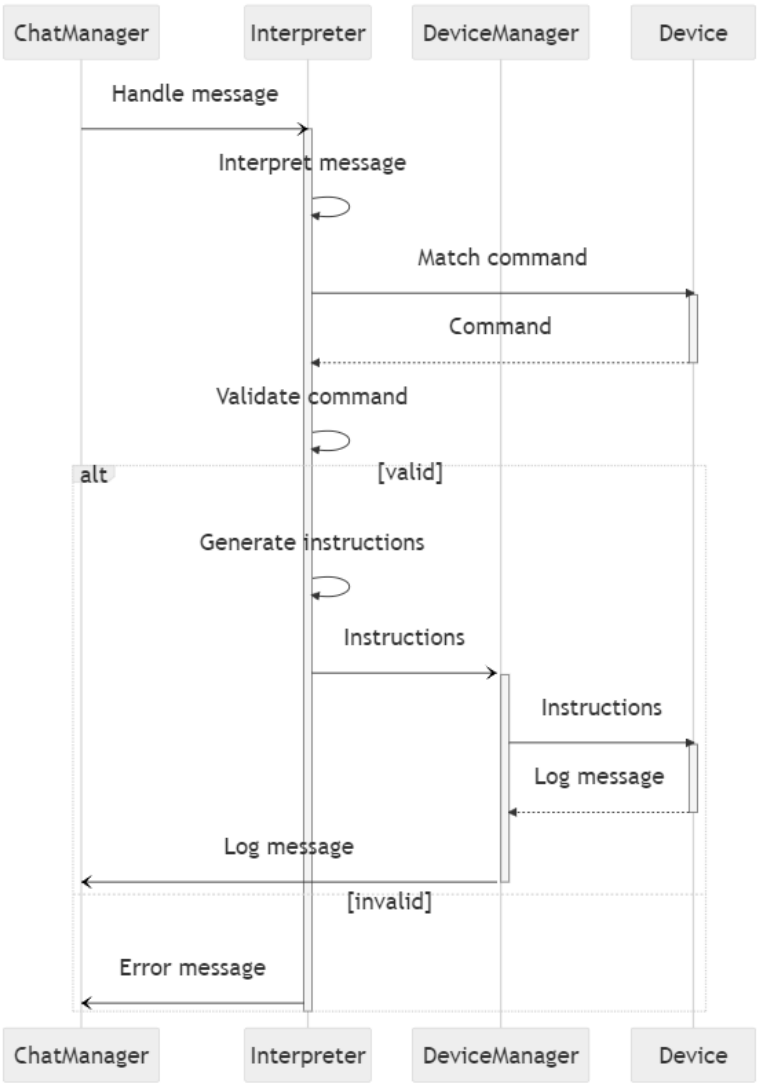

Figure 20. Simplified Interpreter sequence diagram.

5. Implementation

The previous chapters outlined the core requirements of the system as well as the overall architecture of the LSED system. This included designing key elements like `Device Management`, `Chat Management`, or command interpretation. This chapter shows the finished implementation example, including external devices built specifically for that purpose. First, there will be a description of the key technologies used for the implementation. After that, elements and functions of the desktop application will be presented, and lastly, there will be a description of two robots, the cartesian manipulator and the microscope, with the sample selector with the configuration allowing them to interact with each other.

5.1. Used technologies

The list of key technologies used in the implementation of the system:

- Java [15] – The core programming language used to implement the system. Java is a popular general-purpose language well-suited for developing desktop applications. Some key advantages that made Java a good fit for this project:
    - Platform independence – Java code can run on any platform like Windows, Linux, or Mac with the Java Virtual Machine (JVM). This makes the deployment flexible.
    - Strong typing – Java's static typing helps catch errors during compilation that might otherwise cause runtime crashes. This improves robustness and assists with development.
    - Multithreading – Java provides built-in threading support, allowing concurrent processing critical for a real-time system.
    - Ecosystem – A vast ecosystem of open-source libraries could be leveraged to accelerate development.
- JavaFX [16] – A standard GUI library for Java-enabled quick responsive UI construction. It provides:
    - APIs for UI controls like buttons, text, images, etc. These are used to construct the application's screens and interface.
    - Bindings to link UI visual state to data models and view models. This keeps data and views in sync.
    - Layout panes to organize UI controls in flexible ways and make them suitable for all screen sizes.
    - CSS stylesheets to customize visuals like colors, fonts, etc.
    - Canvas and media support for rendering images and videos (including webcam footage).
- jSerialComm [17] – A open-source Java library used to interface with computer serial ports. It provides a complete serial port API that allows LSED to communicate with connected devices over USB or serial interfaces. Some key features that made the jSerialComm suitable for this project:
    - Cross-platform – abstracts underlying system differences for serial communication allowing LSED to work on all major operating systems.
    - Simple API – high-level methods like `openPort()`, `readBytes()`, or `writeBytes()` make serial port communication easier.
    - Events/Listeners – allows async event-based programming by registering listeners that trigger when data is available.

- Port discovery – scanning methods enable easy discovery of connected serial devices.

jSerialComm handled all the low-level serial port protocols and conversions, enabling easy high-level integration with `ExternalDevice` classes.

- SnakeYAML [18] – A Java library for parsing and generating YAML data. LSED uses SnakeYAML to load the YAML device configuration files into instantiated `Device` objects that the application can use. It enabled seamless integration of YAML configuration LSED's Java-based object model.
- Twitch API [19] and Youtube API [20] – Publicly available APIs to enable real-time chat integration with streaming platforms. These are used by the `ChatService` implementations to allow communication between LSED and the streaming chat.
- Jackson Library [21] – A Java library that provides functionality for converting between Java objects and JSON data. In the LSED system, it is used for:
  - Serializing Java objects like Users into a JSON representation for persistent storage.
  - Deserializing the stored JSON back into object instances when required.

  This allows LSED to save application data like users, banned user lists, etc., in a persistent, user-readable JSON format. Jackson provides high-performance JSON conversion capabilities out-of-the-box, making it well-suited for this application's data persistence needs.

5.2. Desktop application

The finished desktop application's graphical user interface can be seen in Figure 21. It consists mainly of elements already mentioned in Chapter 3. Additionally, the GUI is expanded by the second window intended for administrative use. It has input elements that allow system operators to issue commands directly through the external devices' serial ports and send commands that will be executed without waiting in the queue, even when other users are using the system. The additional window is called "LSED Options" and can be seen in Figure 22.

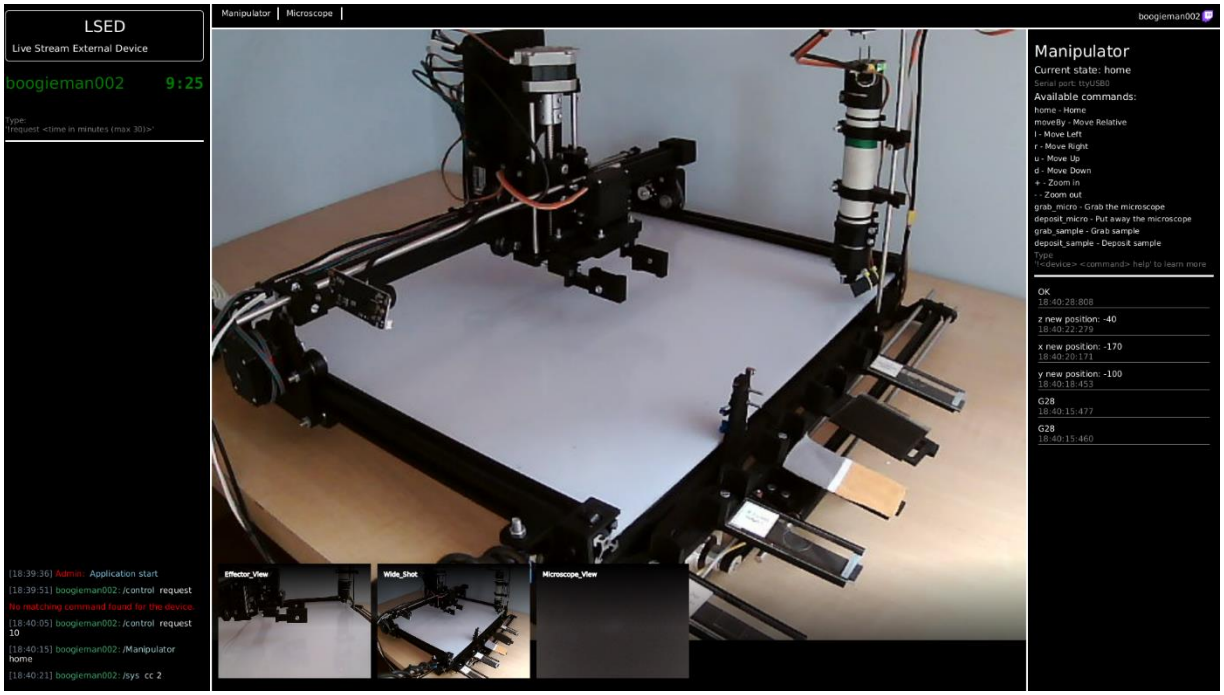

Figure 21. A graphical user interface of the LSED application implementation.

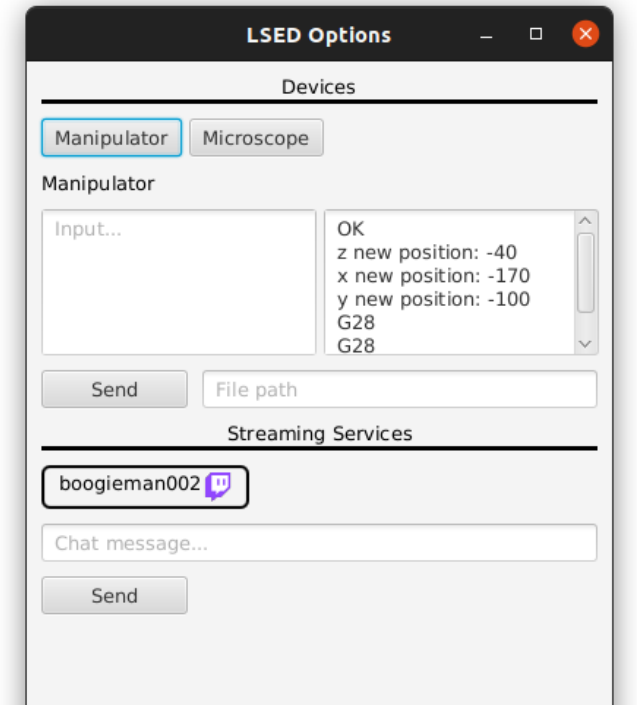

Figure 22. The additional window for the host.

The UI components are bound together using the observer pattern and properties to maintain synchronization. For example, selecting a device from the Devices toolbar in the LSED Options window binds its active camera feed to the main view area, as seen in Listing 5.

Listing 5. Fragment of the AuxiliaryWindow.java highlighting device selection button.

```
  public VBox generateDeviceOptions(){
      […]
      HBox devicesHBox = new HBox();
      devicesHBox.setSpacing(5);
      for(int i = 0; i < deviceManager.getDevices().size(); i++){
        int finalI = i;
        Button deviceButton = new
Button(deviceManager.getDevice(i).getName());
        deviceButton.setOnAction(event -> {
            deviceManager.selectedDeviceIndex.set(finalI);
        });
        devicesHBox.getChildren().add(deviceButton);
      }
      devicesVBox.getChildren().add(devicesHBox);
      […]
      return devicesVBox;
  }
```

Listing 6 shows how the `DeviceManager` class uses the `IntegerProperty` class provided by the JavaFX framework to react to the selected device change. These classes allow, among others, to add event listeners, which will be executed each time the value changes. The new index of the selected device is then broadcasted among subscribers.

Listing 6. Fragment of the DeviceManager class highlighting how it leverages the IntegerProperty class.

```
public class DeviceManager implements Device, MessageSubscriber {
  […]
  public IntegerProperty selectedDeviceIndex = new
SimpleIntegerProperty(0);
  private ArrayList<DeviceChangeSubscriber> deviceChangeSubscribers =
      new ArrayList<>();

  public DeviceManager(ChatManagerMediator mediator, UserManager
userManager) {
      […]
      selectedDeviceIndex.addListener((observable, oldValue, newValue) ->
{
        if(newValue.intValue() == -1) return;
        changeSelectedDevice((Integer) newValue);
      });
  }

  public void changeSelectedDevice(Integer id){
      logger.debug("Change device to device nr: " + id);
      selectedDeviceIndex.set(id);
      deviceChangeSubscribers.forEach(i ->
i.deviceUpdate(devices.get(id)));
  }

  […]
}
```

The subscriber pattern is used in a number of areas throughout the system. It is a software design pattern that enables objects to subscribe to events happening in other objects. In this pattern, there is a publisher object with events other objects may want to know about, like `DeviceManager` with its `selectedDeviceIndex`. Some subscriber objects also wish to receive notifications when the publisher object has a new event. For example, one of the subscribers interested in the selected device index change is `MainWindow` class showing mentioned earlier active camera feed associated with the device.

The main benefits of the subscriber pattern are easy extensibility, separation of concerns, and loose coupling between the publishers and subscribers, since publishers don't need to know about all subscribers explicitly, and new subscribers can be easily added without changing the publisher code.

Another example of leveraging the subscriber pattern and properties is the already-mentioned functionality allowing hosts to send commands directly to the external device. When the command is entered in the LSED Options window, the content is sent via the "`deviceSendCommand`" to the

device, which notifies received message subscribers. This subscriber can be `DeviceMediator`, responsible for containing `ObservableList` of received messages. This list has a listener in `MainWindow`, which updates the log list for the device on the GUI. The sequence diagram shown in Figure 23 shows how different elements are engaged with limited coupling.

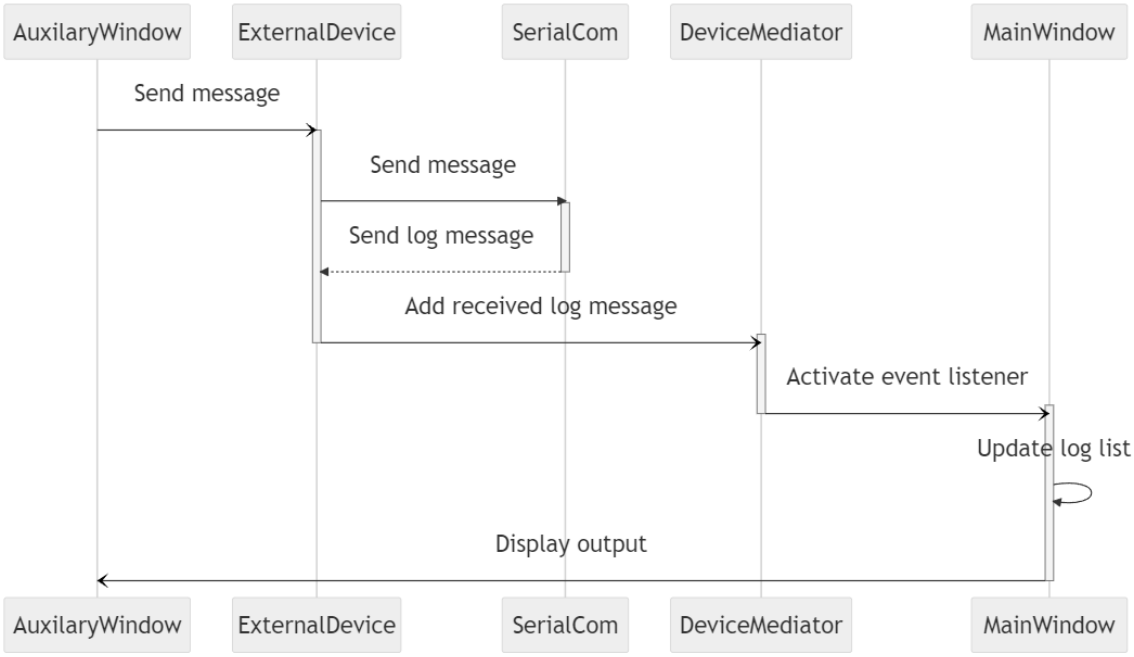

Figure 23. A sequence diagram for sending commands directly to the device.

Several system commands are hard-coded to allow users and admins to interact with the system through live-streaming services. These are defined for device management and user management areas:

- Device management
    - `Change camera` - Change main camera views to one available for selected devices by providing the camera name or index. Each camera view for the device is visible to the users at the bottom of the window so users can easily decide what part of the machine they want to focus on.
    - `Change device` - Change the selected device by providing a name or index. All available devices are seen on the upper part of the window. Changing the selected device shows all its camera views, logs, and available commands. It also shows in what state the device is currently if it happens to be an FSM.
- User management
    - `Request control` - Request control over devices by specifying the number of minutes and entering the queue.
    - `Ban user` - Ban user by username. This is immediately saved in the database file so the information is persistent between sessions.
    - `Unban user` - Unban user by username.

The request control command is the most important one since it allows users to issue commands to the devices added to the system. The workflow begins when a user gives a chat command such as "`!request 10 username`" where the number is the duration in minutes. This command is picked up by the `ChatManager` and routed to the `UserManager` component. Within the `UserManager`, the `handleMessage` method extracts the request time and username parameters from the command text. It encapsulates them in a `UserRequest` object, which pairs the `User` with their requested timeslot. The `UserManager` checks if this user already has a pending request in the queue. If not, it looks up the `User` object in the database. This allows associating permissions and request history with each user. The `UserRequest` is then added to a request queue and implemented as an observable list. `User` interface elements like the request list in `MainWindow` class are listeners on this queue. So the UI automatically displays the updated request list after the addition, as seen in Figure 24.

Figure 24. UI element showing the queue of user control requests.

In the background, the `UserManager` maintains the state of the currently active user controlling the system and their remaining time via a countdown timer. When this timer expires, the `UserManager` dequeues the next `UserRequest` from the queue if available. It then sets the requesting `User` as the new active user, granting them exclusive control. The timer is reset to the requested timeslot duration. Only this user's commands will be passed through to devices while active.

After the time expires, the slot is freed up for the subsequent user request. By cycling through requests in a structured queue, the LSED system allows shared access between users. The encapsulated `UserRequest` object coordinates the request lifecycle throughout.

All messages are formatted and presented on the GUI. To allow users easier distinguish between all types, each is presented differently. Examples of messages and how they are shown can be seen in Figure 25.

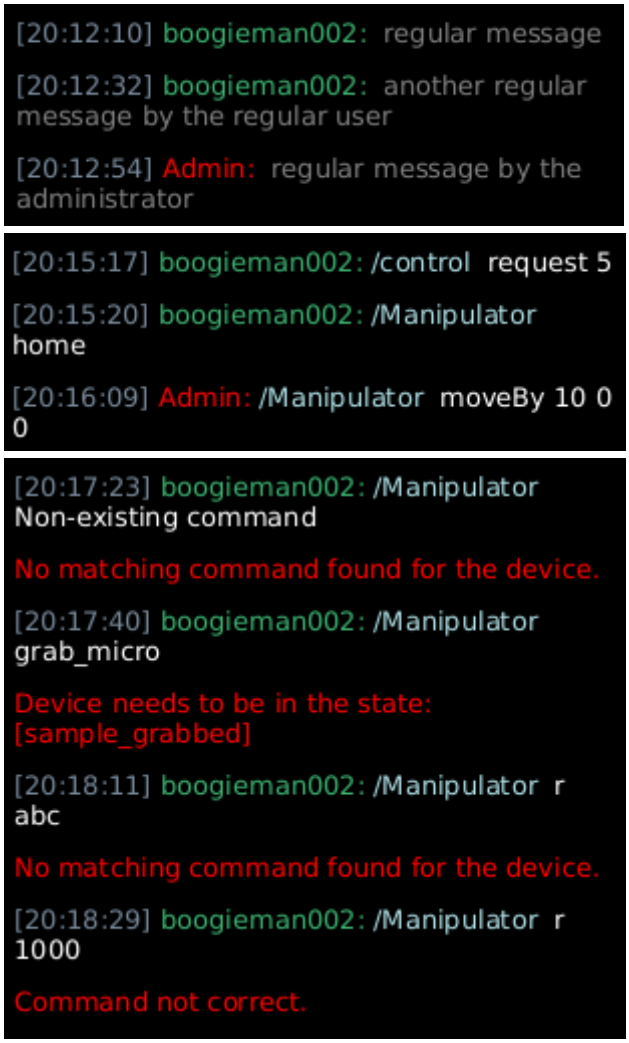

Figure 25. Examples of different types of messages.

5.3. Cartesian Manipulator

Cartesian manipulators, also known as Cartesian robots or gantry robots, are a type of robotic device used for automated handling and positioning tasks. They provide precise linear motion along three orthogonal axes - X, Y, and Z.

The origins of Cartesian manipulators trace back to the 1950s when early computer numerically controlled (CNC) machines were developed. They were some of the first machines capable of automated, programmable control of machine tools. Early industrial applications were in the automotive and aerospace sectors. Today, Cartesian manipulators are widely used across manufacturing, assembly, packaging, laboratories, warehouses, and other domains. Their ability to translate objects linearly with high precision, speed, and repeatability makes them well-suited for pick-and-place, palletizing, machining, testing, and inspection applications. The recent popularization of 3D printing drastically increased the number of cartesian manipulators worldwide.

Figure 26. Example of the cartesian manipulator in the form of a 3D printer. Source: [22].

Structurally, Cartesian robots consist of three linear motion axes, frequently coupled with an end-effector for grasping or tool operation. The linear slides are often belt/screw or rack/pinion driven, powered by servo or stepper motors. This orthogonal 3-axis design provides a cubic workspace and allows relatively simple path planning and control algorithms. Kinematically, Cartesian manipulators have a minimal number of coupled movements, reducing complexity compared to articulated arm robots. Their forward and inverse kinematics can be derived algebraically by simply adding the individual X, Y, and Z displacements. The Cartesian coordinate system used also provides intuitive programming and control.

Cartesian manipulators are valued for their modular frame construction, reconfigurable workspaces, and high positional accuracy. However, they do lack the continuous rotation capabilities of articulated designs. Heavy payloads can also introduce deflection without proper structural reinforcement.

5.3.1. Construction

The manipulator built for this system uses parts similar to the popular 3D printer Creality Ender 3 [15], as shown in Figure 27. It uses NEMA17 stepper motors to drive the motion along each axis. The X and Y axes utilize a belt and V-slot rail system to translate the linear motion from the motors into linear actuator movement. The Z axis in Figure 28 uses a trapezoidal lead screw and linear shaft to provide vertical lift motion from the stepper motor. The vertical Z-axis uses a lead screw well-suited for lifting against gravity and holding position. The horizontal X and Y axes favor a belt drive system that provides high horizontal movement speeds. This separation also enables modular design - each axis is constructed as an independent module that can be separately maintained and replaced.

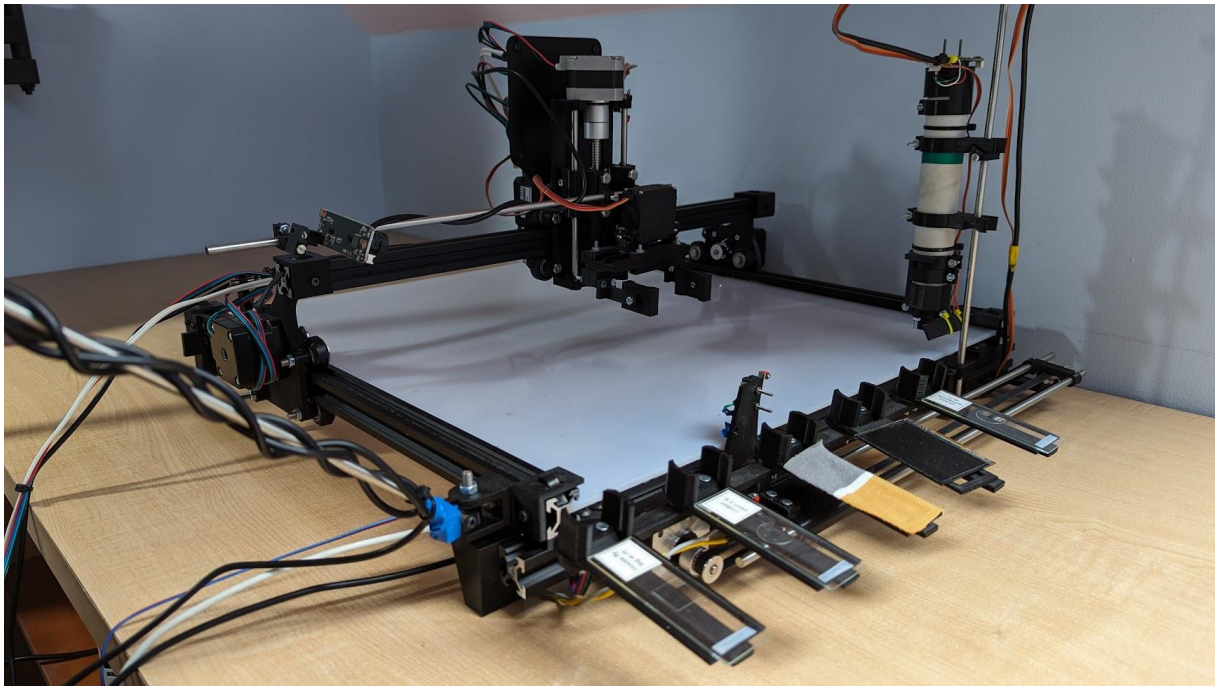

Figure 27. The cartesian manipulator external device.

A custom gripper powered by the servomotor mounted on the effector of the robot allows interaction with the environment. It has paddles shaped so that they eliminate two degrees of motion from similarly shaped objects. Elements that the robot interacts with are designed to fit the gripper arms, as can be seen in Figure 29, which reduces unwanted and uncontrolled movement.

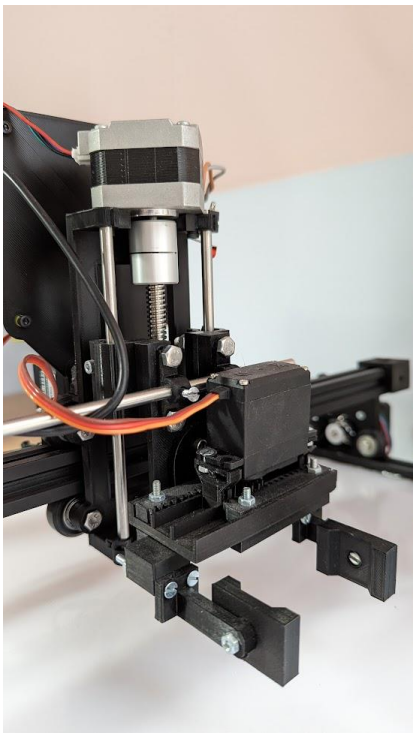

Figure 28. A Z-axis close-up.

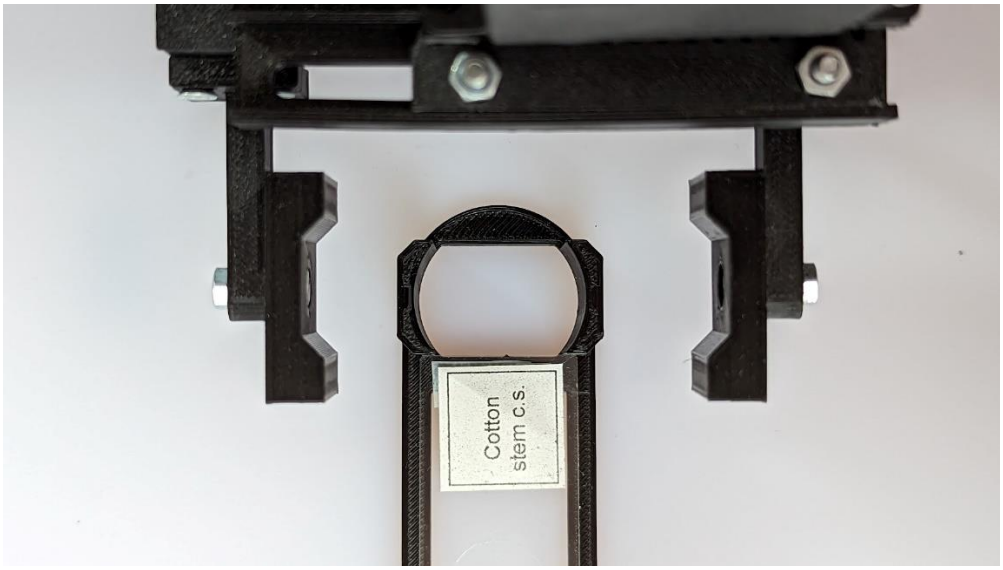

Figure 29. A gripper close-up.

5.3.2. Control system

The Cartesian manipulator utilizes an Atmega168 8-bit AVR microcontroller as the main control system, as seen in Figure 30. The Atmega MCU was chosen for its low cost, adequate processing capabilities, and easy interfacing to peripherals like stepper motors. The MCU handles the stepper motor control loops, processes motion commands received over the UART serial interface, and manages homing routines, limit switches, and other tasks.

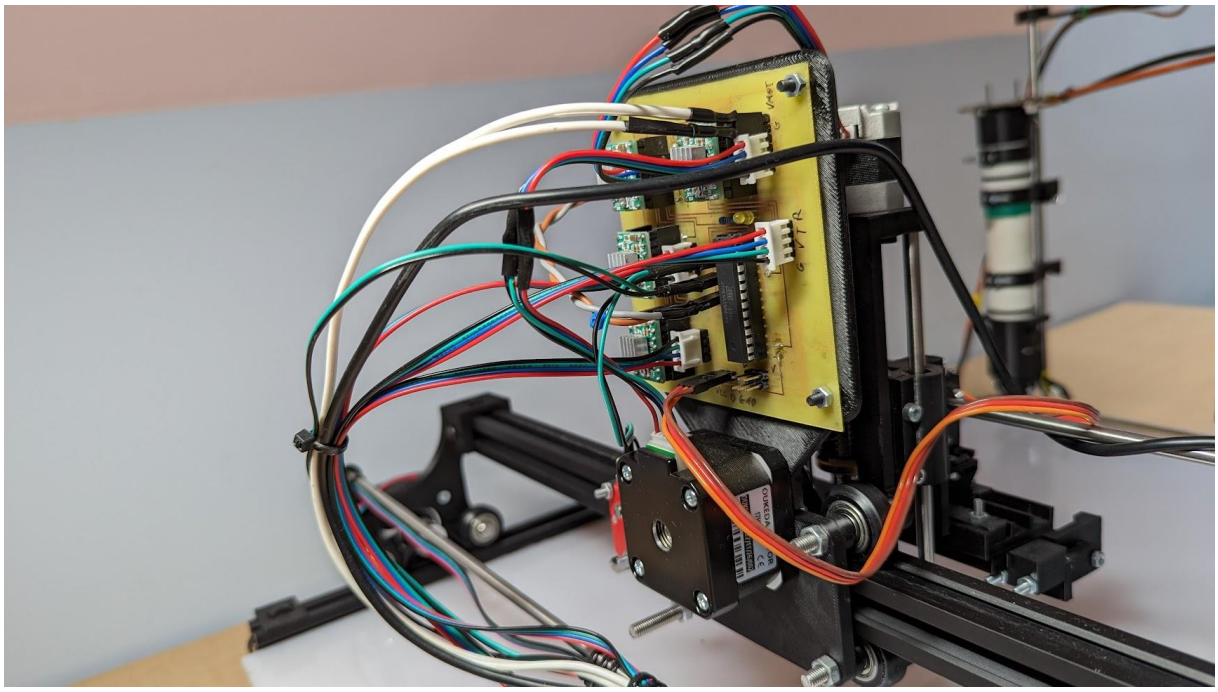

Figure 30. Cartesian manipulator electronics.

The design uses A4988 stepper motor driver carriers to interface the stepper motors. These driver modules convert the logic-level outputs from the MCU to the higher current and voltage levels needed by the motors. They allow configuring parameters like micro-step resolution and motor current. The drivers handle brushless motor commutation and sequencing of phase currents. The MCU uses its hardware timer modules to generate precisely timed step and direction signal waveforms to command the drivers. An interrupt-driven control scheme is implemented to ensure consistent step pulse timing and smooth motor operation while processing other tasks. The MCU UART serial port receives motion coordinate commands over the USB connection to a host PC. A simple parsing routine extracts the target X, Y, and Z positions from the serial data and converts them into a target number of steps to move each motor. Safety limit switches connected to the MCU GPIO pins are used for axis-homing routines and crash protection. To interconnect the microcontroller, motor drivers, and other components, a custom printed circuit board (PCB) was designed.

5.3.3. Firmware

The manipulator's firmware is written in C [23] and compiled using avr-gcc [24] for the Atmega168 AVR microcontroller [25]. It consists of a main control loop that handles stepper motor pulse generation, input command processing, limit switches, and other functions required to operate the robot. Motion control is implemented using a custom A4988 stepper motor driver library that enables precisely timed interrupt-driven step pulse and direction signal to be output to the motor drivers. Serial communication over UART provides the interface for receiving motion commands from the PC host. These commands are parsed by a dedicated routine that extracts the information from the formatted command string. The list of the key motion and configuration commands supported by the Cartesian manipulator firmware:

- `G0 X# Y# Z#` - Linear move to target XYZ position
- `G28` - Home all axes by triggering limit switches
- `G90` - Set to absolute positioning mode
- `G91` - Set to relative positioning mode
- `M` - Open/close gripper proportional to the parameter
- `F#` - Set constant speed in steps/second
- `ACCEL#` - Set acceleration rate in steps/second^2
- `MICROSTEP#` - Set micro-step resolution mode (1, 2, 4, 8, etc)
- `LIMITS X#/Y#/Z#` - Set software limits for each axis
- `ZERO` - Set current position as coordinate origin (0,0,0)
- `STATUS` - Report current position, speed, etc

`G0` moves the manipulator to the specified target XYZ position. An example of this command is presented as a sequence diagram in Figure 31. `G28` homes the axes by pushing into the switches. `G90`/`G91` select absolute vs relative coordinate modes.

`M` opens or closes the gripper proportional to the parameter value. `F` sets a constant stepping speed. `ACCEL` configures the acceleration rate. `MICROSTEP` sets the micro-stepping mode, which impacts resolution and torque.

`LIMITS` define the maximum range of motion along each axis. `ZERO` resets the coordinate system origin. `STATUS` reports back current settings and position.

These commands provide basic but powerful control over most aspects of motion - positions, speed, acceleration, homing, and gripping. The host software combines and sequences these primitive instructions to achieve valuable tasks.

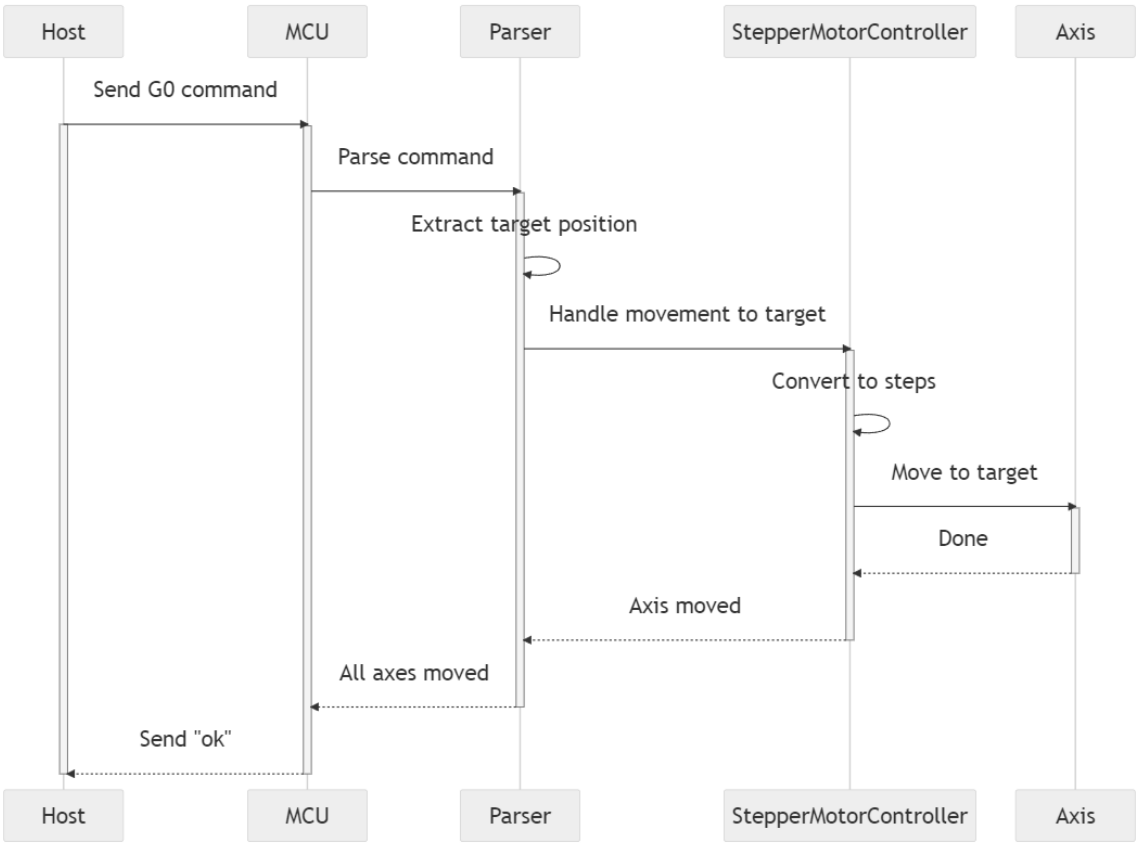

Figure 31. A sequence diagram for the G0 command.

5.3.4. Configuration

The configuration file presented in Listing 7 defines the Cartesian manipulator device and its capabilities within the LSED system, allowing it to be controlled through chat commands.

A key aspect it models is the set of instructions and actions the manipulator can perform. The commands section represents these. For example, the "`Home`" command will return all axes to their reference point by triggering the limit switches. The "`Move Relative`" command translates the device by a specified distance along the X, Y, and Z axes. More complex multi-step procedures are defined using sequenced events. The "`Grab Microscope`" command consists of a series of precision movements to position above the microscope, open/close the gripper at specific points, and lift the microscope for usage. This command encapsulates an entire automated routine utilizing the manipulator's capabilities.

The commands also model concepts like required machine state and command variables. "`Grab Sample`" can only be executed if the current device state is "`home`." The microscope grabs command parameterized certain waypoint positions into reusable variables for clarity.

These commands represent the manipulator itself - its motions, actions, and constraints. When LSED loads this file, it effectively gains an understanding of the physical device and how to operate it. The lower-level implementation details are abstracted away. At runtime, LSED must validate a chat

command against this model and convert it into the equivalent serial G-code instructions. The host system requires no change to support new devices like this manipulator.

Listing 7. A Cartesian Manipulator YAML configuration file.

```
name: 'Manipulator'
portName: ttyUSB0
portBaudRate: 9600
initialState: "start"
confirmation: "OK"

cameras:
  - name: "Effector_View"
    portName: "/dev/video0"
  - name: "Wide_Shot"
    portName: "/dev/video2"
  - name: "Microscope_View"
    portName: "/dev/video4"

commands:
  - name: "Home"
    description: "Home the manipulator in all axes."
    requiredStates: ["start", "home"]
    resultingState: "home"
    prefix: "home"
    output: "G28"
  - name: "Move Relative"
    description: "Move device by distance of millimeters."
    prefix: "moveBy"
    output: "G0 $XAxisDistance $YAxisDistance $ZAxisDistance"
    params:
      - name: "XAxisDistance"
        type: Integer
        range: [-100, 100]
      - name: "YAxisDistance"
        type: Integer
        range: [-100, 100]
        optional: true
        predefined: 0
      - name: "ZAxisDistance"
        type: Integer
        range: [-100, 100]
        optional: true
        predefined: 0

  - name: "Move Left"
    description: "Move device to the left by distance of millimeters."
    prefix: "l"
    output: "G0 -$XAxisDistance 0 0"
    params:
      - name: "XAxisDistance"
        type: Integer
        range: [0, 100]
```

```
- name: "Move Right"
    description: "Move device to the right by distance of millimeters."
    prefix: "r"
    output: "G0 $XAxisDistance 0 0"
    params:
      - name: "XAxisDistance"
        type: Integer
        range: [0, 100]

  - name: "Move Up"
    description: "Move device up by distance of millimeters."
    prefix: "u"
    output: "G0 0 $YAxisDistance 0"
    params:
      - name: "YAxisDistance"
        type: Integer
        range: [0, 100]

  - name: "Move Down"
    description: "Move device down by distance of millimeters."
    prefix: "d"
    output: "G0 0 -$YAxisDistance 0"
    params:
      - name: "YAxisDistance"
        type: Integer
        range: [0, 100]

  - name: "Zoom in"
    description: "Move effector closer to the table by distance of
millimeters."
    prefix: "+"
    output: "G0 0 0 -$ZAxisDistance"
    params:
      - name: "ZAxisDistance"
        type: Integer
        range: [0, 100]

  - name: "Zoom out"
    description: "Move effector further from the table by distance of
millimeters."
    prefix: "-"
    output: "G0 0 0 $ZAxisDistance"
    params:
      - name: "ZAxisDistance"
        type: Integer
        range: [0, 100]
```

```
- name: "Grab the microscope"
    description: "Grab the microscope"
    requiredStates: [ "sample_grabbed" ]
    resultingState: "microscope_grabbed"
    prefix: "grab_micro"
    vars: {
      X1: "150",
      Y1: "0",
      Z1: "10",
      Y2: "-100",
      Y3: "-127",
      Z2: "25",
      Z3: "30",
      X2: "-60",
      Y4: "50",
      Z4: "21.5"
    }
    events: [
              "G91",
              "LIMITX 200",
              "LIMITY 200",
              "LIMITZ 100",
              "G0 $X1 $Y1 $Z1",
              "G0 $X1 $Y2 $Z1", # Go in front of the microscope stand
              "M 100",
              "F 30",
              "G0 $X1 $Y3 $Z1",
              "M 50",
              "G0 $X1 $Y3 $Z2",
              "M 30",             # Grab the microscope
              "G0 $X1 $Y2 $Z3",
              "F 100",
              "G0 $X2 $Y4 $Z3", # Go above bed lights
              "F 20",
              "G0 $X2 $Y4 $Z4",  # Zoom in with microscope
              "F 1",
              "LIMITX -65 -55",
              "LIMITY 45 55",
              "LIMITZ 20 23",
              "G90"
    ]

  - name: "Put away the microscope"
    description: "Put away the microscope"
    requiredStates: [ "microscope_grabbed" ]
    resultingState: "sample_grabbed"
    prefix: "deposit_micro"
    vars: {
        […]
    }
    events: [
        […]
    ]
```

```
- name: "Grab sample"
  description: "Place sample into the work area"
  requiredStates: [ "home" ]
  resultingState: "sample_grabbed"
  prefix: "grab_sample"
  vars: {
      […]
  }
  events: [
      […]
  ]

- name: "Deposit sample"
  description: "Deposit sample back into the sample selector"
  requiredStates: [ "sample_grabbed" ]
  resultingState: "home"
  prefix: "deposit_sample"
  vars: {
    […]
  }
  events: [
      […]
  ]
```

5.4. Microscope

The microscope device provides enhanced visualization and sample handling capabilities that integrate with the Cartesian manipulator. The device is mounted on a stand that allows it to be grasped by the manipulator's gripper, as in Figure 27. Commands like "`Grab Microscope`" and "`Deposit Microscope`" in the manipulator's configuration sequence through pickup and drop-off motions. Once grasped, the manipulator can position the microscope above a sample. The "`Grab Sample`" command places the sample in the work area.

The microscope controls two light sources - bed lights underneath the sample and top lights mounted near the lens, as seen in Figure 32. These are toggled on/off by "`Bed Light Switch`" and "`Top Light Switch`" commands. Brightness is also adjustable. The top lights help to illuminate opaque samples, while bed lights from beneath are suitable for more transparent samples.

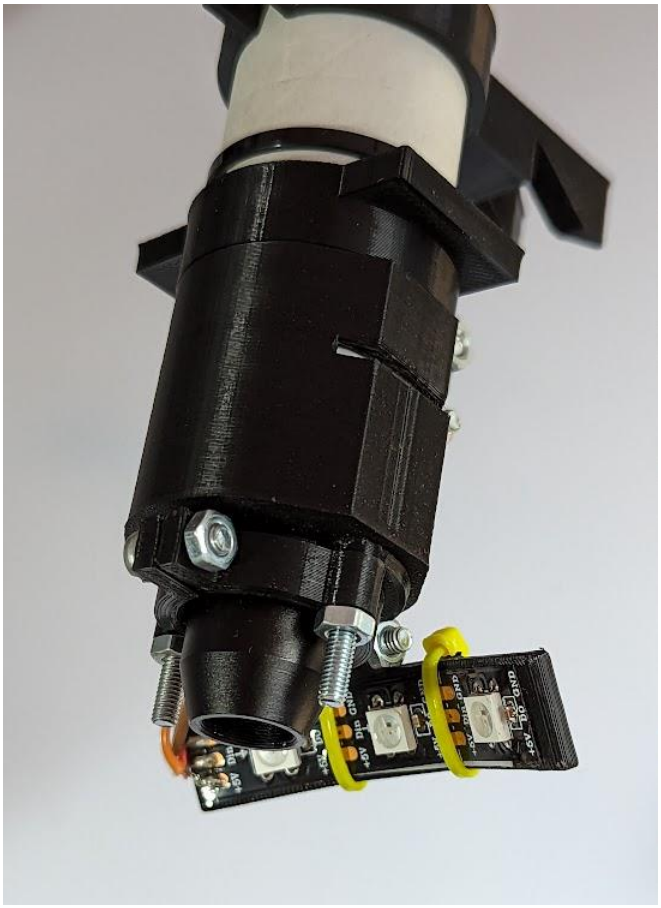

Figure 32. Microscope lens with top light.

A stepper-motor-driven sample selector allows switching between 5 different samples shown in Figure 33. The "`Select Sample`" command rotates this selector to the specified position. It has a toggle switch that tracks when the manipulator takes a sample. This prevents accidental movement of the selector when empty. The manipulator resets it after returning the sample.

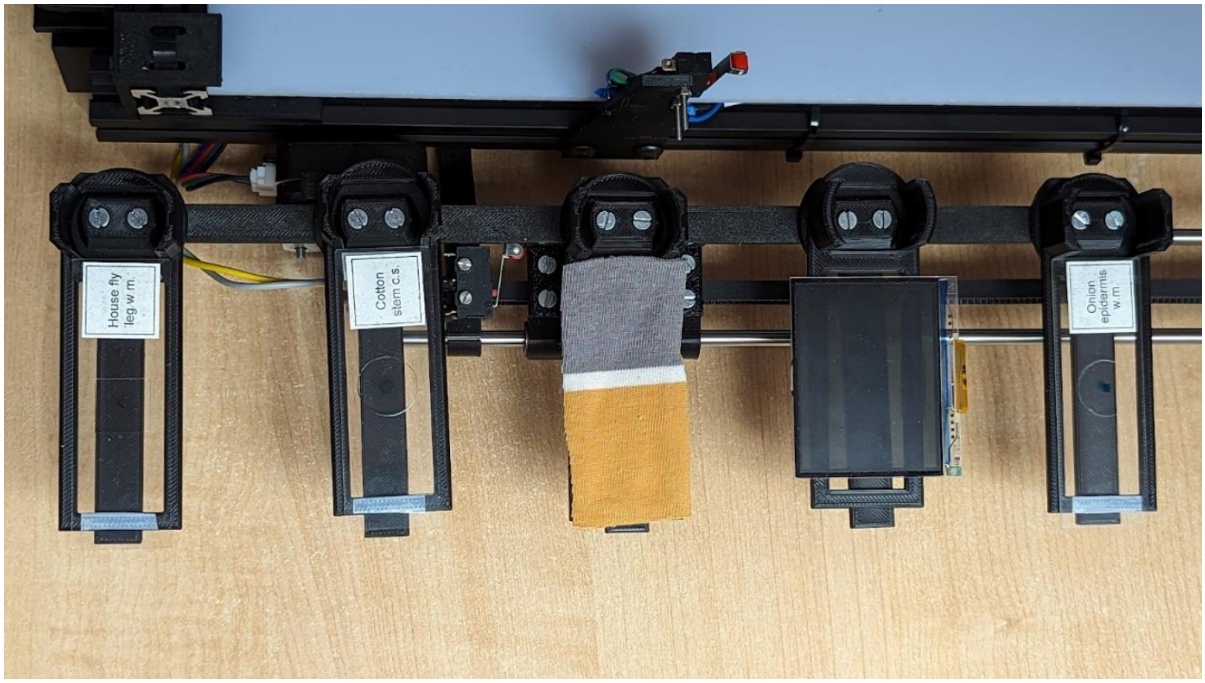

Figure 33. Microscope's sample selector with 5 samples. From the left: house fly leg, cotton stem, a piece of fabric, an LCD from a digital camera, and a piece of onion.

The Arduino firmware processes light and sample selector commands received over serial. It toggles LEDs, actuates the stepper motor, and tracks the sample state.

Together, these features expand the manipulation capabilities. The microscope enhances visualization. The sample selector enables material variety. The integrative commands allow coordinated motions between the devices. The configuration on Listing 8 models the microscope's capabilities in LSED. Chat commands drive the hardware via the Arduino at runtime, just like the manipulator. This showcases the flexible integration of multiple bespoke devices under a unified control scheme.

Listing 8. A microscope YAML configuration file.

```
name: Microscope
portName: ttyUSB1
portBaudRate: 9600
confirmation: "done"
cameras:
  - name: "Main View"
    portName: "/dev/video4"
commands:
  - name: "Bed Light Switch"
    description: "Switch bed light ON and OFF."
    prefix: "bedLight"
    params:
      - name: "State"
        type: String
        values: [ "on", "off" ]
    output: "lgt bed $State"
  - name: "Top Light Switch"
    description: "Switch top light ON and OFF."
    prefix: "topLight"
    params:
      - name: "State"
        type: String
        values: [ "on", "off" ]
    output: "lgt top $State"
  - name: "Select sample"
    description: "Move sample selector to one of 5 possible positions"
    prefix: "sample"
    params:
      - name: "sample_number"
        type: Integer
        range: [1, 5]
    output: "sample $sample_number"
```

6. Summary and conclusion

This thesis presented the design and implementation of the LSED system to enable live-streaming interaction with external devices. The core motivation was facilitating public access to specialized hardware like research equipment over the Internet. To address this problem, a desktop application was developed to manage devices and chat communication across popular streaming platforms. The system modeled device capabilities through configurable commands. It incorporated security measures, including user permissions, validation, and finite state machine constraints. Real-time control was enabled through live video feeds and control queues.

6.1. Contribution

The implementation demonstrated LSED integrating a Cartesian robot manipulator and microscope device. Users could control the hardware in real time via chat commands on Twitch and YouTube. This showcased flexible, secure control of remote hardware through existing live-streaming services. In Figure 34 and Figure 35, there is a GUI with some of the samples viewed through the manipulator-controlled microscope lens as seen by users. Figure 36 is a close-up of the gripper holding the microscope and looking at a sample with bed lights turned on.

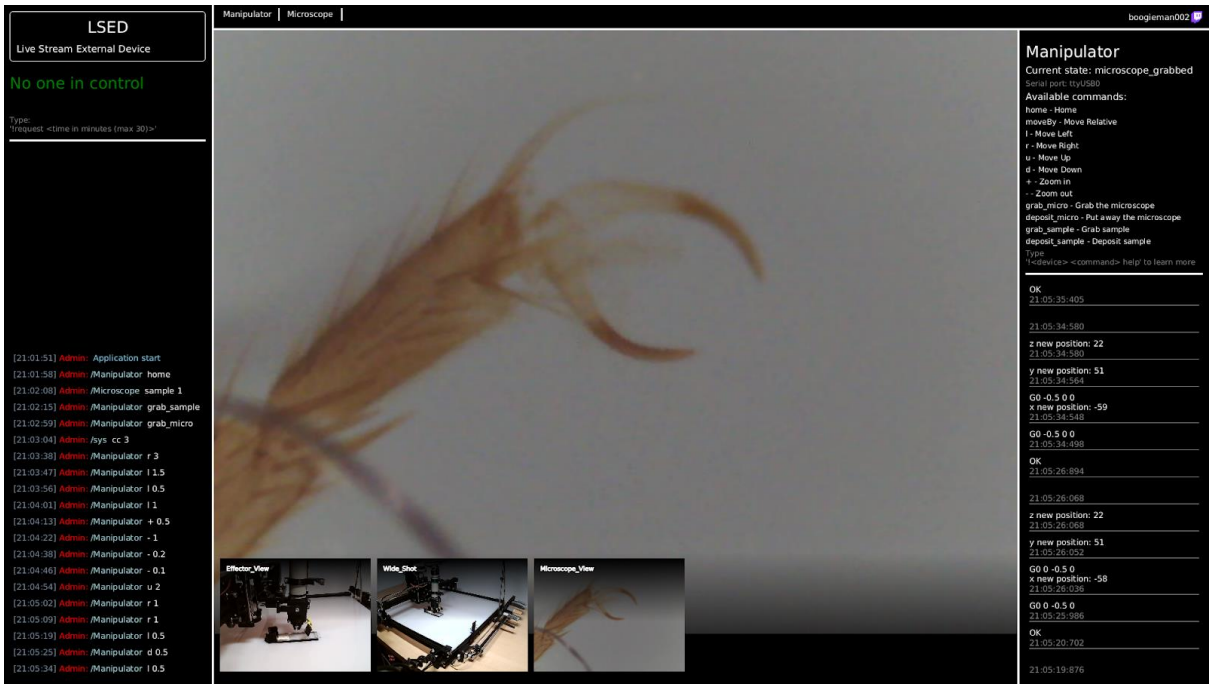

Figure 34. Sample containing a housefly leg as seen on the LSED GUI.

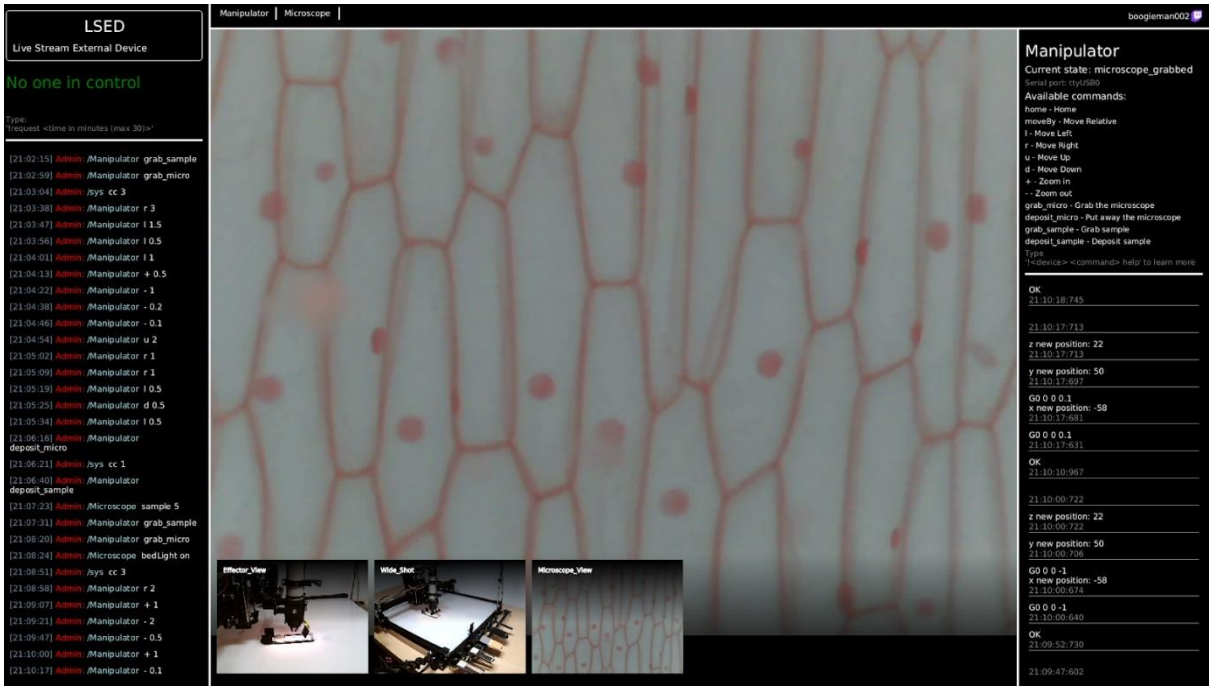

Figure 35. A sample containing cells of onion as seen on the LSED GUI.

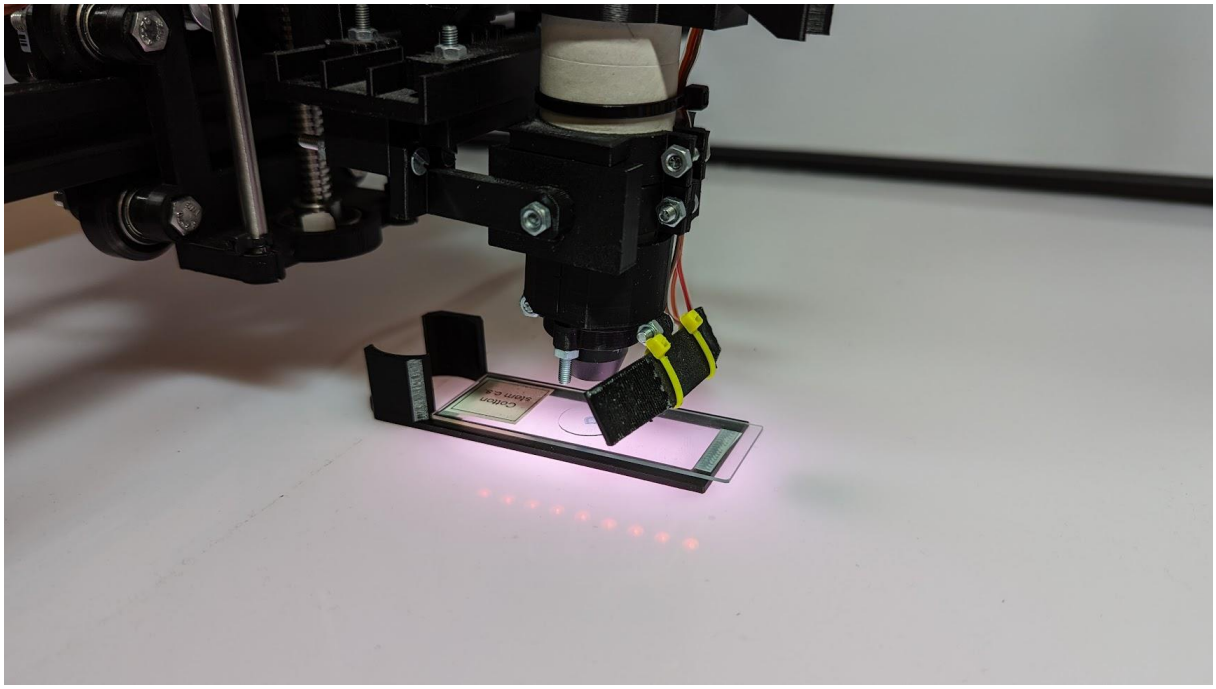

Figure 36. Gripper is holding the microscope, looking at a sample with bed lights turned on.

The key technical contributions of this thesis include:

- A desktop application with GUI for device and chat management. It utilized an architecture that enabled integration with multiple live-streaming platforms through their APIs.
- Device configuration modeling using YAML files. Devices could be added by specifying supporting commands, parameters, and required states.

- A manipulator robot with a gripper designed and built to integrate with the system. Its firmware processed serial commands from the LSED application.
- A microscope device with controllable lighting and a sample selector. It expanded the manipulator's capabilities through coordinated actions.
- Real-time control of devices by users through chat commands and live video feeds. Commands were confirmed immediately on the GUI, and users could analyze the outcome of their commands.
- A command interpreter component to validate and translate chat commands into device instructions. It checked command parameters against the device model.
- User management features include permission levels, control requests, and queues. This prevented concurrent device access and enforced fair sharing.

The implemented LSED system showcased flexible integration of external hardware for real-time control over live streams. The manipulator and microscope served as representative devices, but the architecture supports connecting various custom equipment and its modeling through YAML configuration files. Security and coordinated access were built-in to prevent misuse. Real-time control and feedback enhanced the user experience. The implementation validated the feasibility of remotely sharing specialized devices using the existing streaming infrastructure.

6.2. Possible improvements

While the implementation successfully demonstrated core capabilities, there remain areas for improvement.

One limitation was that the configuration validation at system startup was minimal. Users received limited feedback about errors in their configuration files, mainly through hard-to-read Java trace errors. Enhanced validation and error reporting would allow problems to be identified and corrected more easily.

Additionally, the process of assigning serial ports and cameras was manual. Users had to enter port names through trial and error, which was time-consuming. Improved port handling could auto-detect connected devices and allow users to map ports at startup visually.

The system currently outputs commands directly to serial connections. An alternative approach could be writing commands to files, which are then processed separately. This decoupling would enable commands to be issued in different ways – like executing them over SSH or terminals. It would make the system more adaptable to diverse hardware. The chat commands could also be read from files making the system much more modular and robust. Bots could be written as independent programs that can read and write to chats using various techniques, programming languages, and protocols.

In terms of future work, the system could be expanded to support users' groups so multiple users could collaborate or compete when controlling devices. This would enable team coordination or games using various hardware. Workspace awareness features could allow coordination between concurrent users and shared control modes, further improving system utilization.

Lastly, developing a custom web interface could drastically improve the system's capabilities. Features like user accounts, access permissions, and scalable multi-user device control would be realizable through a server-hosed website backend. This could enable contexts like online lessons or collaborative research with larger user bases utilizing many devices concurrently.

Overall, the system successfully achieved the core goal of the accessible real-time remote hardware control with visual feedback. The idea of device configuration provides a solid foundation, and there are multiple ways in which the system can be refined and expanded. In the future, systems like these can further enhance the opportunities for collaboration given by the Internet.

7. References

[1] "Youtube," [Online]. Available: https://www.youtube.com/. [Accessed 22 05 2023].

[2] "Twitch.tv," [Online]. Available: https://www.twitch.tv/. [Accessed 22 05 2023].

[3] "twitchtracker.com," [Online]. Available: https://twitchtracker.com/statistics. [Accessed 22 05 2023].

[4] "Isotopium," 19 05 2023. [Online]. Available: https://www.isotopium.com/. [Accessed 19 05 2023].

[5] "Emerald Cloud Lab," [Online]. Available: https://www.emeraldcloudlab.com/. [Accessed 19 05 2023].

[6] "OBS Project," [Online]. Available: https://obsproject.com/. [Accessed 31 05 2023].

[7] "Streamlabs," [Online]. Available: https://streamlabs.com/. [Accessed 31 05 2023].

[8] "Outliner Automation," [Online]. Available: https://www.outlierautomation.com/blog/202020/12/16/whats-the-difference-between-opc-ua-and-mqtt. [Accessed 31 05 2023].

[9] "Twitter.com/AnkhBot," [Online]. Available: https://twitter.com/AnkhBot/status/885574840446578688/photo/2. [Accessed 31 05 2023].

[10] "NightBot," [Online]. Available: https://nightbot.tv/. [Accessed 31 05 2023].

[11] "Streamer.bot," [Online]. Available: https://streamer.bot/. [Accessed 31 05 2023].

[12] "Streamer.bot Wiki," [Online]. Available: https://wiki.streamer.bot/en/home. [Accessed 31 05 2023].

[13] "Nandan Technicals," [Online]. Available: https://www.nandantechnicals.com/2022/07/scada-types-functions-scada-system.html. [Accessed 19 05 2023].

[14] "SnakeYAML Documentation," [Online]. Available: https://bitbucket.org/snakeyaml/snakeyaml/wiki/Documentation. [Accessed 31 05 2023].

[15] "java.com," Oracle, [Online]. Available: https://www.java.com/en/download/help/whatis_java.html. [Accessed 11 08 2023].

[16] "openjfx.io," [Online]. Available: https://openjfx.io/. [Accessed 11 08 2023].

[17] "https://fazecast.github.io," Fazecast, [Online]. Available: https://fazecast.github.io/jSerialComm/. [Accessed 11 08 2023].

[18] [Online]. Available: https://github.com/snakeyaml/snakeyaml. [Accessed 11 08 2023].

[19] "dev.twitch.tv," Twitch.tv, [Online]. Available: https://dev.twitch.tv/docs/api/. [Accessed 11 08 2023].

[20] "developers.google.com," google.com, [Online]. Available: https://developers.google.com/youtube/v3. [Accessed 11 08 2023].

[21] [Online]. Available: https://github.com/FasterXML/jackson. [Accessed 11 08 2023].

[22] "farnell.com," [Online]. Available: https://pl.farnell.com/en-PL/creality-3d/ender-3-v2/3d-printer-1-75mm-tf-card/dp/3581851. [Accessed 01 08 2023].

[23] B. Stroustrup, The C++ Programming Language, 2013.

[24] "gcc.gnu.org," GNU, [Online]. Available: https://gcc.gnu.org/wiki/avr-gcc. [Accessed 11 08 2023].

[25] "microchip.com," Microchip, [Online]. Available: https://www.microchip.com/en-us/product/atmega168. [Accessed 11 08 2023].

8. List of Figures

Figure 1. Example of a duck-feeding live stream. 6
Figure 2. Live stream of a Vector robot controlled by chat. 7
Figure 3. A person standing on a game's environment with RC tanks. Source [4] 8
Figure 4. A person controlling his tank and encountering another player. Source: [4] 8
Figure 5. A row of research stations with scientific devices. Source: [5] 9
Figure 6. Example view of the Streamlabs Chatbot GUI. Source: [9] 11
Figure 7. The example view of the NightBot dashboard. Source: [10] 12
Figure 8. Image of the Streamer.bot's comprehensive documentation. Source: [12] 13
Figure 9. Architecture of SCADA System. Source: [13] 13
Figure 10. Mockup of GUI featuring a list of devices and user manual. 17
Figure 11. A graph of two possible states with available actions. 19
Figure 12. Mockup of GUI featuring the control queue. 21
Figure 13. Mockup of GUI featuring a list of commands and logs. 22
Figure 14. Mockup of GUI featuring different camera views. 22
Figure 15. General Architecture View. 24
Figure 16. Device management class diagram. 26
Figure 17. User command sequence diagram. 27
Figure 18. User management class diagram. 28
Figure 19. User control request sequence diagram. 29
Figure 20. Simplified Interpreter sequence diagram. 30
Figure 21. A graphical user interface of the LSED application implementation. 32
Figure 22. The additional window for the host. 33
Figure 23. A sequence diagram for sending commands directly to the device. 35
Figure 24. UI element showing the queue of user control requests. 36
Figure 25. Examples of different types of messages. 37
Figure 26. Example of the cartesian manipulator in the form of a 3D printer. Source: [22]. 38
Figure 27. The cartesian manipulator external device. 39
Figure 28. A Z-axis close-up. 39
Figure 29. A gripper close-up. 40
Figure 30. Cartesian manipulator electronics. 40
Figure 31. A sequence diagram for the G0 command. 42
Figure 32. Microscope lens with top light. 47
Figure 33. Microscope's sample selector with 5 samples. From the left: house fly leg, cotton stem, a piece of fabric, an LCD from a digital camera, and a piece of onion. 48
Figure 34. Sample containing a housefly leg as seen on the LSED GUI. 50
Figure 35. A sample containing cells of onion as seen on the LSED GUI. 51
Figure 36. Gripper is holding the microscope, looking at a sample with bed lights turned on. 51

9. List of Listings

Listing 1. Basic device configuration file. 16
Listing 2. Proposition for command in a device configuration file. 18
Listing 3. A FSM described using commands. 19
Listing 4. A list of webcams added to the configuration file. 23
Listing 5. Fragment of the AuxiliaryWindow.java highlighting device selection button. 33
Listing 6. Fragment of the DeviceManager class highlighting how it leverages the IntegerProperty class. 34
Listing 7. A Cartesian Manipulator YAML configuration file. 43
Listing 8. A microscope YAML configuration file. 49