University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Computer Science	Spring semester, 20.22. Open / ~~Restricted access~~
Writer: Aleksander Pettersen	..... (Writer's signature)
Faculty supervisor: Morten Mossige, Ståle Freyer External supervisor(s):	
Thesis title: Designing and Implementing a Framework for Real-time Robot Controller Clients	
Credits (ECTS): 30	
Key words: WebSocket Robots Single Page Application	Pages: 53 + enclosure: 1 Stavanger, 15 June, 2022 Date/year

Front page for master thesis
Faculty of Science and Technology
Decision made by the Dean October 30th 2009

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Designing and Implementing a Framework for Real-time Robot Controller Clients

Master's Thesis in Computer Science
by

Aleksander Pettersen

Internal Supervisors

Morten Mossige

Ståle Freyer

June 15, 2022

"This quote should be about computer stuff."

My Wife

Abstract

This research paper designs a framework for developing real-time clients that communicate with robot controllers built by students at the University of Stavanger. The framework provides patterns that handle time-sensitive components and demonstrates a technique for functional scalability. A brief section introduces design metrics for user experience. The report presents three use cases covering functionality for actuating a robot and reading its movements in real time. This thesis implemented the three use cases in the spring semester of 2022, and the results show that it is possible to model use cases, but implementing complex use cases requires more effort.

Acknowledgements

I would like to thank my supervisors, Morten Mossige and Ståle Freyer, for their help in writing this thesis.

I thank my lovely wife and five sons for their patience and understanding for all the time spent on finishing my master's degree.

I would also like to thank Sindre Skavhellen and Lars Almås for our collaboration this semester.

Contents

Chapter 1

Introduction

With the emergence of the fourth industrial revolution[1], easier access to robot motors, drivers, and controllers has increased the prevalence of robots in different projects[2]. Advances in System-on-Chip technology have reduced the controllers from specialized and expensive hardware to affordable devices such as the Raspberry Pi[3]. However, as the hardware aspect has been simplified, the software aspect has grown significantly in complexity. Robot controller systems have traditionally been developed for specific operating systems and their intrinsic graphical frameworks[4], such as Windows Forms. However, some modern end-users expect client-side software to be easily available with cross-platform capabilities.

Modern actuators expose real-time data about the motors, metrics traditionally only accessible by probing with an Oscilloscope. Front-end applications can now consume these metrics, and servers often update their state as frequently as 20ms[5]. The frequent data stream forces the clients to start reacting to events from a server instead of having a fire-and-forget relationship in their server interaction. This thesis uses the Cambridge Dictionary[6] definition of a framework, *"a system of rules, ideas, or beliefs that is used to plan or decide something."* Existing development frameworks focus on streamlining the connection between the data source and the presentation, but implementations are either limited to a specialized closed-source technology or require the client to have direct access to the actuator. Therefore, better communication of the problem space and user experience requirements is likely to be attained by modeling a framework for developing real-time clients.

Many development teams have people who do not necessarily have the same working hours or availability. The framework needs to have some strategy to align the developers and a way to present the current state of the shared codebase. The thesis was done in

collaboration with Sindre Skavhellen and Lars Almås[5], who implemented the server-side application.

In this paper, I present a framework for developing real-time clients that send commands to the robot controller in addition to subscribing to its real-time data streams. The framework is based on existing verified research on user experience and uses principles from continuous integration[1] for collaboration. I will answer three research questions:

- RQ1: What is required of a framework to provide clear guidelines for developing real-time applications for a robot controller?
- RQ2: How does the framework scale with high-intensity data streams?
- RQ3: What impact does continuous integration have on the development?

1.1 Outline

The thesis starts by presenting the robots and the system set up in Chapter 2, and it continues to describe related work, types of graphical user interfaces, and Continuous Integration in Chapter 3. Chapter 4 explores existing frameworks and their implementations with an analysis in the closings. Chapter 5 presents the framework design. I describe each element of the framework and give some examples. Chapter 6 models three relevant use cases based on the framework. The use cases were formed in collaboration with Skavhellen and Almås[5]. Chapter 7 presents an implementation of the use cases and the findings. The results are discussed in Chapter 8. In Chapter 9, I present ideas for improvement and further work.

[1] Defined in Section 3.2

Chapter 2

Background

This chapter presents the robots, the system setup, and how the different devices are connected.

2.1 The Robots

This section describes the robots and their capabilities.

2.1.1 Hexapod

The Hexapod, as shown in Figure 2.4, is a robot with six motors connected to two triangles. The upper triangle has a sixty-degree rotation relative to the bottom triangle, giving the upper triangle six degrees of freedom[7]. The Hexapod supports two movement actions; rotation of the upper triangle and positioning of the upper triangle.

The rotation actuation uses "Roll, Pitch, and Yaw" to describe the movement. Roll is a metric for rotation around the X-axis, Pitch is a metric for rotation around the Y-axis, and Yaw is for rotation around the Z-axis. Figure 2.2 gives a visual presentation of the rotation, and is often called joint movement.

The positioning actuation uses "X, Y, and Z" to describe the movement, referring to the position of a point in the middle of the upper triangle. Figure 2.1 shows the point, colored red, in a scenario where the Hexapod has no rotation, but the camera is angled downwards. The positioning actuation is often called linear movement.

2.1.2 Rocker

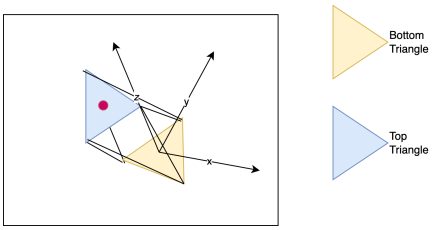

Figure 2.1: The red point is the (x,y,z) position of the linear movement action for the Hexapod.

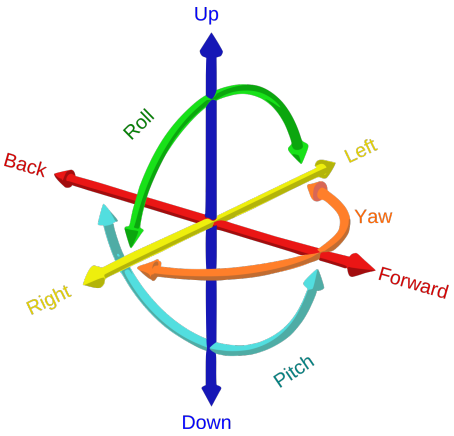

Figure 2.2: Six Degrees of Freedom. By GregorDS[8]

The Rocker, as shown in Figure 2.5, is a robot with a single motor connected to a pivoting platform. The platform is moved by extending and contracting the length of the motor, giving it a single degree of freedom.

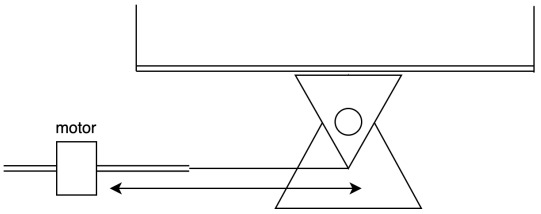

Figure 2.3: Rocker connected to a linear motor.

Figure 2.3 depicts the motor connected to the Rocker. The pivoting point is the center of the rotation movement, and the actuation command consists of a single angle value. Equivalently, the position actuation refers to the extension length of the robot, and the actuation command consists of a single length value. Figure 2.6 shows a still-image sequence of the Rocker's movement where the actuator extends the motor.

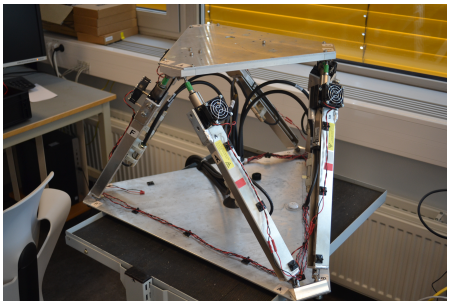

Figure 2.4: Hexapod Robot, photo by Lars Almås.

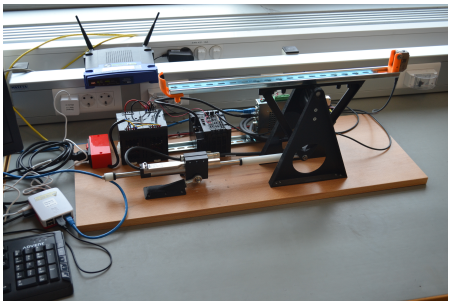

Figure 2.5: Rocker Robot, photo by Lars Almås.

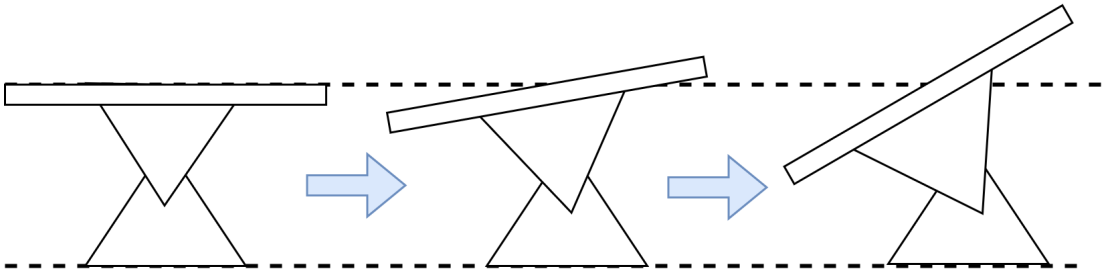

Figure 2.6: Example of a Rocker moving from zero position to an arbitrary angle. The actuator determines the angle by extending its linear motor, as shown in Figure 2.3.

2.2 System Set Up

Figure 2.7 shows the client connecting to the controller via WiFi and using this data link to receive data streams and interact with the controller. The controller connects to a series of actuators using EtherCat[9] over Ethernet. The thesis's responsibility is mainly to design and implement the colored sections in Figure 2.7.

The server implementation[5] exposes a SocketIO[10] endpoint and therefore the client must also use a SocketIO client library. SocketIO is a library that combines different bidirectional communication patterns between client and server. It offers features such as polling and WebSocket, but WebSocket is exclusively used for communication in this thesis.

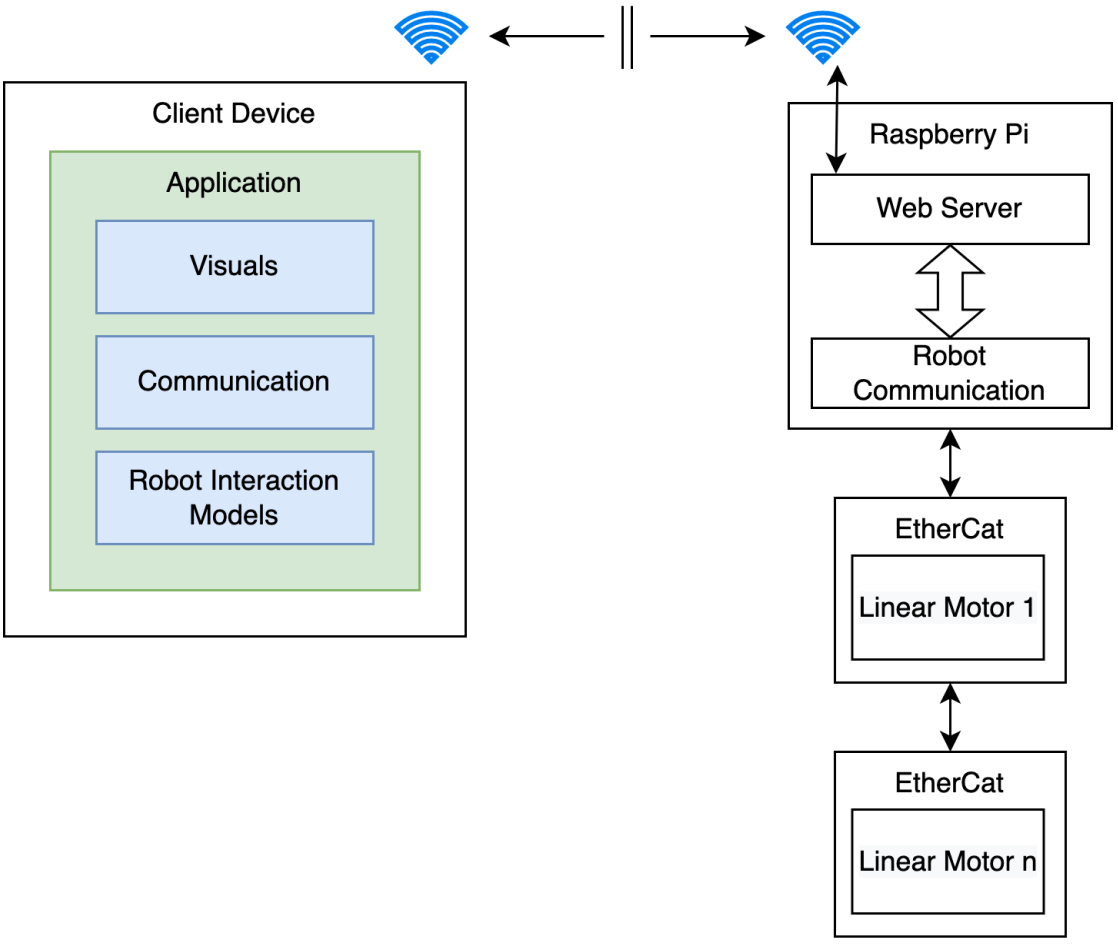

Figure 2.7: Device Connectivity across different media. The client connects via WiFi to the controller, and the controller connects to the actuators over Ethernet.

Chapter 3

Related Work

This section briefly describes different user interfaces and their relevance to this thesis. Additionally, a few core concepts of continuous integration and continuous delivery are presented.

3.1 Client

A client is an interface[11, Chapter 6] that allows end-users to interact with a computer system. The client's main objectives are to present robot values and send actuation commands to the robot controller. This thesis assessed three UI types; command-line interface (CLI), Windows, Icons, Menus, and Pointing device (WIMP), and a browser-based client. The discussion focuses on support for cross-platform and usability.

3.1.1 Command-line Interface

Rogers et al.[11, Chapter–6.2.1] evaluate the CLI as a dated model for user interaction. They identify the CLI as a tool on the operating system level and require the end-user to remember the commands. The CLI limits user interaction to computers and thus fails the cross-platform requirement.

3.1.2 Window, Icons, Menus and Pointing Device Applications

Rogers et al.[11, Chapter–6.2.2] also discuss the WIMP client and identify the challenges of making an application behave consistently across multiple operating systems. The usability of WIMP is considered great and offers great diversity for designing the client.

Frameworks such as Qt[12] provide cross-platform capabilities, but it is a big development environment with a steep learning curve. WIMP is a good candidate, but the learning curve makes it unsuitable for this thesis.

3.1.3 Web Applications

Rogers et al.[11, Chapter–6.2.6] identify web applications as the successor to WIMP systems. They emphasize that web applications run a greater risk of becoming cluttered and that the flow of the application becomes convoluted. The possibilities for layout and design are very flexible, and this allows for carefully crafted components which in turn will give a good usability experience.

The Web application is easily distributed using a web server running on the robot controller, and it does not require the end-user to install additional third-party binaries, except for a browser. Cross-platform and portability are achieved using Responsive Design[13], and all modern browsers have support for the WebSocket API.

Modern technology consists mostly of JavaScript, HTML5, and CSS3. Currently, most modern web applications are implemented in a pattern called Single Page Application (SPA)[14]. In a SPA, the client fetches the application resources from the server once and builds the HTML on the client. Subsequent requests to the server will only fetch data models. In older patterns, the HTML is built on the server[15], and in one pattern, Model View Controller (MVC)[16], most state changes in the client are done by requesting a new page from the server. The two patterns are compared in Figure 3.1. The SPA pattern often has a larger initial payload, but the subsequent requests' payload is smaller, whereas the MVC pattern's payloads are larger throughout due to having the client re-render the entire page on change.

It is important to note that web technology is evolving at incredible speed, and many JavaScript frameworks have relatively frequent major version updates that might break backward compatibility. An advantage of this is that when new technology is available, most JavaScript frameworks are quick to adopt them. One downside of this is that the next developer who contributes to the codebase will have to spend some time updating all the libraries and fixing compatibility issues.

3.2 Continuous Integration

Continuous Integration[17] (CI) has become a widely adopted engineering practice where code is continuously checked into a centralized source code repository. The source

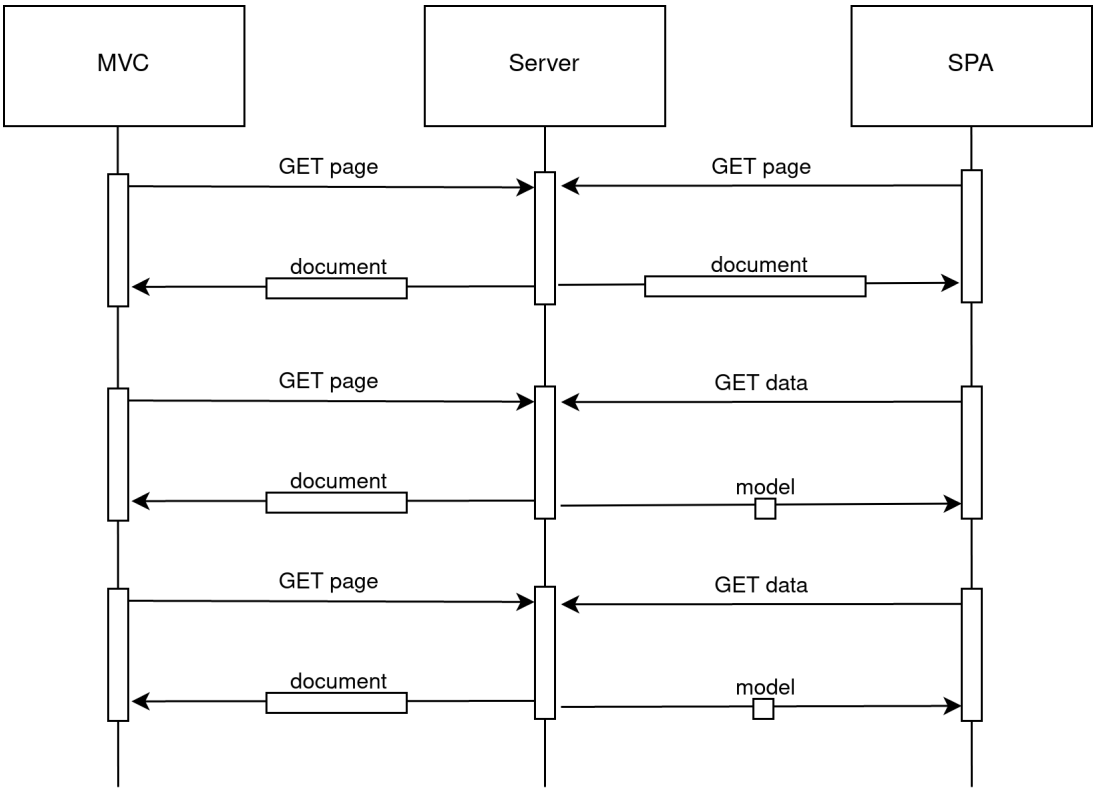

Figure 3.1: Comparison of MVC and SPA.

control system will then trigger an execution flow aimed at running building, testing, and outputting artifacts of the code. This thesis builds on an existing codebase that resides on GitHub with intrinsic support for CI through GitHub Actions.

Mossige et al.[18] model a test system for robots with CI aimed at catching errors as early as possible during development. They use model-based testing (MBT) and Constraint-based testing (CBT) to create test suites. They present a methodology that uses diagrams and test suites for its use cases. Their findings show that using CI in a team increases developer confidence, and bugs are discovered and rectified faster than compared with a non-CI system. Since this thesis does not interact with a robot directly, only the work on CI and parts of the MBT is relevant.

Chapter 4

Existing Frameworks

4.1 Introduction

This chapter presents the framework for developing components and how each component interacts with the server API.

4.2 Existing Work

Vlissides et al.[19] implemented a GUI for reading and mutating the state of a real-time power system simulator. Their approach involved using the Unidraw framework with a direct connection between the GUI and the database. Even though they achieved their goal, Unidraw does not offer any implementation that supports cross-platform between desktop and mobile and is therefore not suitable for implementing the GUI.

Zabierowski et al.[20] model the client-to-server interaction using an N-tier architecture. The N-tier architecture itself is valid for this thesis, but the implementation in CORBA, with SOAP for server communication and WinForms 2.0 on the client, limits its cross-platform capabilities.

Bouraoui et al.[21] make a strong point for building the client using Model Driven Architecture (MDA) for cross-platform support. They use a framework where a Computional Independent Model (requirements and business knowledge) feed a Platform Independent Model (UML, wireframes, and sequence charts) that later feeds an implementation called Platform Specific Model. Their goal is to generate the Platform Specific Model and transform the data from the Platform Independent Model to the target system, e.g., Objective-C, using a metamodel. They suggest using MDA with a User Interface Designer to generate these UIs and perform tests to verify the validity. They are, however, not

able to verify the output from the different tools, but as the idea of building software modules in the style of MDA proved to be a good approach, this thesis will use the same approach for its software modules.

Johnson et al.[22][23] suggest using a software framework called PalCom to implement enterprise-grade UIs. Their main goal is for non-programmers to be able to design UIs based on PalCom services, but their conclusion is that more research is needed to verify their goal. The work and ideas presented align with the concepts described by Bouraoui et al.[21], but it requires both the server and the client to use PalCom framework. This makes PalCom unsuitable as it is incompatible with the work of Skavhellen and Almås[5], which this thesis is integrating with.

Bouraqdai et al.[24] implement a solution using PharoJS and prove that JavaScript is a viable option for cross-platform applications. They use an MDA language and transpile into JavaScript. PharoJS shows great promise, but investigations reveal that it only has 81 stars on its Github repository[25]. This is a strong signal that PharoJS probably is a niche project, and there does not exist a PharoJS library that supports the communication protocols chosen by Skavhellen and Almås[5].

Boe[26] implemented a real-time application for displaying a graph with values from a robot. The paper focuses a lot on graph theory and repeats a lot of information from its sources, but the real-time aspect of the implementation uses a WebSocket connection. The benchmarks indicate that his client hardware had no difficulty with rendering a real-time application using WebSocket and a JavaScript front-end. These findings are confirmed by Soewito et al.[27], who prove and conclude that WebSocket connections offer substantially better performance than HTTP polling.

4.3 Analysis

Even though there exists multiple frameworks for building GUIs using MDA, none of the mentioned frameworks and implementations have been continuously developed to support real-time connectivity with good cross-platform support. WebSocket shows great promise for solving the problem of handling real-time communication between the clients and the server, in addition to having broad browser support. This thesis also has a dependency on the implementation by Skavhellen and Almås[5] that exposes a WebSocket endpoint and an API endpoint.

4.4 Proposed Solution

This thesis proposes to create a development framework for implementing a real-time GUI using web technology. The goals of this framework are to increase the portability and maintainability of a client application, increase confidence in automatic testing, and reduce required development time for features.

Chapter 5

Design

This chapter outlines the design of the framework.

5.1 Framework

The development framework breaks each page into a single "view" with one or multiple "components". Components are further broken down into a collection of "controls". A control is defined as a primitive element, i.e., text box or button. This means that we can describe the systems as follows:

$$view \subset \{components\} \tag{5.1}$$

$$component \subset \{controls\} \tag{5.2}$$

$$control = \{primitive\ elements\} \tag{5.3}$$

5.1.1 Components

Components are domain-specific models that either mutate server-side state or display server-side state. Components that require a robot-specific implementation are responsible for rendering the correct implementation seamlessly without additional information from the view. Each component has a logically independent index file that acts as middleware and is responsible for bootstrapping the correct code, as shown in Figure 5.4. Components are individually validated using automated testing[18] - implemented in an "index.test.js" file. Figure 5.5 shows the expected layout for individual components where "Component-A" has N number of implementations. This will allow multiple robot types to be added without having to alter the implementation of existing robots[28]. Component-B in Figure

5.5 is a robot agnostic component and consequently only exposes a single implementation. All components should implement its logic in an "index.js" file on the root level of the directory. Additionally, a "wireframe.drawio" file should be supplied, and it should contain any mock-ups or state diagrams used by a given component.

Components should follow a few rules:

- A component is responsible for creating the required padding around itself, and as a rule, components should add padding to their bottom.
- A component should not rely on other components to render itself.

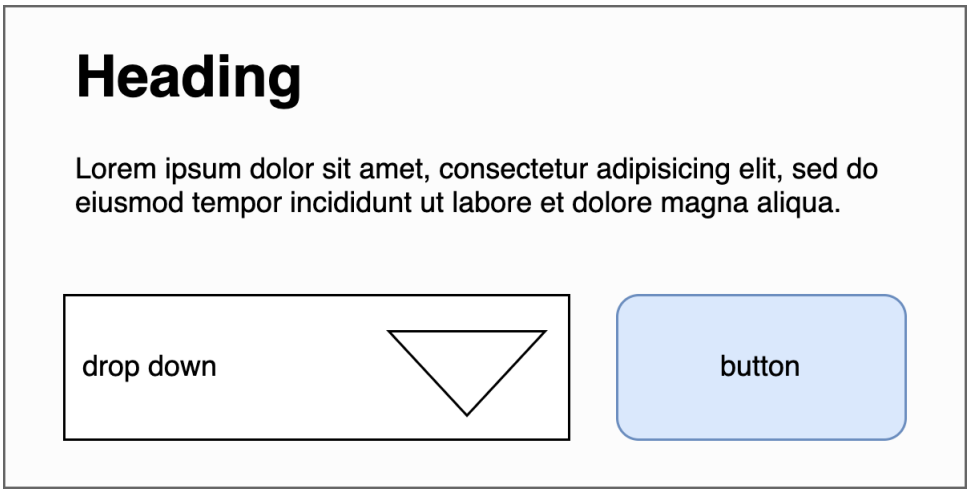

Figure 5.1: Example component with four primitives and required padding. A button, a drop down selector, a heading, and a text paragraph.

Real-time Components

A component is defined as a real-time component when it depends on receiving a continuous data stream from the server. The sequence diagram for real-time components is depicted in Figure 5.2.

High-Intensity Real-time Components

A High-Intensity Real-Time Component is designed to handle use cases where the data stream updates more rapidly than the client is able to render. The component splits its data handling into a separate worker process. This process is responsible for handling data stream messages and batches them while waiting for a threshold to be met before updating the GUI thread. The sequence diagram is shown in Figure 5.3, and the algorithm is described in Algorithm 5.1.

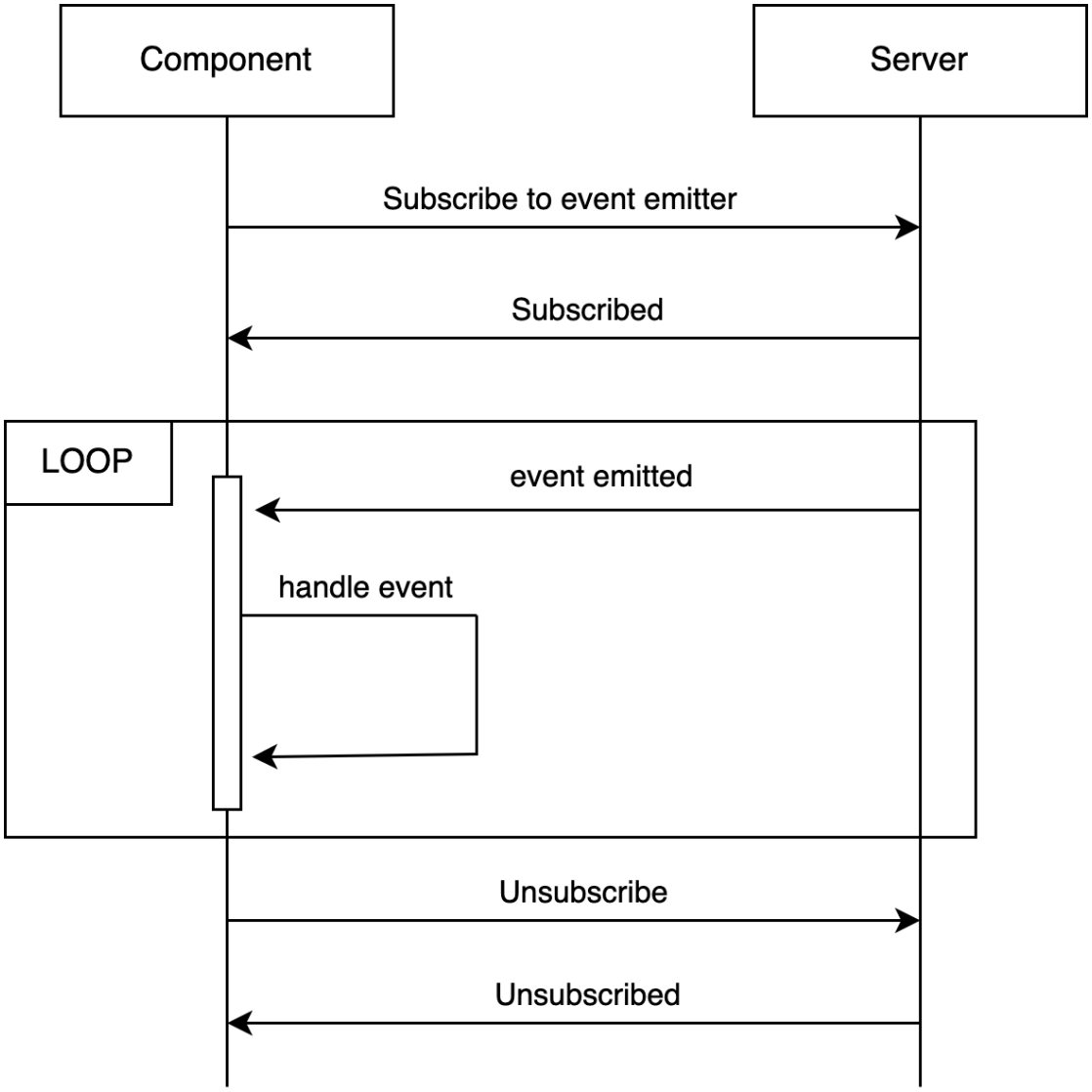

Figure 5.2: Sequence Diagram for Internal Processing Loop for Real-time Components.

Algorithm 5.1 High Intensity Processing Algorithm

```
counter ← 0
batchSize ← N
cutoff ← N
cache ← {}
function HandleDataStreamMessage(message m)
    for signal in m do
        cache{signal} ← signal                    ▷ implement component logic here
        if length(cache{signal}) > cutoff then
            truncateTail(cache{signal}, cutoff)
        end if
    end for
    if ++counter > batchSize then
        counter ← 0
        postMessage(cache)
    end if
end function
```

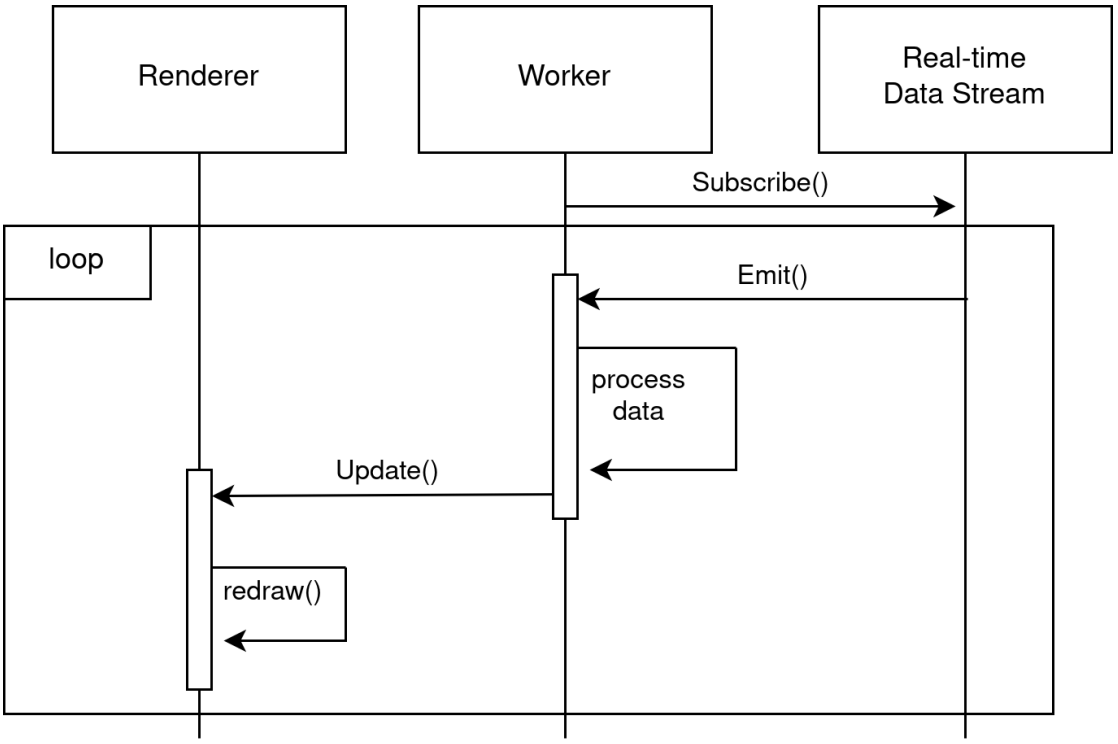

Figure 5.3: Sequence Diagram for Internal Processing Loop for High Intensity Real-time Components.

5.1.2 Design Metrics

Views and components in the framework should be designed with the measures from Pressman[29, Chapter 24.3] in mind, as outlined in Table 5.1. Averaging the measures for each view allows the framework to categorize the views as shown in 6.1 with regards to state and operations. The Unidraw framework[30] emphasizes the distinction between components that manipulate server state and components that read server state. Additionally, the views must consider the cognitive complexity [31, Chapter 8] and the density of components. This suggests that each view should primarily focus on achieving a fixed set of tasks[31, Chapter 8] and not overwhelm the user with information or functionality.

V1	Recognition complexity	Average number of distinct items the user must look at before making a data input decision
V2	Memory load	Average number of distinct data items that the user must remember to achieve a specific objective
V3	Graphics percentage	Average number of graphics media
V4	Word count	Total number of words that appear

Table 5.1: Design Metrics for Views and Components.

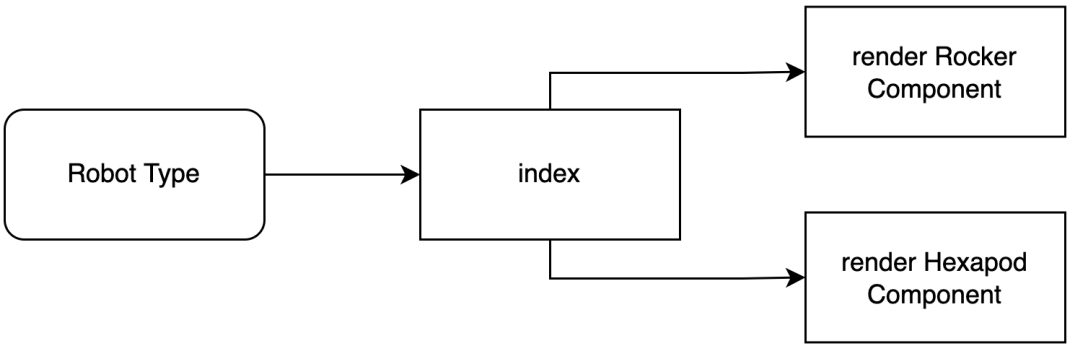

Figure 5.4: Component Index Forking. The index file renders the component that matches the signature of the robot type.

5.1.3 Views

Feature creep[31, Principle 15] has always been a potential issue when designing user-facing applications, and the available screen estate on tablets and phones is much smaller than on desktop computers. An oversaturated view is less likely to fulfill the actual user needs. Therefore, a view is responsible for composing components in a context-dependent presentation[30], and views express an immediate user need and must expose only essential functionality.

A view will fill the entire screen estate of the client device and uses a stacking single-column layout, as shown in Figure 5.7 and Figure 5.6. A view should not have more than three components and a maximum of one high-intensity real-time component. This layout paradigm will likely simplify design work when making cross-platform applications.

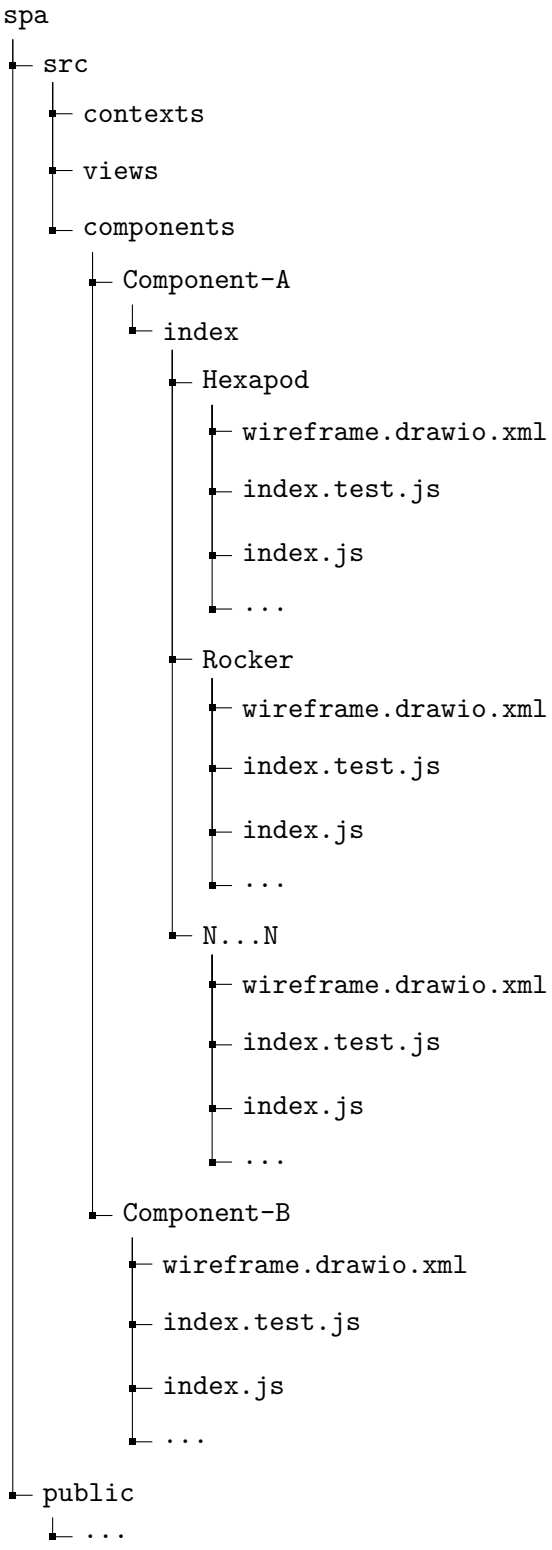

Figure 5.5: Internal Component Structure.

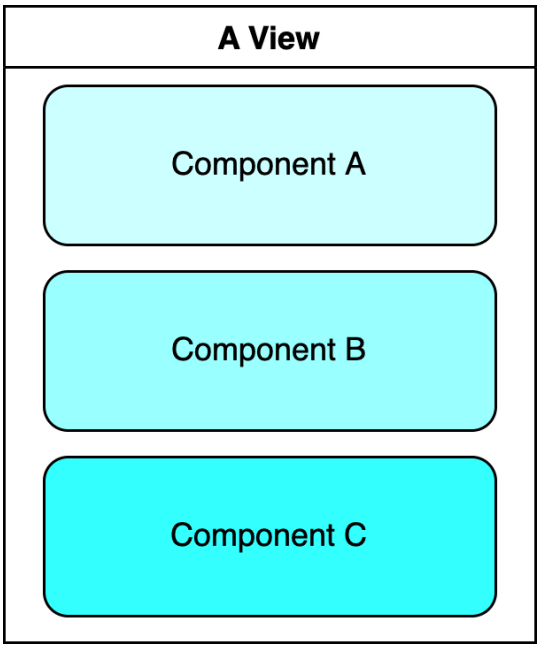

Figure 5.6: The components align in a single-column stacking layout in a desktop browser.

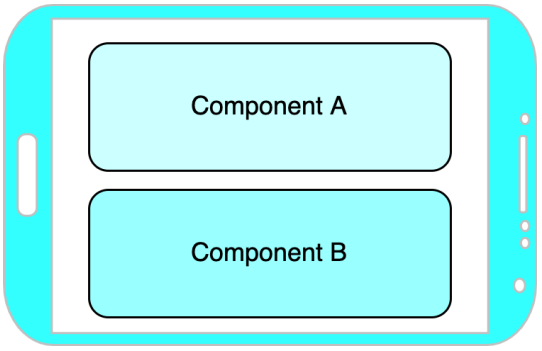

Figure 5.7: The components align in a landscape-oriented mobile browser. The single-column layout ensures consistency across devices.

Chapter 6

Use Cases

This chapter describes three use cases that the real-time GUI should support. The modeled use cases, views, and components are primarily based on the information available in the paper by Skavhellen and Almås[5] and use the methods described in Chapter 5. The first use case is a view for mutating server-side state to move the robot motors, the second use case is a view that monitors real-time updates from the server, and the third is a graphical presentation of the robot. An excerpt of the wireframes is included in the paper.

6.1 View Decomposition

This section contains the views that represent the use cases.

6.1.1 Actuating the robot motors

During operation, an end-user should be able to send commands to the server. A requirement by Skavhellen and Almås[5, Mastership.py] is that any client that wants to mutate the server-side state must acquire a mastership lock on the server.

The server logs several low level messages from its state machine, and robot connectivity events, that are interesting to the end-user. These messages are logged in CLI, but it's also convenient to present it in the client.

Using equation (5.1), we can model this scenario as:

$$Moving\ View \subseteq \{Master\ Control, Move\ Controller, Event\ Log\} \tag{6.1}$$

6.1.2 Monitoring Panel

The server may be connected to a physical robot or a kinetically simulated robot. When connected to a simulated robot, an end-user has no physical confirmation of a robot's movement. Additionally, Skavhellen and Almås[5, Mastership.py] state that only one client is allowed to hold the mastership lock. Therefore, it is advantageous to provide a view with only presentation components that do not require a mastership lock.

The firmware and drivers for the linear motors are not fully developed yet, and as new features are added, monitoring the different response values from the linear motors is interesting to the developers. Traditionally an oscilloscope has been connected to the circuitry, but the Hexapod, with its six separate motors, will require excessive rigging to connect enough oscilloscopes, and oscilloscopes are prohibitively expensive. However, the linear motors used in this thesis use drivers that expose these metrics, meaning that the server can publish the feedback values in real time.

In addition to the oscilloscope and the Event Log components, the view contains a component named "Axes" that displays key run time metrics of each individual motor axis.

Using equation (5.1), we can model this scenario as:

$$Moving\ View \subseteq \{Axes, Oscilloscope, Event\ Log\} \tag{6.2}$$

6.1.3 3D/2D Robot Viewer

The metrics from use case 6.1.2, can also be represented with a 3D/2D component. The 3D/2D insight will likely aid future work with simulated robots.

Using equation (5.1), we can model this scenario as:

$$3D/2D\ Robot\ View \subseteq \{3D/2D\ Viewer\} \tag{6.3}$$

6.1.4 Summary

This section described three different views to be implemented. The views described are decomposed into components in 6.2. The design metrics for the views are listed in Table 6.1

Name	$\overline{V1}$	$\overline{V2}$	$\overline{V3}$	$\overline{V4}$
Mover	High	High	Low	Medium
Monitoring Panel	Medium	Medium	High	High
Robot Viewer	Low	Low	High	Low

Table 6.1: Modelled View Design Metrics.

6.2 Component Decomposition

This section describes a list of components designed using the framework described in 4.4.

6.2.1 Master Control

The Master Control is responsible for acquiring the mastership lock of the robot controller. The component will initially render a button that allows the end-user to initiate the control sequence. Upon receiving the mastership privilege, the component will render a set of control buttons as shown in Figure 6.3. The buttons following a mastership appropriation are described in Table 6.2. Each command is individually verified by the server before execution. Additionally, the Master Control is responsible for maintaining the keep-alive WebSocket connection described by Skavhellen and Almås[5, WSInterface.py]. The Master Control component can act as a gatekeeper and prevent other children components from rendering prior to receiving mastership privilege. The components enveloped by the Master Control Component are hidden until Figure 6.3 reaches the "render" step.

Action Name	Description
Power On	Power robot on
Run Homing	Required after Power On
Start	Resumes execution of command queue
Pause	Pauses execution of command queue
Terminate Sessions	Shuts down the robot

Table 6.2: Available Actions for Master Control after appropriation of mastership.

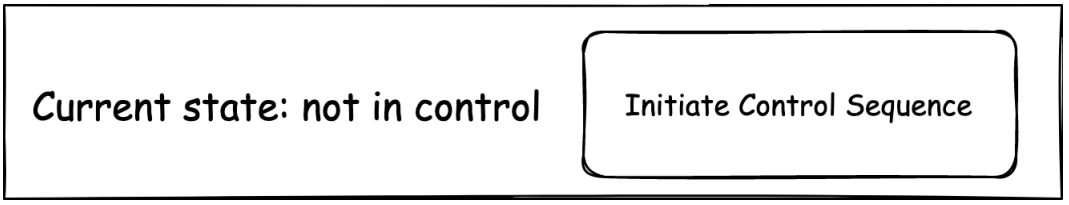

Figure 6.1: Wireframe of Master Control Component without mastership privilege.

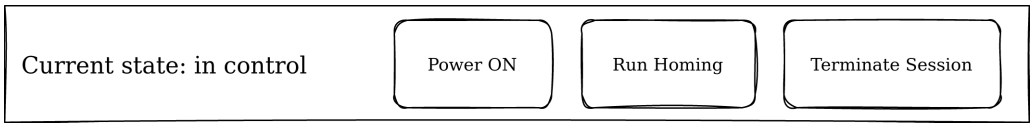

Figure 6.2: Wireframe of Master Control Component with mastership privilege.

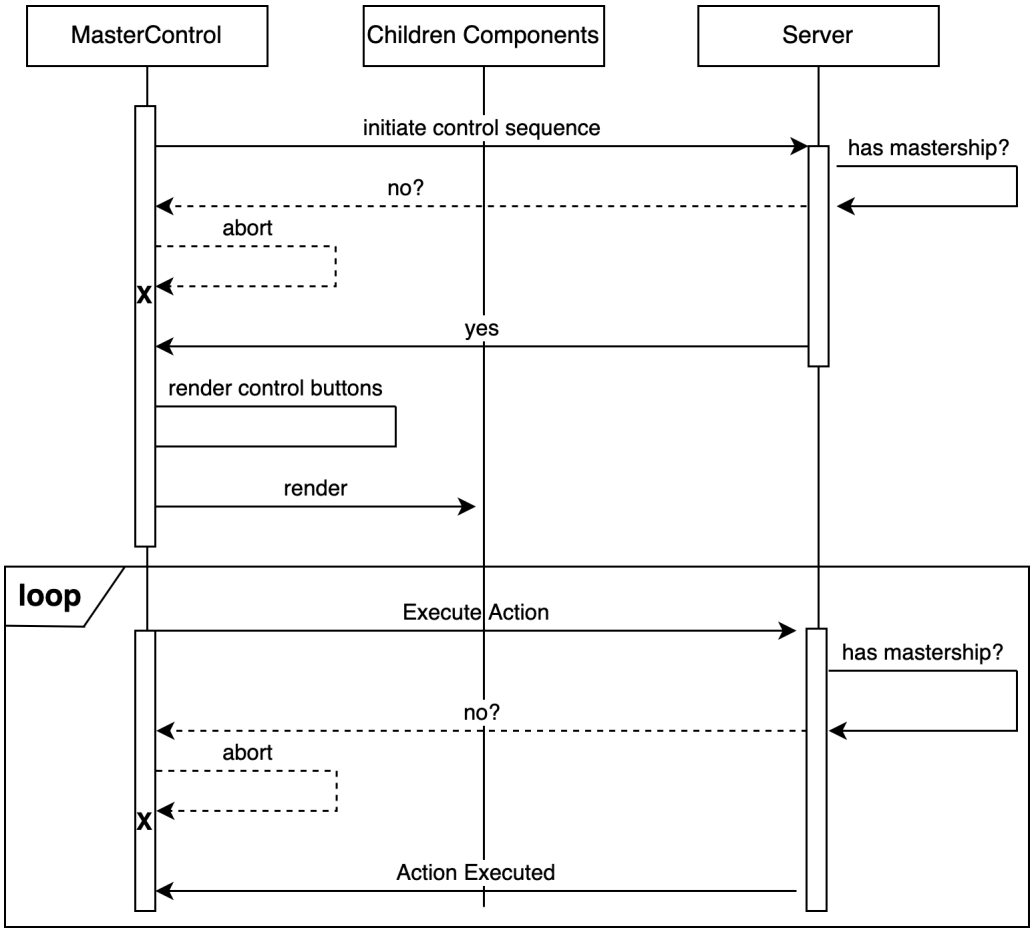

Figure 6.3: Master Control Sequence Diagram.

6.2.2 Move Controller

The Move Controller component is responsible for constructing and sending move commands in accordance with the models presented by Skavhellen and Almås[5, 3.4 and B.4.2]. The server supports two move types; move linear and move joint. The distinction is if the new robot position should be calculated using a linear path or by moving an individual motor. The two robots have very different movement models, and the Hexapod has the most complicated model with six axes compared to the Rocker's single axis.

The linear move command for the Hexapod has a schema that represents the roll, pitch, yaw, X, Y, and Z of the robot, as shown in Figure 6.4. The joint move command has the same basis, except that the individual values represent the extension of a motor.

The linear move command for the Rocker is a lot simpler since the robot only supports a single motor, as shown in Figure 6.5. Identically to the Hexapod, the joint move command only moves the single motor.

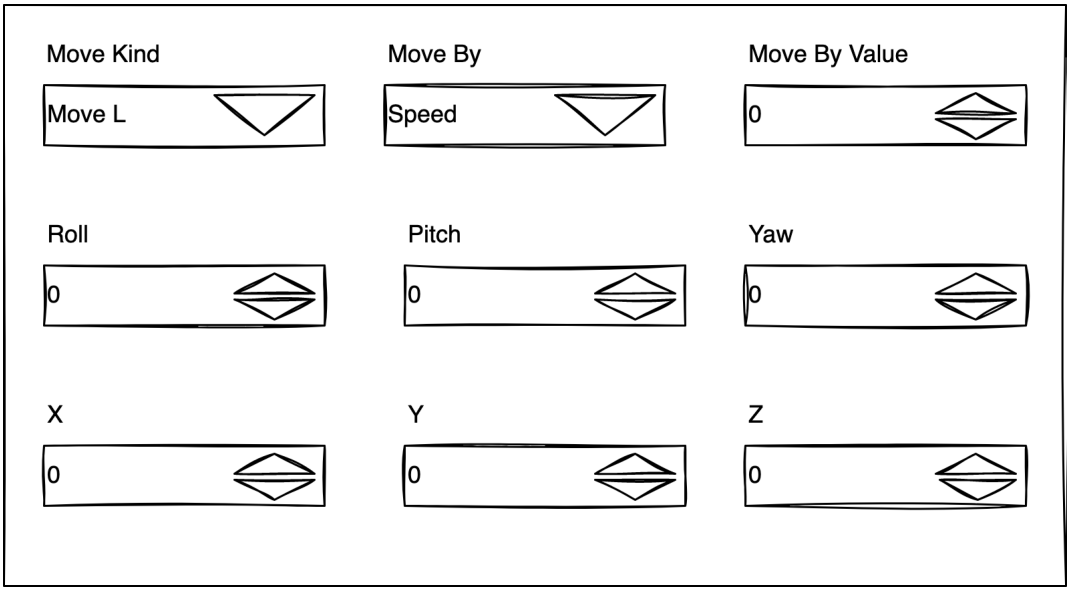

Figure 6.4: Wireframe of Hexapod Move Controller Component with Linear Command.

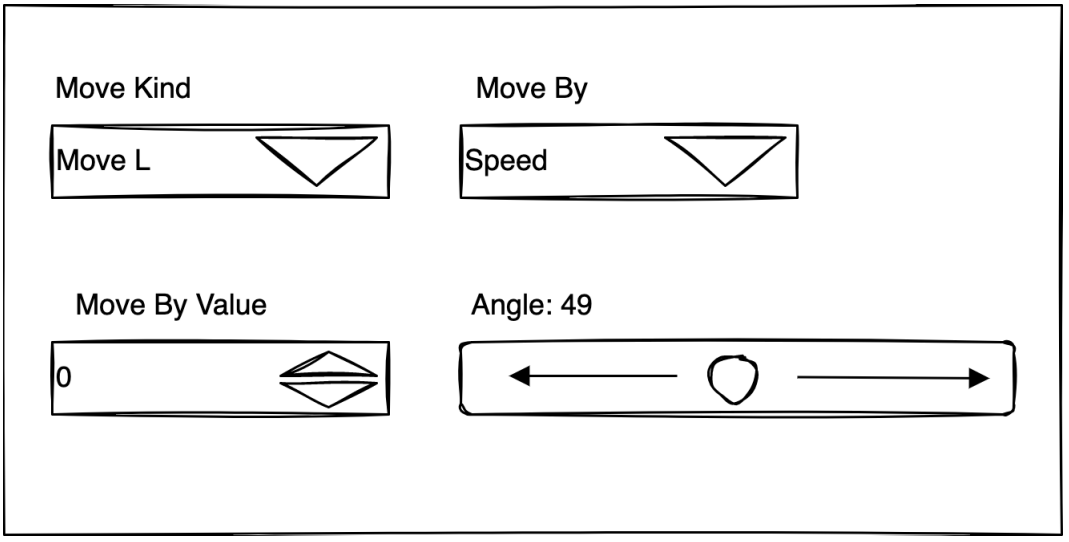

Figure 6.5: Wireframe of Rocker Move Controller Component.

6.2.3 Event Log

The Event Log component subscribes to the "eventlog" WebSocket namespace described by Skavhellen and Almås[5, B.4.1] and continuously updates a text area with the event stream.

The Event Log is considered a real-time component, and its internal processing loop follows the template from Figure 5.2.

6.2.4 Axes

The Axes component connects to the "static_signals" WebSocket namespace described by Skavhellen and Almås[5, B.4.1]. The static signal stream contains the length of the different motors. Each motor will be presented using a range control with its maximum and minimum values derived from "/api/signals/subscription"[5, B.4.1].

The Axes component is considered a real-time component, and its internal processing loop follows the template from Figure 5.2.

6.2.5 3D/2D Viewer

The 3D/2D Viewer component subscribes to the "static_signals" WebSocket namespace described by Skavhellen and Almås[5, B.4.1]. This component has two different implementations; one for the Hexapod and one for the Rocker. The Rocker's movement is presented on a canvas with a 2D figure depicting its orientation and movement. The Hexapod requires a 3D presentation since it supports three dimensional movement patterns. The 3D presentation should also support a movable camera and rudimentary zooming.

The 3D/2D Viewer is considered a real-time component, and its internal processing loop follows the template from Figure 5.2.

6.2.6 Oscilloscope

The Oscilloscope component connects to the "signal" WebSocket namespace described by Skavhellen and Almås[5, B.4.1]. The signal stream may consist of any of the metrics exposed by the robot, and the stream is rendered on a graph continuously.

The Oscilloscope component is considered a high-intensity real-time component, and its internal processing loop follows the template from Figure 5.3 with the extensions that it controls its own signal stream as depicted in Figure 6.6.

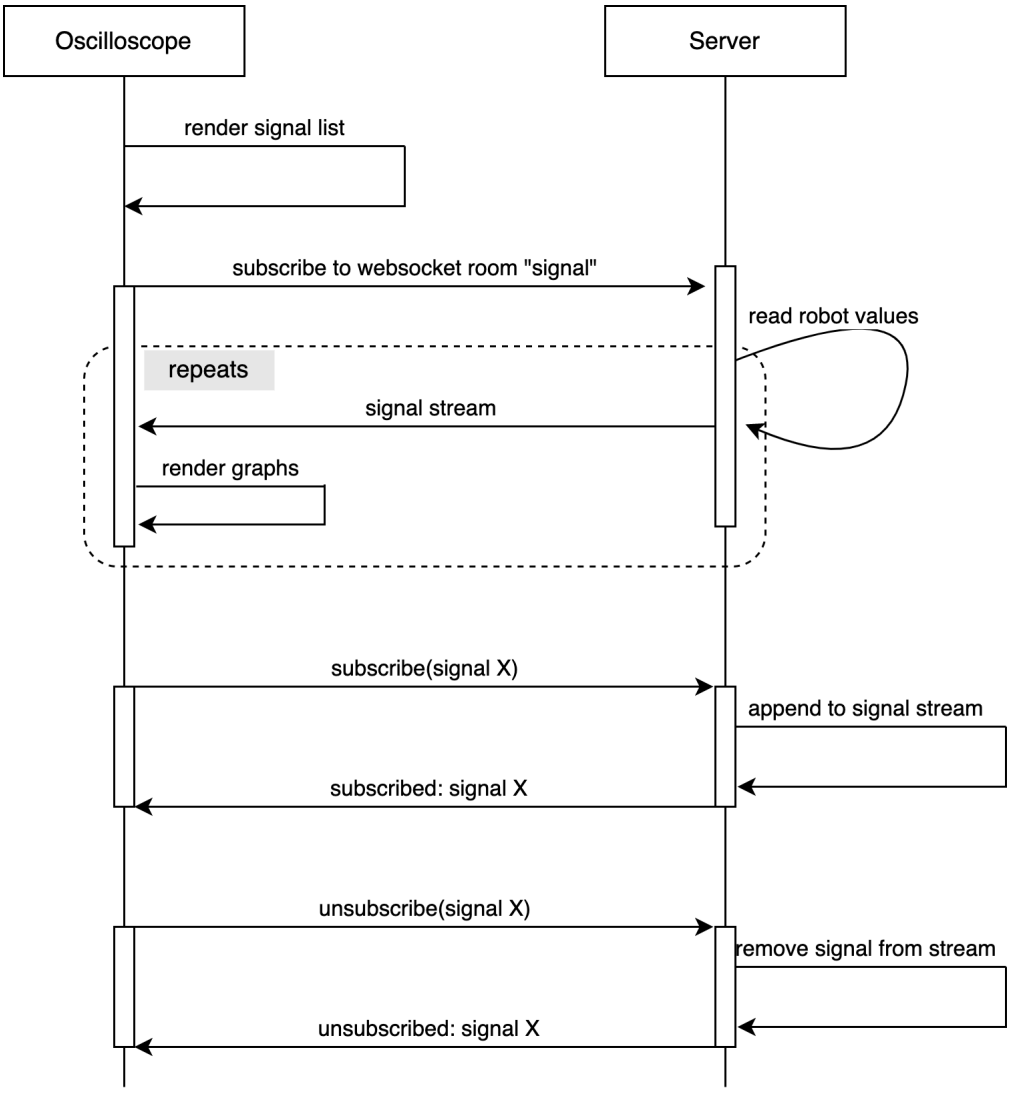

Figure 6.6: Oscilloscope Sequence Diagram.

Chapter 7

Experimental Evaluation

7.1 The Web Application

This chapter presents an implementation of Chapter 6. The first section describes the technology used and includes some excerpts from the implementation. The second section contains the results from the implementation.

7.1.1 Setup and implementation

The framework does not dictate which SPA library to use, but looking at stars, version numbers and maintainers on `www.github.com`, React looks like the strongest candidate with a high star count and a large enterprise as maintainer, as shown in Table 7.1.

React uses a declarative coding style and has a large amount of available support libraries on `https://www.npmjs.com`. React uses a state manager to trigger DOM updates and supports WebWorkers[32].

Name	Maintainer	Version	Stars on Github	License
AngularJS	Google	Deprecated	59.5k	MIT
Angular	Google	v13	80.8k	MIT
VueJS	Community	v2.6	195k	MIT
React	Facebook	v17	186k	MIT

Table 7.1: JavaScript library comparison.

This implementation uses as few external libraries as possible. Some of the key libraries are: React Router[33] for navigation and View separation, shown in Figure 7.1. The routes are:

- https://<server-url>/mover
- https://<server-url>/presentation
- https://<server-url>/viewer

HighCharts[34] draws the graphs used in the Oscilloscope component and Three.JS[35] handles the 3D scenes in the 3D/2D Viewer.

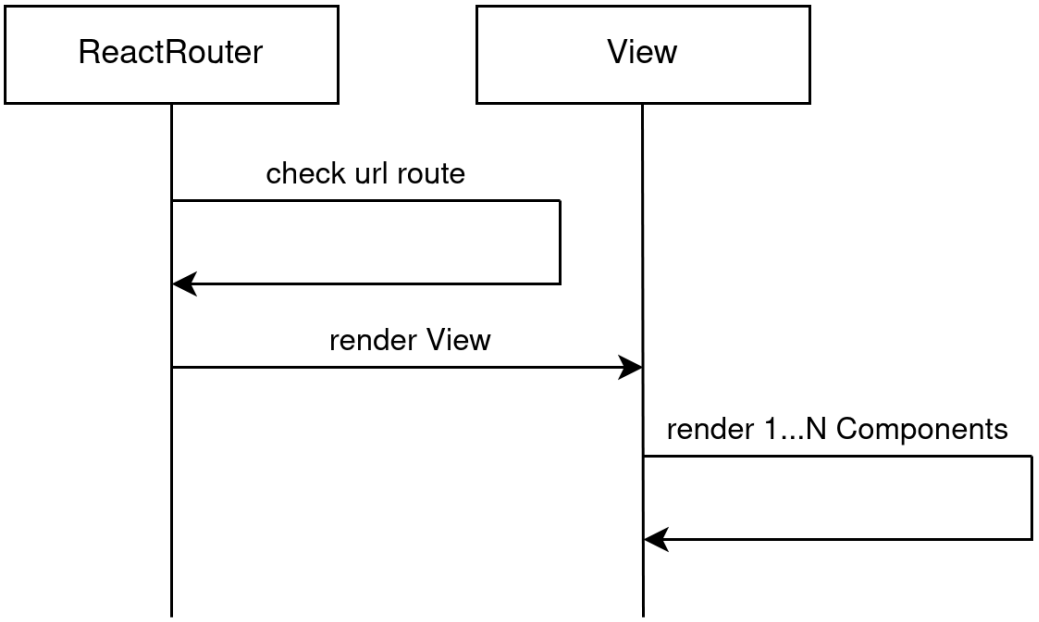

Figure 7.1: React Router interaction with a view.

7.2 Experimental Setup

The server code ran on a Dell R720 with Intel(R) Xeon(R) CPU E5-2690 v2, with 128GB of RAM, and the server was connected via Ethernet to an access point. The server was running Centos 8 Stream (4.18.0-277.el8.x86_64) and used Python 3.8.12. The robot server was configured using the instructions provided by Skavhellen and Almås[5].

The client ran on a Lenovo T14s Gen 1 laptop with AMD Ryzen 7 PRO 4750U, with 32GB of RAM, and the laptop was connected to the access point using 5GHz WiFi. The laptop was running Fedora Linux 35 (5.17.8-200.fc35.x86_64). The laptop was configured using the instructions in the README.md file in the spa directory of the repository. The experiments were executed in Chromium (Version 100.0.4896.127).

The data in the graphs were averaged from three separate runs, and the browser cache was cleared between each run. There was a 3-5% difference in CPU usage when subscribing to

a single signal data stream or subscribing to a multi-signal data stream in the real-time components. Therefore, only a single signal setup is presented in the results.

7.3 Experimental Results

Skavhellen and Almås[5] filmed their experiments and the videos are available here; https://www.youtube.com/watch?v=AxgRVIJXuXc (Rocker) and https://www.youtube.com/watch?v=XUzQpsbdWJU (Hexapod).

The Design Metrics from 5.1.2 are listed in Table 7.2. The solution implemented a CI workflow for the SPA with two steps; build the code and run tests. Figure 7.2 depicts the build results posted on the Github Actions website. The build pipe has a passing rate of 80%. The remaining 20% runs failed because of linting errors. The CI system sends out notifications when anyone pushes code and reports if the build was successful or a failure.

Name	$\overline{V1}$	$\overline{V2}$	$\overline{V3}$	$\overline{V4}$
Mover	Medium	Medium	Low	Medium
Monitoring Panel	High	Medium	High	High
Robot Viewer	Low	Low	High	Low

Table 7.2: Measured View Design Metrics.

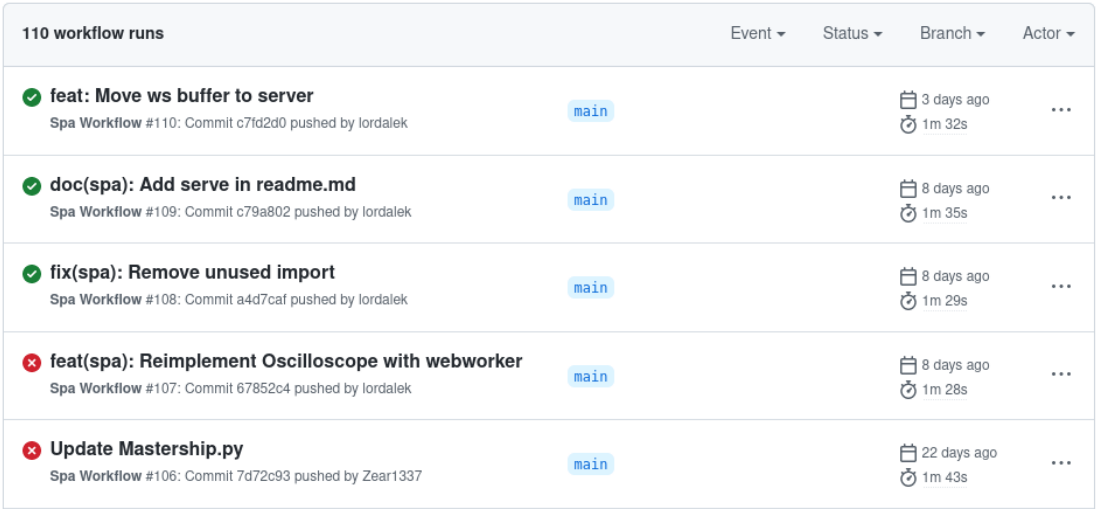

Figure 7.2: Github Actions workflow showing the build status of different commits.

7.3.1 Mover View

The Mover View without mastership is presented in Figure 7.5. The view renders the Hexapod presentation, as shown in Figure 6.4, with all the motors. When "moveJ" is

selected, it switches from the "Roll, Pitch, Yaw, X, Y, Z" to sending individual motor length values. The Rocker presentation is shown in Figure 7.3 and switches to robot length if the "Move Kind" is changed to "moveJ".

The "Run Homing" and "Power ON" buttons work, and the server feedback can be read in the event log. These actions are idempotent since the server maintains its own state machine. The "Pause" and "Start" buttons will respectively pause the robot's execution queue and resume the execution. Pressing "Terminate Session" will turn off the robot and terminate the session.

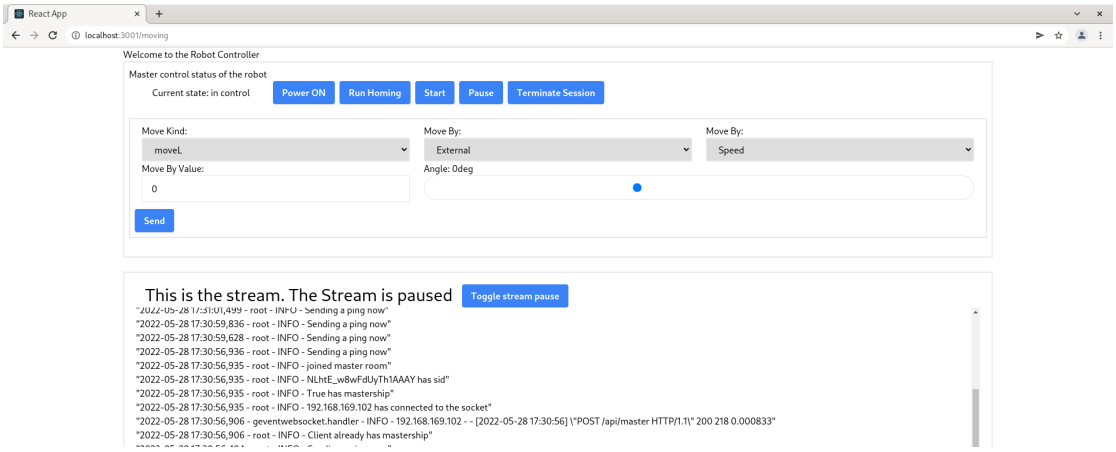

Figure 7.3: Rocker Move View with mastership.

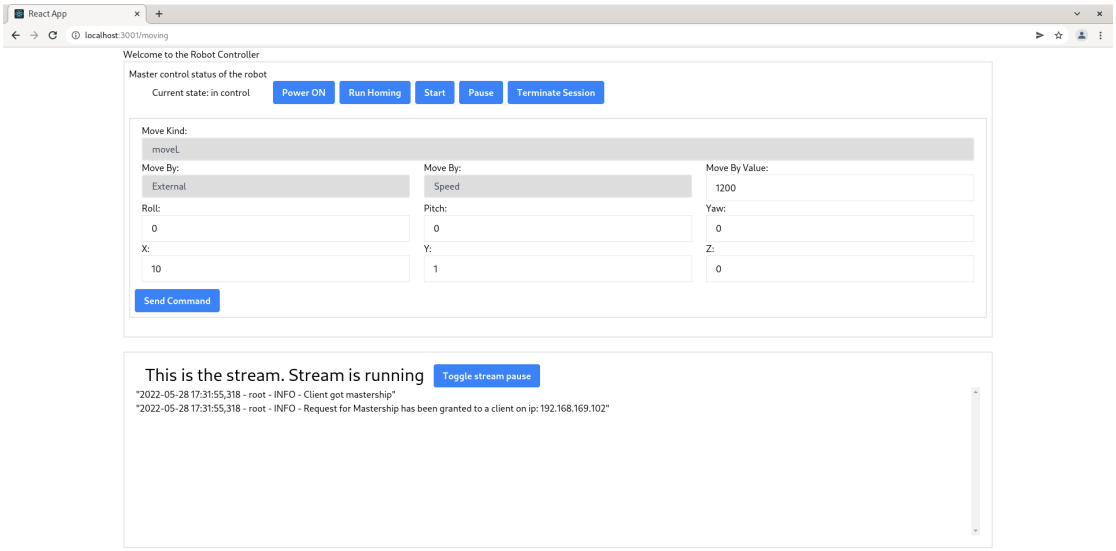

Figure 7.4: Hexapod Move View with mastership.

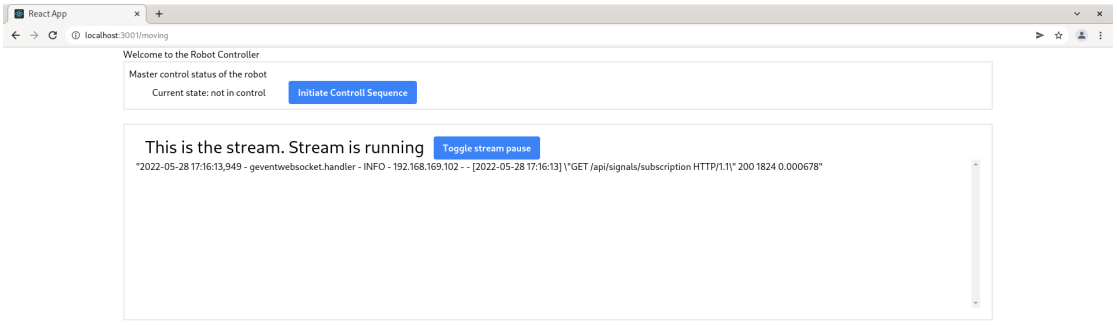

Figure 7.5: Move View without mastership.

7.3.2 Monitor View

The wireless interface on the laptop was monitored during the tests since the Monitor View has multiple real-time components. This was done using `ip -s link show {interface}`, and it showed less than five packets lost per 10 000 packets. While testing the view without batching, the high intensity real-time components will reset after three to four minutes. During testing without batching, the browser became unresponsive and required the tester to kill the browser process. The WebSocket connection was stable when testing with and without batching enabled.

Screenshots of the view is shown in Figure 7.6.

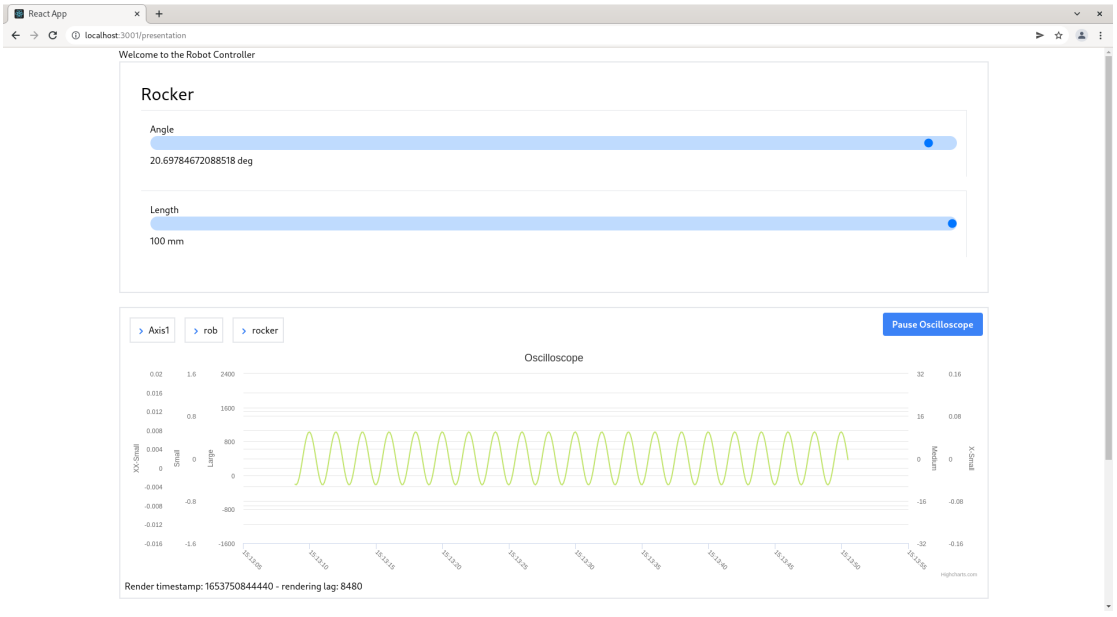

Figure 7.6: Rocker Presentation View.

The server emits data every 20ms, and the Oscilloscope Component uses a worker to handle the data stream. Fifty events per second is not considered high throughput,

but the CPU cost of unwrapping and processing the data structure from the stream is high. Render lag is defined as the time from the client receiving the data frames until the data frame is drawn on the screen. The maximum render lag of the Oscilloscope component is graphed in Figure 7.8 and Figure 7.9. The experiments with batching enabled show an improvement in render lag, but the rendering is still choppy and uneven. The React state manager spends between 70ms and 80ms processing each of the Worker updates, as shown in the Chromium Profiler in Figure 7.10. That figure also suggest that React groups state updates together instead of processing its queue immediately. In Figure 7.7, the queuing is visible as the worker process is doing its work and updating the UI, but React postpones the rendering. The number "12" represents the number of log statements made. The log statements are made in such a short time span that the browser consolidates them into a single line. Notice that there are log statements from the worker process prior to and after the graph render.

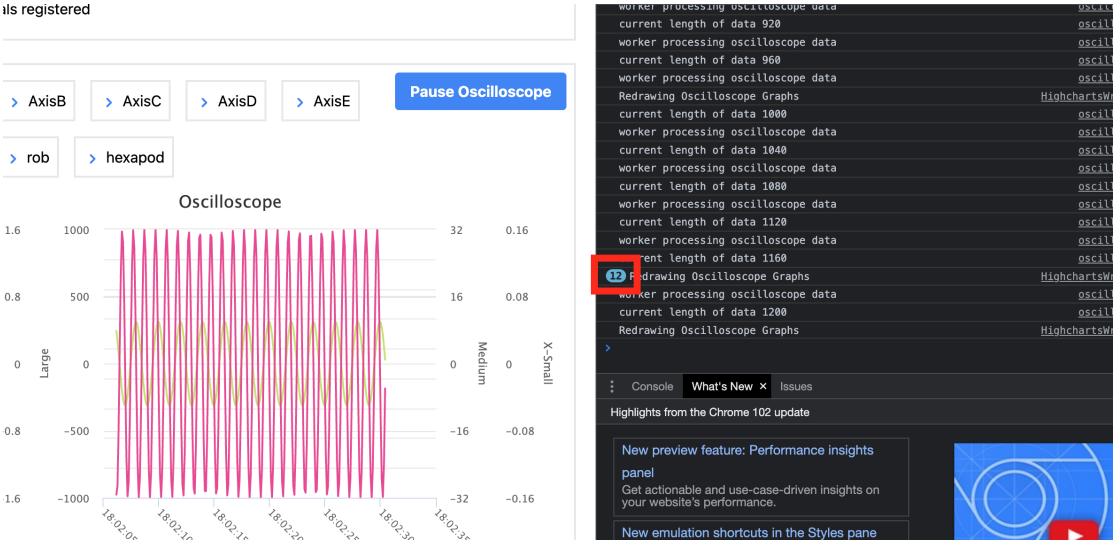

Figure 7.7: React queuing state updates instead of executing immediately.

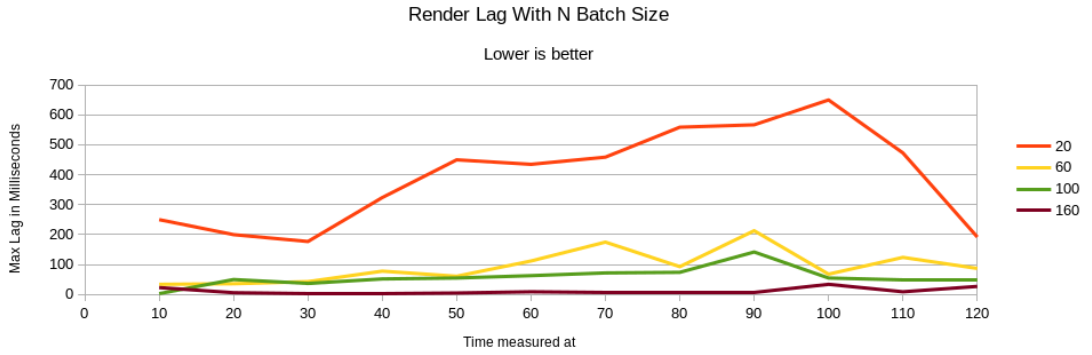

Figure 7.8: Observed Max Lag in Oscilloscope Component with Batching.

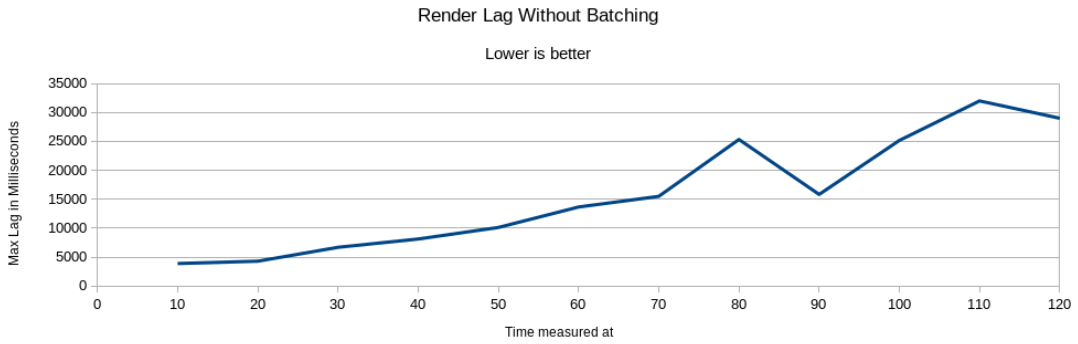

Figure 7.9: Observed Max Lag in Oscilloscope Component without Batching.

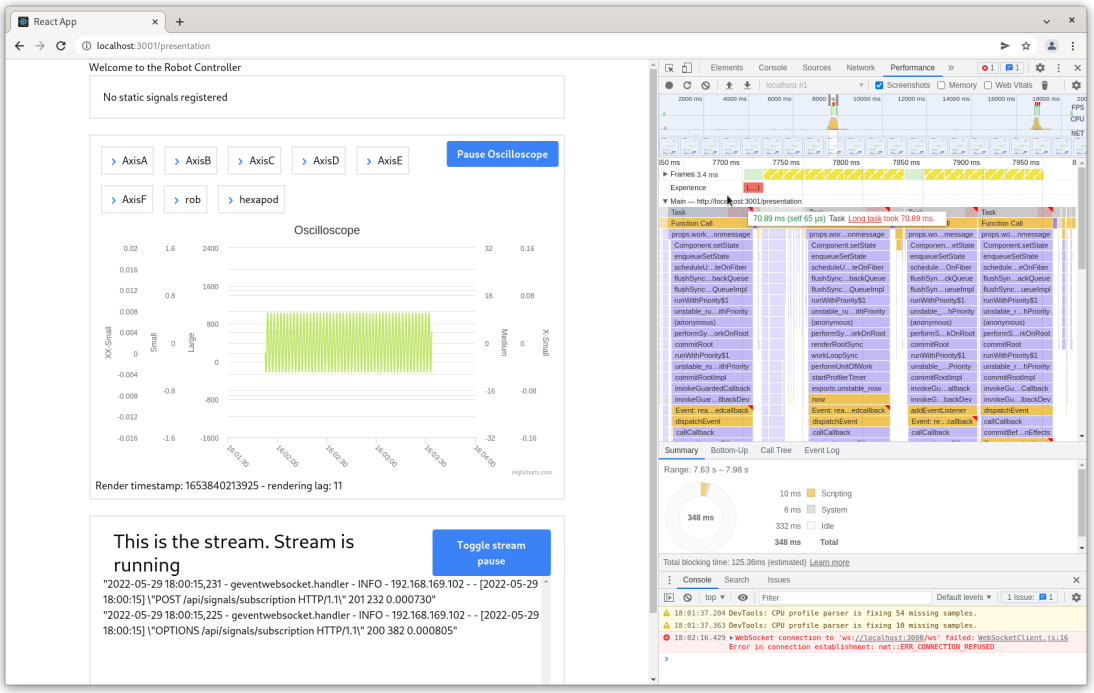

Figure 7.10: Chromium Profiler measuring time used by react to process state updates.

3D/2D Robot Viewer

The screenshot of the robot viewer is shown in Figure 7.11. Zooming works and camera rotation works, but not repositioning the camera. The drawn hexapod does not update.

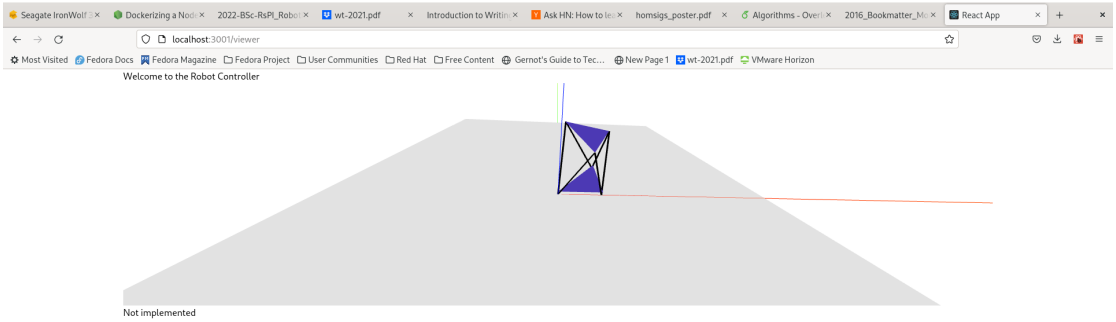

Figure 7.11: 3D/2D Robot Viewer with the Heaxpod 3D viwer on top and the not implemented 2D Rocker component beneath.

Chapter 8

Discussion

This section analyses the experiments from Chapter 7 and compares the results with the framework presented in Chapter 5 and the use cases from Chapter 6. Section 4.4 stated that some of the key points of the development framework are to build real-time applications faster and with good quality. This section also answers the research questions from Chapter 1.

8.1 Research Question 1

What is required of a framework to provide clear guidelines for developing real-time applications for a robot controller?

This research question is best answered by analyzing the use case implementations, but the general notion is that the effort of modelling the individual components with wireframes and sequence diagrams had a positive effect.

The Mover View is considered a success since it is both able to acquire mastership on the server and send move commands based on robot types. Additionally, its event log functions properly and gives good feedback to the end-user. Table 7.2 shows that the view scored lower on V1 and V2 than modeled in Table 6.1. This might be because abstract domain concepts, such as mastership, appear harder to understand in low-fidelity wireframes as compared to functioning components. This aligns with the findings by Vlissides et al.[30]. However, our framework is able to communicate abstract concepts using sequence diagrams, a central part of the work on MDA by Bouraoui et al.[21], thus confirming their model.

The same patterns apply to the Monitor View with the addition that this view has a high-intensity real-time component. Interestingly, the implementation scored higher for

the same metrics that the Mover View scored lower in. This might be an indication that future work on the framework should investigate more about the design metric[5.1.2] guidelines.

The 3D/2D Robot View has good wireframes, and the concepts are easy to understand, but the implementation details are far more complex than perceived by the wireframes. This is a use case where the framework is able to communicate the goal of the view, but it does not translate into the domain of 3D graphics. 3D graphics is a discipline in itself, and it's hard to describe the concepts without polluting the general model with the internals of the 3D libraries used. This might help explain why implementing the 3D/2D Viewer component required a high amount of effort compared to the other components.

Based on the analysis of the use cases, it's possible to conclude that the guidelines in Chapter 5 are sufficient, but some sections require some more research. The design metrics section gives a good understanding of user experience, but no user testing data has been collected. Future work should introduce user-based testing on the low fidelity wireframes to reveal unfitting requirements. The framework also provides a pattern for integrating with real-time data streams, but this is discussed in the next research question.

8.2 Research Question 2

How does the framework scale with high intensity data streams?

Testing show negligible packet loss during operation, which confirms the claims of Soewito et al.[27] and Boe[26]. However, the server implementation in SocketIO limits the client to platforms that supports the SocketIO library. Future work should investigate the possibility of using plain WebSockets to reduce dependency on third-party libraries. With the introduction of HTTP/3 it's possible that future work should use WebTransport[36], replacing WebSockets altogether. Webtransport uses UDP instead of the WebSocket's TCP, and UDP might reduce the general latency of the system, but more research is needed to verify this.

Figure 7.8 shows a steep decrease in render lag around the 100s mark, and this effect is also present around the 80s mark in Figure 7.9. I have not looked deeply into why this is, but since React is open source, it's possible to investigate the source code. React's source shows queuing of state updates[37] instead of immediate execution. A deeper investigation is required for more insight, but that's outside the scope of this thesis.

Figure 7.8 also shows that the larger the batch size is, the smaller the render lag is. This is a two-sided problem; having too small batches overloads the client, but large batches means that the client has to wait longer for updates. Setting the batch size to 160 gives approximately 0ms lag, but the client has to wait 3200ms between updates. This is bad UX, and further work should look into implementing a variable batch size.

The experiment data show that introducing batch processing reduces rendering lag significantly, but React decides to queue its state updates as the Profiler shows. React's state queuing is confirmed in Figure 7.7. This is unfortunate for real-time components and reduces React's usefulness as a real-time component library. With an average processing time of 80ms per cycle, it should be fast enough for real-time components. However, its postponement of state updates results in stuttering during graph drawing and consequently poor user experience. The framework has two models for real-time components, and React renders the Axes component without issues but struggles with high-intensity real-time components. Even though the data in Figure 7.8 indicate that the high-intensity model[5.1.1] is valid, it's not possible to completely confirm the model or algorithms due to React as a library. Future work should try to implement the Monitor View in a different JavaScript library with more accessible performance tuning.

8.3 Research Question 3

What impact does continuous integration have on the development?

One of the positive experiences with our CI was the email notifications. This proved to be a very effective way of communication since most of the team did not work on the codebase simultaneously. Linting proved to be a source of "non-value generating" commits, commits which only remove a trailing white space or add a missing new line at the end of a document. One could argue that linting is needed for a consistent codebase, but as most of the linting errors didn't directly relate to logical code, it becomes a forced practice. Since all of the failing CI runs were due to linting and none due to tests, it indicates that developers run tests locally and honor the test results but forget to run linting locally or ignore it.

The development team experienced few regression bugs, but we used tests in conjunction with CI from project start. The solution lacks a coherent test suite for end-to-end testing. The front-end tests do not have any methodical coupling with the server API. I suggest declaring the API in a model so that both server and client pragmatically generate their bindings dynamically. A declared API aligns with the findings of Mossige et al. [18], but

my paper does not provide any quantifiable measures to confirm the claims of reduced development time.

The consensus among the developers is that CI is a good thing and that it increases visibility and promotes regular check-ins of working code.

Chapter 9

Conclusion

This thesis presents a framework for implementing real-time client applications for a robot controller. The framework presents a set of guidelines and patterns to aid when implementing use cases. The framework is complete, and a continuous integration server integrates with the codebase.

This thesis implemented three use cases, and most of that implementation is complete. The framework showed a good match for the use cases, but some components did not function as intended. This was due to the inner workings of third-party libraries, but future work should try to re-implement those use cases using other libraries.

This thesis also answers three research questions related to the framework and concludes that the framework is a good candidate when developing real-time client applications. The framework provides good flexibility with its views and reusable components, but it does not give any directions on including user testing during development.

The framework produces reusable components, and future work should investigate the possibility of using a layout manager to create views. The layout manager would make the framework accessible to non-programmers and possibly reduce the framework's risk of being forgotten.

List of Figures

List of Tables

Appendix A

Instructions to Compile and Run System

Instructions on how to run this application are located in the README.md and spa/README.md files in the code repository.

Run "python3 rspi/RsPiRobot.py –robot Rocker –servotime 3 –sim" to quickstart a server.

Attached source code in zip format.

Bibliography

[1] Reimund Neugebauer, Sophie Hippmann, Miriam Leis, and Martin Landherr. Industrie 4.0 - from the perspective of applied research. *Procedia CIRP*, 57:2–7, 2016.

[2] Steinar Þór Bachmann, Ragnar Sigbjörnsson, and Jónas Þór Snæbjörnsson. Earthquake simulator earthquake simulator - hexapod, 2010. URL http://jardskjalftamidstod.hi.is/wp-content/uploads/2016/10/R-VoN-2010-Bachmann-et-al.pdf. Last accessed 13 June 2022.

[3] Enver küçükkülahlı and Recep GULER. Open source mobile robot with raspberry pi. *Balkan Journal of Electrical and Computer Engineering*, 3, 12 2015. doi: 10.17694/bajece.29976.

[4] Chengjing Yu, Xudong Ma, Fang Fang, Kun Qian, Shun Yao, and Yanping Zou. Design of controller system for industrial robot based on rtos xenomai. pages 221–226, 06 2017. doi: 10.1109/ICIEA.2017.8282846.

[5] Sindre Skavhellen and Lars Almås. Utvikling av rest interface og robot kontroller logikk på en raspberry pi. 2022. University of Stavanger, Faculty of Science and Technology, Kjell Arholms gate 41, 4021 Stavanger.

[6] Cambridge University Press. framework, 2022. URL https://dictionary.cambridge.org/dictionary/english/framework. Last accessed 15 June 2022.

[7] Wikipedia. Roll, yaw, and roll, . URL https://en.wikipedia.org/wiki/Six_degrees_of_freedom. Last accessed 13 June 2022.

[8] GregorDS. Six degrees of freedom, own work, cc by-sa 4.0. URL https://commons.wikimedia.org/w/index.php?curid=38429678. Last accessed 13 June 2022.

[9] Ethercat Technology Group. Éthercat. URL https://www.ethercat.org/default.htm. Last accessed 13 June 2022.

[10] Socket.IO. Socket.io, 2022. URL https://socket.io/. Last accessed 15 April 2022.

[11] Roger S. Pressman. *Software Engineering, A Practioner's Approach*. The McCraw-Hill Company, 1221 Avenue of the Americas, New York, NY 10020, 2nd edition, 2010.

[12] The Qt Company. Qt, 2022. URL https://www.qt.io/. Last accessed 12 May 2022.

[13] Mozilla Foundation. Responsive design, 2022. URL https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Responsive_Design. Last accessed 17 April 2022.

[14] Wikipedia. Single page application, . URL https://en.wikipedia.org/wiki/Single-page_application. Last accessed 17 April 2022.

[15] Wikipedia. Server-side scripting, . URL https://en.wikipedia.org/wiki/Server-side_scripting. Last accessed 17 April 2022.

[16] Wikipedia. Model view controller, . URL https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller. Last accessed 17 April 2022.

[17] Martin Fowler. Continuous integration, 2006. URL https://martinfowler.com/articles/continuousIntegration.html. Last accessed 17 April 2022.

[18] Morten Mossige, Arnaud Gotlieb, and Hein Meling. Testing robot controllers using constraint programming and continuous integration. 2014. doi: https://doi.org/10.1016/j.infsof.2014.09.009.

[19] M. Rochefort, N. De Guise, and L. Gingras. Development of a graphical user interface for a real-time power system simulator. *Electric Power Systems Research*, 36(3):203–210, 1996. ISSN 0378-7796. doi: https://doi.org/10.1016/0378-7796(95)01033-5. URL https://www.sciencedirect.com/science/article/pii/0378779695010335.

[20] A. Patek, Wojciech Zabierowski, and A. Napieralski. Building winfbase application using windows forms 2.0 and c#.net. 2008. URL https://www.researchgate.net/publication/251883628_Building_Winfbase_application_using_windows_forms_20_and_CNet.

[21] Amina Bouraoui and Imen Gharbi. Model driven engineering of accessible and multi-platform graphical user interfaces by parameterized model transformations. *Science of Computer Programming*, 172:63–101, 2019. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2018.11.002. URL https://www.sciencedirect.com/science/article/pii/S0167642318304301.

[22] Björn A. Johnsson and Gunnar Weibull. End-user composition of graphical user interfaces for palcom systems. *Procedia Computer Science*, 94:224–231, 2016. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2016.08.035. URL https://www.sciencedirect.com/science/article/pii/S1877050916317823. The 11th International Conference on Future Networks and Communications (FNC 2016) / The 13th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2016) / Affiliated Workshops.

[23] Björn A. Johnsson and Boris Magnusson. Towards end-user development of graphical user interfaces for internet of things. *Future Generation Computer Systems*, 107:670–680, 2020. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2017.09.068. URL https://www.sciencedirect.com/science/article/pii/S0167739X17321660.

[24] Noury Bouraqadi and Dave Mason. Test-driven development for generated portable javascript apps. *Science of Computer Programming*, 161:2–17, 2018. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2018.02.003. URL https://www.sciencedirect.com/science/article/pii/S0167642318300595. Advances in Dynamic Languages.

[25] PharoJS. Pharojs, 2022. URL https://github.com/PharoJS/PharoJS. Last accessed 12 May 2022.

[26] Fabian Legland Boe. Real-time graph visualization in a single-page application, 2019. University of Stavanger, Faculty of Science and Technology, Kjell Arholms gate 41, 4021 Stavanger.

[27] Benfano Soewito, Christian, Fergyanto E. Gunawan, Diana, and I Gede Putra Kusuma. Websocket to support real time smart home applications. *Procedia Computer Science*, 157:560–566, 2019.

[28] Wikipedia. Open–closed principle, 2022. URL https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle. Last accessed 17 April 2022.

[29] Roger S. Pressman. *Software Engineering, A Practioner's Approach*. The McCraw-Hill Company, 1221 Avenue of the Americas, New York, NY 10020, 2nd edition, 2010.

[30] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Trans. Inf. Syst.*, 8(3):237–268, jul 1990. ISSN 1046-8188. doi: 10.1145/98188.98197. URL https://doi.org/10.1145/98188.98197.

[31] Mica R Endsley, Betty Bolte, and Debra G. Jones. *Designing for situation awareness: an approach to user-centered design*. CRC Press, Taylor and Francis Group, 6000

Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742, 2003. ISBN 0–7484–0967–X.

[32] Mozilla Foundation. Using web workers, 2022. URL https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. Last accessed 8 June 2022.

[33] Remix. React router, 2022. URL https://reactrouter.com/docs/en/v6. Last accessed 15 April 2022.

[34] Highcharts. Highcharts, 2022. URL https://www.highcharts.com/. Last accessed 15 April 2022.

[35] ThreeJs Authors. Three.js, 2022. URL https://threejs.org/. Last accessed 15 April 2022.

[36] Alan Frindell, Eric Kinnear, Tommy Pauly, Martin Thomson, Victor Vasiliev, and Guowu Xie. WebTransport using HTTP/2. Internet-Draft draft-ietf-webtrans-http2-03, Internet Engineering Task Force, March 2022. URL https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-http2-03. Work in Progress.

[37] Facebook. React enqueue, 2022. URL https://github.com/facebook/react/blob/1ad8d81292415e26ac070dec03ad84c11fbe207d/packages/react-dom/src/server/ReactPartialRenderer.js#L439. Last accessed 2 June 2022.