HTBLuVA Wiener Neustadt
Höhere Lehranstalt für Informatik

DIPLOMARBEIT

Robot Extensible Communication Toolkit

Ausgeführt im Schuljahr 2023/24 von:

Projectmanagement. Rust TCP/UDP and BLE Interface Jeremy Sztavinovszki	5CHIF
Rust gRPC Interface and interacting with in memory sqlite databases Christoph Fellner	5BHIF
C++ gRPC interface and library for communication Maximilian Dragosits	5CHIF
Python gRPC interface and library for communication Timon Koch	5CHIF

Betreuer / Betreuerin:

Harald R. Haberstroh

Wiener Neustadt, am April 2, 2023/24

Abgabevermerk: Übernommen von:

Eidestattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die im Literaturverzeichnis angegeben Quellen und Hilfsmittel verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Wiener Neustadt am April 2, 2023/24

Verfasser / Verfasserinnen:

Jeremy Sztavinovszki

Christoph Fellner

Timon Koch

Maximilian Dragosits

Contents

Acknowledgement

The authors would like to thank their respective families, the HTBLuVA and robo4you for their support throughout the process of creating this diploma thesis.

Diplomarbeit Dokumentation

Author: Dragosits Maximilian

Name der Verfassser	Sztavinovszki Jeremy, Koch Timon, Dragosits Maximilian, Fellner Christoph
Jahrgang Schuljahr	5CHIF/5BHIF 2023/2024
Thema der Diplomarbeit	Robot Extensible Communication Toolkit
Kooperationspartner	F-WuTS

Aufgabenstellung	Erstellung einer Biliothek mit der man die Kommunikation zwischen zwei idividuellen Systemen über TCP, UDP und BLE simplifiziert und erleichtert. Weiters das Testen, Optimieren und Dokumentieren dieser Bibliothek für die Nutzer von RECT.

Realisierung	Realisierung mittles Rust und einer in memory SQLite Datenbank als Backend. Zwei Frontend Bibliotheken in Python und C++ jeweils die sich mit dem Rust Backend verbinden.

Ergebnisse	Die Rust-Schnittstelle und das Python-Frontend sind entsprechend den ursprünglichen Spezifikationen implementiert. Die comm_lib konnte jedoch aus Zeitgründen und der Komplexität für dessen funktionalität nicht implementiert werden. Auch das C++-Frontend konnte nicht in einen funktionierenden Zustand gebracht werden, da es Komplikationen gab und die Zeit nicht ausreichte, diese zu korrigieren.

Möglichkeiten der Einnahme in die Arbeit	HTBLuVA Wiener Neustadt, Dr.-Eckener-Gasse 2, A 2700 Wiener Neustadt

Approbation	Prüfer	Abteilungsvorstand
(Datum, Unterschrift)	DI Harald Haberstroh	Nadja Trauner

Diploma Thesis Documentation

Authors	Sztavinovszki Jeremy, Koch Timon, Dragosits Maximilian, Fellner Christoph
Form Academic Year	5CHIF/5BHIF 2023/2024
Topic	Robot Extensible Communication Toolkit
Co-operations partners	F-WuTS

Assignment of tasks	Creation of a library to simplify and facilitate communication between two individual systems via TCP, UDP and BLE. Furthermore, the testing, optimisation and documentation of this library for the users of RECT.

Realisation	Realisation using Rust and an in memory SQLite database as backend. Two frontend libraries in Python and C++ that connect to the Rust backend.

Results	The rust interface and python frontend are implemented according to original specifications. However the comm_lib could not be implemented due to time constraints and the complexity of the intended functions. Additionally the C++ frontend could also not be brought to a functioning state due to complications and there not begin enough time to correct them.

Accessibility of diploma thesis	HTBLuVA Wiener Neustadt, Dr.-Eckener-Gasse 2, A 2700 Wiener Neustadt

Approval	Examiner	Head of Department
(Date, Sign)	DI Harald Haberstroh	Nadja Trauner

Kurzfassung

Author: Jeremy Sztavinovszki

Diese Arbeit beschreibt, wie ein Kommunikationsprotokoll und Bibliotheken für Robotik implementiert werden können. Sie untersucht verschiedene Implementierungsmöglichkeiten und versucht, mögliche Schwierigkeiten und anwendbare Optimierungen herauszufinden, um eine Methode für die Kommunikation in der Robotik bereitzustellen. Es gibt bereits mehrere Kommunikationsprotokolle und Bibliotheken für die Robotik, jedoch sind sie entweder auf eine bestimmte Sprache spezialisiert oder rechenintensiv auf, der ihre Verwendung auf Plattformen mit geringer Rechenleistung erschwert. Diese Arbeit beschreibt die Entwicklung eines erweiterbaren Kommunikationsprotokolls sowie die Implementierung von Bibliotheken dafür in verschiedenen gängigen Sprachen. Sie beschreibt zunächst den Planungsprozess und präsentiert dann die allgemeine und Netzwerkarchitektur sowie Tests und Benchmarking für die Leistung und Ressourcennutzung der verschiedenen Teile der entwickelten Kommunikationsmethode. Die Arbeit stellt ein vielseitiges und effizientes Kommunikationsprotokoll für die Robotik vor, das bestehende Einschränkungen adressiert und zu Fortschritten in diesem Bereich beiträgt.

Abstract

Author: Jeremy Sztavinovszki

This thesis describes how to implement a communication protocol and libraries for robotics. It explores the different possibilities for implementation and seeks to find out possible difficulties and applicable optimizations in order to provide a method for communication for robotics. There are already quite a few communication protocols and libraries for robotics available, however they are either specialized to a certain language, or have an overhead, that makes them difficult to use on low power platforms. This thesis describes the development of an extensible communication protocol as well as implementing libraries for it in different popular languages. It first describes the planning process and then goes on to showcase the general and network architecture, as well as testing and benchmarking the performance and resource-use for the different parts of the developed communication method. This thesis presents a versatile and efficient communication protocol for robotics, addressing existing limitations and contributing to field advancements.

Chapter 1

Introduction

1.1 Motivation

Author: Jeremy Sztavinovszki
In almost every robotics application nowadays you need some kind of communication. Wether it is a robot communicating its data to a home-base, or two robots sharing data with one another. Over the past years communication has drastically improved with new protocols and technology, such as Bluetooth Low Energy(BLE)[1]. On the other hand there are still people, that don't use any of the communication technologies described before, because most of the established frameworks seem too complex or have too many requirements regarding performance.

1.2 Goal

Author: Jeremy Sztavinovszki
This diploma thesis will explore how to write a communication framework using new technologies, such as BLE with the Transmission Control Protocol(TCP)[2] and the User Datagram Protocol (UDP)[3].
At the end of the project the framework should be useable for sending data between two KIPR Wombat-Controllers[4]. It should be capable to send well structured data, e.g. a JSON[5]-file, as well as streaming data, e.g. a series of pictures, between the robots. When using protocols, that don't make guarantees about the completeness of the received data RECT will not provide extra safeguards to make sure all sent data is also received. RECT should be able

[1] *Bluetooth Low Energy Primer*. 2024. URL: https://www.bluetooth.com/blog/introducing-the-bluetooth-low-energy-primer.

[2] Wesley Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293. URL: https://www.rfc-editor.org/info/rfc9293.

[3] *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768.

[4] *Wombat Controller*. 2023. URL: https://botball-swag.myshopify.com/collections/robot-controller/products/wombat-controller-w-battery.

[5] *What is JSON?*. 2024. URL: https://www.w3schools.com/whatis/whatis_json.asp.

to provide a stable and fast connection at reasonable distances for the selected protocol (e.g. circa 30 meters of line-of-sight, for BLE). RECT should provide libraries for the languages Python and C++ that hide the complexity of communication and provide nice abstractions simplifying its use. For this to be made possible the following requirements have to be met:

- A Rust library for communicating via TCP, BLE and UDP has to be written.
- A Rust gRPC service using the communication library has to be implemented.
- A Python and a C++ gRPC client and wrapper library have to be implemented.

1.3 History

Author: Maximilian Dragosits

1.3.1 Networks

The history of communication and coordination between seperate systems is long and extensive. Nowadays it is an essential function of almost every technological device. In the field of computing it started in the late 1950s with SAGE (Semi-Automatic Ground Enviornment), which was a network of computers and networking technology created by the United States of America military, and it allowed the transfer of radar data nation-wide.[6]
The next major step forward was the beginning of ARPANET in 1969, which served as a connection between multiple north american universities, and laid the groundwork for the modern internet.[7] In more recent times the Internet of Things is commonplace, and is used for the interchange of terabytes of data each day. But there are also many smaller private networks used either for simple processes or sensitive data, that shouldn't be accesible by a theoretical viewer outside the trusted circle.

1.3.2 Protocols

In the current network communication landscape, various protocols enable the smooth transfer of data. One of these protocols is the Transmission Control Protocol (TCP), which has become the dominant choice among Transport layer protocols due to its widespread use on the internet. The TCP/IP protocol, which was first described in RFC 675[8], is the foundation of modern network communication. It establishes a robust framework that underpins the vast expanse of the internet. Alongside TCP, the User Datagram Protocol (UDP), conceived by David Patrick Reed in 1980, is a formidable companion. These protocols, TCP and UDP, have solidified their positions as stalwarts in data transfer methodologies across the internet

[6] Paul E Haigh Thomas; Ceruzzi. *A New History of Modern Computing*. MIT Press, 2021. ISBN: 978-0262542906, p. 89.

[7] Robert Gillies James; Cailliau. *How the Web was Born: The Story of the World Wide Web*. Oxford University Press, 2000. ISBN: 978-0192862073, p. 25.

[8] *Request For Comment 675*. 1974. URL: `https://datatracker.ietf.org/doc/html/rfc675`.

and analogous networks.

The introduction of TCP/IP was a significant moment in the development of network communication. RFC 675 established the basic principles that continue to influence modern internet protocols. UDP, although different in design and use, complements TCP by offering a lightweight option for situations where real-time data transmission is more important than guaranteed delivery. TCP, UDP, and IP collectively form the Internet Protocol Suite, which governs the functioning of the internet and facilitates global connectivity. Their enduring significance in shaping network protocols underscores their indispensable role in our interconnected digital world.

1.3.3 Bluetooth

Establishing seamless communication between multiple mobile and fixed devices in close proximity, along with the formation of Personal Area Networks (PANs), has traditionally presented considerable challenges. To address this issue, Bluetooth technology emerged as a groundbreaking solution. The genesis of this innovative concept can be traced back to 1989 when Ericsson Mobile envisioned a wireless solution for headsets. Following the conceptualization phase, active development began in 1994. In 1997, IBM collaborated with Ericsson to integrate Bluetooth technology into the IBM ThinkPad, recognizing its potential. This collaboration resulted in the incorporation of Bluetooth functionality into both the ThinkPad notebook and an Ericsson mobile phone, marking a pivotal moment in the evolution of wireless connectivity.

In 1999, the first device equipped with Bluetooth functionality, a wireless headset, marked a significant leap forward for the technology. This milestone revolutionised the way devices communicate over short distances and laid the foundation for the widespread adoption of Bluetooth across a myriad of electronic devices, ranging from smartphones to smart home gadgets. The evolution of Bluetooth from its origins as a wireless headset solution to its current status as a crucial element of various technological ecosystems highlights its lasting influence in enabling effortless connectivity in both personal and professional contexts.

1.4 Project Management

Author: Jeremy Sztavinovszki

The obvious choice for working on a group project is using SCRUM[9], because it allows complicated projects to be completed in a fashion, that is able to adapt to new requirements by for example customers. One of the biggest downsides of SCRUM the members of the project saw was the high overhead of planning sprints. For that reason Kanban was chosen for managing the tasks, that need to be done in order to finish the project. Kanban, coming from japanese and meaning signboard or billboard[10] is a form of project management, that chooses to use continuous flow of tasks instead of splitting them up into sprints, like SCRUM.

[9] *What is Scrum*. 2024. URL: `https://www.atlassian.com/agile/scrum`.

[10] *What is Kanban*. 2022. URL: `https://learn.microsoft.com/en-us/devops/plan/what-is-kanban`.

It also lacks distinct roles for team members, that would be defined in SCRUM (e.g. a SCRUM-Master), and uses cycle-time[11], instead of velocity as a metric for performance.

1.4.0.1 Structure of RECT's Kanban Board

RECT's Kanban-Board is hosted on trello, because it offers easy setup and operation. The board consists of the following columns:

- A "Backlog" column for tasks, that are to be done when there is capacity.
- A "Todo" column for tasks, that need to be done soon.
- A "Design" column for tasks, that need to have a specific design (e.g. the Architecture of a TCP-Service) and for thinking about tests, that need to be written.
- A "Doing" column for tasks, that are currently being worked on. This has a limit of 4 tasks, so each member can only work on one thing.
- A "Testing" column for task, that have been roughly implemented, but still need some changes to pass all tests.
- A "Code Review" column for tasks, that have been thoroughly tested and need to be reviewed by a member in order to be merged.
- A "Done" column for tasks, that have been tested, reviewed and merged with the main branch.

1.4.1 Meetings with the projects supervisor

There were weekly meetings that were held with the diploma theses' supervisor Harald Haberstroh on fridays, where the progress of the project and upcoming tasks were discussed. The members of the team could also request feedback on their work and get help, or advice for problems they may have been having.

1.4.2 Hours spent outside of school hours

Apart from these weekly meetings the members of the team met regularily on wednesdays and thursdays after school to work on the project together and discuss problems, that came up between the weekly meetings.

1.4.3 Version Control

Version control was done with the industry standard version control system git and the project was hosted on the schools gitlab server, which also contained submodules hosted on F-WuTS and on Jeremy Sztavinovszki's personal github account.

[11] *Cycle time and Lead time.* 2021. URL: `https://community.atlassian.com/t5/Agile-articles/Lead-time-and-Cycle-Time-and-the-importance-of-Flow-in-Kanban/ba-p/1886848`.

1.5 Outline

Author: Jeremy Sztavinovszki
This *Introduction* is followed by a *Study of Literature*, where the authors describe which books were read, why they were read and what knowledge and insights were gained. After the *Study of Literature* is the *Methods* chapter. This chapter covers the technologies and languages used, why they were chosen and seeks to provide a basic understanding of the technologies. Following that chapter is the *Implementation* chapter, which covers the implementation details of each of the parts and explains design choices and tradeoffs taken. The last two chapters are the *Experiments* chapter and the *Conclusion* chapter. The *Experiments* chapter goes over the experiments performed for the thesis and their results. The *Conclusion* chapter recaps the contents of the thesis and draws a conclusion regarding the thesis' success and findings.

Chapter 2

Study of Literature

Author: Jeremy Sztavinovszki

2.1 The Rust Programming Language

Rust is a general purpose programming language, which provides important safety guarantees for systems programming while remaining as fast as, or faster than established programming languages like C or C++. The "Methods" section covers the reasoning for Rust's usage in RECT in more depth. "The Rust Programming Language"[1], as the name would suggest is the official book for learning rust. It is a great resource for learning the Rust language and is available online as a website, as well as a downloadable PDF, or a Paperback. It is divided into 20 chapters, starting off with the installation of Rust and basic concepts like variables and control flow, and finishing by explaining how to program a web server.

2.1.1 The Basics

Chapters 1 to 9 cover the most basic concepts of Rust. They go over simple I/O programming and explain common concepts such as ownership, borrowing, and lifetimes. They also go over common collections, how to define and use enumerations and structs and how to manage a project in Rust. The most important parts of these chapters are the explanations of ownership, borrowing, lifetimes, and error handling, as these are the concepts most beginners tend to struggle with when learning Rust.

2.1.2 Intermediate Concepts

Chapters 10 to 14 cover more intermediate concepts such as generics, traits, and writing unit tests. Chapter 13 goes over the functional features of Rust, explaining closures, iterators, and how to use them. These chapters are important for understanding the more advanced concepts of Rust and provide the basis for writing better and more efficient code.

[1] *The Rust Programming Language*. 2024. URL: `https://doc.rust-lang.org/stable/book/`.

2.1.3 Advanced Concepts

Chapters 15 to 20 cover the more advanced conecepts of Rust. They go over how to use unsafe code, how to use concurrency, smart-pointers, and many more advanced features of Rust. Chapter 15 covers smart-pointers and why they are useful for memory safety and concurrency. Chapter 16 goes over how to use concurrency in Rust covering the basics of how to start a new thread and then going over how to use message-passing and shared-state concurrency. Chapter 17 covers the object oriented programming features of Rust, explaining how to use traits and how to use them to implement object oriented programming. Chapter 18 goes over patterns and how to use them as a way to control the flow of a program and where they should be used. Chapter 19 goes over the advanced features of Rust, such as macros, unsafe Rust, and advanced traits and types. Chapter 20 is the final chapter of the book providing a kind of final project by first building a single threaded web server, then making it multi-threaded and finally going over graceful shutdown and cleanup of the resources the server used.

2.1.4 Takeaways

"The Rust Programming Language" is a great resource for learning Rust. It provides a great introduction to the language and then gradually goes over more advanced topics. RECT uses many of the concepts explained in the Rust book such as concurrency and error handling, which makes the knowledge gained from the book very useful.

2.2 Getting Started with Bluetooth Low Energy

"Getting Started with Bluetooth Low Energy"[2] is a book giving deep insight into the inner workings of Bluetooth Low Energy. The first four chapters are especially useful to this work and are summed up in the following sections.

2.2.1 Chapter 1. Introduction

The introduction explains the history of Bluetooth Low Energy from its start as Wibree[3], over the adoption by the Bluetooth Special Interest Group, to the status at the time of the writing of the book. The chapter then goes on to explain the difference to Bluetooth Classic. Key Limitations like Data througput and Operation range as well as possible network topologies and the difference of Protocols versus profiles are also explained.

2.2.2 Chapter 2. Protocol Basics

The second chapter goes over the different layers and protocols of Bluetooth Low Energy explaining the importance and use for each of the layers and also covering the Generic Attribute

[2] Akiba Kevin Townsend Carles Cufi et al. *Getting Started with Bluetooth Low Energy*. O'Reilly. ISBN: 9781491949511.

[3] *Is Wibree going to rival Bluetooth?* 2024. URL: `https://electronics.howstuffworks.com/wibree.htm`.

Profile and the Generic Access Profile. It provides explanations for the frequency hopping and modulation done in the physical layer and many other specifics in the other layers.

2.2.3 Chapter 3. The Generic Access Profile

The third chapter explains concepts like roles, such as Broadcaster and Observer, and explains how each of them work. It gives explanations of the modules and procedures included in the General Access Profile. After that section the Security Manager and General Security constructs, such as Address Types, Authentication and Security Modes are covered. Lastly the GAP service and the format in which data is sent over advertisements is covered.

2.2.4 Chapter 4. The Generic Attribute Profile

This chapter goes on to explain the roles defined in the Generall Attribute Profile and covers UUIDs. Then the most important concept, Attributes, are explained. This chapter along with the documentation for the library this work uses for BLE solidified, that L2CAP is more suitable for this work.

2.2.5 The Remaining Chapters

The remaining covers go over specific hardware platforms, debugging tools, application design tools and platform specific mobile programming and were not as important as the previous four chapters. Therefore they are not covered here.

2.2.6 Takeaways

The book provides great explanations of many difficult topics through text as well as visual aids. The main takeaways for this work are, that L2CAP is better suited for its purposes and better general knowledge of the BLE and its stack.

Chapter 3

Methodology

3.1 Wombat

Author: Timon Koch
The KIPR Wombat[1] 3.1 is the Controller currently used in the Botball competition. It consists of a raspberry pi 3b, a display and a casing, designed to include all necessery elements needed for the competition, such as ports for the various sensors, motors and servos allowed to be used and a cutout for the battery needed to power the used parts. Through the use of a raspberry pi 3b it is capable of wireless LAN and Bluetooth Low Energy. The Wombat uses the KISS (KIPR Instructional Software System) Web IDE developed by the KISS Institute for Practical Robotics. The current version supports ANSI C[2], Python and C++. It can be used with a web browser and WiFi on any operating system.

3.2 Python

Author: Timon Koch
Python is a powerful high-level, general purpose, object-oriented progrmming language with a design philosophy that emphasizes the readability of the written code through the use of significant indentation.[3]
It was created by Dutch programmer Guido van Rossum as a successor to ABC[4] and was first released in 1991 with the capability of exception handling. Features such as list comprehensions, reference counting, Unicode support and cycle-detecting garbage collections were introduced with Python 2.0 on 16 October 2000. After this release Python became a community-driven, open source project. Due to a lack of backward compability, the switch from 2.7 to the 3.0 update, released in 2008, caused a big controversy. Folowing this most programs written in 2.7 had to be rewritten to support 3.0 update. Python has gained enormous

[1] *Wombat Controller*.
[2] *ANSI C Standard*. URL: `https://web.archive.org/web/20161223125339/http://flash-gordon.me.uk/ansi.c.txt`.
[3] *Python Homepage*. URL: `https://www.python.org`.
[4] *ABC Programming Language*. URL: `https://homepages.cwi.nl/~steven/abc/`.

Figure 3.1: KIPR Wombat Controller

popularity since the first release in the early 1990s and is used in a wide range of applications, ranging from web development to machine learning. The most important feature of the programming language Python is its support for mulitple programming paradigms, including object-oriented, imperative and functional programming. It was designed to be highly extensible via modules, rather than building all of itfunctionality into its core.

3.3 C++

Author: Maximilian Dragosits

C++ is a precompiled programming language that combines low-level memory management with support for object-oriented, generic, and functional programming paradigms. It was designed by Danish computer scientist Bjarne Stroustrup with a focus on efficiency, perfor-

mance, and flexibility.[5] C++ has become widely used in various domains, including desktop applications, servers, video games, and digital equipment for space exploration.

It was standardized by the International Organization for Standardization (ISO) in 1998 as ISO/IEC 14882:1998 and has since evolved into a powerful programming language. The popularity of C++ has led to the creation of numerous libraries and frameworks, such as Catch2[6] and Doxygen[7]. These frameworks are essential to the project's development and functionality.

C++ was chosen as one of the two frontend languages for this project due to its impressive speed and efficiency, as well as the vast ecosystem of external frameworks and libraries already available. These resources are essential in developing the RECT library, improving the project's capabilities and accelerating the development process. The decision to use C++ was a strategic choice, aligning with the language's strengths and the abundance of tools and resources it offers.

3.4 Rust

Author: Jeremy Sztavinovszki

Rust is a general purpose multi paradigm programming language used in many fields ranging from embedded programming to web development. Although it is a relatively young language, having released its version 1.0 on May 15th 2015, it has seen great adoption from developers and has a big community. The language tries to be as fast as possibly, while still remaining memory-safe, which it achieves using its borrow checker. Even though it is possible to write unsafe code in Rust, that is not checked by the borrow checker, it is custom to keep the unsafe parts as small as possible. Rust has a great ecosystem driven by the Rust Foundation and the Rust community. There are many tools, such as cargo, or rust-gdb, that provide great developer experience. Right now there is now standardized async-runtime, so you normally use runtimes like tokio, async-std, or smol for programming asynchronously.

3.4.1 Documentation

In order to provide an easy understanding of a Rust crate and its functionalities, a documentation is needed. Documentation in Rust works by using comments in the code, which are then parsed by the `cargo doc` command. This command generates a folder called `target/doc` in the project, which contains the documentation in HTML format. Comments which go into the documentation are written in a specific way. The following is an example of this:

[5] *Lecture: The Essence of C++*. 2014. URL: `https://web.archive.org/web/20150428003608/https://www.youtube.com/watch?v=86xWVb4XIyE`.

[6] *Catch2 framework Github*. 2024. URL: `https://github.com/catchorg/Catch2`.

[7] *Doxygen Homepage*. 2024. URL: `https://www.doxygen.nl`.

```rust
//! This is a module/crate-level documentation comment.

/// This function does something
/// # Arguments
/// * `x` - The first parameter
/// * `y` - The second parameter
/// # Returns
/// The sum of `x` and `y`
fn add(x: i32, y: i32) -> i32 {
  x + y
}
```

Listing 3.1: Example of a Documentation Comment

As shown in the example, there are crate-, or module-level and general documentation comments. The crate-level comments are used to describe a whole crate, or module and provide an overview of the functionality of the crate or module. The general comments are used to describe functions, structs, enums, traits, etc. and provide an explanation of the functionality of the specific item. The documentation for the CommLib can be found in

3.4.2 Cargo

Author: Christoph Fellner

Cargo[8] is a build system and package manager for Rust, providing developers with a robust toolset for managing Rust projects. A rust package consists of a `Cargo.toml` file, which contains metadata about the package, and a `src` folder, which contains the source code as rust-files (`.rs`). It is an integral part of the Rust ecosystem and plays a crucial role in managing dependencies, building projects, and facilitating a smooth development workflow. Here's an overview of Cargo's main functionalities:

- 1. Dependency Management: Cargo manages project dependencies by automatically fetching and incorporating external libraries or crates. Developers specify dependencies in the Cargo.toml file, and Cargo ensures the correct versions are used.
- 2. Project Structure: Cargo establishes conventions for organizing Rust projects, defining a standardized layout for directories and files. This consistency helps developers understand and contribute to projects more easily.
- 3. Building and Compilation: Cargo handles the complexities of building and compiling Rust projects. Developers can use simple commands such as `'cargo build'` to compile the code and `'cargo run'` to execute the compiled binary.
- 4. Testing: Cargo supports the integration of unit tests into Rust projects. Developers can use the `'cargo test'` command to run test suites, ensuring the reliability and correctness of their code.

[8] *The Cargo Book*. 2024. URL: `https://doc.rust-lang.org/cargo/`.

- 5. Documentation Generation: Cargo can generate documentation for Rust projects, making it easier for developers to create and maintain documentation for their code. The `'cargo doc'` command generates and hosts documentation, and it can be published on platforms like `docs.rs`.

- 6. Project Initialization: Cargo provides a convenient way to initialize new Rust projects with the `'cargo new'` command. This command sets up a new project with the necessary directory structure and initial files.

- 7. Publishing: Cargo facilitates the process of publishing Rust packages (crates) to the official package registry, crates.io. This makes it straightforward for developers to share their libraries and projects with the wider Rust community.

In essence, Cargo streamlines various aspects of Rust development, offering a standardized and efficient workflow for managing dependencies, building projects, testing code, generating documentation, and publishing packages. Its integration with the Rust toolchain contributes to the language's reputation for being developer-friendly and conducive to building robust and reliable systems.

3.5 gRPC

Author: Maximilian Dragosits
gRPC[9] is an open-source framework that facilitates Remote Procedure Calls (RPC) across diverse environments. It is versatile and can be used in a broad spectrum of use cases, proving invaluable in establishing robust service-to-service connections. gRPC plays a pivotal role in the development of microservices and libraries. This framework is available in 11 different programming languages and provides developers with a flexible and accessible toolset for creating efficient and scalable communication channels.

gRPC's efficiency is central to its straightforward service definition and generation structure, which streamlines the integration process. This simplicity allows developers to focus on the core aspects of their projects, enhancing productivity and code maintainability. Furthermore, gRPC includes pluggable features such as authentication, load balancing, tracing, and health checking. These features provide developers with fine-grained control over service communication, ensuring robust and secure interactions within distributed systems.

In the context of the current project, gRPC plays a pivotal role. Its capability to seamlessly connect clients to backend services is particularly crucial. This feature enables efficient communication between Python and C++ frontends and the Rust backend, creating a cohesive and interoperable system. The project uses gRPC to achieve high communication efficiency, allowing for seamless data exchange between components and improving the overall performance and reliability of the system architecture.

[9] *gRPC Homepage*. 2024. URL: `https://www.grpc.io`.

3.6 Protocol Buffers

Author: Maximilian Dragosits

Protocol Buffers offer a platform-neutral solution for serializing structured data, similar to formats such as XML, JSON, or YAML. They provide a versatile solution that seamlessly interfaces with automatically generated source code across an array of programming languages, catering to developers' preferences. Noteworthy among the supported languages are Java, Kotlin, Python, and various C-based languages, underscoring the broad applicability of Protocol Buffers in diverse development ecosystems.

Protocol Buffers enable developers to efficiently manage and exchange structured data across different platforms and systems. They provide a unified approach to serialization. This approach promotes interoperability, enabling developers to easily incorporate serialized data into their projects, thereby enhancing the efficiency and maintainability of their codebases.

An example of a Protocol Buffer file illustrates the elegance and conciseness inherent in this technology, demonstrating its role in facilitating efficient data interchange.

```
1   syntax = "proto3";
2   package msg;
3
4
5   message From {
6     string conn_name = 1;
7     string topic = 2;
8   }
9
10
11  message To {
12    string conn_name = 1;
13    string topic = 2;
14  }
15
16
17  message Msg {
18    bytes data = 1;
19    oneof fromto {
20      From f = 2;
21      To t = 3;
22    }
23  }
```

Listing 3.2: Excerpt taken from the ProtoBuf files of this project

As can be seen in this example the first part of any .proto file is the definition of the protobuf language version. Either *proto2* or *proto3*. Next the package, which sets the namespace for the later generated code. In this case it will be *msg*. After that any other .proto file can be imported, so that it can be used for the definition of types within the file. Then it is possible to define any amount of the following types and many others not used by this project:

1. **message:** Defines a special data structure that houses multiple variables of potentialy different data types, which can then be used in other enums or services.

2. **enum:** Defines an enum which acts like the equivalent type of structure in other programming languages. This can then be used in other parts of the .proto file.

3. **service:** Defines a Remote Procedure Call (RPC) system. The generated code for this will include service interfaces and stubs to be implemented and used for RPCs.

3.6.1 Protofile message definition

Message types in proto3 are relatively simple to define.

```
1    message message_name {
2        field_type field_name = number;
3     }
```

Listing 3.3: Message definition syntax in Protofiles

First the *message* keyword is used to signify that the following is a declaration for a message type. Then a freely chooseable *message_name* is used as the name for the later resulting message structure. After that any number of fields can be defined within the curly brackets. The *field_type* can be one of multiple supported data types, which includes but is not limited to double, flout, integer, boolean, string as well as bytes. After defining an appropriate *field_name* this format requires the assignment of a number between 1 and 536,870,911 to each field in a message. This is required in order to identify the field after encoding.
There are also three other modifiers, that can be applied to fields:

1. **optional:** If a field with this modifier does not have its value explicitly set later it will instead return a default value. It also possible to check if this it has been set.

2. **repeated:** A field with this modifier can be repeated any number of times within the message and the order of the repetition will be saved.

3. **map:** A field with this modifier acts like a key/value pair with the definition syntax being like that of a C++ map.

Another way of defining fields, that can have a multitude or a currently unkown type, is to use either the *any* or the *oneof* types. The *any* type is then later resolved by Protocol Buffers internal reflection code. *Oneof* is then automatically later defined as one of the given data types within curly brackets placed after the *field_name* is given.

3.6.2 Protofile enum definition

Enums share a lot of the same traits as message types in terms of the defintion syntax.

```
enum enum_name {
    constant_value = number;
}
```

Listing 3.4: Enum definition syntax in Protofiles

Similarly to messages the enum is given an *enum_name* and then any number of *constant_value*s can be defined. All of these constants need an associated value in order to function properly and the first of those needs to have 0 as its number, so that the enum has a default value in case fields have the *optional* modifier. In order to bind multiple *constant_value*s to the same *number* the *allow_alias* option must be set to true. This is done by inserting this line into the enum before any definition of *constant_value*s:

```
option allow_alias = true;
```

Once an enum is defined then it can be used in other parts of the Protofile, as seen in this example:

```
enum Success {
    Ok = 0;
}

enum SendError {
    NoSuchConnection = 0;
    SendFailed = 1;
}

message SendResponse {
    oneof result {
        Success s = 1;
        SendError err = 2;
    }
}
```

Listing 3.5: Excerpt taken from the ProtoBuf files of this project

3.6.3 Protofile service definition

Services allow the easy generation of service interfaces and stubs to then be used by RPC implementations.

```
service service_name{
    rpc rpc_name(message_type) returns (message_type) {}
    rpc rpc_name(message_type) returns (stream message_type) {}
}
```

Listing 3.6: Example of service definition in Protofiles

A service is defined with a *service_name* and after that any number of inidvidual methods.

In order to define the methods first the keyword *rpc* must be used. Then a name for the method is given through *rpc_name* and a parameter for the *message_type* that this method accepts. And then a *message_type* is defined as the return value of the RPC. A stream of a particluar *message_type* can be defined by putting the keyword *stream* before the type.
An example of this would be the SubListen service from this project:

```
1    service SubListen{
2        rpc listen(ListenRequest) returns (ListenResult) {}
3        rpc subscribe(ListenRequest) returns (stream ListenResult) {}
4    }
```

Listing 3.7: Example of service definition in Protofiles

3.7 Nix

Author: Jeremy Sztavinovszki
Nix is one of a couple of things depending on the context. It is either a configuration language, a package manager, an operating system, or a build system. That may seem a bit confusing, but the next section will cover each of these contexts for the sake of clearing up some of the confusion, that may arise from this statement.

3.7.1 History

Nix was first conceived and made a reality in 2003 as a research project by Eelco Dolstra. At first it was just a package manager, that could be run on any distro, but in 2007 it became its own full blown Linux distribution with many other additions to the Nix eco system, such as Hydra[10], a continuous intergration tool, and nixops, a deployment tool for nixos deploying NixOS in a network/cloud[11]. At the time of writing Nix has gotten another big addition in the form of flakes. Flakes are a way of declaratively building anything ranging from Nix packages, over development environments, to system configurations. When a flake is build for the first time it pulls in all of its inputs and writes their commit hashes to a flake.lock file. Through this mechanism of noting the exact commit each input of a flake it is possible to use nix flakes for reproducible builds.

3.7.2 The Language

The language was the first part of Nix, that was implemented and it is arguably the most important part of Nix. Nix is a declarative functional programming language, that is used for defining packages, build processes and configurations for a host of things ranging from reproducible development environments to IT-Infrastructure. Taking an example for the syntax from the official nixos wiki site the language looks something like this:

[10] *Hydra Nixos Wiki Page*. 2023. URL: `https://nixos.wiki/wiki/Hydra`.
[11] *NixOps Github Page*. 2023. URL: `https://github.com/NixOS/nixops`.

```
#1
let
    a = 1;
    b = 2;
in a+b
# result 3

#2
let
    square = x: x*x;
    a = 12;
in square a
#result 144

#3
let
    add = x: y: x+y;
in add 1 2

#result 3

#4
let
    square = {x ? 2}: x*x;
    a = 12;
in square{x=square {};}

#result 16
```

Listing 3.8: Simple Examples of the Nix Language

In examining the provided code, a discernible pattern emerges. The syntax `let <variables> in <statement>` serves as a method for defining values within a forthcoming block and it is quite reminicent of the Haskell programming language. However, this construct encompasses more than mere value assignment. Consideration of each distinct block sheds light on its functionality. Block No. 1 initializes two variables, 'a' and 'b', then performs an addition operation on them. It's noteworthy that if an additional variable 'c' were defined but left unused, it would remain unevaluated due to Nix's lazy evaluation mechanism. Moving to Block No. 2, it defines a function named 'square'. This function accepts a parameter 'x', as indicated by the notation `x: x*x`. Parameter declarations in this context follow the structure `<parameter name>: <statement>`. However, when multiple variables are required, as demonstrated in Block No. 3, the syntax adapts to `<parameter name 1>:<parameter name 2>: <statement>`. This paradigm bears resemblance to lambda calculus[12]. Block No. 4 showcases optional parameters. This feature is denoted by
`{<parameter name> ? <default value>}: <statement>`.

[12] *Lambda Calculus*. 2023. URL: `https://plato.stanford.edu/entries/lambda-calculus/`.

3.7.3 The Package Manager

Nix, now a package manager, is a cross-platform package manager, that claims to have solved a problem called dependency hell[13], by keeping track of which package needs which dependencies. If a package is no longer needed it can automatically be garbage collected. Packages are installed to a directory called the nix store and have a unique hash, which is generated by combining some factors, like dependencies, versions and so on. When defining a package you use the nix programming language and lazy functional programming to declare how to build it, what you need to build it and what files to install through a format called derivations.

3.7.4 The Build System

The following example showcases how to use the Nix language to define a flake containing a package, which can be built and installed on any system running the Nix package manager. The specific example builds the latex files of which this thesis is comprised into a pdf file using a build tool called latexmk.

[13] *Dependency Hell Techopedia.* 2019. URL: `https://www.techopedia.com/definition/27701/dependency-hell`.

```
{
  description = "A flake to build the RECT-Diploma-Thesis";

  inputs = {
    nixpkgs.url ="github:nixos/nixpkgs/nixos-23.05";
  };

  outputs = {self, nixpkgs}:
  let
    system = "x86_64-linux";
    pkgs = nixpkgs.legacyPackages.${system};
  in {
    packages.${system}.default = pkgs.stdenv.mkDerivation rec {
      name = "RECT-Diploma-Thesis";

      src = ./.;

      buildInputs = [
        pkgs.texlive.combined.scheme-full
      ];

      buildPhase = ''
          latexmk -pdf main.tex
      '';

      installPhase = ''
          mkdir -p $out/pdf
          cp main.pdf $out/pdf
      '';
    };
  };
}
```

Listing 3.9: The nix flake, that builds this diploma thesis

The code shown above is a nix flake. It defines the nixpkgs repository as an input and the result of the build process, that is taking place in the mkDerivation block as an output. mkDerivation is a function which takes a name, pname, version, src, buildInputs, buildPhase, installPhase, builder and shellHook as inputs and produces a package which is built in the standard environment (stdenv)[14]. The built derivation (or output of this flake) is then asigned a hash and stored in the nix store on the machine that built it. An example of this would be the following path
`/nix/store/dnx26izplgv46dwg548whh9kj5iz4vvx-RECT-Diploma-Thesis`.

3.7.4.1 Nixpkgs

Of course the defined packages need to be stored somewhere. This is where the nixpkgs repository[15] on github comes in handy. It is a collection of over 80000 packages according to

[14] *mkDerivation Documentation*. 2023. URL: `https://blog.ielliott.io/nix-docs/mkDerivation.html`.
[15] **nixpkgs˙repo**.

repology[16]. The community can contribute their own definitions, or updates to the repository if they found something to be out of date, or missing. Of course when installing a package it is not built from scratch every time, like on source based distros. Instead nixpkgs caches builds of the most popular packages, which then just have to be downloaded onto the users machines.

3.7.5 The Operating System

NixOS is built upon the Nix package manager. It is an independent Linux distribution, which means it is not based on any other Linux distribution like for example Debian, or Arch Linux. What is really special about NixOS is, that the whole operating system with services, programs and all of the needed configurations can be built from one central file, which is written in the Nix Programming Language. With this file stored as a backup a machine running NixOS could be up and running again in no time after a failure and through the usage of Nix Flakes it is possible to reinstall the system exactly the way it was before.

3.8 Bluetooth Low Energy

Author: Jeremy Sztavinovszki
Bluetooth Low Energy (BLE) is a Low-Cost, Low-Bandwidth, Low-Energy technology, that works by transmitting data over the air using a slice of the frequencies available in the Industrial, Scientific and Medical Band[17]. It was introduced in the 4th version of the Bluetooth Version, after being developed by Nokia under the name Wibree and being adopted by the Bluetooth Special Interest Group (SIG). BLE at the time of its release made use of many innovative technologies, for example Frequency Hopping Spread Spectrum[18], which is used to avoid collisions when sending data. A combination of being innovative and marketing lead to BLE seeing a great adoption rate after its initial release. Over the years Bluetooth Low Energy has seen many improvements, such as a Bluetooth Low Energy device being able to take on multiple roles simultaneously in the network.

3.8.1 BLE Layers

3.8.1.1 Protocol vs. Profile

The BLE specification has two important concepts, which it has clearly separated since its inception. These concepts are profiles and protocols. Hereby protocol is used to describe the basic parts upon which the BLE stack builds and can be thought of the horizontal layers of the stack. Profile refers to vertical slices of the stack, that are used for specific use-cases when developing with BLE. Examples of a profile are.

[16] *Nixpkgs Repology*. 2023. URL: https://repology.org/repository/nix_stable_23_05.
[17] *Industrial, Scientific and Medical Band*. 2023. URL: https://www.pcmag.com/encyclopedia/term/ism-band.
[18] *Frequency Hopping Spread Spectrum*. 2023. URL: https://www.analog.com/en/design-center/glossary/fhss.html.

- The Glucose Profile[19], which is used to securely transmit measurement data from e.g. insulin pumps.

- The Find Me Profile[20], which allows devices to find one another

These use specific parts of the BLE protocol to achieve a specific task.

3.8.1.2 Physical Layer PHY

The Physical Layer (PHY) establishes the foundation for BLE communication by defining radio transmission properties such as modulation schemes, frequency bands, and power levels. It optimizes communication for BLE devices by employing techniques like frequency-hopping spread spectrum (FHSS) to mitigate interference and enhance reliability. Advancements in the PHY layer aim to improve data rates, extend range, and enhance spectral efficiency while maintaining low power consumption, which is crucial for the versatility of BLE applications.

3.8.1.3 Link Layer (LL)

The Link Layer(LL) is responsible for managing essential functions such as connection establishment, maintenance, and data transmission between BLE devices. It handles advertising, scanning, and packet acknowledgment, optimizing power usage during data exchange. The efficient handling of packet formatting, acknowledgment, and error handling ensures a robust and reliable communication link, which is pivotal for BLE's energy-efficient operations.

3.8.1.4 Host Controller Interface (HCI)

The Host Controller Interface (HCI) serves as the intermediary between the host (application processor) and Bluetooth hardware on the host side. It defines protocols and commands for seamless data exchange, enabling efficient control of Bluetooth functionalities. The details of the host side are also included.

3.8.1.4.1 Host Side

The HCI on the host side enables communication between the HCI driver and the application processor. It offers a standardised interface that defines the command structures and protocols used by the application to interact with the Bluetooth hardware. This abstraction allows for platform-independent communication and streamlines application development. The Controller Side should also be considered.

[19] *Glucose Profile*. 2024. URL: `https://www.bluetooth.com/specifications/specs/glucose-profile-1-0-1/`.

[20] *Find Me Profile*. 2024. URL: `https://www.bluetooth.com/specifications/specs/find-me-profile-1-0/`.

3.8.1.4.2 Controller Side

The HCI on the controller side translates commands from the host into hardware-specific operations for the Bluetooth controller. It facilitates communication between the host and the Bluetooth hardware, ensuring accurate execution of the required actions. This layer is responsible for managing data transfer between the host and controller, optimizing the transmission process

3.8.1.5 Logical Link Controll and Adaptation Protocol (L2CAP)

The Logical Link Control and Adaptation Protocol (L2CAP) efficiently multiplexes higher-layer protocols over BLE connections. It segments and reassembles data packets, optimizing data transmission efficiency while accommodating diverse application requirements. The role of protocol multiplexing and fragmentation is to ensure efficient data exchange while maintaining BLE's low-energy characteristics.

3.8.1.6 Attribute Protocol and Generic Attribute Profile

The Attribute Protocol (ATT) and Generic Attribute Profile (GATT) play critical roles in defining how data is exchanged and accessed between devices. ATT outlines rules for attribute information exchange, while GATT specifies the structure and mechanisms for accessing these attributes. The structured approach provided by data representation and interaction ensures interoperability across various applications and device types.

3.8.1.7 Security Manager (SM)

The Security Manager (SM) operates within the BLE protocol stack and is responsible for establishing secure connections and managing security-related aspects between BLE devices. It manages processes such as pairing, encryption, and authentication to ensure the confidentiality and integrity of data transmitted over BLE connections. This is critical for safeguarding sensitive information.

3.8.1.8 General Access Profile

The Generic Access Profile (GAP) is a fundamental layer in Bluetooth Low Energy (BLE) that is responsible for device discovery, connection setup, and addressing within the network. It defines device roles and manages how devices interact. GAP also handles device addressing, ensuring unique identification, and manages visibility, pairing, and power modes. GAP promotes interoperability between devices by standardizing essential functions. This ensures smooth communication across diverse BLE devices, regardless of their manufacturers or applications. The layer's importance lies in its ability to provide stability and reliability in BLE networks. GAP defines the following roles for a device to take on:

- Broadcaster. The Broadcaster role is especially useful for applications, that regularily transmit data. It uses Advertising Packets instead of Connection Packets to send data in order for any listening device to be able to read the data without having to establish a conneciton.

- Observer. The Observer role is the counterpart to the Broadcaster. This role could be used for the base station of an alarm system, that receives broadcasts from sensors around the house for detecting intrusions.
- Central. The Central role is used to establish connections with peripheral devices. It always initiates the connections and is in essence the role, that allows devices onto the network. This is usually a smartphone and could be used for establishing a connection to some speakers to play music.
- Peripheral. The Peripheral role sends advertising packets in order for central devices to be able to find it and initiate connections.

3.8.2 Network Topologies

Using the profiles described above, there is a wide range of network topologies that developers can choose from, depending on their objectives. Some of the following topologies excel at saving power, while others are great for scalability and redundancy.

3.8.2.1 Star Topology

In a star topology, a central device (the 'Central') communicates with multiple peripheral devices. This configuration is similar to a hub-and-spoke model, where the central device manages and controls communication with the peripherals. For example, in a smart home scenario, a smartphone (acting as the central) connects to various peripherals such as smart locks, lights or sensors. The central device collects data from these peripherals and can also coordinate their functions. This architecture is simple and provides centralised control, but it relies heavily on the availability and range limitations of the central device.

3.8.2.2 Mesh Topology

BLE mesh networks allow devices to communicate with each other, forming a decentralised network without a central control point. Each device, or 'node', can communicate with nearby nodes, extending the coverage and redundancy of the network. For example, in a smart lighting system, each bulb can communicate with neighbouring bulbs to relay commands or data, ensuring robustness even if a node fails. Mesh networks are scalable, resilient and suitable for applications that require extensive coverage, such as smart buildings or large-scale sensor networks.

3.8.2.3 Hub and Spoke Topology

This architecture involves a central 'hub' device that communicates with multiple 'spoke' devices, each of which communicates only with the central hub. Think of a smart home setup where a central controller (e.g. a smart hub or gateway) manages and interacts with various sensors, smart appliances or actuators placed throughout the home. Each peripheral device communicates only with the central hub, streamlining communication and enabling centralised management.

3.8.2.4 Cluster Tree Topology

The cluster tree topology forms a hierarchical structure that organises devices into clusters, each cluster having a central node. These central nodes can communicate with other central nodes or higher level devices, creating a structured network. In industrial applications, devices within a particular area or zone can communicate with a local coordinator, which then communicates with higher level coordinators or a central system. This hierarchy enables efficient communication in large deployments, providing local and global control.

3.8.2.5 Hybrid Architectures

BLE applications often use hybrid architectures that combine multiple topologies to meet different needs. For example, a smart building might use a mesh network for inter-floor sensor communication, a star network for room-level control (with each room having a central controller), and a hub-and-spoke network for centralised management and integration of various systems within the building.

3.9 WiFi

Author: Timon Koch
WiFi, which stands for Wireless Fidelity, is a technological advancement that has significantly transformed how people and businesses access the internet and communicate wirelessly within a specific range. Wireless networking technology has had a significant impact on connectivity, providing users with greater flexibility and convenience when accessing information, conducting business, and engaging in communication activities. WiFi has become an integral part of modern society, permeating various aspects of daily life, from homes and workplaces to public spaces and transportation systems.

3.9.1 History

The development of WiFi can be traced back to the 1990s when the Institute of Electrical and Electronics Engineers (IEEE)[21] established the 802.11[22] committee to create standards for wireless local area networking (WLAN). In 1997, the initial version of the 802.11 standard was introduced, which provided data transmission speeds of up to 2 megabits per second (Mbps). Over the years, WiFi standards have evolved, including 802.11a, 802.11b, 802.11g, and 802.11n, bringing significant enhancements in speed, range, and reliability. By the early 2000s, WiFi had become widely adopted in both residential and commercial environments, catalysing the wireless revolution and reshaping the landscape of connectivity.

3.9.2 Usage

WiFi technology is widely used in various settings, such as homes, offices, educational institutions, hospitality establishments, and transportation hubs. It allows users to connect

[21] *IEEE*. URL: `https://www.ieee.org/`.
[22] *IEEE 802.11*. URL: `https://standards.ieee.org/ieee/802.11/7028/`.

multiple devices, including smartphones, tablets, laptops, desktop computers, smart TVs, gaming consoles, and IoT devices, to the internet without the need for physical connections. Wireless networks are commonly established using routers or access points, which serve as intermediaries that facilitate communication between connected devices and the internet. This extensive connectivity enables individuals to work remotely, collaborate effectively, access digital content, stream multimedia, and stay connected while on the go, thereby enhancing both personal and professional endeavors.

3.9.3 Function

WiFi operates by transmitting data over radio frequencies within designated bands, primarily the 2.4 GHz and 5 GHz spectrums. Devices with WiFi capabilities use modulation techniques to encode data into radio signals, which are then transmitted and received by nearby routers or access points. The communication process typically employs protocols, such as CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance)[23] , to manage data transmissions and prevent potential signal interference or collisions. WiFi routers are an important tool for managing the flow of data between connected devices and the internet. They utilize advanced technologies such as beamforming and multiple-input multiple-output (MIMO)[24] to optimize signal propagation, coverage, and bandwidth utilization.

3.9.4 Advancements

In recent years, there have been remarkable advancements in WiFi technology driven by the increasing demand for enhanced performance, reliability, and scalability. The introduction of the 802.11ac standard, also known as WiFi 5[25], has ushered in a new era of gigabit-speed wireless connectivity, providing improved throughput and spectral efficiency. Subsequent iterations, such as the 802.11ax standard (also known as WiFi 6)[26], have introduced innovative features like orthogonal frequency-division multiple access (OFDMA)[27] and multi-user multiple input multiple output (MU-MIMO)[28]. These features enable more efficient spectrum utilization and accommodate an ever-expanding array of connected devices. These advancements have enabled WiFi to become a part of ultra-high-speed networking, making it possible to seamlessly stream, play games with low latency, experience immersive virtual reality, and deploy robust IoT solutions.
Looking to the future, it is worth noting that the development of WiFi technology is ongoing. New advancements, such as WiFi 6E, are set to utilize the previously unused 6 gigahertz (GHz) spectrum, which will provide increased bandwidth and reduced interference. Furthermore, it is believed that advancements in mesh networking structures, network optimization driven by artificial intelligence, and secure wireless protocols may enhance the capabilities and

[23] *CSMA/CA*. URL: https://ieeexplore.ieee.org/document/4067995.

[24] *Multiple Input Multiple Output (MIMO)*. URL: https://www.sciencedirect.com/topics/engineering/multiple-input-multiple-output.

[25] *WiFi 5*. URL: https://www.elektronik-kompendium.de/sites/net/1602101.htm.

[26] *WiFi 6*. URL: https://www.arubanetworks.com/de/faq/was-ist-wlan-6.

[27] *OFDMA*. URL: https://www.computerweekly.com/de/definition/OFDMA-Orthogonal-Frequency-Division-Multiple-Access.

[28] *MU-MIMO*. URL: https://www.elektronik-kompendium.de/sites/net/2112061.htm.

resilience of WiFi networks. This could potentially usher in an era of ubiquitous, ultra-fast connectivity that caters to the diverse needs of modern society.

3.10 Libraries

3.10.1 Catch2

Author: Maximilian Dragosits

Catch2[29] is a unit testing framework specifically designed for C++. It seamlessly integrates with C++ code and offers a testing environment that aligns closely with the overall structure of C++ code. Its ability to harmonize with the natural syntax of functions and boolean expressions is noteworthy. Catch2 not only facilitates unit testing but also incorporates micro-benchmarking capabilities, enabling developers to analyze performance in detail.

In the context of our project, Catch2 is an ideal choice for shaping the C++ frontend. Its integration into the development process ensures that unit tests become an integral part of the codebase, fostering a culture of code reliability and maintainability. In addition, the integration of micro-benchmarking in Catch2 is consistent with our project's objectives, enabling thorough performance evaluations. This feature is essential in promoting optimizations, as benchmarks provide valuable metrics for measuring the speed and efficiency of implemented methods.

By using Catch2, our project benefits from a versatile testing framework that not only validates the functionality of the C++ frontend but also enables developers to make informed decisions about enhancements and optimizations. Catch2's seamless integration and benchmarking capabilities make it a valuable asset in ensuring the robustness, performance, and efficiency of the implemented methods within the project.

3.10.1.1 Unit Tests

Unit Tests in Catch2 are defined very similarly as normal functions in C++. This example, pulled from the Github repository of Catch2[30], shows the simplicity of this framework and its integration into codebases.

[29] *Catch2 framework Github.*
[30] *Catch2 framework Github.*

```
#include <catch2/catch_test_macros.hpp>

#include <cstdint>

uint32_t factorial( uint32_t number ) {
    return number <= 1 ? number : factorial(number-1) * number;
}

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( factorial( 1) == 1 );
    REQUIRE( factorial( 2) == 2 );
    REQUIRE( factorial( 3) == 6 );
    REQUIRE( factorial(10) == 3'628'800 );
}
```

Listing 3.10: Example of Catch2 test definition

First the relevant Catch2 headers are included and then a function with a return value is defined. In this case it is the function factorial. This function will be executed by Catch2 during the testing process. Then a test case is a macro defined as:

```
TEST_CASE(string testname, string tags) {...test...}
```

The argument *testname* is a arbitrary name given to the unit test. The behavior of the macro can be modified by inputting values into the *tags* field. In the case of the example above the only given tag is the name of the function to be tested. After this the *TEST_CASE* macro has a curly brackets-enclosed body in which the logic of the test can be defined.
This requires the use of other specific macros included in the Catch2 framework. For example:

```
REQUIRE( function(value) == expected_value );
CHECK( function(value) == expected_value );
```

The two macros described above, REQUIRE and CHECK, operate in a similar way. They both execute the given *function* with the provided *value* or *values* and then assert if the returned data equals true or false according to the provided boolean operator. If it does then it was successful and the rest of the *TEST_CASE* is executed. The difference between REQUIRE and CHECK is however that if a REQUIRE macro fails it throws an exception and the unit test is stopped from executing the remainder of code inside it.

3.10.1.2 Micro-benchmarks

The benchmarking macros in Catch2 are defined similarly to how unit tests are. Benchmarking in itself is a useful practice, that provides a way to measure the performance and speed of a particular function.

```cpp
#include <catch2/catch_test_macros.hpp>
#include <catch2/benchmark/catch_benchmark.hpp>

#include <cstdint>

uint64_t fibonacci(uint64_t number) {
    return number < 2 ? number : fibonacci(number - 1) + fibonacci(number - 2);
}

TEST_CASE("Benchmark Fibonacci", "[!benchmark]") {
    REQUIRE(fibonacci(5) == 5);

    REQUIRE(fibonacci(20) == 6'765);
    BENCHMARK("fibonacci 20") {
        return fibonacci(20);
    };

    REQUIRE(fibonacci(25) == 75'025);
    BENCHMARK("fibonacci 25") {
        return fibonacci(25);
    };
}
```

Listing 3.11: Example of Catch2 benchmark creation

In the example above the unit test macros and benchmark macros from Catch2 are first included in order to be used later. The function to be benchmarked is then defined. After that the TEST_CASE macro is used combined with the [benchmark] tag in order to turn this unit test into a benchmark. In the actual body of the test the function is first tested weather or not it actually works as intended before any benchmarks are done. This is done with the REQUIRE macro, since it throws an exception if the assertion fails, preventing the rest of the benchmark from executing unnecessarily. If all the tests before the benchmarks pass then the actual BENCHMARK macros are executed.

```cpp
BENCHMARK(string name) {
    ... benchmark ...
};
```

As part of the BENCHMARK macro a arbitrary name is given to it, which is then later used as a identifier for the specific benchmark during the execution of the test . Then the actual logic of the benchmark is defined within the curly brackets giving a lot of freedom in how a certain benchmark is executed. Adding a return statement within the benchmark will ensure that the compiler perserves the original order of the statements within the benchmark.

After this is run a summary is automatically output within the command line window. This includes multiple data points that pertain to the execution speed of the tested function:

1. **Samples:** How often the code within the curly brackets of the BENCHMARK macro is repeated in order to build a dataset to calculate the mean execution time.

2. **Iterations:** The amount of iterations performed.

3. **Estimated run time:** The estimated amount of time the code within the benchmark will take to run. Mesured in milliseconds.

4. **Mean/low mean/high mean run time:** The mean time it will take for the code to run as well as the low mean and high mean for this in nanoseconds.

5. **Standard deviation:** The standard deviation from the mean time in nanoseconds,

3.10.2 Doxygen

Author: Maximilian Dragosits

Doxygen[31] is a versatile C++ framework that automates the generation of documentation for C++ code. It can also be used for other programming languages such as PHP, Java, Python, IDL, and Fortran. Doxygen extracts information directly from annotated source code, making the documentation process more efficient and accessible for developers.

Doxygen is notable for its flexibility in delivering documentation. The generated documentation can be accessed through a browser in the form of an HTML document or through a LaTeX document, providing options tailored to developers' preferences. Additionally, Doxygen can derive structure overviews even from non-annotated sources, facilitating a holistic understanding of the codebase. The project's architecture can be visualized more effectively by displaying entity relations through insightful graphs.

A quick overview of a Doxygen implementation shows a well-organized framework that streamlines the documentation process. This implementation improves code comprehension and promotes a collaborative development environment by providing a centralized repository of project documentation. The 3.2 shows how Doxygen's integration simplifies and enhances the documentation workflow, contributing to the clarity and accessibility of the project.

[31] *Doxygen Homepage*.

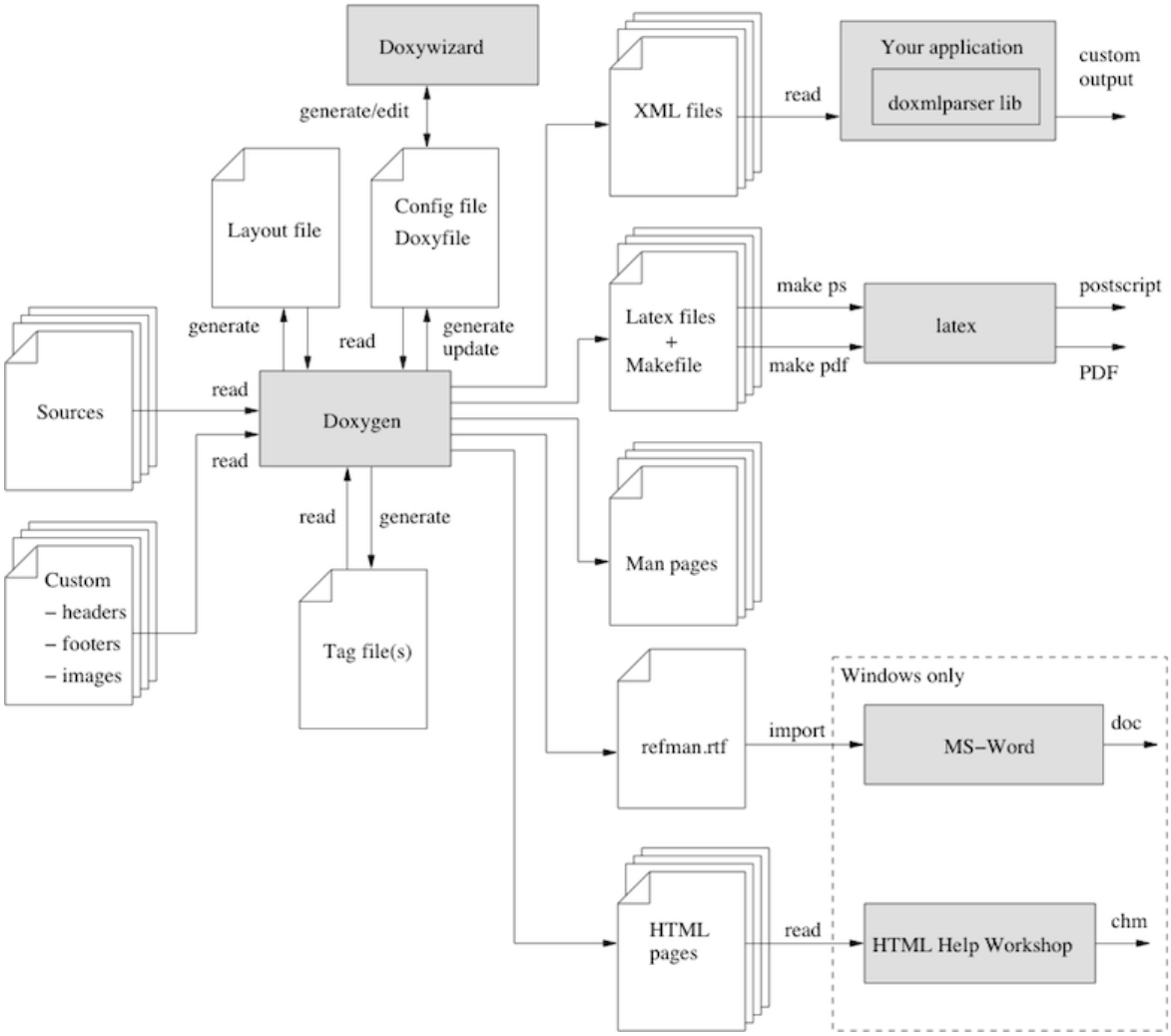

Figure 3.2: Doxygen Infoflow diagramm[32]

First a Doxygen config file, also called a *Doxyfile* is generated in a project using the *Doxywizard* and can be edited using this tool afterwards. This file is later read and updated during the generation process from *Doxygen*. The layout files and tag files are also generated and used by *Doxygen* to facilitate the creation of the documentation.

Sources are the files with the actual source code, that will be part of the documentation, in them. These files are read alongside any other additional *Custom* files. For example the headers for source code files. There are of course many other files that can be included in order to refine the result further.

Finally *Doxygen* uses all this to generate documentation in the specified formats: XML, LaTex, Man, refman or HTML.

3.10.2.1 Doxygen Configuration

The first step in creating documentation with Doxygen is a configuration file. This can be automatically generated by the *Doxywizard* with this command:

```
1    doxygen -g <config-file>
```

In this example *config-file* is a stand-in for the name of the generated configuration file. There are many tags within this config file that change how the end result will look and what sources should be used or ignored. For example the INPUT tag specifies the location that *Doxygen* will search for code and EXCLUDE forbids it to use files contained within the given directories. After editing the file using a text editor or Doxywizard this command can be used to generate the documentation.

```
1    doxygen <config-file>
```

Doxygen normally requires the source code to be annotated in order to generate documentation, but a rough version can be made by setting the tag EXTRACT_ALL to YES. This will extract even the non-annotated classes and functions from the source code.

3.10.2.2 Doxygen Annotation

If the EXTRACT_ALL tag is not enabled then the classes, functions and members within the source files need to be annotated in order to be picked up by *Doxygen* and turned into a documented section. There are many ways of signaling to *Doxygen*, that you want it to produce documentation from a source. In general there are two ways to do this:

1. **Documentation within source:** In this way of annotation special comment blocks are inserted before and after parts of the source code. This will cause the text within this special block to be displayed alongside the class, function, or member within the documentation.

2. **Documentation outside source:** In this way of annotation the special comment blocks are written within a separate file and then connected to the sources by way of a reference. With this method references to the source file need to be written within this distinct file.

In *Doxygen* special comment blocks[33] are similar to C++ comments, but with extra characters in order to be picked up during generation. There are many different ways to define special comment blocks:

Javadoc and Qt style: This way a special block can be definied just like a multi line comment in C++ with a special character after the first *.

[33] *Doxygen Manual section on comment blocks*. 2024. URL: `https://www.doxygen.nl/manual/docblocks.html#specialblock`.

```
/**
 * ...  text ...
 */
```

Or:

```
/*!
 * ...  text ...
 */
```

C++ comment style: This style is similar to a C++ single line comment with an added / or ! at the begining.

```
///
/// ... text ...
///
```

Or:

```
//!
//! ... text ...
//!
```

Visible style: This is a more clearly visible version of the previous two styles.

```
/**********************************
 * ...  text ...
**********************************/
```

Or:

```
//////////////////////////////////////
/// ... text ...
//////////////////////////////////////
```

In documentation outside of the source additional commands are required. At the start of the document file a special comment block needs to have the name of the file to be documented referenced by putting a backslash or an @ symbol before the filename in the first line of the block. After that any number of other special comment blocks can be defined with the first line in each of them being used to reference the desired class, function, member, struct, etc. This is done by first using the equivalent structural commands and then inserting the definition statement of the chosen object. This can be seen in this example:

```
1    /*! \fn int open(const char *pathname,int flags)
2        \brief Opens a file descriptor.
3
4        \param pathname The name of the descriptor.
5        \param flags Opening flags.
6    */
```

In this case *fn* is the structural command, which tells *Doxygen* what type the object is.

The other structural commands that can be seen in this example can be used in both documentation inside sources and outside sources. Using the *brief* command is used to give a short description to the object and *param* in order to signify and describe the parameters of this function. An example of the end result of function annotation can look like is shown by this figure.

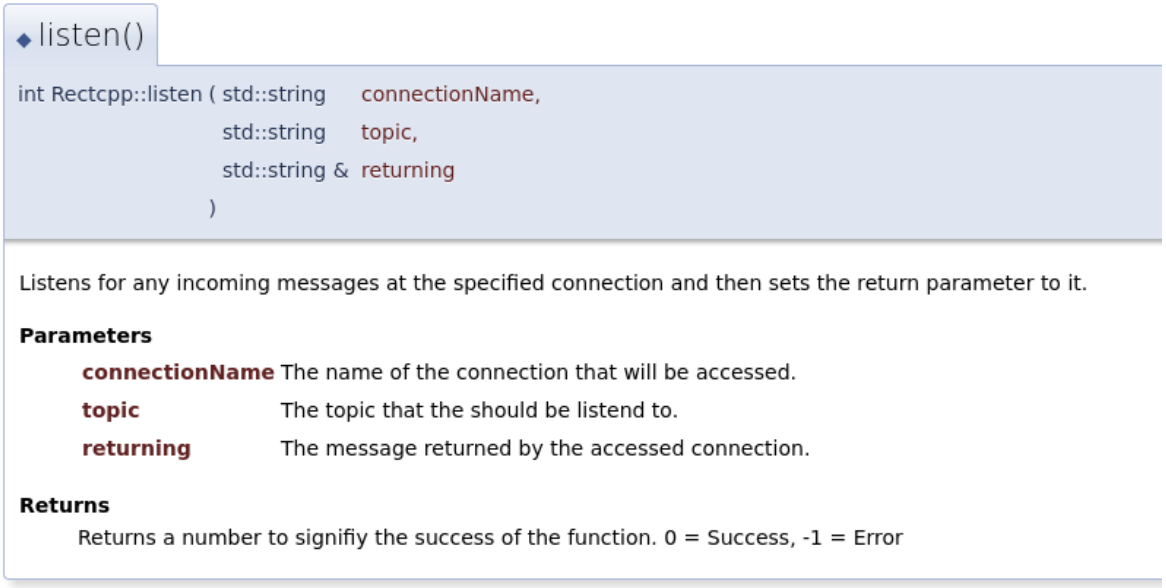

Figure 3.3: Example of function documented using Doxygen

3.10.2.3 Doxygen Parsing

During the documentation generation process, Doxygen parses special comment blocks and translates them into various output formats to create comprehensive documentation. The parser performs several key processes, culminating in the creation of documentation with Doxygen[34].

[34] *Doxygen Usage Manual*. 2024. URL: `https://www.doxygen.nl/manual/index.html`.

1 One of these processes is the conversion of Markdown writing within comment blocks into HTML, ensuring compatibility with web-based documentation platforms. This conversion improves the presentation of textual content, making it more accessible and visually appealing.

2 Special commands embedded within the comment blocks are executed, allowing developers to include custom instructions or annotations. This feature empowers developers to tailor the documentation to specific project requirements, fostering flexibility and customization.

3 Blank lines are handled appropriately. Paragraph separators are strategically used instead of blank lines to enhance the readability and organization of the generated documentation. This helps developers to better understand the information presented.

4 Doxygen generates links to other sections of the documentation based on special comment blocks. The interlinking mechanism enables smooth navigation within the documentation, improving the user experience and promoting a cohesive understanding of the codebase.

5 In situations where the output format is LaTeX, Doxygen converts HTML elements within the comment blocks into LaTeX commands. This transition ensures consistency and compatibility when generating LaTeX documents. It allows developers to choose the format that best suits their documentation needs.

3.10.3 Pytest

Author: Timon Koch
Pytest is a testing framework for the Python programming language. It enables the creation of both simple unit tests and complex functional tests. Key features include test discovery, parameterized testing, and plugin support, making it a popular choice among Python developers.

3.10.3.1 Test Discovery

The framework has the capability to automatically discover and execute test cases through a process known as 'test discovery'. By default, files that begin with 'test_ ' or end with '_test' are recognized as test files, although it is possible to explicitly mention other filenames. It is necessary for test method names to begin with 'test', as Test functions are identified by their names.
It is possible to either execute a specific test by indicating the corresponding identifier or execute all tests simultaneously. In the latter case, pytest will automatically detect and execute all files and functions that contain tests.
By utilizing Pytest's plugin system, it becomes feasible to personalize and explore tests using supplementary mechanisms. The plugins available provide integration with diverse test frameworks, enabling the exclusion of particular tests or designating them for customized behaviors.

3.10.3.2 Fixtures

Fixtures are a useful feature that can help modularise and share setup code across multiple tests, resulting in improved code by increasing readability, maintainability and reusability. They are defined as functions identified by the '@pytest.fixture' decorator. Different scopes may be specified to determine how long the fixture should last.

3.10.3.3 Unit Tests

Pytest provides compatibility with the 'unittest' testing framework. It is possible to discover and run both Pytest-style test functions and unittest test cases in the same project. This can facilitate a seamless transition between the two frameworks while leveraging the additional features provided by Pytest.

3.10.4 Pydoc

Author: Timon Koch
Pydoc is a Python library and command-line tool that generates documentation from Python modules. It extracts docstrings from modules, functions, classes, and methods, and presents them in various formats, including plain text, HTML, and man pages. Pydoc facilitates access to Python code documentation, which can aid in understanding the usage of modules, functions, and classes within a codebase. It is particularly useful for exploring Python library and module documentation without having to leave the command line or an IDE. Additionally, Pydoc can be extended and customized to support additional documentation formats or features, which can be beneficial for developers seeking to optimize their workflow.

3.10.5 Rusqlite

Author: Christoph Fellner
Rusqlite[35] emerges as a pivotal library, facilitating the seamless integration of SQLite capabilities into Rust code, akin to its analog, rust-postgres. This library provides a sophisticated interface, streamlining the execution of diverse database operations within the Rust programming paradigm. From the orchestration of table creation and data insertion to the intricacies of data querying, rusqlite empowers developers with a versatile toolset for database manipulation directly within their Rust codebase.
The decision to adopt rusqlite in our specific use case is underpinned by a meticulous consideration of three critical factors:

1. **Portability:** SQLite's commendable attribute of cross-platform compatibility across diverse operating systems assumes paramount importance in our context at RECT. Given the heterogeneous nature of small controllers operating within our system, the ability to seamlessly deploy SQLite databases across different platforms becomes imperative for maintaining operational uniformity and efficiency.

[35] *SQLite in Rust - rusqlite*. 2023. URL: `https://www.linkedin.com/pulse/sqlite-rust-rusqlite-amit-nadiger`.

2. **Configuration:** In stark contrast to database systems that necessitate intricate setup procedures, SQLite obviates the need for complex configurations. The simplicity inherent in working with SQLite aligns seamlessly with our operational requirements. The nominal requirement for a limited number of tables further amplifies the pragmatic appeal of SQLite. In contrast to configuring a database on each controller, the minimal setup overhead consolidates SQLite as the optimal choice for our streamlined operational demands.

3. **Local:** SQLite distinguishes itself by eschewing the necessity for an external server or intricate installations. Its self-contained package encapsulates all requisite features within a local environment. This intrinsic localization resonates with the operational ethos at RECT, aligning with the need for an efficient and autonomous database solution without the encumbrance of external dependencies.

In our specific instantiation, rusqlite is strategically employed for the persistent storage of the config.json file, a repository of vital data concerning available connections. The judicious selection of rusqlite in this context is grounded in its inherent advantages. The utilization of rusqlite markedly enhances the expediency and security of data access compared to traditional file-based retrieval mechanisms. This methodological choice contributes not only to the enhanced performance of our system but also underscores the commitment to preserving the integrity and reliability of configuration data, pivotal to the seamless functionality of our application within the broader scientific and technological landscape.

3.10.6 Serde

Author: Christoph Fellner

Serde[36] emerges as a pivotal tool in the Rust programming ecosystem, dedicated to the efficient and generic serialization and deserialization (**ser**ialization/**de**serialization) of data structures. Offering a robust foundation for these operations, Serde plays a crucial role in optimizing the handling of data structures within the Rust programming paradigm. For a comprehensive overview of Serde, interested readers can refer to the detailed documentation available at https://serde.rs/.

The salient features of Serde become particularly evident in its capability to facilitate the seamless deserialization of JSON files with efficiency and simplicity. This functionality proves invaluable in our context, where the utilization of data from the config.json file is an integral part of our program. Serde streamlines this process, enabling us to incorporate the data effortlessly into our program with just a few lines of code.

By leveraging Serde, we engage in a streamlined deserialization process wherein the data from the JSON file is efficiently transformed into a custom Rust structure. This bespoke structure, tailored to our specific needs, enables us to harness the deserialized data seamlessly within our program. The utilization of Serde thus transcends mere deserialization, providing

[36] *Serde Overview*. 2023. URL: `https://serde.rs/`.

us with a powerful and adaptable mechanism to work with the data in a meaningful and efficient manner.

In essence, the incorporation of Serde into our workflow is not merely a technical choice but a strategic one, driven by the need for optimized data handling and seamless integration of external data sources. Through Serde, our program gains a robust and flexible capability to decode JSON files, thereby enhancing the overall efficiency and maintainability of our Rust-based application.

3.10.7 Tokio

Author: Christoph Fellner
Tokio[37] serves as a pivotal asynchronous runtime for Rust, addressing the intricacies of asynchronous code execution in the language. In Rust, asynchronous code does not inherently execute independently; rather, it necessitates the use of a runtime like Tokio to effectively function. For an in-depth exploration of Tokio and its capabilities, readers are encouraged to delve into the detailed tutorial available at https://tokio.rs/tokio/tutorial.

The selection of Tokio as our asynchronous runtime is underpinned by several compelling reasons. Chief among them is Tokio's status as the preeminent and widely adopted runtime for asynchronous Rust code. Its widespread usage within the Rust community attests to its robustness and reliability in facilitating asynchronous programming paradigms.

Moreover, Tokio's popularity is bolstered by the abundance of tutorials and educational resources available, making it approachable for developers seeking to harness the power of asynchronous programming in Rust. The simplicity and accessibility of Tokio's interface contribute to its widespread adoption, enabling developers to swiftly grasp and implement asynchronous patterns in their codebase.

One of the key advantages of Tokio lies in its ability to execute multi-threaded asynchronous code safely. This capability is crucial in scenarios where parallelism and concurrency are essential, such as in web servers or applications handling numerous concurrent tasks. Tokio's adept handling of multi-threaded asynchronous code ensures the efficient execution of tasks without compromising safety or introducing race conditions.

In summary, our decision to embrace Tokio as the asynchronous runtime for our Rust project is grounded in its status as the leading runtime in the Rust ecosystem, coupled with its user-friendly design and robust support for multi-threaded async code execution. This strategic choice positions us to harness the full potential of asynchronous programming in Rust, ensuring the responsiveness and scalability of our application.

```
#[tokio::test(flavor = "multi_thread", worker_threads = 2)]
async fn client_test(){
```

[37] *Tokio Tutorial*. 2023. URL: `https://tokio.rs/tokio/tutorial`.

```
    let ser = test_server::run_server();
    println!("Server started");
    let cl = test_client::client_test();
    println!("Client started");

    tokio::select! {
        biased;
        _ = ser => panic!("server returned first"),
        _ = cl => (),
    }
}
```

Using Tokio makes it possible to run multiple async functions at the same time. In the example above we start a server and a client and then wait for the first one to finish. This is done using the tokio::select! macro. The macro takes a list of futures and waits for the first one to finish. In our case we want to wait for the client to finish first, so we panic if the server finishes first. If the client finishes first we just return. Any occoring Errors are cathed and returned as Result. The two functions `run_server` and `client_test` are both async functions. The server function is a simple echo server, that waits for a message from the client and then sends it back. The client function sends a message containing an url and a vote to the server and then waits for the response. Server and Client are connected via a gRPC connection.

3.10.8 Tokio Rusqlite

Author: Christoph Fellner

Tokio Rusqlite[38] stands as an innovative library at the intersection of Tokio and Rusqlite, synergizing their functionalities to enable asynchronous database interactions. This library represents a strategic amalgamation, providing developers with the capability to leverage Rusqlite seamlessly within an asynchronous programming paradigm facilitated by Tokio.

The amalgamation of Tokio and Rusqlite in Tokio Rusqlite addresses the growing demand for asynchronous database operations in Rust. By integrating Tokio's asynchronous runtime with Rusqlite's capabilities, this library empowers developers to perform database operations without blocking the execution of other tasks. This is particularly advantageous in scenarios where responsiveness and concurrency are paramount.

Key features of Tokio Rusqlite include the ability to execute Rusqlite operations asynchronously, allowing for non-blocking interactions with SQLite databases. Asynchronous operations are vital in applications that require efficient handling of concurrent tasks, such as web servers, where multiple requests may be processed simultaneously.

[38] *Crate tokio rusqlite*. 2023. URL: `https://docs.rs/tokio-rusqlite/latest/tokio_rusqlite/`.

The seamless integration of Tokio Rusqlite into our development stack augments the versatility of Rusqlite by enabling asynchronous database operations. This is especially valuable in scenarios where responsiveness and scalability are critical factors. Leveraging Tokio Rusqlite in our project equips us with a robust solution for handling database interactions in an asynchronous manner, aligning with contemporary trends in Rust programming and distributed system architectures.

3.10.9 Tonic

Author: Christoph Fellner
Tonic[39], a Rust library that embodies the principles of gRPC, distinguishes itself with a keen focus on high performance, interoperability, and flexibility. Its seamless integration with the asynchronous paradigm, particularly its compatibility with Tokio, positions Tonic as a versatile tool for developing efficient and responsive distributed systems.

The synergy between Tonic and Tokio is a noteworthy feature, as Tonic is crafted to align seamlessly with Tokio's asynchronous model. This compatibility not only enhances the performance of asynchronous Rust code but also simplifies the integration of Tonic into existing Tokio-based projects. Leveraging async/await functionality, Tonic facilitates a synchronous coding style in an asynchronous environment, making it a natural fit for projects utilizing Tokio.

A distinctive aspect of Tonic lies in its support for gRPC, a high-performance remote procedure call (RPC) framework. Tonic employs Protocol Buffers to describe interfaces, providing a language-agnostic and efficient means of defining the structure of data. This approach not only enhances interoperability but also allows for the generation of necessary Rust code based on the defined Protocol Buffers, automating a significant portion of the development process.

The utilization of Tonic in our project streamlines the definition of interfaces by leveraging Protocol Buffers. This enables us to articulate our interface specifications in a clear and concise manner, and Tonic then takes care of the Rust code generation, reducing manual effort and potential errors.

In essence, Tonic serves as a powerful tool in our technology stack, providing a performant and flexible implementation of gRPC for Rust. Its compatibility with Tokio, support for async/await, and seamless integration with Protocol Buffers contribute to the development of robust and efficient distributed systems. By choosing Tonic, we aim to leverage its capabilities to enhance the performance, interoperability, and maintainability of our Rust-based projects.

The example for Tokio above uses the Tonic library to convert the following Protocol-Buffer-file into Rust code, in order to use it in the client and server functions as Service.

[39] *Crate tonic*. 2023. URL: `https://docs.rs/tonic/latest/tonic/`.

```
syntax = "proto3";
package voting;

service Voting {
    rpc Vote (VotingRequest) returns (VotingResponse);
}

message VotingRequest {
    string url = 1;

    enum Vote {
        UP = 0;
        DOWN = 1;
    }
    Vote vote = 2;
}

message VotingResponse {
    string confirmation = 1;
}
```

In order to translate the Protocol Buffer into Rust code we use the `include_proto!` function from the Tonic library. This function takes the path to the Protocol-Buffer-file and generates the Rust code for the Service.

3.11 Docker

Author: Christoph Fellner

Docker[40] serves as an indispensable tool in modern software development, offering a containerization solution that facilitates the efficient execution of diverse applications within isolated environments known as containers. These containers, encapsulating applications and their dependencies, operate independently, enabling the concurrent execution of multiple containers on a single host system. The versatility of Docker extends from lightweight services like echo servers to complex web applications, making it a versatile choice for a spectrum of development and deployment scenarios.

One of Docker's compelling use cases is the encapsulation of databases within containers. This capability enables the creation of portable and reproducible database environments, fostering ease of testing, development, and even benchmarking. In our specific use case, Docker proved instrumental in benchmarking different databases, allowing us to evaluate and identify the most suitable database for our application's requirements.

Docker's open-source nature and compatibility across major operating systems contribute to its widespread adoption. Its availability on diverse platforms empowers developers to create consistent environments, irrespective of the underlying infrastructure. This feature is par-

[40] *Docker Overview*. 2023. URL: `https://docs.docker.com/get-started/overview/`.

ticularly advantageous in scenarios where multiple applications need to coexist on the same machine, as Docker mitigates compatibility concerns and ensures the isolation of applications from the local infrastructure.

The ability to isolate applications from the host system's infrastructure is a key feature of Docker. This isolation not only enhances compatibility but also simplifies the deployment process. Developers can confidently run multiple applications on a single machine without the fear of conflicts or compatibility issues, streamlining the development and testing phases.

In summary, Docker's role in our project extends beyond conventional application development and deployment; it serves as a pivotal tool for benchmarking databases. Its containerization approach, coupled with the ability to create reproducible environments, positions Docker as a valuable asset in our quest to identify the optimal database solution for our specific use case. The open-source nature and cross-platform compatibility further solidify Docker's standing as a cornerstone technology in contemporary software development and deployment practices.

Chapter 4

Implementation

4.1 General Architecture

Author: Jeremy Sztavinovszki
The general Architecture of RECT looks like 4.1. This is a high level overview of the RECT stack. It shows, that multiple client programs using the libraries provided for RECT send data through gRPC to the *RECT-Backend*, which then handles sending the data to peers. The *RECT-Backend* may also receive data for a Client process, which it will then send to the corresponding client program through gRPC. The following sections will go into more detail about the different parts of the stack and how they were implemented.
4.2 shows how data is sent through the different protocols in RECT. In this diagram the *RECT-Interface* handles the inter process communication with the client programs. The *RECT-Interface* then interacts with components the comm_lib, mainly the ConnectionManager, which is responsible for handling the connections to the database and the LogicHandler. The LogicHandler is responsible for handling the logic for sending and receiving the different types of messages. It receives the messages from the ListenManager, which listens on the different interfaces for incoming messages. The LogicHandler is also directly responsible for sending the messages through the different interfaces. The following section covers these

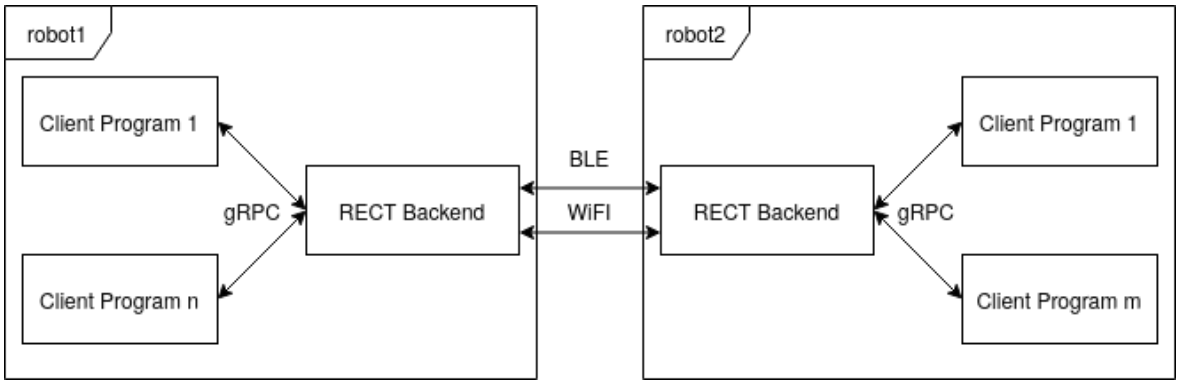

Figure 4.1: RECT's Architecture

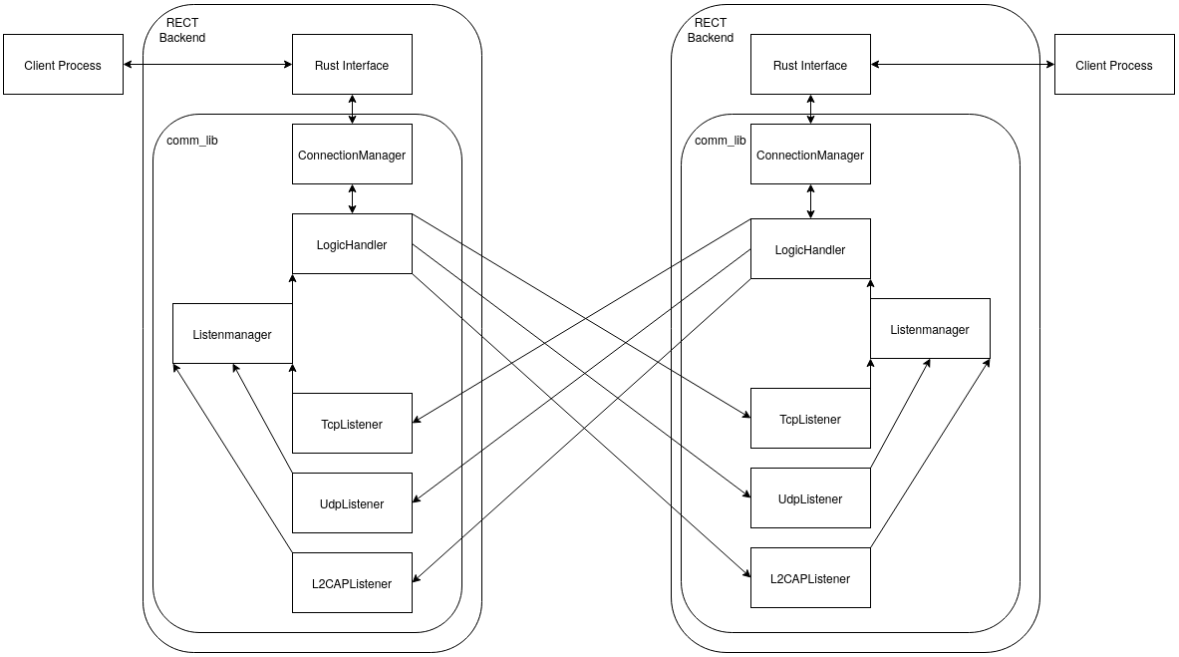

Figure 4.2: The Flow of Data in RECT

components in more detail.

4.2 CommLib

Author: Jeremy Sztavinovszki
The Communication Library, or CommLib for short is the part of the RECT stack, that handles all of the communication between the hosts over traditional protocols, like TCP, UDP and BLE. This requires it to be especially performant. For this reason a highly concurrent design was chosen, which uses the tokio library for asynchronous IO and the bluer library for Bluetooth communication. All of the communication between the components of the CommLib is done through tokio's channels, which means each component can be started and then run parallel to the other components.

4.2.1 Setting up the Library

The first steps of setting up the library are more or less the same as in any other rust project. First the project is initialized with `cargo new --lib <rust-name>`. This creates a new folder with the name specified in `<library-name>` and generates some files like Cargo.toml and src/main.rs. After this step is done the needed libraries for RECT are added to the project through `cargo add <dependency-name> -F <dependency-name>/<feature-name>` these dependencies are the pulled and built by cargo (Rust's build tool) upon the initial build of the project. The first iteration of the project then had the following dependencies:

- tokio
- bluer
- anyhow

All in all the commands used to generate the CommLib project and install all dependencies looked like this:

```
1    cargo new --lib CommLib && cd CommLib
2    cargo add tokio bluer anyhow enum_dispatch -F tokio/full,bluer/full
3    cargo build
```

Listing 4.1: Setup Commands for CommLib

Any application using the CommLib needs to interact with this object in order to be able to use the communication methods provided by the library. To achieve this the CommLib provides the following functionality:

- Setting a connection to a database, which is used to read the connections configured by the clients.
- Receiving simple messages, streams, requests and responses through any of the configured connections.
- Sending simple messages, streams, requests and responses through any of the configured connections.
- Initiating an update of the connections, which checks changes in the configuration found in the database and the connections, that are currently active in the ConnectionManager.

In the following sections the implementation of these functionalities will be covered in more detail.

4.2.2 Datatypes for Handling Communication

In order to have a simple and easy to use interface for the CommLib, the library uses a set of different types to handle the different kinds of event, which can occur during communication. These types are as follows:

- SimpleMessage
- SimpleMessage acknowledgement
- Stream subscribe
- Stream unsubscribe
- Stream data

- Request
- Request acknowledgement
- Response
- Response acknowledgement

These types are implemented as structs in the CommLib which are part of an enum called Message. This enum uses the enum_dispatch library in order to be able to run a set of functions defined in a trait on each of the different structs. Each of the scructs embodies a different use which will be explained in the following sections.

4.2.2.1 SimpleMessage

The simple message contains a message_id, which is a 64 bit unsigned integer, a message, which is a vector, or array of bytes, and a topic which is a String. This message is used to send a message which does not require a response. The message_id is used to identify the message for acknowledgement and to ensure that there are different acknowledgement hashes for two messages which contain the same data. The topic is used to identify the message on the receiving end.

4.2.2.2 SimpleMessage acknowledgement

The simple message acknowledgement is used to acknowledge the receipt of a simple message. It contains an acknowledgement hash, which is computed from the message_id and the data of the message. This hash is then used to verify, that the message was received and insure, that the data in the message is correct.

4.2.2.3 Stream subscribe

The stream subscribe message is quite minimalistic, only containing a topic, which is a String. This message is used to subscribe to a topic on the receiving end. Upon receiving this message the recipient will include the sender in the list of subscribers for a given topic.

4.2.2.4 Stream unsubscribe

Stream unsubscribe is similar to stream subscribe, but is used to unsubscribe from a topic. So instead of adding the sender to the list of subscribers the recipient will remove the sender from the list of subscribers for a given topic.

4.2.2.5 Stream data

The stream data message is used to send data to a subscriber. This message contains a topic, as well as a message, which is a vector of bytes.

4.2.2.6 Request

The request message is used to send a request to a recipient. It contains a request_id, which is a 64 bit unsigned integer, a topic, and data, which is a vector of bytes. The request_id is used to identify the request for acknowledgement and to ensure that there are different acknowledgements for two requests which contain the same data for the same topic. The topic is used to identify the request on the receiving end and relay it to the correct recipient process.

4.2.2.7 Request acknowledgement

Similar to the simple message acknowledgement, the request acknowledgement is used to acknowledge the receipt of a request. It contains an acknowledgement hash, which is computed from the request_id, topic and the data of the request. This hash is then used to verify, that the request was received and insure, that the data in the request is correct.

4.2.2.8 Response

The response message is used to send a response to a request. It contains a response_id, which is a 64 bit unsigned integer, a topic, and data, which is a vector of bytes. The response_id is used to identify the response for acknowledgment and to relay the response to the correct requesting process. The topic is used to identify the response on the receiving end.

4.2.2.9 Response acknowledgement

The response acknowledgement is used to acknowledge the receipt of a response. It is the exact same as the request acknowledgement, except for the name, which is changed in order to differentiate between the two.

4.2.3 Logic for Handling Communication

Datatypes alone however are not enough to implement the handling of the communication. The logic needed to handle the communication is implemented in a couple of different software components on the CommLib. These components are highly concurrent and use many of the features of the tokio library in order to allow the parallel handling of many different tasks. The compoments are:

- ConnectionManager. This is the component handling the database and relaying the commands received over gRPC to the other components of the CommLib.
- LogicHandler. This is the component handling the logic for the different types of messages. It is responsible for keeping track of subscribers to streams and for packing the messages into the correct format for binary serialization and sending them over the corresponding protocol. It receives all messages received on any of the available interfaces (TCP, UDP, L2CAP) through the ListenManager. It also handles the logic for acknowledgements and any future optimizations concerning communication.

- ListenManager. This is the central component for receiving data over the different protocols. It is responsible for setting up listeneres on all interfaces and for relaying any data received to the LogicHandler in the form of Message structs.
- The Listeners. These are the components responsible for listening on their respective interfaces and sending the data to the ListenManager.

The following sections will cover the implementation of these components in more detail.

4.2.3.1 The Listeners

The listeners are a fairly simple component. They are passed a Notify[1] to receive the notification when to stop and a sender for an unbounded channel[2], which is used to send the data received on the interface with the corresponding address the data was received from. The listener can then be started in a new tokio task and will run asynchronously until it receives the notification to stop. Normally there are three listeners running at the same time, one for each of the protocols TCP, UDP and L2CAP.

4.2.3.2 The ListenManager

The ListenManager is a bit more complex than the listeners. It is a singleton and is responsible for setting up the listeners on the correct ports and with the correct addresses (and service uuids in the case of L2CAP). It is also responsible for converting the received data from the listeners from binary into the corresponding Message structs and adding information on what interface the data was received from. This is then sent to the LogicHandler through an unbounded channel.

4.2.3.3 The LogicHandler

The LogicHandler is yet another singleton. It contains a bit more data and logic than the before mentioned components, because of its central role in the CommLib. It holds and populates a hashmap of published topics and subscribers to said topics, which are stored in a Vector of address strings and types of the interfaces the subscribers are connected to. It provides a few public Senders and Receivers in order to be let other components interact with it. There are a few wrapper functions for these Senders and Receivers, which are used in order to reduce the complexity of interaction with the LogicHandler and reduce duplicate code.

4.2.3.4 The ConnectionManager

The ConnectionManager is the last of the singleton components in the CommLib. It is responsible for interacting with the RECT database and for resolving the topics, or connection names received through gRPC to the corresponding addresses and interfaces. It also relays

[1] *Rust Tokio Notify*. 2024. URL: `https://docs.rs/tokio/latest/tokio/sync/struct.Notify.html`.

[2] *Rust Tokio Unbounded Channel*. 2024. URL: `https://docs.rs/tokio/latest/tokio/sync/mpsc/fn.unbounded_channel.html`.

data received through interfaces back to gRPC, as well as calling the correct functions in the LogicHandler.

4.2.4 Facing Problems with Concurrency

Although the comm_lib's structure is designed to be highly concurrent, there are problems in the implementation, which couldn't be resolved in time for the comm_lib to be finished. The main problem is concurrently handling the sending and receiving of messages, while also having to wait for the acknowledgements for previous messages. This means, that while the Listeners for the interfaces, the ConnectionManager and most of the LogicHandler have been implemented, complex tasks like acknowledgements and streams have not been implemented fully.

4.3 RECT Database

Author: Christoph Fellner

4.3.1 Database Structure

The RECT Database is designed like 4.3. This is a Entity Relationship Diagram (ERD) that shows the structure of the database. The database implements a classic star schema, with the connections table as the center of the star. The connections table is linked to the client, method and address tables.
The RECT database is structured according to the widely adopted star model, providing a robust framework for efficiently managing connection data. Comprising four distinct tables, namely connections, client, method, and address, this database architecture ensures comprehensive organization and accessibility of critical information.

The connections table serves as a central hub, linking to the other three tables through their respective IDs and storing unique identifiers for each connection. Meanwhile, the client table houses data pertaining to the clients associated with specific connections, facilitating targeted access to connection information. In the method table, precisely three objects—BLE (Bluetooth Low Energy), TCP (Transmission Control Protocol), and UDP (User Datagram Protocol)—are stored, representing the available connection types within the RECT ecosystem. Additionally, the address table stores essential network information, including IP addresses and ports, with the provision for a null port value in instances where the connection method is BLE.

By adopting this schema, the RECT database optimizes accessibility to connection data for clients by establishing seamless linkages between tables. Leveraging the Rusqlite library, the backend components can efficiently query and retrieve pertinent information, enabling effective communication over the designated connections. Moreover, this database structure facilitates the extraction of various insights and metrics, empowering users to select and analyze data such as:

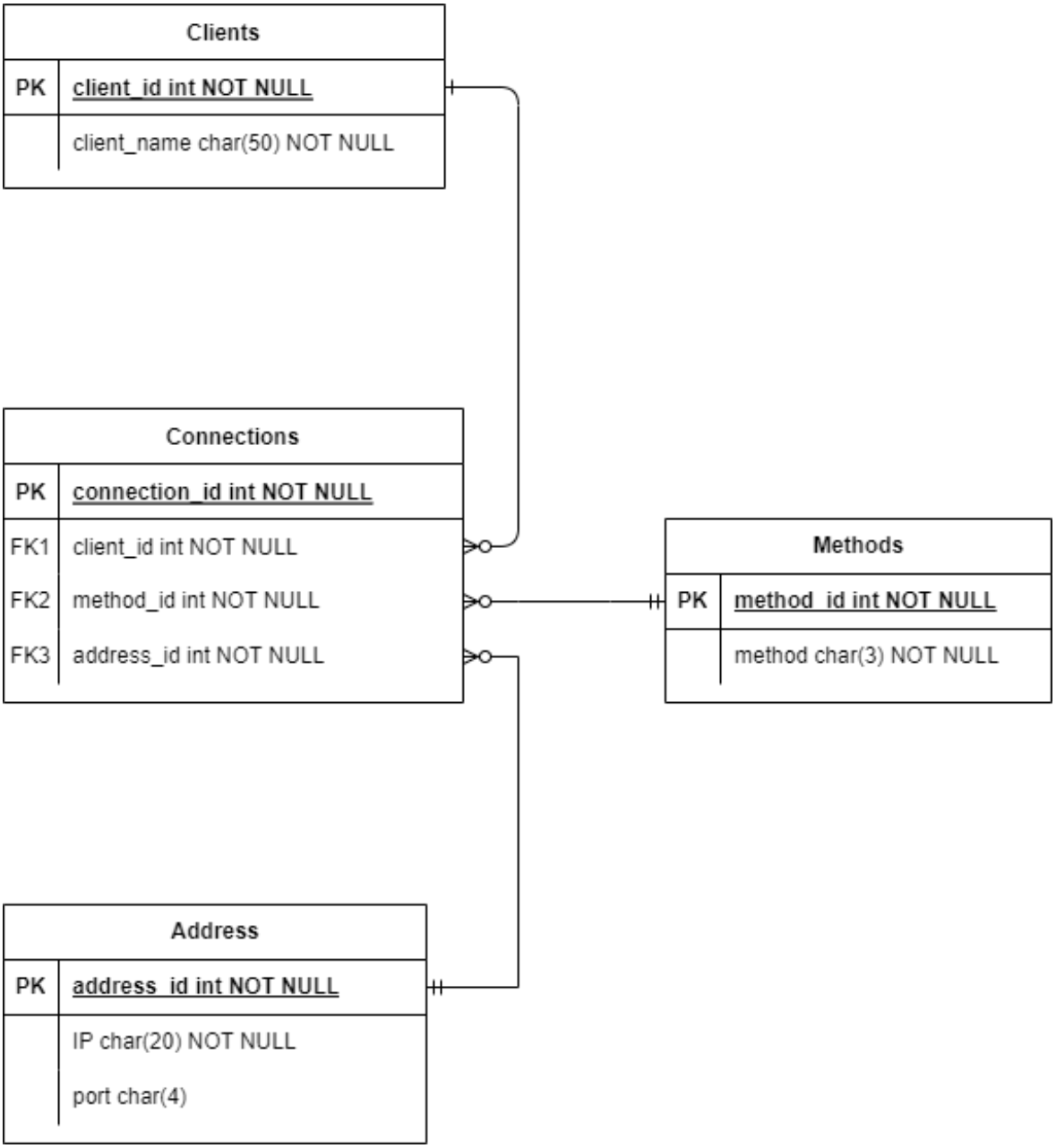

Figure 4.3: Database Architecture of RECT.

How many BLE, TCP or UDP connections are available?

All connections of a specific client.

All clients that have a specific connection.

All connections that are available.

How many connections are available in total?

Are there more connections to the same client?

4.3.2 Database usage

The RECT Database serves as a vital resource accessible not only to the Rust Interface but also to the entirety of the backend infrastructure within the RECT stack. Acting as the backbone for data management, the Rust Interface undertakes the essential task of configuring the database, a process that includes the insertion of data sourced from the JSON file into its tables. This data encompasses a comprehensive inventory of available connections tailored specifically for the local RECT Client system. Each connection within this dataset is meticulously characterized by a unique identifier, a designated communication method (be it BLE, TCP, or UDP), and a corresponding address composed of an IP address and port number.

Upon completing the database setup and population with pertinent connection data, the backend components of the RECT stack rely on this repository to establish and manage communication channels. However, before the backend can effectively harness the capabilities of the database, the Rust Interface must first initialize the database setup and meticulously inject the JSON-derived information. Following this preparatory phase, the Interface seamlessly facilitates backend access to the database via the CommLib, thereby ensuring a streamlined pathway for communication. Subsequently, armed with access to this centralized database, the backend components adeptly navigate and utilize the provided connections to facilitate efficient communication processes tailored to the demands of the RECT ecosystem.

4.4 Rust Interface

Author: Christoph Fellner
One of the main sections of the RECT project is the Rust Interface. The Rust Interface is the part of the RECT stack that is responsible for the communication between the different parts of the stack. That means that the Rust Interface is responsible for the communication between the CommLib and the RECT Database. The RECT Database is part of the Rust Interface, it stores Data about all available RECT connections. The Rust Interface also communicates with the Python Service and the C++ Service. So basically the Rust Interface is responsible for the communication between the frontend and the backend of the RECT stack.

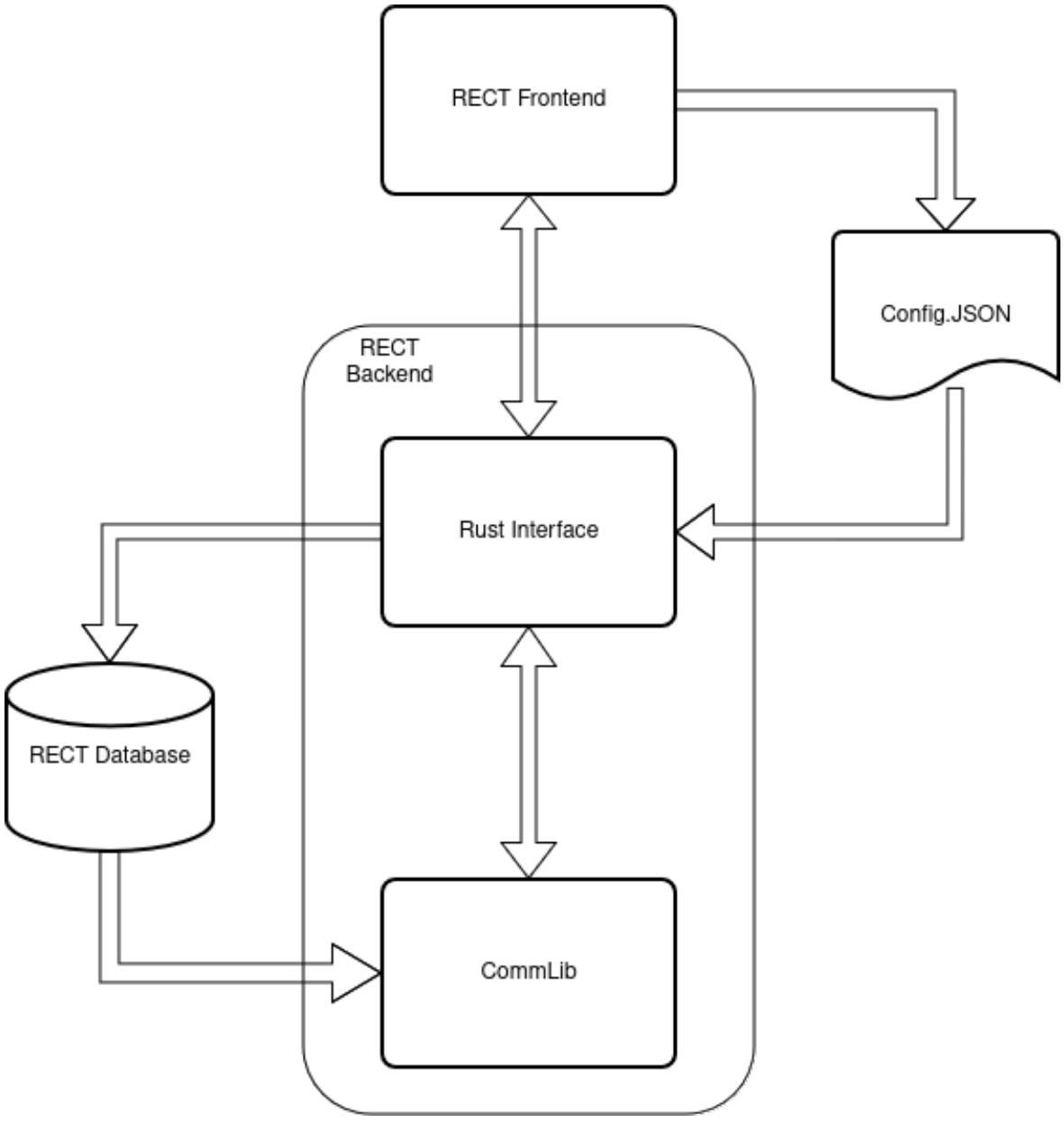

Figure 4.4: Structure of the Rust Interface.

In order to achieve this functionality the first step was to create a Rust module for gRPC communication. The gRPC communication is done with the help of the tonic library. The tonic library is a gRPC library for Rust, which is based on the tokio library. The tokio library is a runtime for writing reliable asynchronous applications with Rust. The gRPC module is responsible for the communication between the Rust Interface and the Python Service and the C++ Service. The gRPC module is also responsible for the communication between the Rust Interface and the CommLib.

The second step was to create a module for the RECT Database. The RECT Database is a in-memory SQLite database. The module uses the rusqlite library with the addition of the tokio-rusqlite library. The tokio-rusqlite library is a asynchronous version of the rusqlite library. The Rust Interface implements the RECT Database, creates the necessary tables and provides the necessary functions to communicate with the database. The module is also responsible for the inserts of the available connections from a JSON file into the database. The JSON file is created by the frontend and contains all available connections for the RECT stack.

For the second step to work the Rust Interface also needs to be able to read the Config.JSON file in which the data for the connections is specified which in turn is stored in the rect database. The Config.JSON file is read by another module of the Rust Interface. The module uses the serde library to read JSON files. The serde library is a framework for serializing and deserializing Rust data structures efficiently and generically. The module defines suitable structures for the data strored in the JSON file and then reads the file and stores the data in the defined structures. The database module then takes this data and stores it in the RECT database.

With these modules completed the Rust Interface is able to communicate with the CommLib, the Python Service and the C++ Service. In conclusion, the Rust Interface plays a crucial role within the RECT project, facilitating communication between various components of the stack. Through the implementation of modules for gRPC communication and the RECT Database, the Rust Interface ensures seamless interaction between the frontend and backend elements. Leveraging libraries such as tonic, tokio, rusqlite, and serde, the interface achieves reliable and efficient communication, handling tasks such as database management and data serialization effectively. Overall, the Rust Interface serves as a vital bridge, enabling smooth operation and integration within the RECT stack.

4.5 C++ Implementation

Author: Maximilian Dragosits
The C++ Implementation is one of the two outward facing components of the RECT stack. Alongside the Python Implementation it serves as a library in order for developers to be able to create robots, that are able to communicate with each other, much easier then before. This is accomplished by abstracting most of the complexities of gRPC behind the *Rectcpp* class.

The class only needs to be initialized with IP-Addresses for the different services that it offers and be given the IP of another of its kind and then it should be a simple act of using the predefined methods within the class in order to effortlessly communicate with other robots or devices running this or the Python frontend implementation.

4.5.1 Rectcpp class

The *Rectcpp* class is the foundation of the C++ implementation, providing users with intuitive functions to control a diverse range of RECT services. These functions simplify the management of multiple gRPC services by condensing them into straightforward calls. For example, the listen method streamlines this process. With only one line of code, users can engage in listening activities while the underlying complexities are abstracted away. This encapsulation not only improves usability but also promotes efficient and robust utilization of RECT's capabilities, enabling developers to focus on their core objectives without being burdened by implementation details. An Example of this is the *listen* function.

```
1   int Rectcpp::listen(std::string connectionName, std::string topic, std::string& returning);
```

The function requires users to provide the name of the connection to be monitored and specify the topic to which the incoming message must adhere. By supplying these parameters to the *listen* function, users establish the criteria for message reception and filtration. When a message that meets the specified conditions arrives at the designated connection and aligns with the prescribed topic, it is retrieved. The function extracts the contents of the received message and returns them to the user as a string. This approach ensures a seamless and structured message handling process, allowing users to manage communication flows efficiently within the RECT framework.

There are two importent facets of this class aside from the simplified interaction with gRPC services. The hosting of its own services and the management of connections to other services. The structure of *Rectcpp* can be seen in this UML-Diagram:

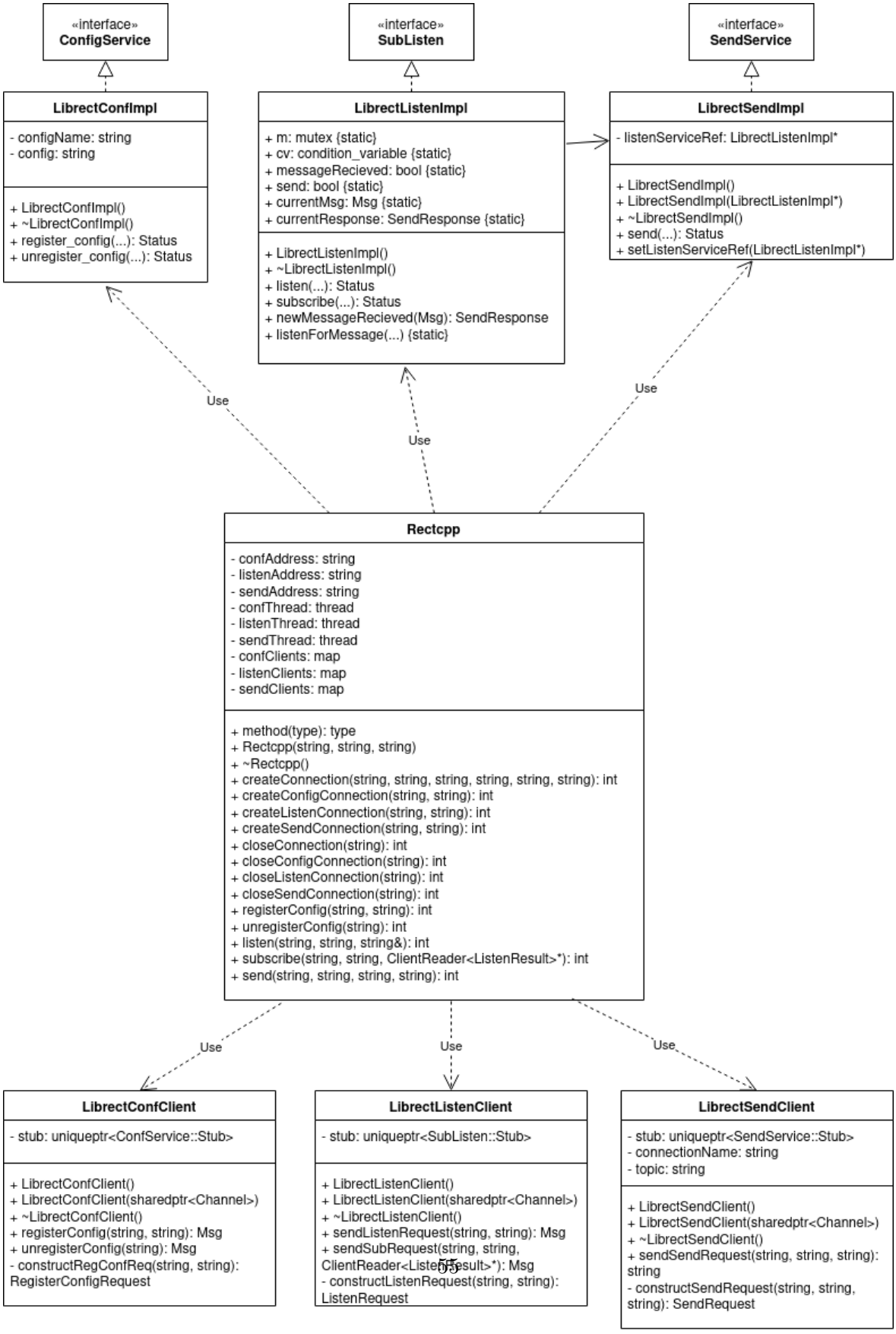

Figure 4.5: UML-Diagram of the *Rectcpp* class

While the users of the library will only directly interact with the methods of the main class, the various additional classes used in order to facilitate the use of remote procedure calls are also displayed in 4.5. All the classes with the *Impl* ending are the implementations of the interfaces automatically generated by ProtoBuf and form the serves, that clients from other instances connect to. In contrast the *Client* classes are for connecting to the servers on other instances that host the relevant services.

4.5.1.1 Construction

When invoking the constructor of the *Rectcpp* class, users must provide the IP addresses and port numbers for the three distinct servers designated to launch the three different services. This crucial initialization step requires the network parameters to be provided in string format. The three services that are integral to this project are:

- Config Service: The Service used to send and recieve config data between two systems.
- Listen Service: The Service used to listen an subscribe to certain topics and then inform the listening clients when a message arrives.
- Send Service: The Service used to send messages with topics to other systems.

During the construction phase, instantiation of the services occurs, followed promptly by the commencement of a dedicated server for each. This initialization process is facilitated seamlessly through the utilization of the Serverbuilder module from the gRPC libraries, which streamlines the setup and configuration of servers. However, despite the robust capabilities of the gRPC Serverbuilder, challenges arose during the development of this library, primarily stemming from the absence of comprehensive documentation regarding the precise methodology for integrating the instantiated classes with the Serverbuilder module. This ambiguity led to several stumbling blocks and intricacies encountered along the path of development.

4.5.1.2 Connections

Connections within this class are managed using a C++ map structure, which allows for the binding of client instances to servers hosting the three services, all under user-assigned names. This approach significantly simplifies the usability of the class, replacing the need for cumbersome IP addresses and port numbers with easily discernible and memorable names.

The creation of connections is achieved through two distinct pathways: the general-purpose *createConnection* function and its more specialized counterparts. The *createConnection* function instantiates connections for all three services at once. In contrast, the more specific functions cater to individual service connections, depending on their designated name. For example, the *createListenConnection* method requires only the name of the connection and the corresponding IP address. This establishes a connection solely with the Listen Service hosted at the provided address. This granular approach streamlines the process of establishing connections and enhances the modularity and flexibility of the Rectcpp class. Users can tailor their interactions with specific services according to their requirements.

4.5.2 Definition of CMake file

In order to compile and generate the gRPC services within this library CMake was used. The CMakeLists file of this project contains the standard CMake commands in order to make it into a C++ libary. Along side this are the lines responsible for ProtoBuf to generate the base classes for the implementation of gRPC.
First the proto files that will be turned into these base classes are registered using these lines:

```
1  file(GLOB RectVOneConf "${CMAKE_CURRENT_SOURCE_DIR}/proto/conf.proto")
2  file(GLOB RectVOneListen "${CMAKE_CURRENT_SOURCE_DIR}/proto/listen.proto")
3  file(GLOB RectVOneSend "${CMAKE_CURRENT_SOURCE_DIR}/proto/send.proto")
4  file(GLOB RectVOneMessage "${CMAKE_CURRENT_SOURCE_DIR}/proto/message.proto")
5  set(PROTO_CONF_FILES ${RectVOneConf})
6  set(PROTO_LISTEN_FILES ${RectVOneListen})
7  set(PROTO_SEND_FILES ${RectVOneSend})
8  set(PROTO_MESSAGE_FILES ${RectVOneMessage})
```

Listing 4.2: Excerpt from CMakeLists file of the C++ portion

The first three lines are for the files pertaining to the services and the last one is for the message type used by them in order to communicate. They are registered using the *file* command with the *GLOB* keyword under a chosen name in order to refer back to them later in the CMake file.

After this each of the services is assigned a library using the *add_library* command and then the required dependencies for them to function are included using *target_link_libraries*. These are in this case *libprotobuf*, *grpc* and *grpc++*.

Finally the *protobuf_generate* command is used to signify to ProtoBuf to generate the base classes when the project is built using CMake.

```
1  protobuf_generate(TARGET rect-conf-service LANGUAGE cpp)
2
3  protobuf_generate(
4      TARGET
5        rect-conf-service
6      LANGUAGE
7        grpc
8      GENERATE_EXTENSIONS
9        .grpc.pb.h
10       .grpc.pb.cc
11     PLUGIN
12       "protoc-gen-grpc=${grpc_cpp_plugin_location}"
13 )
```

Listing 4.3: Excerpt from CMakeLists file of the C++ portion

In this example the *protobuf_generate* is used twice. The first one sets the target files to be used during generation and the programming language for the classes to be created in. The second instance of the command informs it what type of service and the appropriate

extensions for the generated files. The location of the gRPC generation plugin is also given.

This is then repeated for all of the services, that will be created. This is done in order to make it easier to implement gRPC within this library. The implementation of the classes generated by ProtoBuf is then accomplished by the manual creation of a class that extends the previously automatically created service class. This involves coding the functions that were originally defined within the proto files and then made into functions within the service classes.

4.5.3 gRPC Serverbuilder

During the development of the *Rectcpp* class, a critical issue surfaced with the Serverbuilder[3] provided by the gRPC library[4]. The problem stemmed from the method of passing the instance of the service to the builder, resulting in a compilation failure within the library. To ensure the smooth operation and continuous running of the servers, start functions were meticulously crafted for each of the three services. These functions employed threads containing lambda functions, responsible for initiating the servers and awaiting input from remote procedure calls.

Initially, all service classes were designated as private members of the main class and instantiated during its construction. Subsequently, these instances were passed to the functions responsible for spawning the server threads. However, the Serverbuilder[5] employed in these functions could not accommodate this method of passing service objects.

The obscure and extensive error messages compounded the troubleshooting process, prolonging the resolution period. Consequently, numerous approaches were explored to access the pre-initialized objects within the start function, and notably within the lambda function executing the server thread. Techniques ranged from passing objects by reference to accessing class members from the function and copying them into new instances before passing them to the lambda functions. Regrettably, none of these strategies proved effective in resolving the issue.

After considerable effort, a breakthrough occurred when a simplified version of the *Rectcpp* class was reconstructed. The pivotal realization was that constructing the service classes outside the lambda function was counterproductive. Instead, the solution lay in providing all necessary data for their initialization within the start functions, allowing them to be instantiated during the thread runtime. This adjustment finally circumvented the perplexing obstacle, enabling the starting of the servers.

[3] *gRPC Documentation on the Serverbuilder class.* 2024. URL: `https://grpc.github.io/grpc/cpp/classgrpc_1_1_server_builder.html`.

[4] *gRPC Documentation on Github.* 2024. URL: `https://grpc.github.io/grpc/cpp/index.html`.

[5] *gRPC Documentation on the Serverbuilder class.*

4.5.4 gRPC CreateChannel

The other major issue that presented itself during development beside the problem with the Serverbuilder mentioned in the previous section is the throwing of a memory missmanegment error during runtime whenever the *CreateChannel*[6] function from the gRPC library[7] is called within the functions responsible for connecting a client to a server.

The specific line of code, that results in this error looks like this:

```
1  auto channel = CreateChannel(confAddress, InsecureChannelCredentials());
```

As can be seen a gRPC channel is returned by the function with a connection to the specified address. Here the IP-Address and Portnumber of the server to connect to is contained within the variable *confAddress* in the form of a string. The other argument signifies how the channel is to behave during use. During the execution of the Rectcpp library one of two error messages appear in the console before the program shuts down.

```
1  /home/maxi/rect/C++_Frontend/tests/main.cpp:51: FAILED:
2    due to a fatal error condition:
3    SIGSEGV - Segmentation violation signal
4  ===============================================================================
5  test cases: 1 | 1 failed
6  assertions: 1 | 1 failed
7
8  Segmentation fault
```

Listing 4.4: One of the error messages of the Rectcpp class during execution

```
1  terminate called after throwing an instance of 'std::bad_alloc'
2  terminate called recursively
3    what(): std::bad_alloc
4
5  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6  UnitTests is a Catch2 v3.4.0 host application.
7  Aborted
```

Listing 4.5: The other error message of the Rectcpp class during execution

The problem is that during the execution and use of the library a memory error called *std::bad_alloc* is invoked and the process is terminated. Because of the ambiguity of the error message the process of locating the source of the issue took a long time. After many failed attempts of resolving this issue it still persists within the most recent version of the *Rectcpp* library.

[6] *gRPC Documentation on the CreateChannel function.* 2024. URL: https://grpc.github.io/grpc/cpp/namespacegrpc.html#ad62b23e19fdcd13898119cd94818616d.

[7] *gRPC Documentation on Github.*

Unfortunatly a solution to this problem has proved to not be possible within the timeframe of the project. This has lead to the *Rectcpp* class being largly not functional and not being able to connect to other instances of itself or instances of the Python library.

4.5.5 Documentation

Alongside the *Rectcpp* library documentation has also been made for the functions of the library in order to make working with this tool easier in the future. Unfortunatly the majority of this documentation in its current form will be useless due to the issues with the *Rectcpp* class outlined in the previous paragraphs. However it is still available in case someone in the future would like to continue developing this project.
This documentation was created with the help of *Doxygen* and is accessible by looking wihtin the *documentation* folder wihtin the project and opening the *Index.html* file in a browser.

4.6 Python Implementation

Author: Timon Koch
The Python Implementation and the C++ Implementation are the two primary components in the RECT stack that face outward. They serve as libraries that allow developers to create robots that can communicate seamlessly. The Rectpy class simplifies the usage of gRPC. Initializing the class is as simple as providing IP addresses for its various services and specifying the IP of another instance. After initialization, developers can confidently use the pre-defined methods within the class to communicate seamlessly with other robots or devices running either this Python implementation or the C++ frontend.

4.6.1 Rectpy class

The Rectpy class forms the foundation of the Python implementation, providing users with powerful functions to efficiently control various RECT services. These functions streamline the management of multiple gRPC services by reducing them to simple calls.
The send method requires only one line of code for users to send a message, abstracting away underlying complexities. This encapsulation improves usability and enables efficient and robust utilization of RECT's capabilities. Developers can focus on their core objectives without being burdened by implementation intricacies.

```
1   def send(self, connectionName, msg, recipient, topic):
```

The function requires users to specify the connection to be monitored, the message itself, the recipient and the subject of the outgoing messages. By providing these parameters to the send function, users define the criteria for sending a message. This methodology ensures a seamless and structured message sending process, enabling efficient management of communication flows within the RECT framework.
It is also possible to receive messages from a given stream using the listen method. Like the send function, the listen method requires only a single line of code.

```
1   def listen(self, connectionName, topic):
```

To receive a message, the Listen function requires the user to specify the connection to be monitored and the corresponding topic. By passing the necessary parameters to the function, the user defines the criteria for receiving messages.
Two essential aspects of this class, besides the simplified interaction with gRPC services, are the hosting of own services and the management of connections to other services.

4.6.1.1 Construction

When initialising the Rectpy class constructor, users are required to provide the IP addresses and port numbers for the three different servers responsible for launching the three different services. This critical initialisation step requires the provision of network parameters in string format. The three services involved in this project are

- Config Service: Facilitates the exchange of configuration data between two systems.
- Listen Service: Allows subscriptions to specific topics and notifies listening clients when messages arrive.
- Send Service: Allows messages with specified topics to be sent to other systems.

During the build phase, the services are instantiated, followed immediately by the launch of a dedicated server for each service. This initialisation process is seamlessly facilitated by the use of the Serverbuilder module from the gRPC libraries, which streamlines the setup and configuration of servers.

4.6.1.2 Connection

Connection creation is achieved in two different ways: the general purpose createConnection function and its more specialised counterparts. The createConnection function instantiates connections for all three services at once. The more specific functions, on the other hand, instantiate connections for individual services depending on the name given to them. For example, the createSendConnection method will only connect to the Send service hosted at the specified address. This granular approach streamlines the connection establishment process and increases the modularity and flexibility of the Rectpy class. Users can tailor their interactions with specific services according to their needs.

4.6.2 Documentation

In addition to the Rectpy library, extensive documentation has been created using Pydoc for its functions to facilitate future use of this tool. This documentation can be accessed by navigating to the documentation folder within the project directory and opening the associated file.

Chapter 5

Tests

RECT requires a number of different technologies to be used effectively. To find the best technologies for the project, several experiments are needed to evaluate and compare the different methods available.
Author: Timon Koch

5.1 Testing possible Optimization

Author: Timon Koch
In order to find the best possible combination of serialisation and compression methods for use in RECT, different pairs are applied to the text of the King James Bible[1]. The dataset consists of approximately 783,137 words spread over 66 books. The experiment aims to evaluate the effectiveness of the different pairs of compression and serialisation methods. First, the text is encoded using one of the serialisation methods. After serialisation, the data is compressed using one of the compression methods
The setup will maintain a standardised computing environment for consistency and the experiment will be repeated several times for each pair of serialisation and compression methods.
Performance metrics will include serialisation and deserialisation, compression and decompression time, and post-compression size. Statistical analysis will compare the performance of different methods.

5.1.1 Compression

Timon Koch
By reducing the size of data, compression saves storage space, speeds up data transmission over networks and minimises bandwidth requirements, resulting in cost savings and improved operational efficiency. In addition, compressed data is easier to manage, share and access across platforms and devices.

[1] *King James Bible*. URL: `https://www.kingjamesbibleonline.org`.

5.1.1.1 Deflate

The Deflate algorithm[2] is a widely used method of lossless data compression. It achieves compression by eliminating redundancy in the input data through a combination of LZ77[3] and Huffman coding[4] techniques, making it an efficient and versatile compression method used in various applications.
In the first step, LZ77 identifies repeated substrings in the data and replaces them with references to previous occurrences, thereby reducing redundancy. This is followed by Huffman coding, which assigns shorter codes to more frequent symbols and longer codes to less frequent symbols, further compressing the data

5.1.1.2 ZSTD

ZSTD[5], short for Zstandard, is a high performance data compression algorithm developed by Facebook. It is designed to provide both fast compression and decompression speeds while achieving excellent compression ratios, making it suitable for a wide range of applications.
ZSTD uses a combination of advanced compression techniques, including a dictionary-based approach, context modelling and entropy coding. It dynamically builds and updates dictionaries during compression to improve compression efficiency by identifying and exploiting patterns in the data.

5.1.1.3 GZIP

GZIP[6], short for GNU Zip, is based on the Deflate algorithm. While it's optimised for speed, GZIP also provides reasonably good compression ratios, making it a popular choice for applications where both speed and compression efficiency are important.
GZIP typically offers several levels of compression, allowing users to trade off compression ratio and speed according to their needs. Lower compression levels prioritise faster processing times, while higher levels prioritise better compression ratios at the expense of speed.

5.1.2 Buffering

5.1.3 Serialization Formats

Author: Timon Koch

5.1.3.1 JSON

JSON (JavaScript Object Notation)[7] is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is based on a

[2] *Deflate Algorithm.* URL: `https://www.w3.org/Graphics/PNG/RFC-1951`.
[3] *LZ77 Algorithm.* URL: `https://www.gm.th-koeln.de/~hk/lehre/ala/ws0506/Praktikum/Projekt/D_rot/Ausarbeitung.pdf`.
[4] *Huffman Coding.* URL: `https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3`.
[5] *Zstandard.* URL: `http://facebook.github.io/zstd`.
[6] *GNU Zip.* URL: `https://www.gzip.org`.
[7] *What is JSON?*.

Serializer	Pre Compression Size (Byte)	Increase in Size (%)
JSON	4556012	190
YAML	4630774	193
Protobuf	4606085	192
Bincode	5055849	211
Rust UTF-8	4256089	177

subset of the JavaScript programming language[8].

JSON consists of key-value pairs, where keys are strings and values can be strings, numbers, arrays, objects, booleans, or null. It is highly versatile and widely supported across various programming languages and platforms.

Using JSON as the seriation method results in the following:

	Serialization Time (ms)		Deserialization Time (ms)		Compression Time		Deco
Compression	Mean	Variance	Mean	Variance	Mean	Variance	Mea
GZIP Fast	7.421e-2	1.700e-6	4.448e-1	7.124e-6	1.397e-1	1.7e-6	3.43e
GZIP Best	7.380e-2	3.516e-7	2.966e+0	1.485e-3	8.834e-2	5.563e-7	3.443
ZSTD	7.401e-2	8.668e-7	1.138e-1	2.58e-6	4.175e-2	1.534e-5	3.492
deflate	7.421e-2	9.84e-7	2.988e+0	2.081e-3	8.169e-2	3.379e-6	3.625

5.1.3.2 YAML

YAML[9] (YAML Ain't Markup Language) is a data serialisation format commonly used for configuration files, data exchange, and specifying data structures in a clear, concise, and readable manner. It uses indentation and whitespace to represent data hierarchies, making it visually intuitive and easy to understand.

YAML supports multiple data types, including scalars (strings, numbers, booleans), sequences (arrays or lists), and mappings (key-value pairs). It also allows for comments, making it useful for annotating configurations and documentation.

Using YAML as the seriation method results in the following:

	Serialization Time (ms)		Deserialization Time (ms)		Compression Time		D
Compression	Mean	Variance	Mean	Variance	Mean	Variance	
GZIP Fast	5.0937E-01	3.9388E-05	4.5498E-01	2.3392E-05	1.4108E-01	3.2461E-06	5.3
GZIP Best	5.0596E-01	3.3615E-06	3.0110E+00	2.1999E-04	8.9274E-02	1.5003E-06	5.3
ZSTD	5.0769E-01	6.4273E-06	1.1543E-01	2.7384E-06	4.1505E-02	7.0610E-07	5.3
deflate	5.0638E-01	4.1805E-05	3.0265E+00	1.7159E-03	8.3479E-02	7.5117E-06	5.3

[8] *javaScript*. URL: `https://www.javascript.com`.
[9] *YAML*. URL: `https://yaml.org/`.

5.1.3.3 Protobuf

Protocol Buffers (protobuf)[10] is a method of serialising structured data developed by Google. Designed to be efficient, portable and easy to use, it provides a platform-neutral way to encode data for communication between systems or for persistent storage.
Protobuf defines a schema for data structures using a language-agnostic interface description language (IDL)[11]. This schema describes the structure of the data in a concise and platform-independent manner. From this schema, Protobuf compilers generate code in various programming languages, allowing developers to easily work with the defined data structures.
Using Protobuf as the seriation method results in the following:

	Serialization Time (ms)		Deserialization Time (ms)		Compression Time	
Compression	**Mean**	**Variance**	**Mean**	**Variance**	**Mean**	**Variance**
GZIP Fast	2.2997E-02	2.0312E-07	4.8645E-01	5.4953E-06	1.4393E-01	1.0112E-06
GZIP Best	2.2861E-02	3.0166E-07	2.4989E+00	2.6906E-05	9.1720E-02	1.0050E-06
ZSTD	2.3221E-02	5.7291E-07	1.2207E-01	1.0494E-06	4.1794E-02	3.5202E-07
deflate	2.2783E-02	2.4169E-07	2.5021E+00	8.8949E-05	8.4666E-02	2.3459E-06

5.1.3.4 Bincode

Bincode[12] is a binary serialisation format designed for use with the Rust programming language. It allows the serialisation of Rust data structures into a compact binary representation that can be efficiently stored or transmitted. Bincode is designed to be fast and efficient, making it suitable for scenarios where performance is critical, such as networking, file I/O or inter-process communication.
Using Bincode as the seriation method results in the following:

	Serialization Time (ms)		Deserialization Time (ms)		Compression Time	
Compression	**Mean**	**Variance**	**Mean**	**Variance**	**Mean**	**Variance**
GZIP Fast	1.8045E-02	8.4238E-07	4.8633E-01	1.3426E-04	1.4902E-01	7.2413E-07
GZIP Best	1.8991E-02	3.1245E-06	3.6885E+00	4.9743E-03	9.6516E-02	2.1090E-06
ZSTD	1.8366E-02	4.6941E-07	1.3126E-01	2.0827E-04	4.4627E-02	1.4616E-06
deflate	1.8419E-02	1.3100E-06	3.6830E+00	2.1365E-03	9.1972E-02	4.6310E-05

[10] *Protocol Buffers*. URL: `https://protobuf.dev/`.
[11] *Interface Description Language*. URL: `https://www.techtarget.com/whatis/definition/IDL-interface-definition-language`.
[12] *Bincode*. URL: `https://docs.rs/bincode/latest/bincode/`.