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
RECT Tool and Python gRPC interface and library for communication Timon Koch	5CHIF

Betreuer / Betreuerin:

Harald R. Haberstroh

Wiener Neustadt, am February 20, 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 February 20, 2023/24

Verfasser / Verfasserinnen:

Jeremy Sztavinovszki   Christoph Fellner

Timon Koch   Maximilian Dragosits

Contents

Acknowledgement

The authors would like to thank ...

Kurzfassung

Author: Sztavinovszki

Eine der am meisten in der Robotik verwendeten Software Suiten ist das sogenannte Robot Operating System[1] (kurz ROS). Es ist eine Sammlung von Werkzeugen und Packeten, die verwendet werden, um hoch performante Robotik Systeme zu entwickeln und es abstrahiert die Kommunikation zu Topics und Messages, die man in ROS-Bag Files aufzeichnen und abspielen kann. Das erlaubt es entwicklern sich auf die tatsächlichen Probleme der Entwicklung zu konzentrieren und sich nicht darum kümmern zu müssen Kommunikation neu zu implementieren. ROS erlaubt außerdem die simulation Zahlreicher Komponenten von Robotern, welche man meistens als Digitale Zwillinge bezeichnet. Einer der größten Nachteile, die ROS mit sich bringt ist die Komplexität und der Eigensinn, bei der Entwicklung. Diese Komplexität führt außerdem zu Performance einbußen, welche vor allem die Verwendbarkeit auf weniger Leistungsstarken Geräten, wie zum Beispiel dem KIPR Wombat Controller[2], welcher bei Schülern wegen seiner Kostengünstigkeit und Verwendbarkeit zum Einsatz kommt.

Diese Arbeit wird erkunden, wie man eine Software ähnlich zu ROS implementiert. Es werden Probleme, wie zum Beispiel Inter Prozess Kommunikation[3], Kommunikation durch Bluetooth Low Energy (BLE), TCP und UDP, und die Implementation von Kommunikations Libraries für verschiedene Programmiersprachen

[1] **ros-site**.
[2] **wombat-controller**.
[3] **ipc-begriff**.

Abstract

Author: Sztavinovszki

In robotics one of the most important topics over the last couple of years has been communication. Communication doesn't only concern a robot being remote controllable. Mimimi keine Ahnung.

Chapter 1

Introduction

1.1 Motivation

Author: 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. 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. This

1.2 Goal

Author: Sztavinovszki
This diploma thesis will take explore how to write a real-time communication framework using new technologies, such as BLE with TCP and UDP.
At the end of the project the framework should be useable for sending data between two KIPR Wombat-Controllers. It should be capable to send well structured data, e.g. a JSON-file, as well as streaming data, e.g. a series of pictures, between the robots. All sent data should have the option to be compressed with different compression formats ranging from lossless to lossy formats. 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 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.

1.3 History

Author: 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.[1]
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.[2] 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[3], 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 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] **A˙New˙History˙of˙Modern˙Computing**.
[2] **How˙the˙web˙was˙born**.
[3] **rfc˙675**.

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: Sztavinovszki

1.4.1 Kanban

In a project like RECT, that is relatively small, but still complex and needs the members to be able to work independently project management is of utmost importance. Today the most obvious choice for working on a group project is using SCRUM[4], 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[5] is a form of project management, that chooses to use continuous flow of tasks instead of splitting them up into sprints, like SCRUM. It also lacks distinct roles for team members, that would be defined in SCRUM (e.g. a SCRUM-Master), and uses cycle-time[6], instead of velocity as a metric for performance.

[4]**what-is-scrum**.
[5]**what-is-kanban**.
[6]**cycle-time-lead-time**.

1.4.1.1 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 columsn:

- 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 foor tasks, that have been tested, reviewed and merged with the main branch.

1.4.2 Meetings with the projects supervisor

There were weekly meetings that were held with the diploma theses' supervisor Harald Haberstroch 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.

1.4.3 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.

1.4.4 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.

1.5 Outline

Author: Sztavinovszki Beginning with with the Technology section the thesis describes the different libraries, languages and technologies used and explains why the respective technologies have been chosen for RECT. After the Technology section there's a dive into the implementation details and design rationale behind the materialization of the different practical parts of the project. The Implementation section will also include comparisons between the different ways of implementing certain components of RECT from a standpoint of readability and maintainability, but not of performance, which will be discussed in the Testing section of this thesis. As stated, the Testing section of the thesis will include comparisons in performance of different implementations, technologies and field tests of the technology resulting from RECT.

Chapter 2

Study of Literature Sztavinovszki

Author: Jeremy Sztavinovszki

2.1 The Rust Book

2.2 Network Programming with Rust

2.3 Main Takeaways

2.4 Getting Started with Bluetooth Low Energy

Getting Started with Bluetooth Low Energy[1] 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.

2.4.1 Chapter 1. Introduction

The introduction explains the history of Bluetooth Low Energy from its start as Wibree, 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.

2.4.2 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 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.

[1]**ble˙book**.

2.4.3 Chapter 3. GAP(Advertising and Connections)

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.

2.4.4 Chapter 4. GATT(Services and Characteristics)

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.

2.4.5 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.

2.4.6 Main 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

Technology

3.1 Wombat

The KIPR Wombat 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 and can be used with a web browser and WiFi on any operating system.

3.2 Python

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. It was created by Dutch programmer Guido van Rossum as a successor to ABC and was first released in 1991 with the capability of exepction 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 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, performance, and flexibility.[1] 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[2] and Doxygen[3]. 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. The 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 to the development landscape.

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 Cargo

Author: Christoph Fellner
Cargo[4] 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

[1] **lecture˙essence˙cpp**.
[2] **catch2˙git**.
[3] **doxygen˙main˙site**.
[4] **cargo**.

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.
- 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[5] 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

[5]**grpc˙main˙site**.

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.

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, whether using the robustness of Java, the conciseness of Kotlin, the flexibility of Python, or the performance advantages of C-based languages. 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.

```
syntax = "proto3";
package msg;


message From {
  string conn_name = 1;
  string topic = 2;
}


message To {
  string conn_name = 1;
  string topic = 2;
}


message Msg {
  bytes data = 1;
  oneof fromto {
    From f = 2;
    To t = 3;
  }
}
```

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 within this will be stored in when it is converted into a programming languages code is defined. In this case it will be *msg*. After that you can import any other .proto 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 then .proto file.

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

3.6.1 Protofile message definition

Message types in proto3 are relatively simple to define.

```
message message_name {
    field_type field_name = number;
  }
```

First the *message* keyword is used to signify that the following is a declaration for a message type. Then a freely choose able *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 Protobufs 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 are share a lot of the same traits as message types in terms of the defintion syntax.

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

Similarly to messages the enum is given an *enum_name* and then any number of *constant_values* 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 cases like fields with the *optional* modifier. In order to bind multiple *constant_values* 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_values*:

```
option allow_alias = true;
```

Once an enum is defined then it can be used in other parts of the Protocol Buffer, 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;
    }
}
```

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) {}
}
```

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:

```
service SubListen{
    rpc listen(ListenRequest) returns (ListenResult) {}
    rpc subscribe(ListenRequest) returns (stream ListenResult) {}
}
```

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[6], a continuous intergration tool, and nixops, a deployment tool for nixos deploying NixOS in a network/cloud[7]. 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:

```
#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 3.1: Simple Examples of the Nix Language

In examining the provided code, a discernible pattern emerges. The syntax

[6] **hydra**.
[7] **NixOps**.

`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[8]. Block No. 4 showcases optional parameters. This feature is denoted by
`{<parameter name> ? <default value>}: <statement>`.

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[9], 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.

[8] **lambda˙calculus**.
[9] **dependency˙hell**.

```nix
{
  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 3.2: 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)[10]. 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[11] on github comes in handy. It is a collection of over 80000 packages according to

[10]**nixMkDerivation**.
[11]**nixpkgs˙repo**.

repology[12]. 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[13]. 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[14], 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.

- The Glucose profile, which is used to securely transmit measurement data from e.g. insulin pumps.

[12]**repology˙nixpkgs**.
[13]**ism**.
[14]**fhss**.

- The Find Me profile, 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.

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

3.9.1 History

The history of WiFi, which stands for Wireless Fidelity, spans decades and involves numerous technological breakthroughs, standards developments, and the evolution of wireless communication. It is widely acknowledged that the story of WiFi begins with the quest for convenient and efficient wireless connectivity, driven by the ever-growing need for mobility and flexibility in computing and communication devices.
The origins of WiFi can be traced back to the late 19th and early 20th centuries, with the invention of radio waves and subsequent advancements in wireless communication technologies. Pioneers such as Nikola Tesla, Guglielmo Marconi, and Heinrich Hertz contributed significantly to the advancement of wireless communication principles.
During the 1970s, the concept of local area networks (LANs) became increasingly popular, particularly with the invention of Ethernet by Robert Metcalfe at Xerox PARC. Ethernet initially used coaxial cables and later switched to twisted pair wiring, which allowed computers to communicate within a limited physical area. This innovation laid the foundation for the eventual development of wireless LANs.
In the 1980s and 1990s, several wireless LAN protocols were developed, including proprietary ones like WaveLAN by NCR Corporation and the IEEE standard 802.11, which served as the foundation for WiFi. However, these early protocols had some limitations such as limited range, slow data rates, and interoperability issues.
The term 'WiFi' was coined in 1999 by the Wi-Fi Alliance, a non-profit organization established to promote wireless LAN technology and interoperability. The alliance included major technology companies such as Apple, Cisco, Nokia, and Symbol Technologies. In 1997, the IEEE released the first WiFi standard, IEEE 802.11, which provided a framework for wireless communication using radio frequencies.
The IEEE 802.11 standard has undergone several amendments aimed at enhancing data rates, security, and reliability. Key milestones include 802.11a (1999), which operated in the 5 GHz frequency band and offered higher speeds than previous versions, and 802.11b (1999),

which operated in the 2.4 GHz band and became widely adopted due to its compatibility with existing devices.
WiFi started to gain widespread adoption in the early 2000s, fueled by the proliferation of mobile devices, laptops, and the growing demand for wireless internet access in homes, businesses, and public spaces. The introduction of WiFi-enabled consumer electronics, such as smartphones, tablets, and smart TVs, further accelerated its adoption.
In recent years, there have been advancements in WiFi standards resulting in faster data rates and extended range. Standards such as 802.11n (2009) introduced multiple-input multiple-output (MIMO) technology, which significantly improved throughput and range. Subsequent standards such as 802.11ac (2013) and 802.11ax (Wi-Fi 6, 2019) have further enhanced speed, capacity, and efficiency.
WiFi has become an integral part of modern life, enabling wireless internet access in homes, offices, schools, airports, cafes, and public spaces worldwide. Its impact on how people work, communicate, and access information has been significant, driving digital transformation across various industries and sectors.
In the 2020s, WiFi 6E (802.11ax) emerged as the latest standard, introducing support for the 6 GHz frequency band, which significantly expands available spectrum and reduces congestion in wireless networks. As a result, it is expected to provide even higher data rates, lower latency, and better performance in dense deployment scenarios.
Looking ahead, the future of WiFi is likely to be shaped by advancements in technologies like Internet of Things (IoT), 5G integration, artificial intelligence (AI), and Wi-Fi 7 standards. These developments are anticipated to improve connectivity, security, and user experience in the increasingly interconnected world.
In conclusion, it can be said that the history of WiFi demonstrates remarkable human ingenuity and innovation in the pursuit of wireless connectivity. WiFi has transformed the way we connect, communicate, and interact with the world around us, from its humble beginnings as a nascent technology to its ubiquitous presence in today's digital age.

3.9.2 Usage

The use of WiFi has become widely adopted in modern society, transforming the way people connect to the internet, communicate, and interact with technology. WiFi provides wireless connectivity, allowing users to access the internet and network resources without the limitations of physical cables. Its usage spans various sectors and applications, greatly impacting daily life, business operations, and technological advancements.
WiFi is a widely used technology in consumer electronics, including smartphones, tablets, laptops, smart TVs, and smart home devices. These devices rely on WiFi to access the internet, stream media, download content, and communicate with other devices within a local network. In modern gadgets, WiFi has become an essential feature, providing users with the flexibility to stay connected and productive while on the go.
WiFi is considered essential for home networking, as it enables households to establish wireless internet connections for multiple devices simultaneously. WiFi routers and access points distribute internet connectivity wirelessly, allowing family members to browse the web, stream videos, play online games, and connect IoT devices without the hassle of wired connections.

WiFi has become synonymous with home entertainment, productivity, and smart living, powering the connected homes of today.
Wireless connectivity is often considered a necessity in a variety of settings, such as offices, campuses, retail establishments, and public venues, particularly in the business world. WiFi networks can facilitate collaboration, communication, and productivity by enabling employees to access corporate resources, email, cloud services, and applications from anywhere within the premises. Enterprises utilise WiFi for a variety of purposes, including guest access, location-based services, asset tracking, and IoT deployments. This can lead to improved operational efficiency and enhanced customer experience.
WiFi is frequently utilized in educational institutions, including schools, universities, libraries, and training centers. WiFi-enabled classrooms can enhance digital learning initiatives, online research, collaborative projects, and multimedia presentations. Both students and educators can benefit from wireless access to educational resources, e-books, online courses, and interactive learning platforms. The implementation of WiFi in educational settings has the potential to improve the educational experience by promoting connectivity, engagement, and innovation in learning environments.
WiFi hotspots are frequently available in public areas such as airports, coffee shops, restaurants, hotels, malls, and transportation hubs. These locations offer WiFi access to customers, travelers, and visitors, enabling them to stay connected while on the go. This can improve customer satisfaction, loyalty, and time spent in commercial establishments, which can create opportunities for digital marketing, location-based services, and personalized experiences.
WiFi technology plays an important role in healthcare settings, as it enables wireless communication, data exchange, and patient care delivery. Hospitals, clinics, and medical facilities use WiFi for electronic health records (EHR), telemedicine consultations, medical imaging, remote monitoring, and wearable devices. WiFi-enabled healthcare solutions can enhance efficiency, accessibility, and quality of care while ensuring compliance with privacy and security regulations.
The utilization of WiFi connectivity is considered to be of great importance for the widespread adoption of Internet of Things (IoT) devices, which serve a variety of purposes such as data transmission, control, and automation. WiFi networks are also utilized in smart homes, smart cities, and industrial IoT deployments to connect sensors, actuators, cameras, and devices for monitoring and management purposes. The integration of various IoT applications, such as home automation, energy management, traffic control, environmental monitoring, and infrastructure optimization, can be made possible through the use of WiFi.
WiFi enhances the entertainment and hospitality industry by providing seamless access to digital content, streaming services and interactive experiences. Hotels, resorts, theme parks and entertainment venues are deploying WiFi networks to provide guests with high-speed Internet access, in-room entertainment, mobile applications and concierge services. WiFi connectivity increases guest satisfaction, engagement and loyalty, driving revenue opportunities and brand differentiation in the competitive hospitality market.
In summary, the use of WiFi permeates every aspect of modern society, empowering individuals, businesses and communities with wireless connectivity, communication and innovation. As technology continues to evolve, WiFi will remain a critical enabler of connectivity-driven experiences, digital transformation and societal progress in the digital age.

3.9.3 Function

3.9.4 Advancements

3.10 Libraries

3.10.1 Catch2

Author: Maximilian Dragosits
Catch2[15] 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, shows the simplicity of this framework and its integration into projects.

```
#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]" ) {
```

[15]**catch2˙git**.

```
        REQUIRE( factorial( 1) == 1 );
        REQUIRE( factorial( 2) == 2 );
        REQUIRE( factorial( 3) == 6 );
        REQUIRE( factorial(10) == 3'628'800 );
    }
```

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, which is then later during the running of the test printed alongside the results of the macro. Tags can be given to the test by inputting one or multiple tags into the *tags* field and change the behavior of the macro accordingly. 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.

```
#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);
    };
}
```

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.

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

As part of the BENCHMARK macro a arbitrary name is given to it, which is then later during the output of the test used as a identifier for the specific benchmark. Then the actual logic of the benchmark is then defined within 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 doesn't mess with the test.

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:** The amount of times 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[16] 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 online through an HTML website or offline 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 image shows how Doxygen's integration simplifies and enhances the documentation workflow, contributing to the clarity and accessibility of the project.

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, footers for the following document or images to be used in it. 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:

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

[16]**doxygen˙main˙site**.

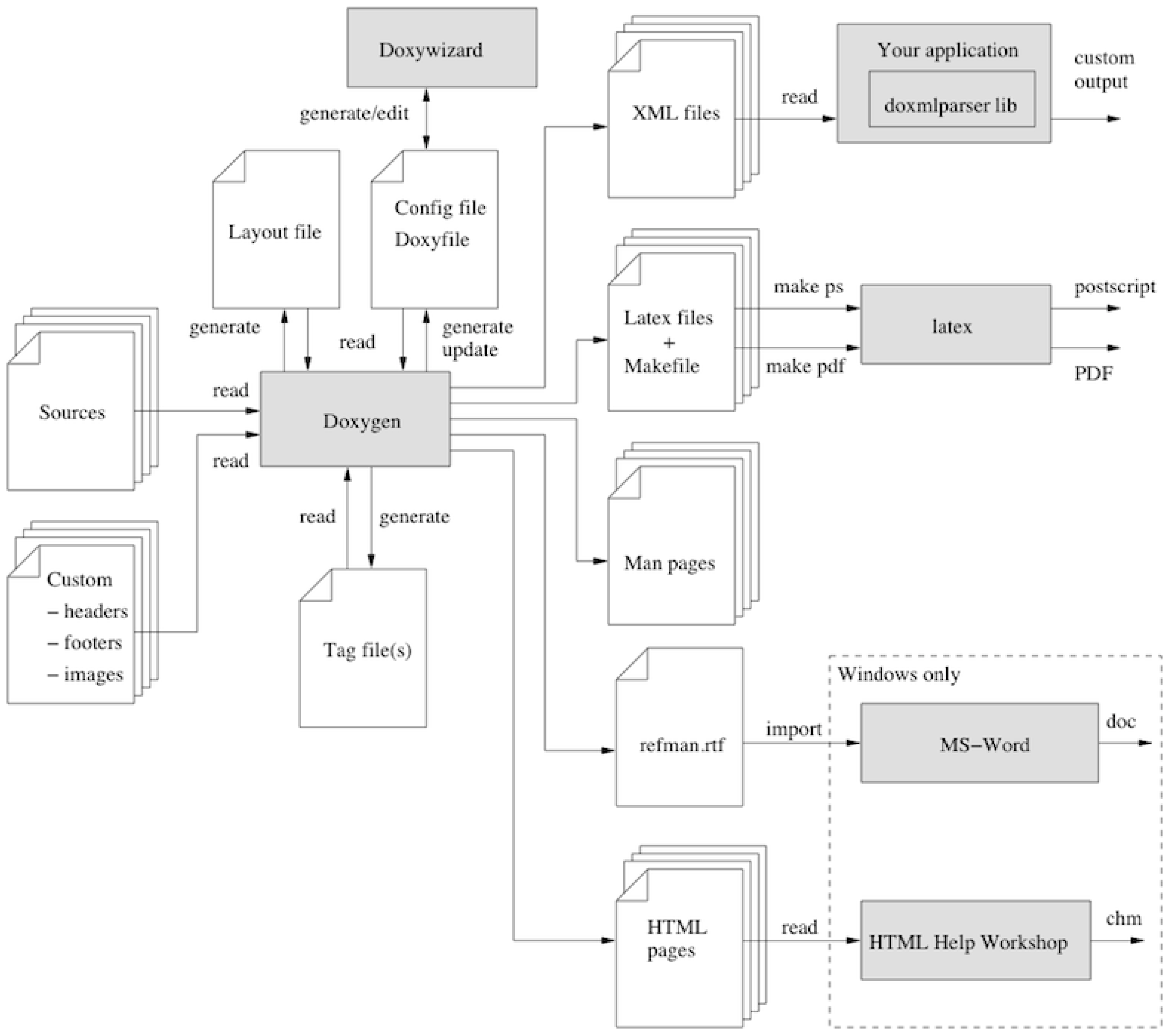

Figure 3.1: Doxygen Infoflow diagramm[17]

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.

```
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 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 *.

```
/**
 *   ... 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:

```
/*! \fn int open(const char *pathname,int flags)
    \brief Opens a file descriptor.

    \param pathname The name of the descriptor.
    \param flags Opening flags.
*/
```

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.

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.

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.

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.

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.

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.

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 Rusqlite

Author: Christoph Fellner

Rusqlite[18] 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.

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.

[18]**rusqlite**.

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.

3.10.5 Serde

Author: Christoph Fellner
Serde[19] 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 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.

3.10.6 Tokio

Author: Christoph Fellner
Tokio[20] serves as a pivotal asynchronous runtime for Rust, addressing the intricacies of asyn-

[19]**serde**.
[20]**tokio**.

chronous 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(){
    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.

3.10.7 Tokio Rusqlite

Author: Christoph Fellner
Tokio Rusqlite[21] 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.

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.

3.10.8 Tonic

Author: Christoph Fellner
Tonic[22], 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.

[21]**tokiolite**.
[22]**tonic**.

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.

```
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.

3.10.9 Docker

Author: Christoph Fellner
Docker[23] 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 particularly 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

[23]**docker**.

Docker's standing as a cornerstone technology in contemporary software development and deployment practices.

Chapter 4

Implementation

4.1 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. In order to avoid premature optimization however, the first part of this section on the implementation of the CommLib will only cover the first versions of the code written to get the Library to work. After the first implementation there will be benchmarks and some profiling, in order to get a grasp on which aspects of the library need to be optimized. The second section will then cover how these results were incorperated into designing a more polished version of the CommLib.

4.1.1 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:

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

Listing 4.1: Setup Commands for CommLib

4.1.2 First Implementation

The first part of the implementation that was tackled was to create an abstraction layer over the existing protocols that RECT uses. in order to have a nice and clean interface to work with and to avoid having to implement each feature separately for the protocols. Of course, there was a bit of a problem with UDP because it is not meant to send structured data, so there is no feature parity between TCP and BLE. between TCP and BLE and UDP in this respect, and UDP is only used to send unstructured data streams. To encapsulate the structured and unstructured data sent over the and unstructured data sent over the common interface, there needs to be a way to convert the data, whether structured or not, into and from bytes in a way that is performant and has minimal overhead. and has minimal overhead. In order to meet the above requirements, the following structure has been implemented.

4.1.2.1 Messages and Packets

The first thing taken into consideration for designing a data and class structure that is able to be sent over all of the protocols is the MTU's of the different protocols. For TCP and UDP the MTU, or maximum transmission unit, is defined by the Maximum Segment Size Option (MSS), which is technically limited to 65535 bytes (64KB), but as defined in RFC 2675[1] an MSS value of 65535 is defined to be interpreted as infinity and to be determined by Path MTU Discovery[2]. For BLE the MTU is defined by the L2CAP and can be anywhere from 23 to 65535, but the packet is fragmented and recombined by the L2CAP for transmission, which practically makes it infinite.
Another requirement for the interface was to be able to encapsulate the mechanism of sending a request to a peer and receiving a response. To do this, a pair of messages was implemented. This pair of messages, aptly named Request and Response, contains the data needed to make this work. This means that the request contains information about what topic to call on the remote machine, the data needed to fulfil that call, and an identification of the calling process, while the response contains the returned data and the ID of the client to which it should be returned. This identification is used to avoid confusion as to which process the data should be returned to, for example, when multiple processes are requesting data from the same remote service.
The identification mechanism used in the above case of sending data to a single recipient can also be adapted to send messages to multiple recipients. The only adaptation required for the requesting and responding application to be able to send and receive broadcasts is to

[1] **rfc2675**.
[2] **rfc9293**.

create a proxy object on the receiving side that acts as a single receiver to receive the data for multiple connections and then forward that data to all subscribing processes.
The last use case covered by CommLib is the sending of a continuous stream of data. An example of this would be a sensor continuously sending data. To avoid the overhead of sending values over protocols used in CommLib, it was decided to require a separate interface, e.g. a socket, that is used only for streaming the described data. This means that when the receiving side is establishing a connection, it only needs to look at the identifier, e.g. the IP address, of the connected peer to decide which process to pass the data to, instead of reading a connection name from the packet sent, minimising the size of the packets that need to be sent.

4.1.2.2 The ConnectionManager

The most important component in the whole CommLib is the ConnectionManager. It is a singleton object, which as the name would suggest manages all of the needed and used connections that are requested by the client programs. Because the ConnectionManager is a singleton and because it must be able to handle being in concurrent execution environments the Rust borrow-checker has very special requirements for how it is accessed. To meet these requirements the static variable holding the ConnectionManager singleton has needs to be wrapped in several objects to ensure it is thread-safe, as well as making sure, that it is initialized and freed, when there are no references left to its smart-pointer.

```rust
pub static CONNECTION_MANAGER: Lazy<Arc<Mutex<ConnectionManager>>> =
    Lazy::new(|| Arc::new(Mutex::new(ConnectionManager::new())));
```

Which leads to the above code, which has specifies, that the ConnectionManager is wrapped in the following types.

- Lazy. This type allows any object held by it to be initialized only once and only when it is first called upon[3]
- Arc. Arc stands for atomic reference counter and is a type of smart-pointer used in rust, when pointers need to be thread safe[4]
- Mutex. The mutex ensures, that only one process at a time can hold a reference to the held object in order to prevent issues such as race-conditions[5]

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. But what functionality does the ConnectionManager provide?

- Setting a connection to a database, which is used to read the connections configured by the clients.

[3]**once-cell-lazy**.
[4]**rust-arc**.
[5]**tokio-mutex**.

- Receiving streams, requests and responses through any of the configured connections.
- Sending 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.

4.1.3 Profiling and Benchmarking

4.1.4 Polishing

4.1.5 Documentation

4.2 RECT Database

Author: Christoph Fellner

4.2.1 Why SQLite?

The choice of a database system is a critical decision in the development of any application, and for RECT, the decision to use SQLite as the backend database is grounded in a thoughtful consideration of specific requirements tailored to the nature of small controllers.

Given the diverse landscape of database systems, each with its unique set of advantages and drawbacks, a comprehensive evaluation of options is imperative. RECT's emphasis on catering to small controllers immediately guides the criteria for selecting a suitable database. Two key considerations emerge prominently: memory efficiency and self-containment.

Small controllers typically operate within constrained resources, making memory efficiency a paramount concern. SQLite, renowned for its lightweight nature and minimal memory footprint, aligns seamlessly with this requirement. Its design prioritizes efficiency, ensuring optimal performance even in resource-limited environments.

The self-contained nature of SQLite further contributes to its suitability for RECT's use case. Unlike some database systems that necessitate complex setup procedures and external dependencies, SQLite operates as a standalone, serverless database engine. This simplicity not only facilitates ease of use but also aligns with the desire to avoid intricate configurations. RECT benefits from a database solution that is straightforward to use and configure, enabling a seamless integration into the development workflow.

While the focus on memory efficiency and self-containment narrows down the pool of potential databases, SQLite emerges as an optimal choice that strikes a balance between these requirements. The careful consideration of these factors positions SQLite as a reliable and pragmatic choice for RECT, providing the necessary functionality without introducing unnecessary complexity.

In conclusion, the selection of SQLite for RECT's backend database is a result of a meticulous comparison of different options, with a keen focus on the specific needs of small controllers. The prioritization of memory efficiency, self-containment, and ease of use collectively affirm SQLite as the ideal database solution, ensuring optimal performance and simplicity in the context of RECT's development environment.

4.2.1.1 SQLite

SQLite[6] stands out prominently as a frontrunner when it comes to selecting a memory-efficient database, and for good reasons. As a small, fast, and serverless database engine, SQLite aligns perfectly with the requirements of applications like RECT, particularly those designed for small controllers.

The serverless nature of SQLite is a noteworthy feature. Unlike some database systems that necessitate a separate server process for operation, SQLite operates in a self-contained manner. This characteristic not only simplifies deployment and configuration but also contributes to its efficiency and suitability for resource-constrained environments.

Originally developed in the year 2000 by D. Richard Hipp for the US Army, SQLite has evolved into an open-source database engine. Its implementation in the C programming language renders it highly portable, allowing it to run seamlessly on a myriad of platforms. This broad compatibility makes SQLite an excellent choice for applications that need to be deployed across diverse environments.

The compatibility between SQLite and Rust is facilitated through the rusqlite library, providing a native and ergonomic interface for Rust developers to interact with SQLite databases. This seamless integration ensures that the benefits of SQLite, such as its speed and efficiency, can be harnessed effortlessly within Rust projects.

Moreover, SQLite offers the flexibility to store the database in memory, enabling rapid access and retrieval of data. The asynchronous access to SQLite databases further enhances its versatility, allowing applications like RECT to efficiently manage and interact with data in an asynchronous programming paradigm.

In essence, SQLite's combination of speed, efficiency, serverless operation, and broad platform compatibility positions it as an ideal choice for applications that prioritize memory efficiency. For RECT, SQLite, with its inherent qualities and compatibility with Rust through rusqlite, emerges as a robust and fitting solution for the backend database, contributing to the overall efficiency and performance of the application.

[6]**sqlite**.

4.2.1.2 PostgreSQL

Postgres[7], also known as PostgreSQL, stands as a widely adopted backend database for web applications and websites. Renowned for its robust features and scalability, PostgreSQL operates on a client/server architecture, where the database is managed by a dedicated server process. This architecture allows multiple clients to interact with the database concurrently, even when accessing the same data simultaneously. The inherent support for handling multiple clients makes PostgreSQL particularly well-suited for scenarios where asynchronous access to the database from multiple threads is crucial.

The origins of PostgreSQL trace back to the Berkeley Computer Science Department at the University of California in 1986. Initially named Postgres, the project evolved over time and eventually became known as PostgreSQL, although the colloquial shortening to Postgres is still common. In 1996, the project transitioned to an open-source model, relying on the contributions of a dedicated group of volunteers for maintenance and improvement. Operating seamlessly on major operating systems, PostgreSQL has garnered a reputation for its reliability and adherence to standards.

In the realm of Rust, PostgreSQL finds compatibility through the rust-postgres library, which provides a native interface for Rust developers to interact with PostgreSQL databases. The integration of Rust and PostgreSQL is further optimized for asynchronous programming through the tokio-postgres library. This library, built on top of Tokio, enhances the performance of asynchronous interactions with PostgreSQL databases via Rust, making it well-suited for modern, concurrent application scenarios.

The client/server architecture of PostgreSQL, coupled with its robust Rust libraries, positions it as a formidable choice for applications that require a backend database with support for concurrent access and asynchronous operations. While it may have started as a project nearly four decades ago, PostgreSQL continues to evolve and thrive, maintaining its relevance in the dynamic landscape of web development and database management.

4.2.1.3 MySQL

MySQL[8] has established itself as a heavyweight in the realm of backend databases, with giants like YouTube, Facebook, and Twitter relying on its robust capabilities. Widely utilized for storing data from web services, MySQL operates on a client/server architecture similar to PostgreSQL, featuring a singular server process managing the database and facilitating access for multiple clients.

The MySQL project traces its roots back to 1994 when Michael Widenius and David Axmark initiated its development. Initially conceived as a fork of the mSQL database, MySQL underwent a significant transformation and was eventually rewritten from scratch. Since 2010, MySQL has been under the development umbrella of Oracle. While the project remains

[7]**postgres**.
[8]**mysql**.

open-source, Oracle also offers an enterprise version of MySQL with additional features and support.

In response to concerns about the direction of MySQL's development under Oracle, the original developers embarked on a new venture called MariaDB. This project represents a fork of MySQL and maintains full compatibility with its predecessor. The availability of MariaDB provides users with an alternative that adheres to the principles of open-source development.

For Rust developers seeking to interface with MySQL, the mysql library, complemented by an extension called `mysql_async`, offers a convenient and native Rust interface. The `mysql_async` library, built on the Tokio framework, specifically caters to asynchronous client access, aligning with modern programming paradigms that emphasize concurrent and non-blocking operations.

In summary, MySQL's widespread adoption by major players in the tech industry underscores its reliability and scalability as a backend database. The project's history, marked by its evolution under different entities, has given rise to alternative options such as MariaDB. The existence of Rust libraries like mysql and `mysql_async` further enhances MySQL's accessibility and usability within the Rust programming ecosystem, enabling developers to seamlessly integrate MySQL into their applications.

4.2.1.4 Comparison

The table provides a concise comparison of key features among SQLite, PostgreSQL, and MySQL, shedding light on their architectural, compliance, support, and use-case distinctions:

Features	SQLite	PostgreSQL	MySQL
Architecture	File Based (Self-contained)	Client/Server	Client/Server
ACID Compliance	Always	Always	Only with InnoDB and NDB Cluster storage engines
In-memory Support	Yes	No	Yes
Editions	Community (Free) with option of pro support	Community with option of commercial support	Community, Standard, and Enterprise
Popular Use-cases	Low-Medium Traffic Websites, IoT and Embedded Devices, Testing and Development	Analytics, Data Mining, Data Warehousing, Business Intelligence, Hadoop	Web Sites, Web Applications, LAMP stack, OLTP-based applications
Key Customers	Adobe, Facebook, and Apple	Cloudera, Instagram, and ViaSat	GitHub, Facebook, and YouTube

This comparison underscores the diverse strengths and use cases of each database system. SQLite's file-based, self-contained architecture makes it suitable for low to medium traffic websites, embedded devices, and development environments. PostgreSQL, with its client/server architecture and robust feature set, caters to analytics, data mining, warehousing, and business intelligence needs. MySQL, available in various editions, is widely employed in web applications, LAMP stack environments, and OLTP-based applications. Each database system has its own unique advantages, making the selection contingent on the specific requirements and scale of the intended use.

After we looked into the three possiblyties mentioned above we created an enviroment to benchmark the different databases. We used docker to create a container for each database and then ran a benchmark test on each of them. The benchmark test was a simple test that inserted 1000 rows into a table and then read them again. The benchmark is purposefully simple, because we won't be using the database for complex queries. More about the benchmark tests can be found here.

4.2.2 Database Structure

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:

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?

4.2.3 Database usage

The RECT Database serves as a pivotal 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.

4.3 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 it responsible for the communication between the frontend and the backend of the RECT stack.

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 JSON file. The JSON file is read by another module of the Rust Interface. The module uses the serde library to read the JSON file. 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.

4.4 Rust Service

4.4.1 Documentation

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.

```
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.

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 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 library 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:

```
file(GLOB RectVOneConf "${CMAKE_CURRENT_SOURCE_DIR}/proto/conf.proto")
file(GLOB RectVOneListen "${CMAKE_CURRENT_SOURCE_DIR}/proto/listen.proto")
file(GLOB RectVOneSend "${CMAKE_CURRENT_SOURCE_DIR}/proto/send.proto")
file(GLOB RectVOneMessage "${CMAKE_CURRENT_SOURCE_DIR}/proto/message.proto")
set(PROTO_CONF_FILES ${RectVOneConf})
set(PROTO_LISTEN_FILES ${RectVOneListen})
set(PROTO_SEND_FILES ${RectVOneSend})
set(PROTO_MESSAGE_FILES ${RectVOneMessage})
```

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.

```
protobuf_generate(TARGET rect-conf-service LANGUAGE cpp)

protobuf_generate(
    TARGET
      rect-conf-service
    LANGUAGE
      grpc
    GENERATE_EXTENSIONS
      .grpc.pb.h
      .grpc.pb.cc
    PLUGIN
      "protoc-gen-grpc=${grpc_cpp_plugin_location}"
)
```

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 provided by the gRPC library. 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 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 and frustration, 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 seamless integration and functioning of the servers.

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* function from the gRPC library 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:

```
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 Protnumber 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.

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.

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. 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

4.6.1 Documentation

4.7 Implementation Comparison

Chapter 5

Tests

Author:

5.1 General Benchmarks

5.1.1 Throughput

5.1.2 Packet Loss

5.2 Optimization

5.2.1 Compression

5.2.2 Buffering

5.2.3 Serialization Formats

5.2.4 Database Benchmarks

Author: Christoph Fellner
RECT uses a register to store data about available connections and useable ports of other controllers. This register is stored in a in-memory database. In order to find the best database solution for our use case, we compare SQLite, PostgreSQL and MySQL. Using docker as testing environment we can easily compare the different databases.
In order to test the databases without any influences from outside, we used docker container for each database and their corresponding client program. The client program is written in rust and uses the corresponding rust librarys for the individual databases. However since SQLite is a file based database, we just used a volume to store the database file, together with the rust program in one container. The rust program connects to the database and executes the queries. The program measures the time it takes to execute the query and prints it to the console. Because we are using docker, we can easily track the resource usage of the database container during the query.

Chapter 6

Conclusion

Author:

6.1 Recap

6.2 Conclusion

6.3 Outlook

Index

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —

Breite = 100 mm
Höhe = 50 mm

— Diese Seite nach dem Druck entfernen! —