An Open Source Smart home Platform

Niklas Harnish

12/05/2023

Supervisor: Dr Amna Asif

B.Sc. (Hons) Computer Science

Number of words = 0
This includes the body of the report only

Declaration of Originality

Put some text similar to the following in here:

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted work, I consent to this being stored electronically and copied for assessment purposes, including the School's use of plagiarism detection systems in order to check the integrity of assessed work.

I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Name:

Date:

Abstract

Put your abstract here. You should create a short abstract (200 words at maximum) which is on a page by itself. The abstract should be a very high-level overview: for example 1–2 sentences on the aims of the project, 1–2 sentences on the kind of design, implementation, or empirical work undertaken, and 2–3 sentences summarising the primary contribution or findings from your work. The abstract appears in the front matter of the report: after your title page but before the table of contents.

what should go in here:

aims

design

implementation

findings and primary contribution

If you want to dedicate to someone in particular

Acknowledgements

General acknowledgements . . .
your supervisor, your family, your friends, . . .

Contents

List of Figures

List of Tables

todo:

- add definitions for some terms: Crates vs Libraries, Traits vs Interfaces, Methods vs Functions and how they will be used within this paper

1 Introduction

Will write a couple of words about the project here, similar projects, inspirations etc. Citation here to remember how to do it :)

1.1 Aims & Objectives

When researching available smart home technology, one major gap I came across was the availability of open source software. While options exist for someone interested in connecting their proprietary device to an open source platform (view), there was no solution for anyone looking to build their own device and then connect it to an open source hub. In fulfilling this goal, to build an open source platform for both devices and the hub they will connect to, there are multiple objectives that will need to be met along the way:

1. Create a Library and API (Application Programming Interface) for building smart home devices.

2. Build a Server with an API for the smart home devices to communicate with. This will act as a hub and will control clients connected to it.
   a) This API should be well documented, so a user can interact with the hub, without using the Library.

3. Create a frontend, which will be populated with devices currently connected to the smart home. It will also be used to control clients connected to the server.
   a) The API provided by the server for this frontend should also be easy to use, so the user can create their own frontend environment.

4. The code of all of the above should be hosted in a public repository, with instructions for how to build and use every component of the system.
   a) An appropriate license should also be selected for this repository, so the code within it can be copied or modified by third parties.
   b) This repository should provide important links and provide information on the inner workings of the system, to support interested parties.

1.2 Project Overview

Each bullet point below would give a small summary of the section

1. **Background Research**
2. **Design of the System & Technology Decisions**
3. **Implementation**
4. **Results**
5. **Conclusion and Reflection**

2 Literature Review

2.1 IoT System Architectures

Kamienski et al. describe a simple three layer architecture of an IoT system in [15]. Within this architecture, the top layer is the "Input System", from which any data that will influence the decisions of the IoT system will come from. Included in this are sensors, but also user facing interfaces. The second layer, known as the "Process System", is where any algorithms are run and system behavioral decisions are made. The goal of this layer is to gain an "improved understanding of the system where the data comes from" [15]. The bottom layer is the "Output System", which are where decisions made by the Process System will be enacted. This is often represented as the devices connected to the IoT system.

This three layer architecture is expanded upon by Bansal and Kumar within [6], where three more architectures are described which expand upon the ideas within the three layer architecture. They are however more specialized than the three layer architecture. The first of these is a "Middleware Based" architecture, which can take many forms, but is usually combined with another type of architecture, with a middleware layer. The different types are described in detail by Zhang et al. in [21]. The second is known as a "Fog Based" architecture, where certain tasks, usually those with less processing requirements, are calculated on device to reduce latency. More computationally expensive tasks are however calculated on a server in the cloud [18].

The most relevant architecture to this report is known as a "Service Based" architecture (SBA). The SBA is defined around the concept of the Service Oriented Architectural (SOA) style [13] of software design. SOA is defined by the Open Group Foundation as an "architectural style that supports service-orientation", where a service is a "logical representation of a repeatable business activity that has a specified outcome" [4]. Each service is a "black box" any device interacting with it. Other devices use interfaces and API endpoints to make requests to the service and receive a result. A SOA is composed of many different services. In SBA, services are used to offer device functionality using interfaces, often using web based concepts such as SOAP or REST APIs [9]. This allows devices with different capabilities and purposes to interact with the same system, allowing for an IoT system that is more flexible.

2.2 The Smart Home System

Sethi and Sarangi [18] define six components that need to be present within a social IoT setting. A social IoT system is defined as a IoT system where devices form relationships with other devices. While our smart home system will not be a social IoT system, some of these concepts are still of interest. These are:

1. ID: the device within the system needs to have a way of identifying it.

2. Meta-data: the device should have information regarding its form and purpose

3. Security Controls: the system should have some way of distinguishing between different users. It should also be able to distinguish what types of devices it can connect to or can connect to it.

4. Service Discovery: each device should be able to discover other devices connected to the system and what services they offer.

There are some specific constraints specific to Smart Homes. Reliability is a key concern, due to the lack of a trained professional being available to fix any issues that arise. This is contrast to more industrial IoT settings, where there might be someone to fix any issues that arise. Another concern is the security and privacy of the system. Due to smart homes inherently having access to sensitive data (due to their position in someone's home), one must ensure that the system is both ethically sound and secure. The issue of security is further discussed in Subsection 2.4.

> further add to this section, just not sure what yet

2.3 APIs and Web Interfaces

The book "Designing Web APIs" makes an important distinction about APIs that can often be forgotten by developers. "Although APIs are designed to work with other programs, they're mostly intended to be understood and used by humans writing those other programs" [14] (Chapter 1). Do to this reality, one must remember to design APIs appropriately. To help the API designer in doing that, there are multiple pre-defined architectural standards that they can use.

2.3.1 Request Response

"Request Response" APIs (RRA) expose their interface through a web server, to which clients can make requests. A client will request data and will receive a response from the server. Common formats for requests and responses include JavaScript Object Notation (JSON) and Extensible Markup Language (XML). [14].

One popular type of RRA is known as Representational State Transfer (REST). Two important properties of REST is that it is used in Client-Server scenarios and that every request is stateless [11]. This means that every API request from the client to the server, must contain all information required to complete that request, without the server storing any of that information. Instead, all state is stored on the client. While this constraint might seem strange, it makes any API implementing REST easily scalable, and potentially easier/faster to build. The downside being inherently increased network traffic, with less control application behavior. [11].

Another popular implementation of a RRA is known as the Remote Procedure Call Architecture (RPC). The key difference between RPC and REST is that RPC is about making an action on the server. In REST the client supplies the server with the information required to take an action, whereas using RPC the client tells the server what action to take. RPC APIs can usually express more nuance in their requests and are generally stateful. While RPC usually uses JSON or XML for requests and responses, there are multiple implementations, such as Google's gRPC and Apache Thrift, which do not. These are usually serialized and therefore consume less network traffic than non-serialized formats such as the aforementioned JSON [14].

2.3.2 Event Driven

Event driven APIs go in a different direction than RRAs. Instead of the client continuously requesting information from the server, the client registers with the Server once, then whenever there is an update the server sends the client a message notifying it of an update. This completely resolves the need for polling, the client continuously requesting updates from the server, which is often present in RAA API designs [14].

Web sockets are a type of Event Driven API that utilize a bidirectional TPC connection between server and client. Unlike the previously mentioned API styles, a connection on a web socket stays active until closed [5]. Due to the bidirectional TCP connections both client and server can send one another packets, even at the same time [14]. This is in contrast to REST and RPC protocols, where only the client can contact the server. This comes at the downside of scalability, as a server must maintain a connection with every device that is connected with the server. Additionally, there are also issues when a client is on an unstable connection, as web sockets expect a client to stay connected, with the client having to reinitiate the connection if it is dropped [14].

The final type of API that deserves a short mention is the Web Hook. Web hooks work in an unconventional manner, where the initiator of an exchange gives a URL to their own API endpoint. This URL is then used by the receiver. Whenever a new event for the initiator occurs, the receiver will send information about the event to the URL [14]. While at face value this may seem like an obvious solution in an IOT based environment, where devices are often waiting for an event from the central server, it makes less sense when one realizes that every device connected to this webhook will need to host some sort of HTTP server, to receive the API requests. This makes it unsuitable, especially in environments where IOT devices are low powered, embedded systems devices. Web hooks are often used in server to server communication, as in such a scenario they are trivial to set up (as servers will most likely already be setup to receive API requests) [14].

2.4 Security

Requires more research and testing to find a good solution, will be written and implemented later.

2.5 Networking

I don't think this section is very interesting, so I will probably not write about this in background, however it will be mentioned in the implementation, that I left it mostly up to the user

2.6 Open Source and Licensing

Not sure about this section either, perhaps speak about how some open source projects are managed and different possible licenses. This does not sound interesting either.

4

Add section about which one I chose: RPC, specifically gRPC

Remove this text and write this section

Remove this text

Remove this text

write another section about the general planned architecture of

3 Design

3.1 Technology Choices

This section will discuss choices that have been made throughout the project regarding technology used and justifications for their usage.

3.1.1 Rust

There were a few requirements when choosing an appropriate programming language for this project:

1. Performance: There are two aspects to performance within this project. Performance considerations and optimization are vital on IOT devices themselves, due to their limited on-board processing power. On the other hand, while performance on servers is definitely important, it is significantly easier to scale server-performance, by simply adding more servers (horizontal scaling) or by improving the hardware of any individual server (vertical scaling), than it is to improve performance of an IoT device. This is especially true of an IoT device that is already deployed.

2. Stability: Another important requirement when choosing a language is the stability of code written in the language. This does not necessarily mean that code written in any language is inherently unstable. This requirement is more of a consideration about if a language enables and encourages a programmer to write code that is memory-safe and handles errors correctly. This is important in an IoT environment, as devices are expected to run for long periods. What is the point of a security camera if it's software crashes every couple days, due to an obscure memory out of bounds error?

3. Security: While no language is inherently "hack-proof" or secure, there are ways a language can encourage behaviors that can lead to better outcomes in security. A blog-post by the "Microsoft Security Response Centre" states that 70% of all vulnerabilities assigned a CVE (Common Vulnerabilities and Exposures) each year are due to memory corruption errors [20]. In the post the languages "C" and "C++" are specifically referred to as being part of this problem. As mentioned in subsection 2.4, security is of particular importance in smart home systems, so ensuring a method or language that enables secure code is chosen is of particular importance.

4. Ease of Use and Comfort: While not particularly important in the final product, having a language that is easy to develop in can make the developer experience easier and can lead to faster iteration on ideas, perhaps leading to a better final result. That being said, developer familiarity with a language can more than make up this difference. A seasoned C++ developer will be able to iterate faster and produce a better product in C++, than if they are using an "easy" language, that they are not as familiar with.

After some deliberation, the language that was chosen for this project was Rust. While C/C++'s performance rivals and often surpasses Rust, the difference is often quite marginal [17], due to all three being compiled to machine code. What makes Rust different, is its headline feature, known as the "borrow checker". While the details of the borrow checker are out of scope of this paper, it can ensure that at compile-time, the code is memory safe. While there are ways to circumvent this (using the "unsafe" keyword), this has to be explicitly done. Due to the code being guaranteed memory safe at compile time, outside the aforementioned unsafe blocks, Rust code is known for its ability to run long-term without running into crashes. Additionally, Rust code is a popular choice for embedded devices, due to being able to compile without a standard library (this has to be enabled), giving it flexibility in a project such as this.

I have decided to use Rust for both the Server and Clients. While Rust might be a somewhat obvious choice for IoT clients, it is less so for servers. Due to the "borrow checker", while Rust might be safe, it is often said to be harder to write than traditional languages. This makes it more questionable as the primary language for the server, due to it being a less constrained platform (view performance section above). While servers are theoretically almost infinitely horizontally scalable, in practice this is often not the case, especially in a smart-home's case, as they have to fit in someone's home and generally should be affordable. Therefore, ensuring that the server can run on as many devices as possible, be it a Raspberry-Pi, or a modern desktop, is an important goal to strive for. The "difficulty" aspect of Rust can be largely counteracted by personal familiarity with the language. Additionally, having a stable server is very important, especially since all IoT devices will need to frequently communicate with this server, an aspect which Rust excels at. Rust also has a thriving community of libraries (known as crates), which using the language gives access to. For these reasons I have chosen Rust as the primary programming language of the Server and Client parts of this project.

Note that the words crate and library will be used interchangeably throughout this report, due to them being very similar concepts, but library being more familiar to non-rust users.

3.1.2 gRPC

gRPC is an RPC implementation released by Google in 2015. It uses Protocol Buffers (protobufs) as an interface definition language (IDL), to define services on servers, that clients can then call, such as any RPC library. The server runs a gRPC server and the client runs the gRPC client [12]. Protobufs are compiled to many different languages, with many different libraries available for these languages, that automate much of the process. In a performance comparison between REST, gRPC, websockets, GraphQL, gRPC came out ahead in many different metrics [16]. These include:

- inserting one value into a database
- fetching one value from a database
- fetching one hundred elements

in both native and containerized tests. In fact, gRPC was the most performant internet communication protocol in all metrics apart from memory usage.

Due to its performance and cross-language support I have chosen gRPC as the internet communication protocol for this project. The specific library used for this project is

known as "Tonic". Tonic is a Rust gRPC crate that includes both a gRPC server and client. It also utilizes "prost" to compile protobuf files into Rust code, without having to interface the protobuf compiler itself. All protobuf files are compiled as a compilation step of the server and client, eliminating the need for external build scripts (this is only partially true, view Sub Section 4.1.1).

3.1.3 Typescript & VueJS

While a Command Line Interface (CLI) frontend, written in Rust, will be made available, the main focus will be on the Graphical User Interface (GUI). To ensure that it can run on a variety of systems and is relatively easy to create, it will be web based, using Javascript at runtime. However, it will be written in Typescript. Typescript is a superset of Javascript, that compiles to Javascript and leaves no trace of types behind. Typescript provides a robust type-system, including, but not limited to [7]:

- Structural type equivalence, instead of Javascript's by-name type equivalence
- Types and concepts for object-based programming
- Type operators

All of these, while not guaranteeing that the program will be type-safe at runtime, help a developer design more robust and long term solutions generally associated with statically typed languages.

In conjunction with Typescript a web-development framework will be used. Web frameworks are libraries for Javascript that allow easier development of websites and web apps, often incorporating HTML (HyperText Markup Language) and CSS (Cascading Style Sheets) code into Javascript code. They also provide reactivity, meaning that if a variable changes in the code, that change can easily be reflected on the site. This can be done in most frameworks by simply using the variable in the HTML code, something that standard HTML does not support (methods of doing this differs between frameworks).

Most of the choice between different web-frameworks comes down to personal preference and familiarity with a framework. That being said there can be performance differences between different frameworks, that could make a difference on some systems. While there is a lack of formal experiments on framework performance, an informal experiment [8] showed that while there is a performance difference between different frameworks, it should not be the primary decision maker. When the difference when creating 1000 rows between vanilla JavaScript and VueJS is 32 milliseconds, the disparity will not be noticeable to the end user. For this reason, and personal familiarity with the framework, I have decided to use VueJS as the frontend development framework for this project.

3.1.4 Raspberry Pi

write this

3.2 System Architecture

The overall system architecture can be seen in figure 3.1. This system architecture consists of two main components, the server and the client. Here the server represents the main controller of the system. The client is any device connected to the system (there most likely will be multiple). One peculiarity, is the fact that the frontend is contained by the server part of the system. This is because, in the current design, by default the frontend

Figure 3.1: System Architecture
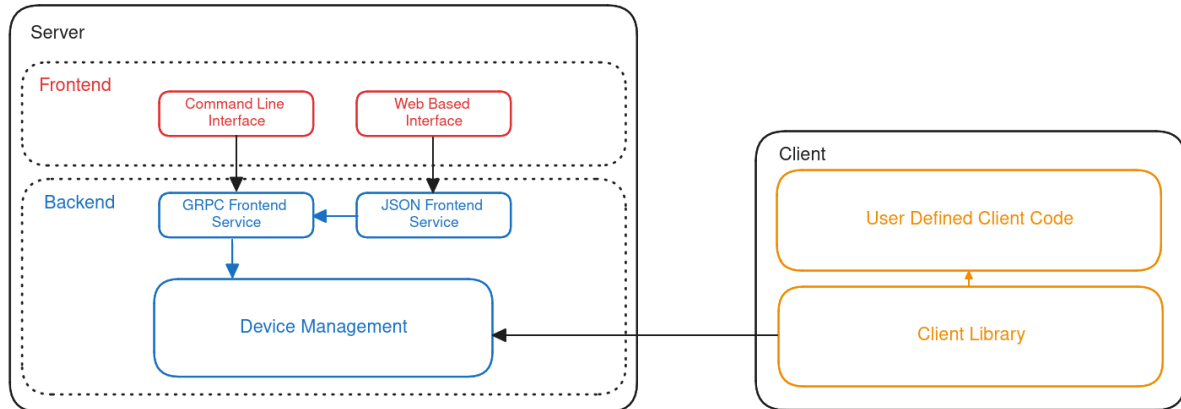
will be run on the same machine as the backend, with frontend API endpoints (located in the frontend services) being open on the localhost loop back address. This can however be easily changed by a user, by simply changing the address the frontend API endpoints are hosted on.

The arrows in the architectural diagram represent the direction of contact. For example, a client can contact the backend and the backend may respond, however the backend cannot contact a device, it can only reply to a request. The only deviation from this rule is within the client. At startup, the user will supply the client library with callback functions, which the library can call. This means that the user defined code must be able to contact the client library during setup. However, once the client library has finished setup and has created contact with the backend, the user defined code can no longer interact directly with the client library. The library will then simply call any code supplied to it during setup, without having any direct contact with user defined code.

3.3 Security

As previously discussed, security is an important concern within a smart-home environment. The user of the system is putting trust into the system to behave as it is meant to, while keeping their personal data safe from outside intruders. That is why special care will be put into the security aspects of the system. This will come in a two pronged approach using certificates and signatures.

3.3.1 Certificates

A certificate is a tool for verifying that a client is who they say they are. It is generated by the server and can be verified by the server. This is useful, for example, if the client goes offline and at a later time wants to re-aunthenticate with the server. They can send the certificate provided to them earlier and the server can verify it. The certificate scheme that will be implemented is based on [10].

Figure 3.2 shows the certificate exchange protocol that will be used, where cK is the client key pair and sK is the server key pair. K(pub) represents the public key part of the key pair.

Figure 3.2: Certificate Exchange
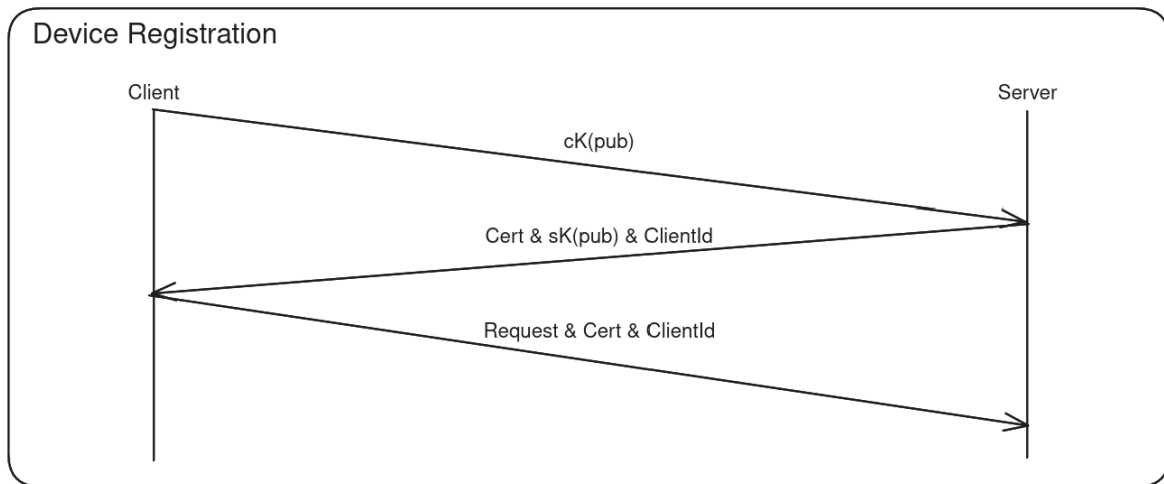
At startup the server and client will both generate an asymmetric key pair using the Rivest, Shamir, Adleman (RSA) public key system. Then, during registration, the client will send the server their public key. The server then creates csr where $csr = cK(pub) + ClientId$. Csr is then hashed using SHA256 hashing to create the certificate: $certificate = H_x(csr)$. This certificate is then sent back to the client, along with the server public key and their client identifier (which is a UUID randomly generated by the server). Now, if the client goes offline and wants to re-register, instead of having to repeat the registration process again, they can simply include their certificate instead and be verified by the server. Verification is quite simple. The client simply re-generates csr using the client's public key and client-id, then re-hashes them. If the new hash is the same as the certificate, then the client is verified.

3.3.2 Signatures

Signatures allow both the server and client to verify that the messages they are receiving are not forged and sent by a third party. The signatures will use the key-pairs generated and exchanged during registration (view certificate exchange), to sign every message with a hash, that can be independently verified by the other party. The message sender will use their private key to sign their message and the receiver will use the public key send during registration to verify the message. This signature scheme is based on concepts described within [10] and [19].

Every signature will consist of three important parts:

1. Unix Timestamp - This will be used to prevent replay attacks. Every message will include the Unix timestamp of when it was sent and this information will be included in the signature. The receiver can then check the age of the message and discard it if it is beyond a certain threshold.

2. Message Contents - This is used to ensure that the contents of the message stay the same between sender and receiver. Because hashing the entire contents of a message might be costly, some subset of the contents must be chosen. This will depend on the message type.

3. Certificate - The certificate is included in the hash, however not in the message. This is an extra layer of security, as both server and client have an independent version of the certificate, meaning that an attacker would also need to obtain this certificate somehow, along with the private key of the sender.

Every message will include a signature, which must first be verified by the receiver before the information within the message is processed. This is help ensure that data can not be tampered with between sender and receiver.

3.4 Web Frontend

Figure 3.3 shows a mockup what the home-screen of the web frontend should look like. The main focus should be on the devices, which will be represented by boxes on the home screen. The device's available capabilities will be shown as buttons or sliders, configurable from the device itself. As the focus of this project is mainly on the system and the API of the device server will be available, custom web frontends should be easy to make, for a more customizable experience. Due to the whole project being open-source, this frontend can be built upon by other users, or companies, interested in customizing it.

Figure 3.3: Home Screen Mockup
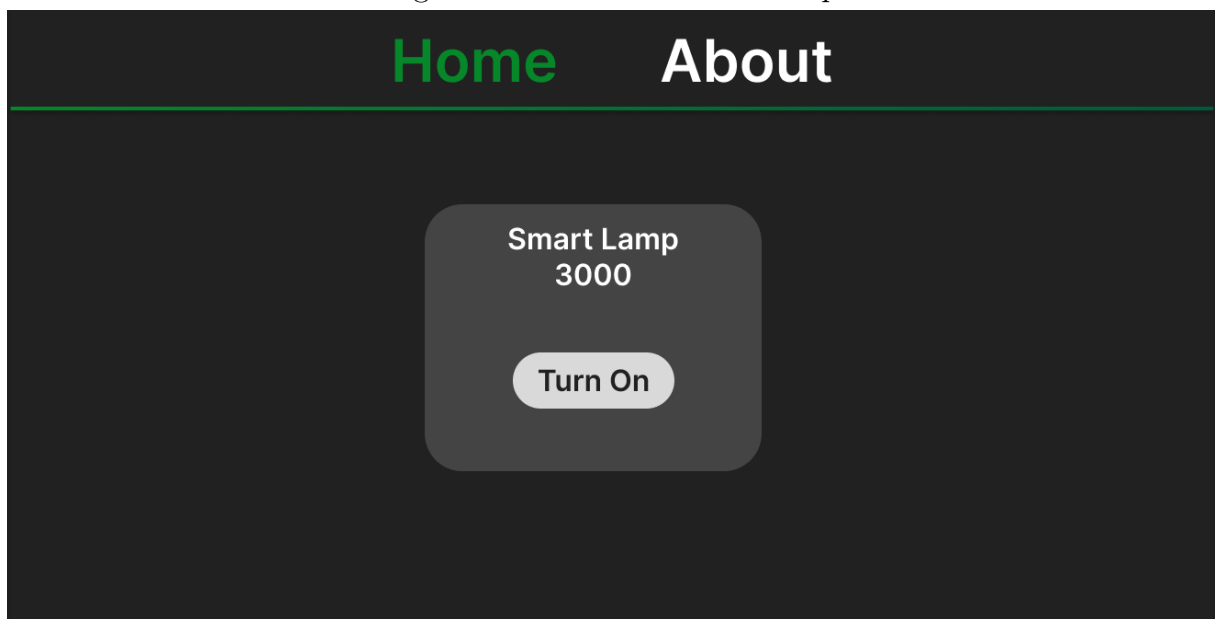
4 Implementation

4.1 Backend Server & Device Management

Due to the server being the largest part of this project, and practically being a requirement for testing the client and frontend functionality, this will be focused on first. This section will describe defining and compiling protobuf files for API endpoints, server startup, how IOT devices are registered, security mechanisms and how concurrency was handled within the server.

4.1.1 Protocol Buffers and Tonic

Protobufs are defined within ".proto" files. They use a fairly basic syntax, where the user can define a service using the "service" keyword. A service can contain many functions which a client can call, these are defined using the "rpc" keyword. You can also define the arguments this function takes and what it returns. Structs are defined using the "message" keyword, where each field is separated by a semicolon. Fields can have multiple modifiers, including optional, repeated and map. Repeated marks a field as possibly being a list, or array structure. Map marks a field as being a map data structure. Finally, messages from other files can be imported, using the import keyword and then giving the file-name [3]. For an example of a protobuf file view ***protos/iot/registrationService.proto***:

```
1  syntax = "proto3";
2  package iot.registration;
3  import "types.proto";
4
5  service RegistrationService {
6      rpc Register(
7          RegistrationRequest
8      ) returns (RegistrationResponse);
9  };
10
11 message RegistrationRequest {
12     string public_key = 1;
13     string name = 2;
14     repeated iot.types.DeviceCapabilityStatus
15         capabilities = 3;
16 }
17
18 message RegistrationResponse {
19     string public_key = 1;
20     string client_id = 2;
21     string certificate = 3;
```

I think this section is probably fine, just talk some more about what these protobufs and method stubs will be used for

```
22  }
```

Here we define a service called "RegistrationService", with a function called Register. This function is called by IOT clients when they first attempt to connect to the server which takes a RegistrationRequest as a parameter and returns a RegistrationResponse. The file also defines two "messages", the aforementioned RegistrationRequest and RegistrationResponse. While most of the fields within them are fairly obvious, RegistrationRequest.capabilities warrants further explanation. This field is repeated, meaning it is an array data structure. This array contains the type DeviceCapabilityStatus, which is defined in iot.types. We can see this type is imported at the top of the file, from "types.proto". This is the definition of DeviceCapabilityStatus:

```
1  message DeviceCapabilityStatus {
2      bool available = 1;
3      string capability = 2;
4  }
```

In summary, if an IOT device wants to register with this server, they will need to call the Register server stub. The server stub takes one parameter, the RegistrationRequest message. This message requires the client to give it's public key, it's display name and an array of "DeviceCapabilityStatus". The Register function then returns a RegistrationResponse. For more information on this topic, view subsection 4.1.3

To compile these server stubs to Rust we will be using the aforementioned Rust "Tonic" crate. To do this first we must add the "Tonic" crate to our project, then add "tonic-build" to our build dependencies (view Cargo Documentation). We then create a build.rs file, which is run when the Rust compiler (Cargo) builds the program. You can view the file **backend/build.rs** below:

```
1  fn main() -> Result<(), Box<dyn std::error::Error>> {
2      tonic_build::compile_protos(
3          "../proto/iot/types.proto"
4      )?;
5      tonic_build::compile_protos(
6          "../proto/iot/registrationService.proto"
7      )?;
8      tonic_build::compile_protos(
9          "../proto/iot/requestUpdateService.proto"
10     )?;
11     tonic_build::compile_protos(
12         "../proto/iot/deviceControlService.proto"
13     )?;
14
15     tonic_build::configure()
16         .type_attribute(
17             ".",
18             "#[derive(serde::Deserialize, serde::Serialize)]"
19         )
20         .compile(
21             &[
22                 "../proto/frontend/registrationService.proto",
23                 "../proto/frontend/frontendTypes.proto",
```

```
24                 "../proto/iot/types.proto",
25                 "../proto/frontend/deviceControlService.proto",
26             ],
27             &["../proto/frontend", "../proto/iot"],
28         )?;
29     Ok(())
30 }
```

In line 2-13 we are telling tonic_build, using prost under the hood (which in turn calls the protobuf compiler), to compile the files required for Device & Server communication. We do this by calling the *compile_protos* function. The files in the folder *proto/iot* are the ones used by IOT devices. This is quite simple to use, however is technically a build script, making the statement in subsection 3.1.2 partially untrue. Line 15-28 are where frontend protobuf files are compiled. These require the traits (Rust equivalent to interfaces in other languages) "Serialize" and "Deserialize" to be implemented for all structs compiled from the files. We therefore require this special syntax when compiling them. For more information on why they need these traits, view subsection 4.3.3.

Note that the words crate and library will be used interchangeably throughout this report, due to them being very similar concepts, but library being more familiar to non-rust users.

Finally, protobuf server-stubs and messages can be imported into Rust using a macro provided by tonic. View below how ***backend/src/server/registration.rs*** imports the *protos/iot/registration.proto* file:

```
1 use crate::types::types;
2 pub mod registration_service {
3     tonic::include_proto!("iot.registration");
4 }
```

In order to import the compiled protbuf file, we simply call the "include_proto" macro (it being a macro is indicated by the **!**). Note that we need to import the types module first, as this module contains the compiled *protos/iot/types.proto* file, which is imported by *registration.proto*. Additionally, instead of giving the file path to the protobuf file, we simply give the module name, which is also defined in the protobuf file.

4.1.2 Server Start Up

This section will describe what is done when the server first starts. It will give a general overview of startup and the details of starting a gRPC and HTTP server in Rust. For more details, view the main function within ***/backend/src/server/server.rs***.

Startup of the server is currently quite simple:

Command Line Arguments

Command line (CLI) arguments are parsed using a Rust library called "Clap". Currently the only commandline argument that can be passed is running the optional HTTP server, which enables using JSON for the frontend (view subsection 4.3.3), however there are plans to include other arguments, such as giving an IP address or Port for the gRPC server to run on as an argument. Currently, the computer's IP address is discovered automatically using a crate called "local-ip-address".

Figure 4.1: Simplified Diagram of Server Startup
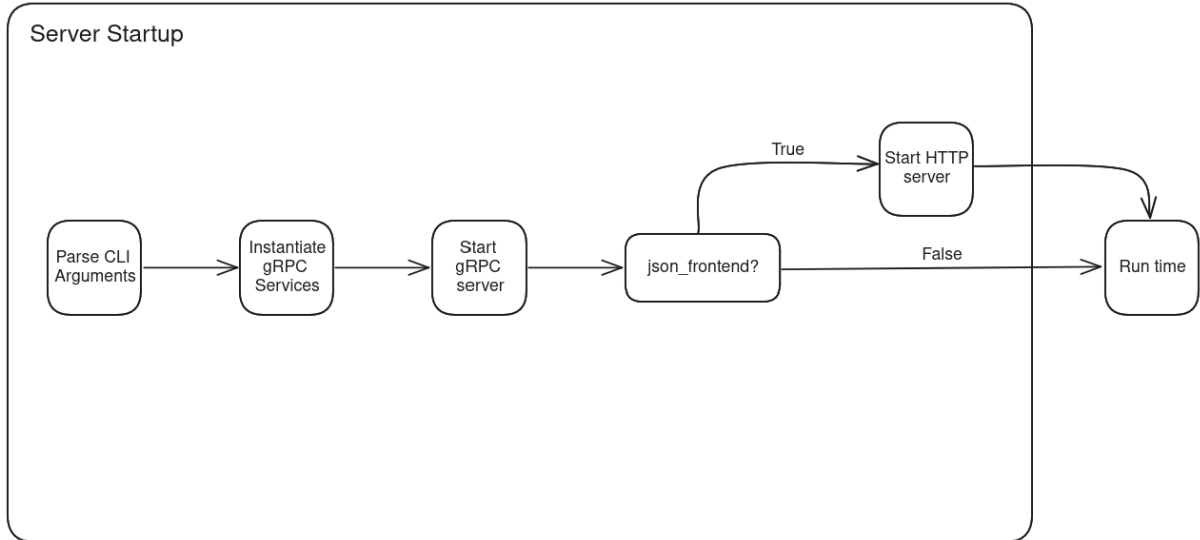
Creating gRPC Services

Next gRPC services are instantiated. GRPC services in Rust take the form of structs that implement the trait (interface) defined as a service within a protbuf file. Take for example, the "RegistrationService" from *protos/iot/registrationService.proto*, defined in subsection 4.1.1:

```
1  service RegistrationService {
2      rpc Register(
3          RegistrationRequest
4      ) returns (RegistrationResponse);
5  };
```

This is implemented on a struct in Rust like this (code snippet from *backend/src/server/registration.rs*):

```
1  use self::registration_service
2      ::registration_service_server::RegistrationService;
3
4  #[async_trait]
5  impl RegistrationService for ClientRegistrationHandler {
6      async fn register(
7          &self,
8          request: tonic::Request<
9              self::registration_service::RegistrationRequest
10         >,
11     ) -> RPCFunctionResult<
12         self::registration_service::RegistrationResponse
13     > {
14         //code goes here
15     }
16 }
```

14

Where "RegistrationService" is the trait we are implementing and "ClientRegistrationHandler" is the name of the struct it is being implemented on. If the trait is implemented correctly, this struct is now a service and, after being handed to the gRPC server struct, this function can then be called using an RPC through gRPC. In other words, any gRPC client that is connected to this gRPC server can now call this function, as long as their programming language supports it.

Starting the gRPC Server

After implementing all services defined in the protobuf files on appropriate structs, the gRPC server can now be started. This is done in the following code snippet from */backend/src/server/server.rs*:

```rust
1  use registration_service_server::RegistrationServiceServer;
2
3  let registration_service =
4      registration::ClientRegistrationHandler::new();
5  let grpc_server = tonic::transport::Server::builder()
6      .add_service(
7          RegistrationServiceServer::new(
8              registration_service,
9          )
10     )
11     .serve_with_shutdown(
12         grpc_address,
13         tokio::signal::ctrl_c().map(drop)
14     )
15     .await;
16
17 println!("Started GRPC Server on {}", grpc_address);
```

Note that the above code snippet is abbreviated for readability, it is however still valid Rust code and gives a good representation of what is done in server.rs.

The Server struct we are instantiating on line 1 comes from the transport module within the tonic crate. We invoke the builder method on the Server struct, a very common pattern within the Rust ecosystem. In fact, the same pattern is used instantiate a new IoT device in the client library for this project (view section 4.2). After invoking the builder method, which returns a new Server struct, we add structs that implement the aforementioned services to it, using the "add_service" method. In our case, we want to hand it a "RegistrationServiceServer" struct, provided by the protobuf file. The "new" method on the "RegistrationServiceServer" takes one argument, which is a struct that implements the "RegistrationService" trait, which is the trait we implemented on our ClientRegistrationHandler earlier. We therefore hand it the variable "registration_service" which is of the type "ClientRegistrationHandler".

The "server_with_shutdown" method consumes the Server struct and runs the server, on the address handed to it in the parameter. In this case it is the variable "grpc_address", which is a string which contains the device's IP address and a port. The second parameter simply tells Rust to drop the gRPC server (shutdown and gracefully free the memory associated with it) when the key combination crtl-c is pressed. This is an easy way to implement this behavior in Rust when working with multiple threads (view subsection 4.1.5).

Once we have added all services to the Server struct, it must be awaited using the "await" keyword. For more information on how this works view subsection 4.1.5.

Starting the HTTP Server

Starting the HTTP server works almost the same as the gRPC server, using the same builder pattern. However, this time instead of using the Tonic crate, we are using a crate named "Actix", a self-described "powerful, pragmatic, and extremely fast web framework for Rust" [1]. View part of the "run_json_frontend" function, found in the file *backend/src/server/server.rs* below:

```
1  const JSON_ADDRESS: &str = "localhost:50052";
2
3  let result = actix_web::HttpServer::new(move || {
4      actix_web::app::App::new()
5          .app_data(web::Data::new(json_state.clone()))
6          .service(json_registration::json_registration)
7  })
8  .bind(JSON_ADDRESS)?
9  .run()
10 .await;
```

Note that the above code snippet is abbreviated for readability, it is however still valid Rust code and gives a good representation of what is done in server.rs.

Due to this builder function requiring more Rust specific knowledge to understand, it will be simplified here. It is functionally the same as the one seen from Tonic. First we create a new HTTPServer struct from the actix_web module. We then hand it our services using the "service" method. What is different here, is that we are actually handing it callback functions, instead of structs with traits implemented. To view an example of these services, view subsection 4.1.3. From there we bind this server to the constant string JSON_ADDRESS. This is an IP Address defined at compile time, the definition however is found above the web server. The server is running on localhost, as it is meant for communication with the web frontend. We then call the run method and await the result. In reality the server is being awaited on a different thread than the gRPC server, so the two can run concurrently. This is not shown in the above code snippet.

For code beyond the above provided abbreviated snippets view the file *backend/src/server/server.rs*, which contains the main function for the server.

4.1.3 Device Registration

This section will give an overview of the server's response to a device registration attempt. For information about how registration is performed on a device, view subsection 4.2.2

Figure 4.2 shows the process of the server receiving a registration request from a client:

When a device wants to connect to the server, it uses the "Register" function from the "RegistrationService" through gRPC. To do so, it must first create a "RegistrationRequest", the contents of which are detailed within the *proto/iot/registration.proto* protobuf file:

Figure 4.2: Simplified Diagram of Device Registration
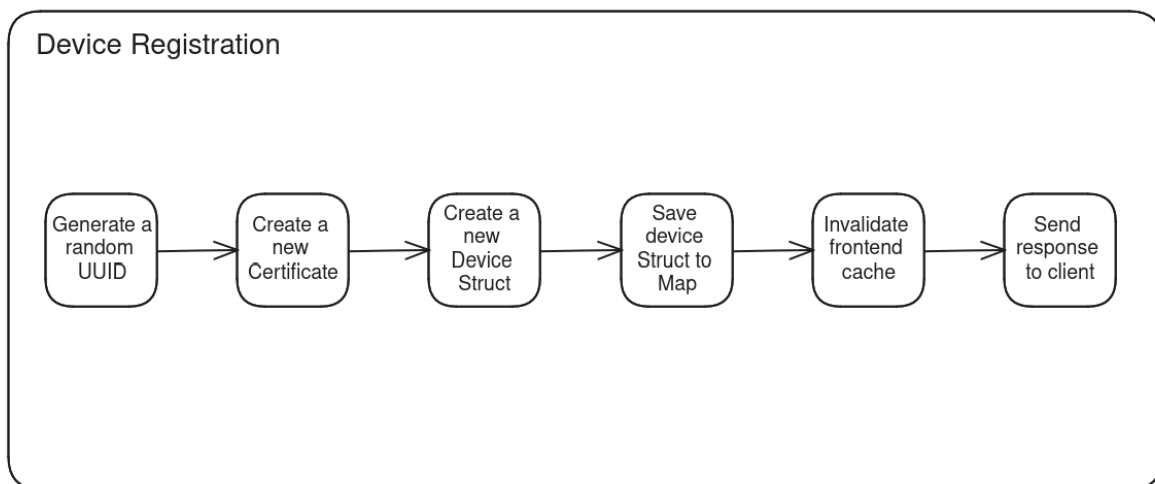

```
1  message RegistrationRequest {
2      string public_key = 1;
3      string name = 2;
4      repeated iot.types.DeviceCapabilityStatus
5          capabilities = 3;
6  }
7
8  //from proto/iot/types.proto
9  message DeviceCapabilityStatus {
10     bool available = 1;
11     string capability = 2;
12 }
```

This is the same protobuf code detailed in subsection 4.1.1. The "RegistrationRequest" contains the device's public key and its display name. The public key will be used for encryption and signing, the name is what the device is called on the frontend-interface. The final field is an array of type "DeviceCapabilityStatus", which is a message (struct) with two fields. A capability in this system represents something the device can do. This takes the form of a string. If a capability is available, that means the server relays that capability to any frontend devices connected to the server. If it is not available, the server still keeps that information, but does not relay it to frontends.

A simple example of a capability is one for a lamp. A lamp could have two capabilities, "turn on" and "turn off". If the lamp is off, the capability "turn on" is available to the user. If the user decides to invoke that capability, the lamp will turn on, the "turn on" capability will no longer be available and the "turn off" capability will become available.

This system has much room for extension, with some glaring features that need to be added being:

- Optional variables attached, such as a float value. Could be used for turning up a speaker for example.
- Access levels, only an admin can see this capability.

still planning to implement this before deadline, add more stuff here

17

The device sends this registration request to the server. With the information from the request, the server can then register the device in a HashMap, which maps the device UUID to the Device struct, following the steps described in figure 4.2. The Device struct is defined in */backend/src/server/device.rs* as:

```rust
use crate::types::types::DeviceCapabilityStatus;
#[derive(Clone)]
pub struct Device {
    pub name: String,
    pub uuid: Uuid,
    pub stringified_uuid: String,
    pub active_capabilities: Vec<DeviceCapabilityStatus>,
    pub inactive_capabilities: Vec<DeviceCapabilityStatus>,
    pub device_public_key: rsa::RsaPublicKey,
}
```

The structure of a device is quite self-explanatory, the only thing of note is having the UUID as two fields, one time as the UUID struct and once as a string. This is due to the type conversion being done very often in certain sections of code, it made sense to do this conversion once and reuse it. Note that in the current system, a device needs to re-register every time the server restarts. It doesn't need to re-register on device restart, as long as the certificate and UUID are retained. An easy way for the server to save devices, so they can be remembered after restarting would be to serialize the hash map which maps UUIDs to Device structs and then to load this at startup. However, this would require more advanced logic to decide when to kick inactive devices out of the hash map and could lead to unforeseen bugs. Therefore, it is another possible extension of the project for the future.

4.1.4 Device & Server Communication

Device and Server communication happens through polling. Every 0.5 seconds the device sends a request through the "RequestUpdateService" to the server. The service is defined in the */protos/iot/requestUpdateService.proto* file:

```proto
service RequestUpdateService {
    rpc PollForUpdate(PollRequest) returns (PollResponse);
};

message PollRequest {
    string certificate = 1;
    string uuid = 2;
    repeated iot.types.DeviceCapabilityStatus
        updatedCapabilities = 3;
};

message PollResponse {
    PollingOption hasUpdate = 1;
    repeated Update updates = 2;
};

```

```
17  enum PollingOption {
18      UNKNOWN = 0;
19      NONE = 1;
20      SOME = 2;
21      DEVICE_NOT_FOUND = 3;
22  }
23
24  message Update {
25      string capability = 1;
26      bool hasValue = 2;
27      int32 value = 3;
28  }
```

The "RequestUpdateService" has one function, named "PollForUpdate". It accepts a "PollRequest" as a parameter and returns a "PollResponse". When a device requests an update from the server, it must provide it's UUID and certificate, so the server knows what device is requesting and so the server can ensure that the device is who it says it is. Additionally, it provides it's updated list of capabilities. Below is some pseudocode to show how the server creates a response (you can view the real function in file **back-end/src/server/polling.rs**):

add signature to this

```
1   def poll_for_update(self, request):
2       device = self.connected_devices.get(request.uuid)
3       if device == null:
4           return PollResponse(
5               has_update: PollingOption::DeviceNotFound,
6               updates: [],
7           )
8
9       if !request.updatedCapabilities.is_empty():
10          active_capabilities, inactive_capabilities = (
11              request.updatedCapabilities.partition(
12                  (capability) => capability.available == True
13              )
14          )
15          device.replace_capabilities(
16              active_capabilities,
17              inactive_capabilities
18          )
19
20          self.frontend_cache_valid = false
21
22      updates = self.updates.get(request.uuid)
23      if updates.is_empty():
24          return PollResponse(
25              has_update: NONE,
26              updates: [],
27          )
28
29      updates_clone = updates.clone()
```

```
30        updates.clear()
31
32        return PollResponse(
33            has_update: SOME,
34            updates: updates_clone
35        )
```

In summary, when the poll_for_update function receives a request, it first checks if the device has updated capabilities. This can happen if something has changed with the device since the last poll or since registration. If the device has updates capabilities, then they are replaced and the frontend cache is invalidated. The frontend cache being invalidated means that some values need to be recalculated next time a frontend requests information about this device, instead of simply using the previous values.

Next the function checks if any events are available for the device in question. Events in this case are generated by the frontend. For example, when the frontend activates capability "turn_on", an event is created on the server for the corresponding device. When the device then requests updates, this event is sent to the device and the device can respond accordingly. Within the PollResponse struct (view the protobuf definition above), the "hasUpdate" field corresponds to the status of events. If the server responds with NONE, this means that no events are available, if it responds with SOME then the updates field contains updates. While the client could simply check for the length of the updates array, "hasUpdate" also provides some error handling, such as reporting if the device was not found or if some error occurred.

When the client receives the response to their poll, they can then decide what to do with the events they have received, the server does not define any behavior. It simply reports to the client any events that have happened. This choice was made to enable a programmer to better define device behaviour themselves and prevents issues with the server and device being out of sync about what behaviors are available. The downside of this, is that the server cannot guarantee to the frontend that anything has happened when a button is clicked. It also cannot give feedback until the event is processed by the IoT device. Another downside is that it leads to more calculation being done on the IoT device itself.

4.1.5 Threads & Concurrency

write this

4.2 Device Library & Example Device

Another goal of this project was to create a library that can be used by other programmers, to easily create an IoT device that can connect to this system. This library was written for use with Rust. It is published within the Rust library ecosystem website, found at *crates.io*. The library entry on *crates.io* can be found be **here**.

4.2.1 Using the Library

The main purpose of the library to is to abstract details of how the client & server connection works away from the programmer. While they can still view the source code due to it being open source, they should not need to know the inner workings of the

some of this code has outdated code fix that

20

library to be able to use it. The example below, which can be found within the repository for the example device **here**, demonstrates usage of the library to create a simple IoT device. The purpose of this device is to toggle an LED, attached to a general purpose input output (GPIO) pin, on and off when the appropriate event is received. This code is made to run on a Raspberry Pi and will therefore not run on non Raspberry Pi Devices. It uses a combination of the NOSHP_Client library (the library created within this paper) and rrpal, a library commonly used to interact with the Raspberry Pi's GPIO in Rust. View the entire code for this simple IoT device and an in-depth explanation of it below:

```rust
use NOSHP_Client::{
    client::{ClientHandler, Request, State},
    client_config::{ClientConfig, ParsedConfig},
};
use std::error::Error;
use rppal::gpio::{Gpio, OutputPin};

struct ExampleState {
    led_pin: OutputPin,
}
impl State for ExampleState {}
impl Default for ExampleState {
    fn default() -> ExampleState {
        return ExampleState {
            led_pin: Gpio::new()
                .unwrap()
                .get(GPIO_LED_PIN)
                .unwrap()
                .into_output(),
        };
    }
}

const GPIO_LED_PIN: u8 = 2;
const CONFIG_PATH: &str = "./example_config.toml";
#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let config =
        ClientConfig::load_config(CONFIG_PATH).unwrap();

    let client_handler = ClientHandler::new()
        .add_callback("Turn On", Box::new(turn_on_led))
        .add_callback("Turn Off", Box::new(turn_off_led))
        .run(config)
        .await
        .unwrap();

    return Ok(());
}

```

```
41  fn turn_on_led(state: &mut ExampleState, req: Request) {
42      state.led_pin.set_high();
43      println!("set pin {} to high", GPIO_LED_PIN);
44  }
45  fn turn_off_led(state: &mut ExampleState, req: Request) {
46      state.led_pin.set_low();
47      println!("set pin {} to low", GPIO_LED_PIN);
48  }
```

State

On line 8 the state struct of the program is defined. State will later be shared between callback functions. This can be defined by the user to fit their needs. The only requirement is that the user defined state struct must implement two interfaces. The first is called "State" and is imported on line 2 from the NOSHP_Client library (our library). This State interface is currently empty, it has no required functions, it exists to make future additions to the library easier to integrate. If it is decided that a function is required on the state struct, it can be easily added to the interface and the user will get a compile time error that is easy to understand. The second interface is called Default. It simply defines the default implementation of the struct, in this case constructing a new "ExampleState" (the name of our State struct), with the "led_pin" field set to the appropriate GPIO pin (defined on line 24), in this case where the pin the LED is connected to. This will then later be used by callback functions (view Callback Functions), to turn on the LED, without having to construct a new GPIO struct every time.

Configuration

Next we will have a look at the main function. On line 28 the configuration for the client is loaded from a file. This file is defined in a constant on line 25, under *./example_config.toml*. The configuration files for the client library are written in Tom's Obvious Minimal Language (TOML), a language often used in Rust projects. Its usage is similar to languages such as JSON or YAML (Yet another markup language). The configuration file looks like this:

```
1  device_name = "Pi"
2  server_ip = "http://192.168.0.0:2302"
3
4  [capability."Turn Off"]
5      available = true
6
7  [capability."Turn On"]
8      available = true
```

This configuration is loaded using the "load_config" method from our library, imported on line 3.

The Client Handler

On line 31, we finally construct the client handler, by calling the "new" method on the "ClientHandler" struct, imported from our library on line 2. Calling the new method

initializes the variables inside the ClientHandler, including calling the default method on our State. This might be confusing to someone who is new to Rust, as we never informed the ClientHandler of the ExampleState struct. In this case Rust uses type inference, to infer what struct we are trying to use as our client's state. It infers this information from the callback functions we pass to it using the "add_callback" method, as they define one of their parameters to be "ExampleState". If we were to remove these calls to "add_callback", Rust would throw an error at compilation, stating that we need to specify the type of State. The definition of "ClientHandler" looks like this (from *client_library/src/client.rs*):

```rust
pub type Callback<S: State> = fn(&mut S, Request);
pub struct ClientHandler<S: State> {
    callbacks: FxHashMap<String, Box<Callback<S>>>,
    state: S,
    server_ip: Option<String>,
}
```

ClientHandler is a struct that accepts one generic argument, named S. S needs to implement the interface State (which in turn requires an implementation of the interface Default). "ClientHandler" has three fields:

- Callbacks - A hashmap used to map capabilities (stored as Strings), to user defined callback functions.

- State - Of type S, used to store the state of the program.

- Server_ip - Of type option, stores either a string representation of the server's IP address and port, or None. It has the option type, due to there being multiple ways of giving the ClientHandler the server's IP address, however the client will not start if server_ip is None when the "run" method is called.

Once the ClientHandler has been constructed, we use the builder pattern mentioned in subsection 4.1.2, a very common Rust pattern. There are a few methods currently implemented on the ClientHandler, which we can call using this pattern. The most common one is the "add_callback" function. We use this to give the ClientHandler a pointer (signified by the "Box" struct in Rust) to our function, along with the capability (defined in the config.toml file) it should be called for. If the user does not add a custom callback, the default callback is used instead, which looks like this (snippet from *client_library/src/client.rs*):

```rust
let callback = self.callbacks.get(&update.capability);
match callback {
    Some(v) => v(&mut self.state, request),
    None => println!(
        "Received signal to {}", update.capability
    ),
}
```

In summary, this snippet gets the callback from the callbacks hashmap defined on the ClientHandler, if the callback is defined by the user we call it, otherwise *"Received signal to Turn On"* (for the capability "Turn On") is printed.

The "Client Handler" has some other useful functions which the user can call. Some examples of these are:

- Set_state - Allows the user to set the state of the program to the non-default implementation. Can be useful if the state is determined programatically. This could allow the machine to change behavior at runtime. An example of this is having different behaviour depending on where the device is located.

- Set_server_ip - While the IP can simply be set in the config and read at runtime, it can instead of be passed using this function. This allows the user to programmatically determine the IP address, for example using network discovery to discover the IP address of the server, instead of using static IPs like in this example.

These methods can be useful, but are not required for using the library. Once all callbacks have been and other additional user parameters have been added, the run method is called. The run method accepts the configuration (obtained through the load_config function) for the device as a paramter and returns a future, which must be awaited using the await keyword (view subsection 4.1.5). This allows the ClientHandler to be used concurrently to any other tasks the client has to perform.

add a smaller blurb about what the ClientHandler does internally

Callback Functions

Finally, the callback functions, used in the "add_callback" method on the ClientHandler are defined. The ClientHandler struct accepts callback functions, that accept two parameters and return void (defined as having no return value in rust). The first parameter must be a Struct that implements the interface State, which the "ClientHandler" uses to infer the type of State in the above example. The second parameter must be of type Request. This is a simple struct that is currently not in use, but will be used in the future for the frontend to pass specific parameters to the client. For example, a slider's value on the frontend could be passed within the Request parameter. As mentioned, this is not implemented yet.

The functions in this case are fairly simple. They use the GPIO pin stored within the ExampleState to turn on and off the LED connected to the GPIO pin, using the methods defined in the rppal library.

4.2.2 Device Registration

While device registration is described in detail within subsection 4.1.3, how the library handles registration on the client side will be briefly touched upon within this section.

While the code for client registration section is significantly more simple than that of the server registration, it has again been translated to pseudo code for consistency. The real code snippet can be found within *client_library/src/client_registration.rs*:

```
1  def register_self(
2      public_key,
3      capabilites,
4      device_name,
5      server_ip
6  ):
7      client = RegistrationServiceClient::connect(server_ip)
8      registration_request = new RegistrationRequest(
9          name, public_key, capabilities
10     )
11
```

```
12      response = client.register(registration_request)
13      return new ServerConnection(
14          response.client_id,
15          response.public_key,
16          response.certificate
17      )
```

The RegistrationServiceClient is the counterpart to the RegistrationServiceServer used by the gRPC server and is imported from the generated code from the RegistrationService protobuf file. We use the client returned by the "connect" method on the RegistrationServiceClient, to call the RPC function "register" (view subsection 4.1.1 for more information on how RPC functions are defined). This then returns the fields required for communication with the server. Finally, we return the ServerConnection struct which is a basic struct from the client library defined as:

```
1  pub struct ServerConnection {
2      pub uuid: String,
3      pub server_pub_key: rsa::RsaPublicKey,
4      pub security_certificate: String,
5  }
```

which stores information required for communication with the server.

4.3 Web & CLI Frontend

This section will describe how both the Web and Command Line Interface (CLI) frontends were implemented. It will also discuss specific struggles that were encountered when attempting to create these, and solutions or workarounds to these.

4.3.1 Command Line Interface Frontend

The first frontend that was implemented was the CLI frontend. It allows a user to control any IoT devices connected to the server from a simple command line enviornment. I was also very useful for testing purposes, as it allowed me to test that the system was working from a simple interface.

Running

Running the CLI frontend is fairly simple. After compiling the binary from Rust with the compiler in release mode, simply run the binary with the flag "*–server-address*" (shorthand is -s), inputting the gRPC address of the server as the server-address argument. This is made easy as the server outputs the address the gRPC server is running on at startup. For example:

```
1  $ ./frontend -s 192.168.0.1:2302
```

will run the frontend if the server's IP address is *192.168.0.1:2302*.

Using the Interface

Using the interface is quite simple. At startup the user will be greated with two options:

```
1 Connecting...
2 Connection Successful
3 Welcome to Nik's Smart Home System
4 Your Device id is: 9404bec6-6a07-4374-bb34-f31e5809e348
5
6 What would you like to do?
7 1. Control a device
8 2. Quit
```

If they select *Control a device* (by entering the number 1), the frontend will fetch any devices connected to the server.

```
1 Fetching available devices...
2 0: Pi
3 1: Quit
4
5 What device would you like to control?:
```

Currently there is only one device connected to the server, with the name "Pi". If we select to control the "Pi", we will be met with the screen:

```
1 Heres what you can do:
2 0: Turn Off
3 1: Turn On
4 2: Quit
```

Under the hood, the frontend is fetching any capabilities defined within the config.toml being used by the device we are attempting to control. For more information how this works viewsection 4.2. If we select any of the capabilities, we will be met with the text:

```
1 Making request....
2 Operation was successful
```

If the request was unsuccessfully received, then an error message will be printed instead:

```
1 Making request....
2 There was an error:
3 status: Unavailable, message: "error trying to connect:
4 tcp connect error: Connection refused (os error 111)",
5 details: [], metadata: MetadataMap { headers: {} }
```

If the request was successful, then the next time the client polls the server, they will receive the event that selected and the callback function attached to that event will be called, in this case turning on or off an LED connected to the device. For more information on this view section 4.2.

Implementation

This section will give a description of how the CLI frontend functions and some design decisions made throughout.

The CLI frontend is relatively simple when compared to the rest of this system. All logic is contained in the file *backend/src/frontend/frontend.rs*. It uses gRPC to communicate with the server, using the same gRPC server as the devices. In fact, any device could also act as a frontend and vice-versa. The frontend API is defined within

protos/frontend. For example, the defintion for frontend definition is defined in ***protos/frontend/registrationService.proto*** and looks like this:

```proto
syntax = "proto3";
package frontend.registration;
import "frontendTypes.proto";

service FrontendRegistrationService {
    rpc Register(RegistrationRequest)
        returns (RegistrationResponse);
    rpc GetConnectedDevices(ConnectedDevicesRequest)
        returns (ConnectedDevicesResponse);
};

message RegistrationRequest {
    string device_name = 1;
}

message RegistrationResponse {
    string client_id = 2;
}

message ConnectedDevicesRequest {
    string client_id = 1;
}

message ConnectedDevicesResponse {
    repeated Device devices = 1;
}

message Device {
    string device_name = 1;
    string device_uuid = 2;
    repeated frontend.types.DeviceCapabilityStatus
        capabilities = 3;
}
```

The frontend gRPC API has two functions, "Register" and "GetConnectedDevices". The Register function is used to register the frontend with the server. The frontend will then assign it an identifier, with which it can make requests. The GetConnectedDevices function is used by the frontend when the user requests connected devices. The server will respond with the device name, it's ID and any capabilities the device has. The server only responds with capabilities that are currently available.

4.3.2 Web Frontend

The web frontend was built using a combination of Typescript and VueJS. It mainly serves as a proof of concept, minimum viable product web-frontend, as it is expected that most users would build their own frontend for their own needs. It also serves to demonstrate how a user can access and use the frontend API to build a web frontend.

Running

The web frontend can be run from the **vue_frontend** directory. Using the node packet manager (NPM), using the command: `npm install`. Once all dependencies have been installed, protobuf files must be compiled. This can be done using the bash script **vue_frontend/build_proto.sh**. From there the site can be hosted using: `npm run dev`, and can be accessed at localhost:5173.

Design

The final design of the frontend is fairly close to the original design proposed within figure 3.3. View figure 4.3 for the final design (note that for readability reasons it has been zoomed in considerably).

Figure 4.3: Web-frontend Home Screen Final
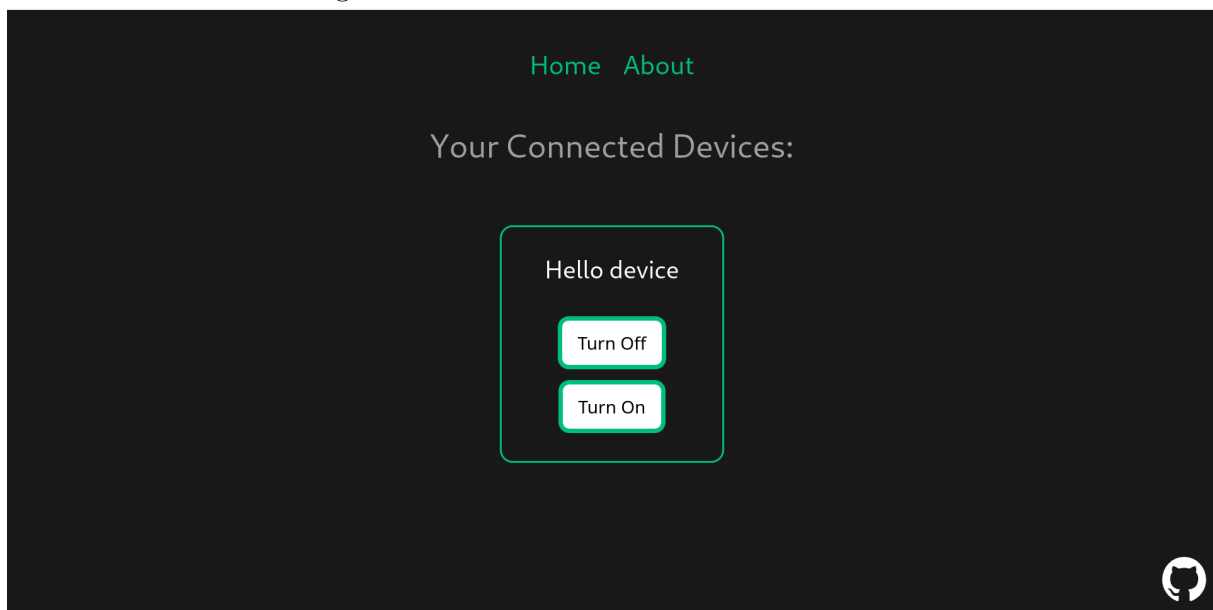
The designs look fairly similar, with the only real difference being Github icon in the bottom right corner, that serves as a link to the project's repository. This allows the user easy access to documentation, or the ability to file a bug report in the Issues section of the repository.

Calling the frontend API from the Web

There are two ways to call the frontend API, one through JSON using an HTTP server, the second through gRPC. For the reasoning behind this view subsection 4.3.3. The web-frontend uses the JSON API to make calls to frontend API endpoints. All this means for a user of the system, is that they will need to start the server with the *"--json-frontend* flag, to enable the HTTP server that allows for JSON calls.

Once the HTTP server has been enabled, calling the frontend API is fairly easy. The vue_frontend (which houses all web frontend files) contains a simple script which can used to generate JS files and with Typescript type definitions from the protobuf definitions.The script uses a Javascript library called ProtobufJs, to generate these files.

Once these files are available, all types used in the CLI frontend are now also available, making calling the frontend API very easy. Below is a typescript code snippet from *vue_frontend/src/backend_calls/registration.ts*:

```typescript
1  import { API_REGISTRATION_ADDRESS } from "@/api_call_links";
2  import { Result, errAsync, okAsync } from "neverthrow";
3  import { frontend } from "@/generated/generated";
4
5  export async function registerSelf(deviceName: string)
6  : Promise<
7      Result<frontend.registration.RegistrationResponse, Error>
8  > {
9      const req =
10         new frontend.registration.RegistrationRequest({
11             device_name: deviceName,
12         });
13
14     const res = await fetch(API_REGISTRATION_ADDRESS, {
15         method: "POST",
16         body: JSON.stringify(req),
17         headers: {
18             "Content-type":
19                 "application/json; charset=UTF-8",
20         },
21     });
22
23     try {
24         const parsed:
25             frontend.registration.RegistrationResponse =
26                 await JSON.parse(await res.text());
27         return okAsync(parsed);
28     } catch (e) {
29         return errAsync(new Error("Malformed Api Response"));
30     }
31 }
```

In this typescript snippet we construct a new "RegistrationRequest", send it in the JSON format to the server. We then await a response and parse it into a "RegistrationResponse" and return that. The address used for this is defined in the *api_call_links.ts* file. In this case being "/api/frontend/registration", with the fetch request being made to the domain the server is on (in this case http://localhost:5173). due to the "/api" prefix in the address, the request is automatically forwarded to the HTTP server. Without the "/api" prefix, the site at "/frontend/registration" would be requested instead, returning a 404 Not Found error, instead of an API response. All of this is handled by the Vite web server, which is hosting the website.

4.3.3 Using JSON for the Frontend API

Throughout this report a JSON frontend API has been alluded to, this subsection will explain why it was necessary to implement this in the first place. GRPC uses the HTTP/2

protocol to communicate, which is supported by web browsers. However, some features in HTTP/2 required by gRPC are not exposed by web browser [2] for a variety of reasons, which lead to the creation of gRPC web. GRPC web functions slightly differently to normal gRPC and is not compatible with normal gRPC, due to the HTTP/2 issues. Because of this, a proxy between the normal gRPC server and gRPC web needs to be used to communicate.

Instead of using a seperate program as a proxy, I decided to use a JSON to GRPC translation proxy instead, running on the server. This makes it possible to use a web-based frontend, without having the additional restrictions imposed by gRPC web. The downside of this, is that another HTTP server needs to be spawned, if the user decides to use JSON. Additionally, all benefits of using GRPC with the frontend are lost, as JSON is sent anyway. That being said, there are some benefits.

The JSON proxy I have created is completely transparent to the server, as the proxy calls gRPC functions. Additionally, the protobuf files are still used in combination with Typescript. This means that nothing in the packets is changed between the proxy and the server, the packet is simply parsed and forwarded as a gRPC packet. Another positive side effect of the JSON proxy, is that most frontend developers are significantly more familiar with working with JSON than gRPC, making it easier to create a custom frontend for the system. No extra program needs to be installed, or started when using a web-based frontend, one simply needs to include a flag when starting the server and the rest is handled automatically. Finally, if a user wants to create a web frontend using gRPC web, this can easily be done, with no changes required to the server code.

View an example below of how JSON calls to the server are translated to gRPC function calls (from the file **backend/src/server/web_json_translation/json_registration.rs**):

```rust
#[actix_web::post("/frontend/registration")]
pub async fn json_registration(
    req_body: String,
    state: actix_web::web::Data<TranslationClientState>,
) -> impl Responder {
    let parsed_req:
        json_registration_service::RegistrationRequest =
            match serde_json::from_str(&req_body) {
                Ok(r) => r,
                Err(_e) => return
                    HttpResponse::BadRequest()
                        .body("Unable to parse request"),
            };

    let response = {
        let mut registration_service_client
            = state.registration_client.lock().await;

        registration_service_client
            .register(parsed_req).await
    };

    let response = match response {
        Ok(r) => r.into_inner(),
```

```
25        Err(e) => return HttpResponse::InternalServerError()
26              .body(e.to_string()),
27      };
28
29      return HttpResponse::Ok()
30          .body(serde_json::to_string(&response).unwrap());
31 }
```

While this code looks complicated (partially due to formatting restrictions), it is actually quite simple. We parse the JSON request into a RegistrationRequest struct. If fails we send an error to the sender. We then forward the request to our "registration_service_client", which is a gRPC client registered as a frontend with the server. This is contained in a Mutex, due to the fact that it may be accessed concurrently, so we need to wait for it's lock to be available first. Finally, we check that the response is not an error. If it is not, we turn the response to a string and forward it to the sender, otherwise we forward the error message.

In subsection 4.1.1 it was mentioned that the protobuf compiler needed to implement the interfaces Serialize and Deserialize on the protobuf files in the frontend folder. The reason for this can be seen in the code snippet above. Requests that are in the form of JSON need to be turned into their corresponding Rust structs. This is done using the "serde_json" library. If the "Serialize" and "Deserialize" traits are not implemented on structs from the protobuf file, then JSON requests will not be parsed. In fact, due to Rust's type system, the program will not even compile.

4.4 Security

This section will briefly showcase implementations of the protocols described within subsection 3.3. The code snippet is from the file *backend/src/server/certificate_ signing.rs*

4.4.1 Certificates

```
1 use rsa::{
2     pss::{BlindedSigningKey, Signature, VerifyingKey},
3     sha2::Sha256,
4     signature::{
5         Keypair, RandomizedSigner,
6         SignatureEncoding, Verifier
7     },
8     RsaPrivateKey,
9 };
10 //CSR = (device_id + device_public_key)
11 pub fn gen_certificate(&self, csr: &String) -> String {
12     let mut rng = rand::thread_rng();
13     let certificate = self
14         .signing_key
15         .sign_with_rng(&mut rng, csr.as_bytes())
16         .to_string();
17     certificate
```

```rust
18  }
19  //Verifies a certificate
20  pub fn verify_certificate(
21      &self,
22      certificate_from_device: String,
23      certificate_authentic: String
24  ) -> bool {
25      let certificate_authentic =
26      Signature::try_from(
27          &*certificate_authentic.as_bytes()
28      ).unwrap();
29
30      match self.verification_key.verify(
31          &certificate_from_device.as_bytes(),
32          &certificate_authentic
33      ) {
34          Ok(_) => true,
35          Err(_) => false,
36      }
37  }
```

In the gen_certificate function, we are using the RSA Rust library, specifically the Probabilistic Signature Scheme (pss) module, to generate a digest of csr. PSS is an encoding method for digital signatures, meant to be a replacement for the popular PKCS#1. Within the verify certificate function we simply compare the signature we know is authentic (stored when it is sent to the device), to that provided by the device. If they are the same, then we know the device is authentic.

4.4.2 Signatures

Signatures are similarly simple. To generate a signature for a message being sent we use the sign_data function (from ***backend/src/server/certificate_ signing.rs***):

```rust
1   ///returns the signature and timestamp of the signature
2   pub fn sign_data(&self, data: String) -> (Vec<u8>, u64) {
3       let timestamp = get_timestamp();
4       let data = timestamp.to_string() + &data;
5
6       let mut rng = rand::thread_rng();
7       let signed = self.signing_key.sign_with_rng(
8           &mut rng, data.as_bytes()
9       );
10      (signed.to_vec(), timestamp)
11  }
```

As mentioned in the comment on line one, this function takes data in the form of a string, concatenates it with the current time-stamp (in Unix time), then signs it with the same pss key used to sign data in the gen_certificate function. It then returns the the signature in the form of an Array of bytes, along with the timestamp.

Due to more complexity in the verification function, it will be provided in pseudo code. For the original Rust code, view the file ***backend/src/server/certificate_ signing.rs***.

```
1  def verify_signature_update_request(
2      client_verifying_key,
3      certificate,
4      updated_capabilities,
5      client_timestamp
6      signature
7  ):
8      server_timestamp = get_timestamp()
9      if (
10         server_timestamp - client_timestamp
11         > SIGNATURE_EXPIRATION_SECONDS
12     ):
13         return false
14
15     capability_string = updated_capabilities.reduce(
16         (acc, capability) => {
17             acc + capability.to_string()
18         }
19     )
20     to_check_against = client_timestamp
21         + capability_string
22         + certificate
23
24     return client_verifying_key.verify(
25         to_check_against,
26         signature
27     )
```

The current expiration time for a signature (and in extension message) is 10 seconds. This ensures that an attacker cannot use a replay attack to maliciously control the system and if something goes wrong during packet transfer, that outdated packets don't influence behavior. The client's verifying key, which is used to check the validity of the signature, is derived from the client's public key during registration.

4.5 Networking

write this

5 Testing & Evaluation

5.1 Using the System

To test Aims and Objectives 1 & 2, **finish this section**
   for All tests in this section a Raspberry Pi 4b was used to host client code, running Raspberry Pi OS Lite (with no desktop environment), which was released on December 11th 2023. The server and frontend were hosted on a Laptop with an Intel i5-12450H processor, running Fedora 39 Linux. The Raspberry Pi's hostname (visible in screenshots) is "nikpi".

5.1.1 Controlling one connected device

Perhaps the most obvious and basic test is to create a device, using the Rust client library, have it run on a device and to connect with that device to a running server.

Testing Setup

A simple client was created, with two capabilities "Turn On" and "Turn Off". The callback functions attached to these capabilities would simply log to the standard output the name of the capability they are attached to. The Raspberry Pi is connected to the same Wi-Fi network as the server. The port 2302 has opened on the laptop's firewall to ensure the Raspberry Pi can make requests to the gRPC server. The Raspberry Pi is being controlled through a secure shell connection (SSH) (visible on the left side of screenshots). View figure 5.1 for the exact configuration of the client.

Figure 5.1: Client's configuration file


```
nik@nikpi:~/dev/test_client_lib $ cat example_config.toml
device_name = "Hello device"
server_ip = "https://192.168.167.110:2302"

[capability."Turn Off"]
    available = true

[capability."Turn On"]
    available = true
```

Testing

Figure 5.2 shows the successful connection and certificate exchange between the client on nikpi and the server. Note that the server ip that the client is connecting to is set within the client's configuration file (view figure 5.1). In this case the server (on the right side of figure 5.2) is being started with the "json-frontend" flag, as discussed within subsection 4.3.3, this flag is required to use the web-based frontend.

Figure 5.2: Establishing a connection

The next important test is viewing the web frontend, to see that the device is correctly displayed on the frontend, with the appropriate capabilities. This can be seen in figure 5.3, where the device, named "Hello device" (view configuration file) can be seen with it's two capabilities "Turn Off" and "Turn On". The web-frontend is being hosted by a Vite server, through using the command "npm run dev", on localhost port 5173. The frontend is being rendered by the Chromium web browser.

Figure 5.3: Web Frontend with the connected device
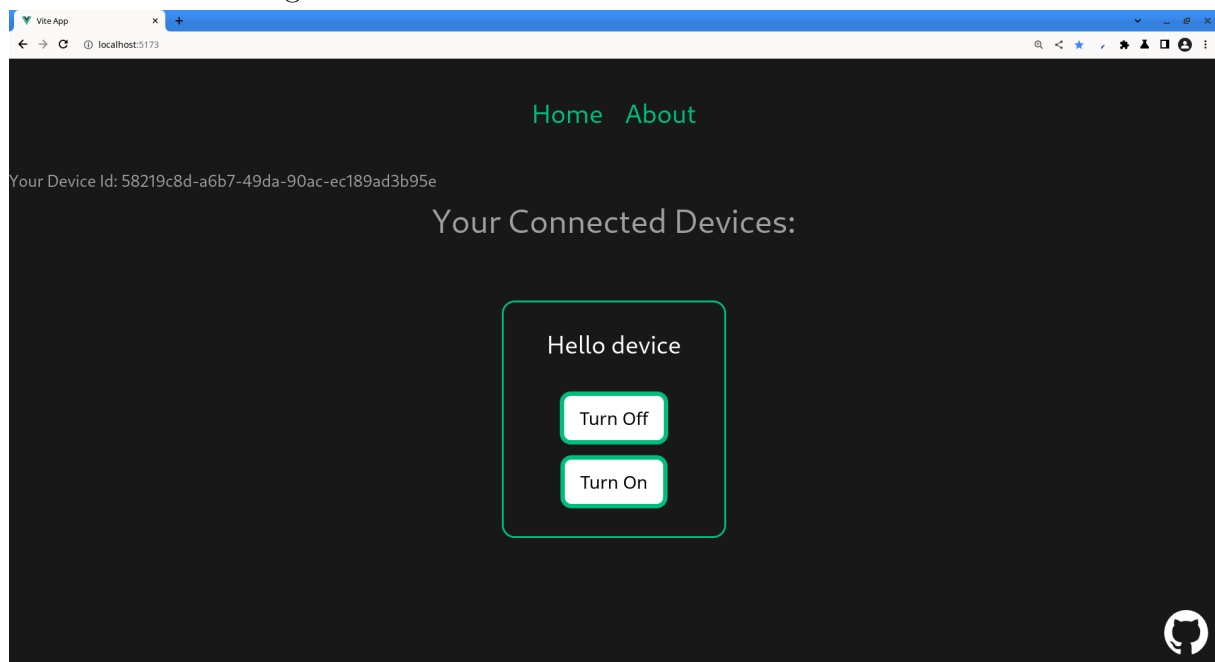
The final test for this subsection is to attempt to trigger the two listed capabilities, by clicking the buttons. This should send a JSON packet to the JSON proxy server, which is then forwarded to the gRPC server. Once this request has been processed, the server will add it to the list of updates for the specified client. The client will then eventually poll the server and receive the update. It will then call the callback function associated

with that capability. For this specific test both buttons will be clicked, starting with the "Turn Off" button, then the "Turn On Button". As shown in figure 5.4 this works as expected and the associated callback functions are called.

Figure 5.4: Triggering capabilities


```
nik@nikpi:~/dev/test_client_lib $ ./target/release/test_client_lib
Attempting to establish connection with: https://192.168.167.110:2302
Received Certificate
Successfully established Connection
Turn Off
Turn On
```

You can view the complete code for the client used in this example in the appendix. Due to the simplicity of setting up a client using the NOSHP-Client library, its only 47 lines long.

5.1.2 Running with One connected Device over Wifi

5.1.3 Running with multiple connected devices

5.1.4 Sending incorrect signatures

5.2 Performance Testing

6 Conclusion

Bibliography

[1] Actix.

[2] Grpc web. https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-WEB.md.

[3] Language guide (proto 3).

[4] Soa source book.

[5] Websockets living standard, Jan 2024.

[6] Sharu Bansal and Dilip Kumar. Iot ecosystem: A survey on devices, gateways, operating systems, middleware and communication. *International journal of wireless information networks*, 27(3):340–364, 2020.

[7] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *ECOOP 2014 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 257–281. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[8] Ryan Carniato. Javascript frameworks, performance comparison 2020. *Medium*, Dec 2020.

[9] Ing-Ray Chen, Jia Guo, and Fenye Bao. Trust management for soa-based iot and its application to service composition. *IEEE transactions on services computing*, 9(3):482–495, 2016.

[10] George F Coulouris. *Distributed systems : concepts and design.* International computer science series. Addison-Wesley, Boston ;, 5th ed. edition, 2012.

[11] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. *UCI Donald Bren School of Information & Computer Sciences*, 2000.

[12] gRPC Authors. Introduction to grpc. Accessed: 2024-Feb-15.

[13] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *IEEE transactions on services computing*, 3(3):223–235, 2010.

[14] Brenda Jin. *Designing Web APIs : building APIs that developers love.* O'Reilly, Beijing, 1st edition. edition, 2018.

[15] C. Kamienski, R. Prati, J. Kleinschmidt, and J.P. Soininen. Designing an open iot ecosystem. Belem, Brazil, 2019.

[16] Lukasz Kamiński, Maciej Kozłowski, Daniel Sporysz, Katarzyna Wolska, Patryk Zaniewski, and Radosław Roszczyk. Comparative review of selected internet communication protocols. *Foundations of computing and decision sciences*, 48(1):39–56, 2023.

[17] Ignas Plauska, Agnius Liutkevičius, and Audronė Janavičiūtė. Performance evaluation of c/c++, micropython, rust and tinygo programming languages on esp32 microcontroller. *Electronics (Basel)*, 12(1):143, 2023.

[18] Pallavi Sethi and Smruti R. Sarangi. Internet of things: Architectures, protocols, and applications. *Journal of electrical and computer engineering*, 2017:1–25, 2017.

[19] Andrew S. Tanenbaum. *Distributed systems : principles and paradigms*. Pearson Education Limited, Harlow, England, second edition, pearson new international edition. edition, 2014.

[20] MSRC Team. A proactive approach to more secure code. *MSRC Blog | Microsoft Security Response Center*, Jul 2019.

[21] Jingbin Zhang, Meng Ma, Ping Wang, and Xiao dong Sun. Middleware for the internet of things: A survey on requirements, enabling technologies, and solutions. *Journal of Systems Architecture*, 117:102098, 2021.

A Original Project Proposal

Put a copy of your proposal here

B Another Appendix Chapter

B.0.1 Client Library Used for Experiments in Subsection 5.1.1

```rust
use std::error::Error;

use NOSHP_Client::{
    client::{ClientState, NoshpClient,
        Request, UserDefinedState},
    client_config::{ClientConfig, ParsedConfig},
};

#[derive(Default)]
struct ExampleState {
    text: String,
}
impl UserDefinedState for ExampleState {}

const CONFIG_PATH: &str = "./example_config.toml";
#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let config = ClientConfig::load_config(CONFIG_PATH);
    let config = match config {
        Ok(r) => r,
        Err(e) => {
            eprintln!(
                "Error loading config: {}", e.to_string()
            );
            println!("Loading default config...");
            ParsedConfig::default()
        }
    };

    let client_handler = NoshpClient::new();
    client_handler
        .set_state(ExampleState {
            text: String::from("hello world"),
        })
        .add_callback("Turn On", Box::new(turn_on_led))
        .add_callback("Turn Off", Box::new(turn_off_led))
        .run(config)
        .await
        .unwrap();
```

```rust
40
41     return Ok(());
42 }
43
44 fn turn_on_led(
45     _state: &mut ClientState<ExampleState>,
46     _req: Request
47 ) {
48     println!("Turn On")
49 }
50
51 fn turn_off_led(
52     _state: &mut ClientState<ExampleState>,
53     _req: Request
54 ) {
55     println!("Turn Off")
56 }
```