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

Contents

List of Figures

List of Tables

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 Background

2.1 IoT System Architectures

Kamienski et al. describe a simple three layer architecture of an IoT system in [13]. 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" [13]. 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 [5], 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 [18]. 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 [16].

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 [11] 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" [3]. 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 [8]. 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 [16] 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

Add section about which one I chose: Service Based Architecture

ask if this is ok with citation as its quite similar

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.

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" [12] (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). [12].

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 [9]. 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. [9].

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 [12].

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 [12].

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 [4]. Due to the bidirectional TCP connections both client and server can send one another packets, even at the same time [12]. 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 [12].

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 [12]. 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) [12].

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.

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

Remove this text and write this section

Remove this text

Remove this text

3 Implementation

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 [17]. 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 [15], 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.

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 [10]. 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 [14]. 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

not sure I like the list style here

step of the server and client, eliminating the need for external build scripts (this is only partially true, view Sub Section 3.2.1).

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 [6]:

- 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 [7] 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 Server & Hub

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.

3.2.1 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. [2] 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 capabilities = 3;
15 }
16
17 message RegistrationResponse {
18     string public_key = 1;
19     string client_id = 2;
20     string certificate = 3;
21 }
```

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.

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:

```rust
fn main() -> Result<(), Box<dyn std::error::Error>> {
    tonic_build::compile_protos(
        "../proto/iot/types.proto"
    )?;
    tonic_build::compile_protos(
        "../proto/iot/registrationService.proto"
    )?;
    tonic_build::compile_protos(
        "../proto/iot/requestUpdateService.proto"
    )?;
    tonic_build::compile_protos(
        "../proto/iot/deviceControlService.proto"
    )?;

    tonic_build::configure()
        .type_attribute(
            ".",
            "#[derive(serde::Deserialize, serde::Serialize)]"
        )
        .compile(
            &[
                "../proto/frontend/registrationService.proto",
                "../proto/frontend/frontendTypes.proto",
                "../proto/iot/types.proto",
                "../proto/frontend/deviceControlService.proto",
            ],
            &["../proto/frontend", "../proto/iot"],
        )?;
    Ok(())
}
```

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 3.6.1.

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.

3.2.2 Server Start Up

This subsection details what the server handles at startup, including starting the GRPC
server, creating server stubs and creating a JSON web server (again, for an explanation
on why this is necessary view subsection 3.6.1). Below you can see how I instantiate a
new gRPC server instance within ***backend/src/server/server.rs***:

```
1   let grpc_server = tonic::transport::Server::builder()
2       .add_service(
3           registration_service_server
4               ::RegistrationServiceServer::new(
5                   registration_service,
6           )
7       )
8       .add_service(
9           request_update_service_server
10              ::RequestUpdateServiceServer::new(
11                  polling_service,
12          )
13      )
14      .add_service(
15          frontend_registration_service_server
16              ::FrontendRegistrationServiceServer::new(
17                  frontend_registration_service,
18          ),
19      )
20      .add_service(
21          frontend_device_control_service_server
22              ::FrontendDeviceControlServiceServer::new(
23                  device_control_service,
24          ),
25      )
26      .serve_with_shutdown(
27          grpc_address,
28          tokio::signal::ctrl_c().map(drop)
29      );
```

maybe
add a
section
on how
cli argu-
ments are
parsed if i
need more
words
(probably
not)

```
30
31  println!("Started GRPC Server on {}", grpc_address);
32
33  let grpc_handle = tokio::spawn(async move { grpc_server.await });
```

The Server struct comes from the tonic crate and transport module. 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 (view section 3.3). After invoking the builder method, which returns a new Server struct, we add services to it. A service is simply an instance of a struct which implements a trait (interface) generated by the protobufs. For an example of this, view subsection 3.2.3. These are created above the code snippet.

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 containg the device's IP address and the user specified 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 3.2.5).

Once we have added all services to the Server struct. We can then "await" the gRPC server handle on another thread. For more information on how this works, view subsection 3.2.5.

The JSON HTTP server (view 3.6.1) works almost the same, using the same builder pattern. 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      let cors = actix_cors::Cors::permissive();
5      actix_web::app::App::new()
6          .app_data(web::Data::new(json_state.clone()))
7          .wrap(cors)
8          .service(json_registration::json_registration)
9          .service(json_registration::json_registration_options)
10         .service(json_get_devices::json_get_connected_devices)
11         .service(json_get_devices
12             ::json_get_connected_devices_options
13         )
14         .service(json_device_control::json_device_control)
15         .service(json_device_control
16             ::json_device_control_options
17         )
18  })
19  .bind(JSON_ADDRESS)?
20  .run();
21
22  println!("Successfully Started JSON API Layer on {JSON_ADDRESS}");
```

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. To host this HTTP server we are using the popular crate named "Actix", a self-described "powerful, pragmatic, and extremely fast web framework for Rust" [1]. First we create a new HTTPServer struct from the actix_web module, while handing it a callback function. We then again add services with the service method on the App struct from the app module and returns it to the HTTPServer struct. To view an example of these services, view subsection 3.2.3. 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.

Apart from creating service structs and parsing command-line arguments, instantiating and running the gRPC and HTTP server are the main things done at startup. For code beyond the above provided snippets view the file ***backend/src/server/server.rs***, which contains the main function for the server.

3.2.3 Device Registration

This subsection serves three purposes: to show how a service/server-stub is implemented within Rust, how the JSON server communicates with the gRPC server-stubs and to provide an explanation of how device registration is handled by the server. For this purpose it will show two Registration code snippets, first the gRPC service, then the JSON service. View the code-snippet below from ***backend/src/server/registration.rs***:

```
5  pub type ThreadSafeMutable<T> = Arc<tokio::sync::Mutex<T>>;
6  pub struct ClientRegistrationHandler {
7      connected_devices: ThreadSafeMutable<ConnectedDevicesType>,
8      connected_device_uuids: ThreadSafeMutable<Vec<String>>,
9      signing_service: certificate_signing
10         ::CertificateSigningService,
11     _public_key: rsa::RsaPublicKey,
12     string_public_key: String,
13     _device_count: ThreadSafeMutable<u128>,
14     frontend_cache_valid: ThreadSafeMutable<bool>,
15 }
```

While this struct looks fairly complex at first glance, with a little bit of understanding it is easy to understand. Within this snippet we create a new struct named the ClientRegistrationHandler. This struct is prefixed with the "pub" keyword, meaning it is accessible from outside this "module" (or file in this case). In Rust everything is private by default, unless prefixed with a pub keyword. The same goes for fields within the struct. These are all private, so they cannot be accessed from the outside. The only way to access these fields is through methods implemented on the struct. Additionally, some of the types used within the struct should be talked about.

- Arc - an atomic reference counting pointer. A reference counting pointer, is a pointer to a piece of memory on the heap, with a counter attached to it. For every instance of that pointer that currently exists, this counter is incremented. Every time the pointer goes out of scope the counter is decremented. Once the counter hits 0, meaning there are no more instances of the Arc, the memory within the pointer gets freed. The atomic part of the Arc simply means that it can be shared across

threads safely, as incrementing this counter is an atomic (uninterruptible) operation, preventing race conditions. A non-atomic Rc also exists in Rust.

- Mutex - A primitive used for mutual exclusion, a common programming concept to prevent race conditions in concurrent/parallel tasks. Only one task at a time can access memory within a Rust Mutex.

- ThreadSafeMutable - a custom type defined on line five. This combines an Arc and Mutex, which enables safe sharing of data between threads. When a task wants to access the memory within the ThreadSafeMutable, it must first dereference the Arc (usually done automatically in Rust), then lock the Mutex. This ensures that only one task can use this memory at a time. If the locked Mutex goes out of scope, the Mutex is free to be used again by a different task.

- Vec - a Vector, otherwise known as a resizable array.

- u128 - an unsigned 128-bit integer

The underscore prefix in front of fields, simply indicate to the Rust compiler that certain fields are currently not in use, but will eventually be used. This silences a compiler warning.

```
16  use self::registration_service
17      ::registration_service_server::RegistrationService;
18
19  #[async_trait]
20  impl RegistrationService for ClientRegistrationHandler {
21      async fn register(
22          &self,
23          request: tonic::Request<
24              self::registration_service::RegistrationRequest
25          >,
26      ) -> RPCFunctionResult<
27          self::registration_service::RegistrationResponse
28      > {
29          let request = request.into_inner();
30          let client_id: Uuid = generate_new_id();
31
32          let csr = client_id.to_string() + &request.public_key;
33          let new_certificate =
34              self.signing_service.sign_certificate(csr);
35
36          let capabilites = request.capabilities;
37
38          let device_public_key: rsa::RsaPublicKey =
39              serde_json::from_str(&request.public_key).unwrap();
40
41          let (active_capabilities, inactive_capabilities) =
42              capabilites
43              .into_iter()
44              .partition(|capability| capability.available);
45
```

```
46        let device = device::Device::new(
47            request.name,
48            active_capabilities,
49            inactive_capabilities,
50            client_id,
51            device_public_key,
52        );
53
54        let response = registration_service::RegistrationResponse {
55            public_key: self.string_public_key.clone(),
56            client_id: device.stringified_uuid.clone(),
57            certificate: new_certificate.await,
58        };
59        return Ok(tonic::Response::new(response));
60    }
61 }
```

3.2.4 Security

3.2.5 Threads & Concurrency

3.3 Device Library

3.4 Example Device

3.5 Networking

3.6 Web & CLI Frontend

3.6.1 Using JSON for the Frontend API

Write about here: - Show Json service in Rust - Why its bound on localhost - Why protobuf compiler needs Serialize and Deserialize on these structs

3.7 Open Source

Bibliography

[1] Actix.

[2] Language guide (proto 3).

[3] Soa source book.

[4] Websockets living standard, Jan 2024.

[5] 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.

[6] 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.

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

[8] 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.

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

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

[11] 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.

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

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

[14] 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.

[15] 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.

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

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

[18] 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.

ask about SOA source book citation

A Original Project Proposal

Put a copy of your proposal here

B Another Appendix Chapter

This could be about your experiments