UNSW
AUSTRALIA

School of Computer Science and Engineering
Faculty of Engineering
The University of New South Wales

A Modern IEEE 2030.5 Client Implementation

by Ethan Dickson

Supervised by Jawad Ahmed
Assessed by Nadeem Ahmed

Thesis C Report
Submitted November 2023

Thesis submitted as a requirement for the degree of
Bachelor of Engineering in Software Engineering

Abstract

IEEE 2030.5 is a standardised application-layer communications protocol between end-user energy devices, and electric utility management systems. In recent years, the protocol has seen unanimous adoption, and proposed adoption by DNSP's in Australia, and the United States of America. This thesis will discuss how we have leveraged modern programming techniques, and the Rust programming language, to produce a safe, secure, robust, reliable, and open-source implementation of a IEEE 2030.5 client library that can be used to develop software for use in the smart grid ecosystem.

Acknowledgements

I would like to thank Jawad Ahmed and Nadeem Ahmed for their guidance as supervisor and assessor, respectively.

I would also like to thank Neel Bhaskar for his work on an IEEE 2030.5 server implementation as part of research at UNSW in 2022.

I would also like to acknowledge and thank all contributors to free and open-source software, especially in the Rust ecosystem. This thesis wouldn't have been possible without them.

Finally, I would like to thank my friends and family for their support throughout my degree, and this thesis.

Abbreviations

API Application Programming Interface

ARENA Australian Renewable Energy Agency

BSGIP Battery Storage and Grid Integration Program

CA Certificate Authority

CBC-MAC Cipher Block Chaining Message Authentication Code

CCM Counter with CBC-MAC

CPUC California Public Utilities Commission

CPU Central Processing Unit

CSE Computer Science and Engineering

CSIP-AUS Common Smart Inverter Profile Australia

CSIP Common Smart Inverter Profile

CVE Common Vulnerabilities and Exposures

DER Distributed Energy Resources

DNS Domain Name System

DNSP Distribution Network Service Provider

DNS-SD Domain Name System - Service Discovery

DRLC Demand Response and Load Control

ECC Electric Curve Cryptography

EPRI Electric Power Research Institute

EXI Efficient XML Interchange

FFI Foreign Function Interface

FS Function Set

HAN Home Area Network

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IANA Internet Assigned Numbers Authority

IEEE Institute of Electrical and Electronics Engineers

I/O Input / Output

IoT Internet of Things

IP Internet Protocol

LFDI Long Form Device Identifier

SFDI Short Form Device Identifier

MRID Master Resource Identifier

mTLS Mutual TLS

OOP Object-Oriented Programming

OS Operating System

PEN Private Enterprise Number

REST Representational State Transfer

RFC Request For Comment

RPIT Return Position Impl Trait

RPTIT Return Position Impl Trait In Trait

RSA Rivest-Shamir-Adleman

SEP Smart Energy Profile

SERCA Smart Energy Root Certificate Authority

SE Smart Energy

SIP Smart Inverter Profile

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

UNSW University of New South Wales

URI Uniform Resource Identifier

WADL Web Application Description Language

xmDNS Extended Multicast Domain Name System

XML Extensible Markup Language

XSD XML Schema Definition

Contents

List of Figures

Chapter 1

Introduction

Society faces a growing need for reliable, sustainable and affordable electricity. One such way we have attempted to address this problem is via the invention of the 'smart grid', an electric grid assisted by computers, where by communication is enabled between electric utilities and end-users via a computer network, such as the Internet.

The portion of the smart grid existing in the end-user environment are end-user energy devices. This category of end-user energy devices further encompasses the category of "Distributed Energy Resources", devices that deliver AC power to be consumed in the residence and/or exported back to the electric grid. Examples include solar inverters, household solar batteries, and biomass generators [oEE18].

Through the use of distributed energy resources, fossil fuel based energy generation can be more readily replaced with clean and renewable energy, the need for which is of growing importance as we seek to address the threat of global climate change. Of great importance to the success of Distributed Energy Resources is their integration, and ongoing management as part of the broader electric grid.

End-user energy devices may require communication with electric utilities for the purpose of managing electric supply and demand, monitoring usage, and ensuring end-users are compensated, and charged, for their energy supply and demand, respectively.

2030.5 is an IEEE standardised communication protocol purpose built for securely integrating end-user energy devices, and therefore Distributed Energy Resources into the wider electric grid. Since it's inception in 2013, the protocol has seen both minor and major revisions, and has seen unanimous adoption by DNSPs in both Australia, and the United States of America - in the Australian Common Smart Inverter Protocol (CSIP-AUS) [Age23] and Californian Smart Inverter Protocol [Lum16], respectively.

Thus, the goal of this thesis is to implement a safe, secure, reliable and performant framework for developing IEEE 2030.5 clients for use in the smart grid ecosystem.

Chapter 2

Context

IEEE 2030.5 is but one protocol designed for communication between end-user energy devices and electric utilities. Although 2030.5 is a modern protocol, it is not wholly new or original. Rather, it is the product of historically successful protocols designed with the same aim. In this section we'll examine the predecessors to IEEE 2030.5, and their influence on the standard.

2.1 Smart Energy Profile 1.x

Developed by ZigBee Alliance, and published in 2008, Smart Energy Profile 1.x is a specification for an application-layer communication protocol between end-user energy devices and electric utilities. The specification called for the usage of the "ZigBee" communication protocol, based off the IEEE 802.15.4 specification for physical layer communication. [All13,]

The specification was adopted by utilities worldwide, including the Southern California Edison Company, who purchased usage of the system for $400 million USD. [Hei08,]

According to SunSpec in 2019, over 60 million smart meters are still deployed under ZigBee Smart Energy 1.x, with 550 certified SEP 1.x products. [All17,]

2.2 SunSpec Modbus

Referenced in the specification as the foundations for IEEE 2030.5 is the Sunspec Alliance Inverter Control Model, which encompasses the SunSpec Modbus Protocol. Sunspec Modbus is an extension of the Modbus

communication protocol, also designed for end-user energy devices, and was published, yet not standardised, in 2010. The protocol set out to accomplish many of the same goals as IEEE 2030.5 does today. Tom Tansy, chairman of the SunSpec Alliance pins the goal of the protocol as to create a 'common language that all distributed energy component manufacturers could use to enable communication interoperability'. [Tan21,]

2.3 IEEE 2030

IEEE 2030 was a guide, published in 2011, to help standardise smart grid communication and interoperability, and describe how potential solutions could be evaluated. A major goal of these potential communication protocols is that, by their nature of existing in the end-user environment, they were to prioritise the security of all data stored and transmitted, such that communication between electric utilities cannot be intercepted, monitored or tampered by unauthorised users.

Furthermore, protocols used were to ensure that an electric grid denial of service cannot be brought about by attacks on smart grid communication infrastructure [SVC+20] [oEE11,].

2.4 Producing IEEE 2030.5

In the interest of interoperability with a future standard, the SunSpec alliance donated their SunSpec Modbus protocol to form IEEE 2030.5. Simultaneously, ZigBee Alliance was looking to develop Smart Energy Profile 2.0, which would use TCP/IP. At this point the SunSpec Alliance formed a partnership with ZigBee Alliance, and IEEE 2030.5 was created as a TCP/IP communication protocol that is both SEP 2.0, and interoperable with SunSpec Modbus [Tan21,].

Likewise, IEEE 2030.5 works to improve the security of smart grid communication protocols, and the guiding principles put forward by IEEE 2030.

The extensibility of the protocol was also considered in it's design. IEEE 2030.5 provides a standard method for extending it's functionality, such that legislative, state-specific, and proprietary extensions of the standard can be developed whilst retaining the protocol's core design. Examples of this include the (later discussed) CSIP and CSIP-AUS. Both allow for clients & servers to be deployed under a different model from that described in the specification. For example, CSIP-AUS extends the possible structured data that can be communicated between clients & servers to better fit the requirements and needs of electric utilities in Australia.

With IEEE 2030.5 extending and combining these existing, widely used, protocols, we're reassured it actually solves the problems faced by utilities and smart grid device manufacturers alike, and wasn't created in a vacuum, unaware of real world requirements, or the needs of device manufacturers and electric utilities.

Chapter 3

Background

3.1 High-Level Architecture

The IEEE 2030.5 protocol follows a REST API architecture, and as such, adopts a client-server model.

Transmitted between clients and servers are 'Resources', all of which are defined in a standardised XSD. Despite the client-server model, IEEE 2030.5 purposefully does not make distinctions between client resources and server resources, as to avoid them having differing behaviours on each. Rather, a server exposes resources to clients, while clients retrieve, update, create and delete resources on servers. Servers communicate with many clients, and under a set of specified requirements, clients can communicate with multiple servers.

Being the product of existing technologies, the IEEE 2030.5 resources cover a wide range of applications. As such, the specification logically groups resources into discrete 'function sets', of which there are twenty-five. The device manufacturers or electric utilities implementing IEEE 2030.5 need only communicate resources from function sets relevant to their needs.

The specification defines two methods by which clients retrieve resources from server. The default method has clients 'poll' servers for the latest versions of resources on a timed interval. The second, more modern, and more scalable method, has clients 'subscribe' to a resource, after which they will be sent notifications containing any changes to the subscribed resource from the server, without needing to poll.

Despite this, only some resources can be subscribed to, and of those whether it can be subscribed to can be further restricted by the server. For that reason, it is practically always required that clients employ both polling,

and subscriptions when maintaining the latest instance of a resource [Wei21] [oEE18].

3.2 Protocol Design

Resources are transmitted between clients and servers using HTTP/1.1, over TCP/IP; optionally using TLS. As a result, the protocol employs the HTTP request methods of GET, POST, PUT and DELETE for retrieving, updating, creating and deleting resources, respectively.

The specification requires that SSL/TLS certificates be signed using ECDSA, that keys be exchanged using ECDHE, and that encryption be done as per AES 128, using the "Counter with CBC-MAC" (CCM) authenticated encryption algorithm. It also requires that the keys be generated using a specific elliptic curve, referred to as "secp256r1". This cipher suite uses "authenticated encryption with associated data" [oEE18]. Unlike most common implementations of TLS, supplied certificates are not only used to verify a server's identity, but are also used to verify a client's identity, using a protocol colloquially referred to as "Mutual TLS" or mTLS. [Wei21]

The standardised resource schema is defined using XSD, as all transmitted resources are represented using either XML, or EXI, with the HTTP/1.1 `Content-Type` header set to `sep+xml` or `sep-exi`, respectively.

In order to connect to servers, clients must be able to resolve hostnames to IP addresses using DNS. Similarly, the specification permits the ability for clients to discover servers on a local network using DNS-SD.

3.3 Usage

The IEEE 2030.5 function sets cover a wide range of applications and uses, aiming to support as many end-user energy devices as possible. A subset of these possible use cases are as follows:

- (Smart) Electricity Meters can use the 'Metering' function set to 'exchange commodity measurement information' using 2030.5 resources [oEE18].
- Electric Vehicle chargers may wish to have their power usage behaviour influenced by resources belonging to the 'Demand Response and Load Control' function set.
- Solar Inverters may use the 'Distributed Energy Resources' function set such that their energy output into the wider grid can be controlled by the utility, as to avoid strain on the grid.

Chapter 4

Adoption

Further proving that the IEEE 2030.5 protocol is worth implementing is it's adoption by electric utilities, as well as the tariffs and guidelines created by government energy regulars who mandate it's use. Consequently, many implementations of the protocol exist already, with the vast majority of them proprietary, or implementing proprietary extensions of the standard. In this section, we will examine both.

4.1 California Public Utilities Commission

'Electric Rule 21' is a tariff put forward by CPUC. Within it are a set of requirements concerning the connection of end-user energy production and storage to the grid. In this tariff, it is explicitly clear that "The default application-level protocol shall be IEEE 2030.5, as defined in the California IEEE 2030.5 implementation guide" [GC23,]. Given that the state of California was among the first to adopt the protocol, it's likely that future writers of legislation or tariffs will be influenced by Rule 21, particularly how they have extended the protocol to achieve scale in the realm of smart inverters.

Relating directly to use of the IEEE 2030.5 protocol at scale are the high level architecture models defined in the California SIP implementation guide.

Individual/Direct Model

Under the individual / direct model there is direct communication between an IEEE 2030.5 compliant client, in this case a solar inverter, and a IEEE 2030.5 compliant server, hosted by the electric utility. This model alone

does not impose any additional restrictions over those already existing in Rule 21. It requires the inverter to be a 2030.5 Client, and be managed individually by the server. IEEE 2030.5 already accounts for clients existing on the end-users local area network, and for those devices to have intermittent internet access, or for those devices to regularly idle ('sleepy devices'). [oEE18]

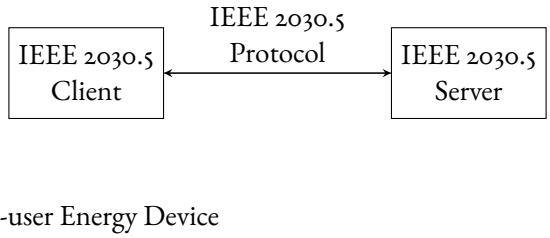

Figure 4.1: The Individual/Direct IEEE 2030.5 Model, as defined by California SIP.

Aggregated Clients Model

The aggregated clients model, outlined in the implementation guide, is preferred for use by electric utilities. Under this model, the 2030.5 client is but an aggregator communicating with multiple smart inverters, acting on their behalf. The rationale behind this model is to allow utilities to manage entire geographical areas, or a specific model of end-user energy device as though it were a single entity.

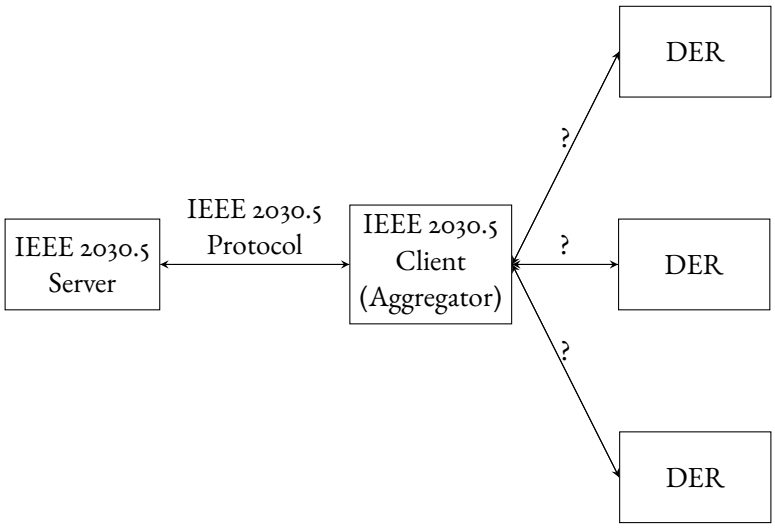

Figure 4.2: The Aggregated Clients IEEE 2030.5 Model, as defined by California SIP.

The IEEE 2030.5 server is not aware of this aggregation, as the chosen communication protocol between an aggregator client and an end-user energy device is unspecified and out of scope of the model. [Gro16] [All17].

Under this model, aggregators may be communicating with thousands of IEEE 2030.5 compliant clients. For this reason, the California SIP mandates the subscription/notification retrieval method be used by clients, rather than polling, wherever technically possible [Gro16]. This is done in order to reduce network traffic, and of course, allow for use of the protocol at scale. Given this, the aggregating IEEE 2030.5 client is likely to be hosted in the cloud, or on some form of dedicated server.

4.2 Australian Renewable Energy Agency

"Common Smart Inverter Profile" (CSIP-AUS) is an implementation guide developed by the "DER Integration API Technical Working Group" in order to "promote interoperability amongst DER and DNSPs in Australia". The implementation guide requires DER adhere to IEEE 2030.5, whilst also mandating the client aggregator model, and recommending the use of the subscription/notification mechanism where possible. These are very similar requirements to that of American CSIP [Age23,].

Importantly, the Australian Common SIP extends upon existing IEEE 2030.5 resources whilst still adhering to IEEE 2030.5. As per IEEE 2030.5, resource extensions are to be made under a different XML namespace, in this case `https://csipaus.org/ns`. Likewise, extension specified fields are to be appropriately prefixed with `csipaus`.

The additional fields and resources in CSIP-AUS work to provide DERs with support for `Dynamic Operating Envelopes`, bounds on the amount of power imported or exported over a period of time [Age23,].

4.3 SunSpec Alliance

As of present, the specification behind the aforementioned SunSpec Modbus is still available and distributed, as it's "semantically identical and thus fully interoperable with IEEE 2030.5". The primary motivation for implementing SunSpec Modbus is it's compliance with IEEE 1547, the standard for the interconnection of DER into the grid [All21,].

4.4 Open-source implementation

Despite IEEE 2030.5's prevalence and wide-spread adoption, the vast majority of implementations of the standard are proprietary, and thus cannot be distributed, modified, used, or audited by those other than the rightsholder, with the rightsholder typically providing commercial licenses for a fee.

For that reason, any and all open-source contributions involving IEEE 2030.5 actively work to lower the cost of developing software in the smart grid ecosystem. These contributions are necessary in ensuring the protocol can be as widely adopted as possible, as to incorporate as many end-user energy devices into the smart grid as possible.

In this section we will discuss existing open-source implementations of the standard for use in client devices.

4.4.1 Electric Power Research Institute Client Library: IEEE 2030.5 Client

One of the more immediately relevant adoptions of the protocol is the mostly compliant implementation of a client library by EPRI, in the United States of America. Released under the BSD-3 license and written in the C programming language, the implementation comes in at just under twenty-thousand lines of C code. Given that a IEEE 2030.5 client developed using this library would require extension by a device manufacturer, as to integrate it with the logic of the device itself, EPRI distributed C header files as an interface to the codebase. For the purpose of demonstration and testing, they also bundled a client binary that can be built and executed.

The C codebase includes a program that parses the IEEE 2030.5 XSD and converts it into C data types (structs) with documentation. This is then built with the remainder of the client library.

The implementation targets the Linux operating system, however, for the sake of portability, EPRI defined a set of header file interfaces that contained Linux specific APIs, such that they could be replaced for some other operating system.

These replaceable interfaces include those for networking, TCP and UDP, and event-based architecture, using the Linux `epoll` `syscall`, among others.

Notably, this implementation uses minimal third-party libraries, choosing to implement XML, EXI, HTTP, the XSD parser, and DNS-SD, effectively from scratch, using just low-level linux interfaces and OpenSSL. This has the advantage of producing more self-contained code, and may have allowed for niche optimisations to be made.

However, since these implementations are not as commonly used as more popular third party opens-source libraries it is unlikely they have been subjected to extensive testing.

The implementation states, in it's user manual, that it almost perfectly conforms to IEEE 2030.5 according to tests written by QualityLogic. The one exception to this is that the implementation does not support the subscription/notification mechanism for resource retrieval (as of this report). This is particularly unusual given that the California SIP mandates the use of subscription/notification under the client aggregator model, a model of which this implementation was targeted for use in.

One potential pain point for developers utilising this library is the ergonomics of the interface provided. The C programming language lacks many features present in modern programming languages that can be used to build more ergonomic and safe interfaces. For instance, the codebase's forced usage of global mutable state for storing retrieved Resources, goes against modern design principles, and would be easily avoided in a more modern programming language.

Furthermore, being written in a language without polymorphism, be it via monomorphisation, dynamic dispatch or tagged unions, C forgoes a great deal of type checking that could be used to make invalid inputs to the interface compile-time errors, instead of run-time errors or invocations of undefined behaviour.

For example, due to the lack of tagged unions (sum types) in C, many functions exposed to users accept a untyped (void) pointer, which is then cast to a specific type at runtime, providing no compile-time guarantees that that type conversion is possible, or that the underlying input is interpreted correctly, or even that the given pointer points to memory that the process is capable of reading and/or modifying.

Regardless, the library is sufficiently modular, providing interfaces across multiple C headers, where users of the library need only compile code that is relevant to their use-case. For example, developers building IEEE 2030.5 clients that do not handle DER function set event resources are not required to compile and work with the code responsible for managing them.

Additionally, the usage of the `epoll` `syscall` and the library's state-machine centric design lends itself to the scalability of the client, allowing it to handle operations asynchronously, and better scale when operating under the client aggregation model [Ins18,].

4.4.2 Battery Storage and Grid Integration Program: envoy-client

A more modern implementation of the protocol is `envoy-client`, developed by BSGIP, an initiative of The Australian National University. Of note is that this client has been open-sourced ahead of the release of their IEEE 2030.5 Server library implementation `envoy`. The release provides a very bare implementation of core IEEE 2030.5 client functionality. The library is written in Python, and therefore provides a far more modern interface than that of the EPRI library [BSG21].

The library was released publicly in 2021, and has seen virtually no updates since. It is possible the client will see a major update when the `envoy` library is released, as improvements to 2030.5 test tooling were indicated as being developed [Cut22].

Despite being written in Python, a programming language with support for asynchronous programming, async await syntax is not present in the codebase. For that reason, it's likely the user of the library will be required to wrap the provided codebase with async python in order for it to scale as a client aggregator. In theory, it's also likely that the Python Global Interpreter Lock would impact the ability for the client to take advantage of multiple threads, and potentially lead to performance issues at scale.

Despite the library's current, incomplete, state, a dynamically typed programming language lends itself reasonably well to the nature of 2030.5 client-server resource communication, as it allows for deserialisation of XML resources to dynamically typed Python dictionaries, where each key and value can have a different type, using the python "Any" type. This lack of type-checking on resources leads to faster development times, and is part of BGSIP's justification for the client & server being implemented in Python.

Under this design, the checking of XML attributes and elements is left to the user of the library - they must ensure that the given XML resource is of the same type as expected, and that it contains the expected fields.

By the very nature of dynamic typing, Python provides no guarantees that fields of resources accessed are present or valid. Unchecked accesses to fields may lead to runtime errors.

With minimal dependencies, and without a fixed implementation of TLS, the library is as portable as Python, and the Python requests library.

Given this, the library would be an acceptable tool for quickly testing an IEEE 2030.5 Server implementation, but would likely struggle in production use, such as when aggregating on behalf of clients, due to it's incompleteness.

Chapter 5

Design

Given the context and background surrounding the IEEE 2030.5 protocol, we can begin examining the high-level considerations, constraints, and assumptions we make in designing our implementation.

5.1 Considerations

5.1.1 Client Aggregation Model

Of great importance to our design is that we have developed a client library primarily for building client aggregators, the IEEE 2030.5 model preferred by electric utilities.

Despite this being the case, we do not wish to make decisions that would directly impact the ability for our client library to be used under the individual / direct model, such as where our library runs on a lightweight Linux distribution in the end-user environment. Therefore, where possible, we design an implementation capable of operating under both. One example of this we'll later discuss is retaining support for IEEE 2030.5 "sleepy devices".

5.1.2 Security

Improving the security of end-user energy devices was a major motivator behind the development of IEEE 2030 and 2030.5.

Parallel to that measure of improving security, when considering a programming language for our implementation, we want to ensure that we are upholding the security-focused design of the protocol.

In 2019, Microsoft attributed 70% of all CVEs in software in the last 12 years to be caused by memory safety issues [Mil19,]. For that reason, we wish to develop our implementation in a programming language that is memory safe. A language where there are fewer attack vectors that target memory unsafety. For that reason, when implementing IEEE 2030.5 in Rust, we limit our development to the subset of Rust that provides memory safety via static analysis, 'safe' Rust. This is opposed to using 'unsafe' Rust in our client, where raw pointers can be dereferenced, and as such the compiler is unable to provide guarantees on memory safety. When writing unsafe Rust, soundness and memory safety of unsafe code must be proven by hand.

Furthermore, in developing our implementation we are required to adhere to many other standards. In the interest of security, and also correctness, we neglect implementing these standards ourselves, and instead defer to more commonly used, publicly audited, and thoroughly tested implementations of HTTP, SSL, TLS and TCP.

An exception to this is our usage of OpenSSL, which in 2023 alone has had 19 reported CVEs [Ope23b]. In section 7.3 we discuss alternatives to this.

5.1.3 Modularity

IEEE 2030.5 is a specification with a wide range of applications. It aims to provide functionality for coordinating virtually all types of end user energy devices. Under CSIP and CSIP-AUS, it can be deployed in different contexts (See Section 4.1), and can require software on both the side of the electric utility, and the end-user energy device.

For this reason, instead of developing any single binary application, we've instead designed and implemented a set of Rust libraries (called 'crates'). These libraries provide an abstraction for interacting with IEEE 2030.5 servers and resources, allowing users to build IEEE 2030.5 client applications.

XML Serialisation & Deserialisation Library: SEPSerde

IEEE 2030.5 resources can be communicated between clients and servers as their XML representations. This means we require the ability to serialise & deserialise Rust data types to and from XML. Fortunately, there

already exists a popular Rust crate for this purpose, built for use in embedded communication protocols, called `YaSerde` [Lum23,].

However, this library does not perfectly fit IEEE 2030.5 requirements. To address this, we have forked YaSerde, and developed a crate `SEPSerde`, operating under a very similar interface to YaSerde, but instead producing XML representations of resources that conform to IEEE 2030.5.

We do not intend for library users to interact directly with this crate, it merely forms the basis of our common library implementation.

Note that we are yet to complete renaming, in most contexts the library will still be referred to as `YaSerde`.

The source code is available at: `https://github.com/ethanndickson/yaserde`

Common Library: sep2_common

IEEE 2030.5 resources are used in both clients and servers. For that reason, we've developed a common library with a Rust implementation of the IEEE 2030.5 XSD. This common library includes Rust data types annotated to use our `SEPSerde` crate. This allows these resources to be serialised and deserialised to and from XML.

This common library, `sep2_common`, can then be easily integrated and implemented in a future IEEE 2030.5 server implementation, as to avoid resources being implemented and stored differently on either. For the sake of modularity, and to avoid unnecessarily large binaries when compiled, this crate comes complete with compile-time flags (called Crate 'features') for each of the resource packages in IEEE 2030.5, where packages correspond to function sets.

The source code and documentation for this library is available at: `https://github.com/ethanndickson/sep2_common`

Client Library: sep2_client

The potential use cases for IEEE 2030.5 are broad, as it's designed to be able to coordinate as many different types of end-user energy devices as possible. Every implementation of a IEEE 2030.5 Client will behave differently to fit the the end-user energy devices it targets, and the model under which it is deployed. If a IEEE 2030.5 Client is deployed under the "Aggregated Clients Model" it will need to communicate with the end-user energy

devices themselves via some undefined protocol. If a IEEE 2030.5 Client is deployed under the "Individual/Direct Model" the very same client will be responsible for modifying the hardware of the device itself accordingly. Clearly, it is impossible for us as developers to implement the resulting logic for directly interacting with the electric grid.

We have therefore produced `sep2_client`, a framework for developing IEEE 2030.5 Clients, regardless of the specific end-device, and regardless of the model under which it is deployed.

Much like `sep2_common`, we provide compile-time flags for different function sets. For example, clients that don't run a Subscription / Notification server, need not compile it.

The source code and documentation for this library is available at: `https://github.com/ethanndickson/sep2_client`

5.1.4 Open-source Software

Our implementation aims to address the lack of open-source IEEE 2030.5 software and tooling. Consequently, it's crucial that our codebase is well documented, and well tested, as to encourage and enable users to contribute modifications and fixes, and to reassure potential users of it's correctness.

When choosing an open-source license for which we distribute our implementation under, we consider the licenses of all other software we depend on. The Rust programming language is released under both the MIT license, and the Apache 2.0 license.

Furthermore, all our Rust library dependencies are licensed under either the MIT license, or both the MIT license and the Apache 2.0 license. The exception is OpenSSL, which is only licensed under Apache 2.0.

Therefore, we are given the ability to release our implementation under either license, or both.

As part of our research, we discovered a rationale for MIT & Apache 2.0 dual licensing being overwhelmingly used by Rust crates. This rationale claims that Apache 2.0 improves the interoperability of Rust crates in terms of their licensing, while the MIT license allows for software developed downstream to be licensed under the GPLv2, whereas the Apache 2.0 does not. [Arl19] [Tri].

For that reason, we follow the vast majority of Rust libraries, and the Rust programming language itself, and have released our `sep2_common` crate, and our `sep2_client` crate under both the MIT and Apache 2.0 li-

censes, allowing library users to choose the license that best their needs when releasing downstream of our implementation.

Ergonomic Interface

In in the interest of creating a quality Rust crate that can be released publicly, our implementation strives to produce an ergonomic interface.

For example, despite being a strongly typed language, we could have made the decision to pass resource field and attribute validation onto the user by parsing all XML as generally as possible, however, that would force library users to write verbose error handling, as they might do in a dynamically typed programming language. Instead, we leverage the fact that all IEEE 2030.5 data types are specified in a standardised XSD, and therefore their attributes and fields are known ahead of time, and provide appropriate Rust data types for all valid XML inputs.

Similarly, as we'll discuss, we provide Rust enums & bitflag implementations for integer enumerations and bitmaps once types are parsed, instead of raw integers.

Further details of how we produce an ergonomic interface are detailed throughout Chapter 6.

5.1.5 I/O Bound Computation

When designing our implementation, we consider that IEEE 2030.5 clients are I/O bound applications. Furthermore, with the expectation that our client library will be used to develop clients operating under the client aggregation model, it is necessary that our implementation is able to scale alongside a large proportion of I/O bound operations as it interacts with multiple end-user energy devices, and potentially multiple servers.

For that reason, we require an abstraction for event-driven architecture, such that computations can be performed while waiting on I/O.

Events a client instance are required to listen for include, but are not limited to:

- Input from aggregated clients, or local hardware, requiring the creation or updating of resources locally.
- Scheduled polling to update local resources.

- Event resources, starting and ending, indicating that clients engage in a specific behaviour over a given interval.

- Network events, such as an updated resource being pushed to the client via the subscription / notification mechanism, or receiving the response from a sent HTTP request.

For that reason, we implement our client library using async Rust, which provides us with the advertised 'zero cost' abstraction for asynchronous programming.

Rust provides runtime-agnostic support for `async/await` syntax. When attached to an runtime, async Rust can use operating system event notifications, such as via `epoll` on Linux, in order to significantly reduce the overhead of having to poll for new events.

Furthermore, an async Rust runtime allows us to take advantage of multiple OS threads, and therefore multiple CPU cores, as to best accommodate for the scale that's required by the client aggregator model.

We are reassured this approach to be sensible, as it is the approach shared by the EPRI C Client implementation. EPRI claims their library to be performant as it leverages asynchronous events via `epoll` and state machines.

5.1.6 Reliability

By the nature of the protocol, all IEEE 2030.5 software used must be reliable. All expected errors are to be recovered from gracefully, as client instances must run autonomously for extended periods of time. Failure to do so could possibly lead to a denial of service for electric grid consumers.

For that reason, we leverage Rust's compile-time guarantees that assist the reliability of software. For example, in Rust, expected errors are to be handled at compile time. The Rust tagged union types 'Option' and 'Result' force programmers to handle error cases in order to use the output of a process that can fail. Comparatively, a language like C++ uses runtime exceptions to denote errors, as is the case in the C++ standard library. C++ does not require programmers to handle these exceptions at compile-time, it does not express these exceptions at the type-level like Rust does.

Safe Rust, through it's type system, also eliminates the possibility of a data race when working with multiple threads of execution, further improving the reliability of our implementation.

5.1.7 Operating System

Despite the desire to write code that is portable, our code requires a good deal of operating-system-specific functionality. As such, we will target a single operating system.

Of great consideration when choosing an operating system is the aforementioned 'Aggregator' model for IEEE 2030.5, where by our client would be most likely deployed on a dedicated server, or in the cloud. In this circumstance, it's very much likely an operating system running on the Linux kernel is to be used.

Furthermore, in the event a client is being developed under the Individual/Direct model, there exist very lightweight Linux based operating systems for low-spec devices. For that reason Linux based operating systems are the best candidate for our targeted operating system.

We are fortunate enough that we get, for 'free', a great deal of portability, simply by the nature of the Rust programming language, and the open-source libraries we use. Both have implementations for a wide variety of common operating systems [Fou23b] [Tok23b]. Of note is that the vast majority of our code is as portable as Rust. Whilst the remaining portion is as portable as the Tokio runtime.

In the event a library user wishes to use our implementation on an unsupported operating system, our code is released under open-source licenses, allowing them to modify it for their use case.

5.2 Assumptions

In designing our client, we've made a set of assumptions on library user expectations, and IEEE 2030.5 Client behaviour that is not present in the spec.

5.2.1 Rust Usability

This thesis is designed to be understood without extensive knowledge of the Rust programming language. Despite this, a major assumption is that users of our libraries will have a strong understanding of the Rust programming language, as is required to write elegant and performant, asynchronous Rust code.

To make the best use of our implementation, users will need to be be familiar with common design patterns when working with concurrent, asynchronous Rust code. This in of itself requires a solid understanding of single-threaded Rust code.

However, our implementation will not be accompanied by explanations or documentation pertaining specifically to the Rust programming language, as many sets of this freely available, and are almost certainly of higher quality than what we could produce.

We will however, produce thorough documentation and examples for using our libraries.

5.2.2 Notification Routes

When developing our Subscription/Notification mechanism, as part of the `sep2_client` crate, we've made the assumption that library users will not want to a use single HTTP route that handles all incoming notifications. Instead, developers would have each notification containing a different resource sent to a different route. That is, subscription resources would each use different `notificationURI` values.

We make this assumption as the specification makes no mention of whether a single route must be used, yet the examples in the specification show all notifications forwarded to a single URI. With a single route, each incoming HTTP request would first need to parse the body of the request to determine how to handle it, and then deserialise the resource. Using multiple routes simplifies this logic, requiring only the `Host` header be inspected before beginning deserialisation.

If the library was being developed in a dynamically typed programming language, or if XML parsing was done untyped, the single route would be a more reasonable approach. This is not the case.

We therefore assume the single route usage is purely for the purpose of the example, and that in reality, client developers do not desire this functionality.

We also assume that, due to only a small subset of IEEE 2030.5 resources being subscribable, that parametrized HTTP routes are not required, and that all routes are simply a relative URI interpreted literally.

5.2.3 DNS-SD

IEEE 2030.5 states that a connection to a server can be established by specifying a specific IP address or hostname and port. In the event cannot be provided, the specification states that DNS-SD can be used to query a network for servers, whilst providing clients with the ability to only query for servers advertising support for specific function sets.

As it stands, there is little perceived value in this functionality. Our client library targets the client aggregation model and as such client developers will almost certainly be capable of supplying the address and port of a server. The client manual for the EPRI IEEE 2030.5 Client Library shares the same belief. [Ins18]

We therefore assume this feature is simply not required by the majority of developers, and is therefore not included in our implementation as of this report. However, it will be required for full adherence to IEEE 2030.5.

5.3 Constraints

5.3.1 Generic Interface

As we develop an implementation of IEEE 2030.5 as part of this thesis, we note that we are somewhat removed from the potential use cases of our software. The lack of free and open-source IEEE 2030.5 resources online means we have no real measure, or way to determine how one might want to use our library.

Fortunately, we have the existing open-source implementations to refer to. For example, the EPRI library interface was likely designed with better understanding of possible use cases, and as such, it has been appropriate to use it as a guide when designing our own interface.

Furthermore, as a general rule, we prioritise designing a highly generic interface that minimises the restrictions placed on library users as much as possible, as to support incorporating our libraries into as many differently designed Rust programs as possible, and not force any one program structure.

5.3.2 EXI

IEEE 2030.5 Resources can additionally be communicated between clients & servers as their EXI representations. EXI is a binary format for XML, aiming to be more efficient (Measured by number of bytes sent for the same payload, and computation required to decode) by sacrificing human readability. As of present, there exists no Rust library for producing EXI from XML or from Rust data types, and vice-versa. Developing a Rust EXI library fit for use in IEEE 2030.5 is a large enough of an undertaking to warrant it's own thesis, and as such, is not included in our implementation.

Chapter 6

Implementation

Given the context and background of the IEEE 2030.5 protocol, and the high-level design decisions we've been required to make, we can discuss the technical design decisions present in our final implementation.

6.1 Common Library

The common library, `sep2_common`, contains all functionality that we have determined to be common to both IEEE 2030.5 clients and servers.

6.1.1 Resource Data Types

The primary purpose of the common library is to provide Rust representations of resources, which are described precisely in an XSD. Resources range from data that may be used by the electric utility, such as the current load of the device, to resource metadata, such as the rate at which a client should poll a server for a given resource, or what URI endpoint can be used to access a given resource, in the case of a Link resource. In IEEE 2030.5, these data structures are separated into packages for each of the function sets.

Whilst IEEE 2030.5 makes no mention of the object oriented programming paradigm, OOP inheritance underpins the design of all resources. Data types use both multi-level and hierarchial inheritance. For the purpose of having developers reuse code, many base types appear in the 'common' package; these are data types extended by many others.

Representing Resources in Rust

Rust, despite being influenced by OOP languages, does not posses the notion of a class, like in languages like C++ or Java. As such, Rust does not define shared behaviour of types through inheritance from a common parent class. Rather, shared behaviour is defined through traits. In this case, Rust shared behaviour refers solely to the methods that we can call on that type - the traits a type implements.

In this sense, Rust polymorphism does not concern itself with what a type is or what that type stores, it concerns itself only with the traits it possesses.

Traits themselves do support a form of inheritance. Traits can require that other traits be implemented for a given type. However, this does not change the fact that traits only represent behaviour - there is no way to have a data structure inherit the internal members of another.

```
pub trait SEResource {
    fn href(&self) -> Option<&str>;
}
```

Figure 6.1: A Rust trait representing the IEEE 2030.5 "Resource" data type.

```
pub trait SEList: SEResource {
    fn all(&self) -> Uint32;
    fn results(&self) -> Uint32;
}
```

Figure 6.2: A Rust trait representing the IEEE 2030.5 `List` data type.

In Figure 6.2, we have a Rust trait that describes the behaviour of the List base type. All lists are resources, and thus we have a trait bound that all types implementing SEList must first implement the SEResource trait.

The prefix `SE` (for 'Smart Energy') simply differentiates the trait from the concrete type with the same name.

This is the extent of native inheritance in Rust. We can specify the exact behaviour of types that belong to a trait in detail, but we cannot influence how that behaviour is achieved.

Emulating inheritance in Rust

As a result, we're forced to emulate the inheritance of data structure members in Rust, of which there are two approaches:

- Composite an instance of the base type into type definitions.
- Repeat all inherited members in type definitions.

Regardless of the approach, we would still not have have polymorphism using resource base-types as supertypes. To allow for polymorphism a trait must be defined for each base-type. That trait must then be implemented for all types that extends that base type. This is unavoidable duplicate code - although Rust provides ways for which it can be generated at compile-time.

Inheritance via Composition

If we had implemented the first of the two approaches, we could have made use of an existing Rust library to reduce the amount of boilerplate required to implement polymorphism. This library operates on the basis that inheritance can be replicated via composition. If a data type were to contain a member that implements a given trait, there is no reason for that outer struct to not be able to implement that trait by simply calling upon the underlying member.

```
#[inheritable]
pub trait SEResource {
    fn href(&self) -> Option<&str>;
}

pub struct Resource {
    href: Option<String>
}

impl SEResource for Resource {
    fn href(&self) -> Option<&str> {
        self.href.as_str()
    }
}
```

Figure 6.3: Rust code required to represent the IEEE 2030.5 "Resource" data type using 'inheritance-rs'.

```
#[derive(Inheritance)]
pub struct List {
    #[inherits(SEResource)]
    res: Resource,
    all: Uint32,
    results: Uint32,
}
```

Figure 6.4: A Rust data type representing an IEEE 2030.5 `List` using 'inheritance-rs'.

Figures 6.3 and 6.4 show how this library, `inheritance-rs` [HM19,] is used to reduce the boilerplate necessary to inherit data members. In Figure 6.3, we mark the SEResource trait as 'inheritable' and then implement that trait on a type that holds the necessary members, our bare minimum 'base' type. In Figure 6.4, we compose an instance of that base type into a type that would normally inherit from it. Then, we tell the library to generate the code, at compile time, that would allow List to implement the SEResource trait. This generated code simply calls the underlying SEResourceObj member when the href function would be called on a list.

The major flaw in this approach is that for every single type that is used as a base type, a trait, and a base implementation of that trait needs to be written. Given that there are just under 700 data types in the IEEE 2030.5 schema, we had to consider alternatives.

XSD to Rust types

If we were to implement the second approach, an existing Rust library can be used to automate the process of defining data types altogether. This of course draws on the fact that the IEEE 2030.5 XSD is entirely self-contained, and follows XSD guidelines by W3C. As such, generating Rust data types from it is a reasonable approach. One such way to automate this process would be to design and implement our own XSD parser and struct generator. Fortunately, we are not the first to require such a tool.

On the Rust public crates registry there are several XSD parsers, many of which existing to solve very similar problems; implementing standardised communication protocols in Rust. However, for that reason, many of these implementations are developed until they meet the creators needs, at which point the tool is no longer maintained.

Of the most complete parsers, one particular implementation stands out. This crate supports multi-level and hierarchial inheritance and makes reasonable assumptions on the internal representations of primitive data

types. xsd-parser-rs by Lumeo, was created for use in their Open Network Video Interface Forum Client, software with requirements not dissimilar to that of IEEE 2030.5. [Lum22,]

```
#[derive(Default, PartialEq, Debug, YaSerialize, YaDeserialize)]
#[yaserde(namespace = "urn:ieee:std:2030.5:ns")]
pub struct List {
    // The number specifying "all" of the items in the list.
    // Required on a response to a GET, ignored otherwise.
    #[yaserde(attribute, rename = "all")]
    pub all: Uint32,

    // Indicates the number of items in this page of results.
    #[yaserde(attribute, rename = "results")]
    pub results: Uint32,

    // A reference to the resource address (URI).
    // Required in a response to a GET, ignored otherwise.
    #[yaserde(attribute, rename = "href")]
    pub href: Option<String>,
}
```

Figure 6.5: A Rust data type representing an IEEE 2030.5 List as generated by xsd-parser-rs.

Figure 6.5 is what xsd-parser-rs produced for the List resource, without any modification. In this figure, it's parsed that the List type inherits from the Resource type, and included the href data member accordingly. It's also included the documentation as found in the schema.

Compared to inheritance in other languages, this has types include their parent type data members in their own type definitions. Despite this, the type definitions are far more readable, and align more closely with the output of the client - inheritance is 'flattened' out.

```
<List xmlns="urn:ieee:std:2030.5:ns" all="0" results=0"
href="/sample/list/uri" />
```

Figure 6.6: An XML representation of an IEEE 2030.5 List data type.

Figure 6.6 shows the List data type were it serialised into XML.

xsd-parser-rs does not perfectly fit our needs. However, it is released under the MIT license, giving us the freedom to fork the library, modify it accordingly, and manually adjust the output as required. As part of our common library implementation we've:

- Modified the tool to use the Rust Option type where the XSD specifies minOccurs=0 and maxOccurs=1.

- Modified the tool to provide us with the names of all types in a type's inheritance tree.
- Modified the Derive output on all data types to automatically implement traits where useful.
- Manually implemented Rust enums for all types that are integer enumerations.
- Manually implemented Rust `bitflags` for all integer bitmap types.

Due to the large amount of manual work required on the outputted code it is unlikely we will use `xsd-parser-rs` again, unless the XSD undergoes a major revision, and that major revision cannot be simply implemented with modern text replacement tools.

Integer Bitmaps

IEEE 2030.5, for the sake of efficiency, has resources encode many boolean values together in a single integer, a bitmap of booleans (colloquially a 'bitflag'). Whilst we must still maintain how these integers are stored internally, we have the opportunity to provide them in a more ergonomic interface. Thus, we use the `bitflags` crate, a popular Rust crate that wraps integers and provides convenience functions for comparing them (intersections, unions) and constructing them using enum-like syntax, whilst retaining support for bitwise operators [Bit23].

Resource Polymorphism

As mentioned, we've used `xsd-parser-rs` to automate the process of resolving an inheritance tree in the XSD. This has enabled us to add polymorphism via base types to our implementation.

We add Resource polymorphism to our common library as it can be implemented relatively easily, and assists both ourselves in writing more generic code (such as the Event scheduler), and users of our library. One example is that users could now store all "Subscribable" resources in a single container, and use that as the set of active subscriptions.

Therefore, as part of `sep2_common`, we define a set of Rust traits that mirror base type resources.

Since IEEE 2030.5 does not use multiple inheritance (inheriting from more than one base type), if we were to mirror base types to traits exactly, we would be implementing a confusing level of redundancy in traits. For example, `RespondableSubscribableIdentifiedObject` inherits from `RespondableSubscribableObject`,

which provides it with three new fields that allow it to be 'identified'. Similarly, `RespondableIdentifiedObject` inherits from `RespondableObject`, but provides it with the same Identified fields. In this example, Resources become 'identified' in two different ways. Rather than have two traits that provide the same behaviour, but with two different supertraits, we simplify and provide a single `SEIdentifiedObject` trait. We do this for all such redundant definitions of fields.

```
trait SEIdentifiedObject: SEResource {
    fn mrid(&self) -> &MRIDType;
    fn description(&self) -> Option<&String32>;
    fn version(&self) -> Option<VersionType>;
}
```

Figure 6.7: Implementation of the IdentifiedObject trait.

To implement these traits for all relevant types we use Rust's procedural macros - Rust code that runs at compile time producing more code to be compiled. A straight-forward implementation gives us the ability to derive any of these traits on a type that possesses all the required fields.

Manually adding these Derive annotations to all our data types would be extremely time consuming, and is unfeasible. Instead, we use our `xsd-parser-rs` fork to evaluate all the inheritance in the inheritance tree of a specific type, and use it to automate this process entirely.

```
#[derive(
    SERandomizableEvent,
    SEEvent,
    SERespondableSubscribableIdentifiedObject,
    SEIdentifiedObject,
    SESubscribableResource,
    SERespondableResource,
    SEResource,
)]
struct EndDeviceControl { ... }
```

Figure 6.8: The output of the modified `xsd-parser-rs` for `EndDeviceControl`.

The result is all IEEE 2030.5 resources include trait implementations defining behaviour shared between them.

6.1.2 Resource Serialisation & Deserialisation

IEEE 2030.5 resources can be sent as their XML representations over HTTP. This means our common library requires the ability to serialise and deserialise resources to and from the appropriate XML. This means HTTP requests containing resources are sent with the `Content-Type` header set to `application/sep+xml`.

On the public Rust crates registry, there exists a popular XML serialisation and serialisation library purpose built for use in embedded communication protocols called `YaSerde` [Lum23,]. Despite being developed by different teams, the syntax to use `YaSerde` on Rust data types is auto-generated by `xsd-parser-rs`, making it ideal for our use case.

Referring again to Figure 6.5, we see `xsd-parser-rs` has qualified our struct with the appropriate namespace, such that YaSerde serialisation will include it, and deserialisation will expect it. Furthermore, names of data members have been specified as attributes, rather than child elements in the resulting XML; determined by the parser as per the XSD. Figure 6.6 is an example of what the List type looks like when serialised to XML.

`YaSerde` does not perfectly fit our needs. However, it is also released under the MIT license, giving us the freedom to fork it, and add our required functionality. As part of our work we've modified `YaSerde` (`SEPSerde`) in order to:

- Serialise Rust enums as their underlying integer representations.
- Allow for Rust trait objects (dynamic dispatch) to be created on the serialise and deserialise traits.
- Serialise and deserialise IEEE 2030.5 `HexBinary` primitives as hexadecimal with an even number of digits.
- Allow `bitflags` generated bitmaps to be serialised and deserialised.
- Allow for data types that use Rust generics to be serialised and deserialised (Notification & NotificationList).

All done without requiring a manual per-type implementation.

Notification

The 'Notification' resource is a container for delivering resources to clients via the Subscription / Notification method. This means a Notification resource is generic, it contains some other resource as one of it's fields. To

represent this as a type, we use Rust generics, where the generic type is bound by the 'SEResource' trait we've defined.

```
struct Notification<T: SEResource> {
    resource: Option<T>,
    ...
}
```

Figure 6.9: Notification resource implemented using Rust generics.

This type representation uses Rust's monomorphisation, and therefore it being generic has no runtime overhead.

However, this poses some challenges for `YaSerde` which was not written with this sort of use case in mind. Even more so, the XML representation of a Notification resource is unique, in terms of how it expresses the type of the inner resource.

```
<Notification xmlns="urn:ieee:std:2030.5:ns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <subscribedResource>/upt/0/mr/4/r</subscribedResource>
    <Resource xsi:type="Reading">
        <timePeriod>
            <duration>0</duration>
            <start>12987364</start>
        </timePeriod>
        <value>1001</value>
    </Resource>
    <status>0</status>
    <subscriptionURI>/edev/8/sub/5</subscriptionURI>
</Notification>
```

Figure 6.10: Example Notification<Reading> resource from IEEE 2030.5.

Figure 6.10 shows a Notification resource that contains a Reading resource. Of interest is that the inner resource is always contained in an XML element with the name "Resource", and the type of the inner resource is instead given by the 'type' attribute, provided by the `XMLSchema-Instance` namespace.

The implication of this is not only do we need to modify `SEPSerde` to handle generic Rust types, but we also need a way to add this `xsi:type` attribute accordingly.

When setting the `xsi:type` attribute, we note that we require the name of the resource as a string. This means for every resource we also need to somehow bundle along with it a string literal of the name of the type. We do

this by having the procedural macro define a function that contains that string literal.

Throughout our implementation, we've also found additional use for this function, using it to provide the name of the resource as additional context when creating logs in generic functions.

In order to instruct `SEPSerde` to place this `xsi:type` attribute as required, we add a new procedural macro attribute to the library, `generic`.

```
#[derive(
    YaSerialize, YaDeserialize, SESubscriptionBase, SEResource,
)]
#[yaserde(
    namespace = "urn:ieee:std:2030.5:ns",
    namespace = "xsi: http://www.w3.org/2001/XMLSchema-instance"
)]
pub struct Notification<T: SEResource> {
    #[yaserde(rename = "Resource")]
    #[yaserde(generic)]
    pub resource: Option<T>,
    ...
}
```

Figure 6.11: Notification resource implementation.

Figure 6.11 shows this in use. We append the additional namespace as required, and include the `generic` attribute to the resource field. The result is that an instance of a `Notification<T: SEResource>` serialises and deserialises successfully.

NotificationList

Unlike the Notification resource, the NotificationList resource's purpose is more ambiguous. In IEEE 2030.5 a NotificationList resource is described as a "List of Notifications" and that it is a "List Resource that supports multiple types". This could be interpreted in a few ways:

- Each Notification in a Notification List can contain a different inner resource.
- All Notifications in a NotificationList must contain the same resource.
- A NotificationList is a Notification of a List resource, where that List resource is split up into multiple individual notifications.

The latter two interpretations would lead themselves to the most trivial implementation, a NotificationList need simply have a single generic type. The first interpretation would be, as a result of Rust's type-system, very much non-trivial, and would require each Notification to store a SEResource trait object (dynamic dispatch), as to force each Notification within the list to have the same concrete type. Fortunately, as mentioned in 5.2.1 we assume developers would want to use different routes for different notifications, making the first interpretation irrelevant, as all notifications on a single route will contain the same inner resource.

However, that does not yet clarify which of the latter two are correct. In any case, this is merely the behaviour of the NotificationList after being deserialised, and not something we as library developers need to worry about. For that reason we implement NotificationList as simply a list of Notifications, all with the same inner resource.

Exploring Dynamic Deserialisation

Dynamic deserialisation was a feature we explored implementing in our `YaSerde` fork; the ability to determine how a given XML string should be deserialised at runtime, instead of having to know at compile-time, as is required in the current implementation.

The protocol's WADL XML document outlines all the routes a IEEE 2030.5 server can provide, and what type of resource is expected on that route. Coupled with the assumption that library users implementing the Subscription / Notification mechanism will use different routes for different incoming resources, we can deduce that the type of all incoming XML resources can be determined at compile time, and thus the need to inspect Resources and determine how they should be deserialised at run-time isn't at all necessary.

Regardless, as part of this thesis we explored how this might be accomplished in Rust, in order to determine if it would work to improve the ergonomics of our common library interface.

In order to have a Rust function that deserialise all valid resources, even if the type is not known, we require that the function returns a single concrete type. For this we could use trait objects, Rust dynamic dispatch, which has the function return a 'fat' pointer, which contains a pointer to the resource (usually on the heap) and pointer to the table containing function pointers for the corresponding trait, often called a 'virtual method table'.

Implementing this function with a trait object return type would require a significant time investment, adjusting the existing `YaSerde` code to have it it return a `Box<T>` (heap allocated), instead of `T` (stack allocated).

An alternative to using trait objects would be to have a Rust tagged union (an enum) of all (141) top-level resource

types. Every resource would therefore have the same concrete type once wrapped in that enum.

Regardless of the approach, we would also need to implement some form of data structure mapping the names of resources to code that deserialise based off the type of the resource - we could likely auto-generate this.

If a trait object was used, the return type of the function doesn't provide users with anything immediately useful, in order to inspect the contents of the resource, they would need to downcast the trait object to a concrete type, which requires them to first know the concrete type at compile-time, bringing us back to the original problem.

The Rust enum approach improves this situation. When combined with Rust match statements library users can filter to a concrete resource as needed. However, with 141 different resources it's likely this filtering would be very verbose. Given that client developers should know the type of all incoming resources at compile time, implementing this functionality ourselves would force library users to handle this filtering themselves, even if they don't necessarily require it.

For that reason, we won't be implementing any form of dynamic deserialisation. If users require any of the approaches we've discussed, they have the ability to implement it themselves, or more reasonably, switch to a dynamically typed language where this model can be more easily supported, as we saw in the `envoy-client`.

6.1.3 List Ordering

IEEE 2030.5 defines list resources as simply a list of resources that can be retrieved in a single request. For each list resource, the specification defines how the list should be ordered, which values should be used as keys, and the precedence of those keys. [oEE18]

Therefore, for all resources that are used in a list we've implemented the Rust `Ord` trait, which defines a total order (as opposed to a partial order), that will be used when using default sort methods on the list itself. We store list resources in Rust `Vec`s.

Unfortunately, these keys are not described in any easily parsable way, and was instead implemented manually.

```
impl Ord for DERControl {
    fn cmp(&self, other: &Self) -> std::cmp::Ordering {
        // Primary Key - primacy (ascending)
        match self.interval.start.cmp(&other.interval.start) {
            core::cmp::Ordering::Equal => {}
            ord => return ord,
        }
        // Secondary Key - creationTime (descending)
        match self.creation_time.cmp(&other.creation_time).reverse()
             {
            core::cmp::Ordering::Equal => {}
            ord => return ord,
        }
        // Tertiary Key - mRID (descending)
        self.mrid.cmp(&other.mrid).reverse()
    }
}

impl PartialOrd for DERControl {
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering
        > {
        Some(self.cmp(other))
    }
}
```

Figure 6.12: Ordering implementation for the DERControl trait.

Figure 6.12 shows our implementation of this, where we first implement a total order, and then use that to implement the required partial order (of note is that there is now no "None" case for the partial order).

To provide an ergonomic way for working with lists, and ensure lists are always sorted, we use our procedural macros to provide a common interface for working with list resources.

List resources contain a Rust `Vec` internally, which can be pushed to directly, but does not retain the sortedness. Instead, we provide a `add` function on all lists that inserts and retains the required order. This function also updates the 'all' field of the resource (the length of the list), to ensure it remains consistent with the actual contents of the list. Similarly, we provide a `remove` function maintaining the same invariants. For all other, less common use cases, library users will be required to update the list invariants as necessary.

```
pub trait SEList: SEResource {
    type Inner: Ord;
    fn all(&self) -> Uint32;
    fn all_mut(&mut self) -> &mut Uint32;
    fn results(&self) -> Uint32;
    fn results_mut(&mut self) -> &mut Uint32;
    fn list_as_slice(&self) -> &[Self::Inner];
    fn list_mut(&mut self) -> &mut Vec<Self::Inner>;

    /// Add an item to the contained list, maintaining invariants
    fn add(&mut self, item: Self::Inner) {
        self.list_mut().push(item);
        // 'very fast in cases where the slice is nearly sorted'
        self.list_mut().sort();
        *self.all_mut() = Uint32(self.all().get() + 1);
    }

    ///Remove an item from the contained list, maintaining
       invariants
    fn remove(&mut self, idx: usize) -> Self::Inner {
        *self.all_mut() = Uint32(self.all().get() - 1);
        self.list_mut().remove(idx)
    }
```

Figure 6.13: The SEList trait, with default add and remove methods.

6.1.4 CSIP-AUS

Implemented in `sep2_common` are the IEEE 2030.5 extensions required by CSIP-AUS. These take the form of two additional resources, and additional fields on a subset of existing resources. When extensions are used, IEEE 2030.5 requires that the XML namespace for the extension be included with the resource, and that extension specific fields be prefixed accordingly. For CSIP-AUS this is the `https://csipaus.org/ns` namespace, and the `csipaus` prefix [Age23].

We make CSIP-AUS extensions available behind the `csip_aus` crate feature (compile flag). When activated, the Rust compiler adds the namespace to existing IEEE 2030.5 resources, and adds the prefix to their respective fields. It also makes the new resources available for use, also with the correct namespace and prefix. Using existing functionality of `YaSerde`, adding these extensions is straight-forward.

As per the Energy Queensland SEP2 Client Handbook, and the CSIP-AUS publication by ARENA, these resources, and our support for the client aggregation model, are the extent to which we, as library developers, can assist users in developing CSIP-AUS compliant clients [Que23] [Age23].

```rust
#[yaserde(namespace = "urn:ieee:std:2030.5:ns")]
#[cfg_attr(
    feature = "csip_aus",
    yaserde(namespace = "csipaus: https://csipaus.org/ns")
)]
pub struct DERCapability {
    /// Bitmap indicating the CSIP-AUS controls implemented
    #[cfg(feature = "csip_aus")]
    #[yaserde(rename = "doeModesSupported")]
    #[yaserde(
        prefix = "csipaus",
        namespace = "csipaus: https://csipaus.org/ns"
    )]
    pub doe_modes_supported: DOEControlType,
    ...
}
```

Figure 6.14: DERCapability annotated with optional CSIP-AUS extensions.

```xml
<EndDevice xmlns="urn:ieee:std:2030.5:ns"
           xmlns:csipaus="https://csipaus.org/ns">
    <changedTime>0</changedTime>
    <csipaus:ConnectionPointLink  href="/edev/1/cp" />
    <sFDI>0</sFDI>
</EndDevice>
```

Figure 6.15: Example XML representation of an EndDevice resource with a CSIP-AUS ConnectionPointLink.

Figure 6.15 shows the output of our common library when an EndDevice is constructed with a ConnectionPointLink resource, and when the `csip_aus` compile flag is enabled.

6.1.5 MRID Generation

IEEE 2030.5 Servers and Clients are both capable of creating resources. Resources where multiple instances exist, and must be differentiated from one another, contain a "Master Resource Identifier" (MRID), a 128 bit integer that is globally unique. As such, library users will require a way of reliably creating cryptographically unique MRIDs. One requirement of the MRIDs is that they contain a IANA Private Enterprise Number (PEN) in the least significant 32 bits.

For that reason, we provide users of our library with a function with the signature `fn mrid_gen(pen_id: u32) -> MRIDType` that produces a new MRID each time it's called.

IEEE 2030.5 requires that MRIDs are sufficiently cryptographically unique, such that the risk of an MRID

collision is minimised. The EPRI C implementation defines an algorithm for generating MRIDs that ensures, at the very minimum, that two MRIDs produced by a single instance of the program will not collide, at least for the first 2^{32} MRIDs generated. It does this by using a global, mutable, atomic integer that increments each time an MRID is generated. Since EPRI was able to release their MRID generation algorithm as part of their library, we assume it's sufficient, and implement a very similar algorithm in Rust.

```
static MRID_COUNT: AtomicU32 = AtomicU32::new(0);

fn mrid_gen(pen_id: u32) -> MRIDType {
    let id: u128 = /* Generate Random u128 */
    let time: u128 = /* Get Unix Timestamp */
    let count: u128 = /* Fetch and increment MRID_COUNT */
    HexBinary128(
        time << 32 | id << 32 |
        count << 32 | pen_id as u128)
}
```

Figure 6.16: Rust implementation of an MRID generator, inspired by EPRI's.

Going forward, it would be ideal to calculate the possibility of a global collision with this algorithm, as to verify that it is sufficiently cryptographically unique. This is currently out of scope for this thesis.

6.1.6 Testing

`sep2_common` is developed alongside a suite of tests, ensuring all resources can be serialised and deserialised, and that their XML representations adhere to IEEE 2030.5.

When running LLVM source-based coverage on `sep2_common` alone, the `grcov` report claims we have 59% coverage, or 1647 /2791 lines of code covered by tests. The vast majority of the missing coverage is code from compile-time code generation from libraries other than ours. In the vast majority of cases, this code is already tested. Regardless, we strive to improve this figure going forward.

MRID Generation

We provide a test verifying that the PEN ID can be obtained from a given MRID using bitwise operators, and that there are no collisions when generating several million MRIDs on a single instance of the program. This is to be expected, as we use include a global, mutable counter in the final value.

Testing SEPSerde

As expected, by the nature of serialisation and deserialisation, we have no compile time guarantees either will succeed for a given data type. Therefore, our first test suite for the common library is to test that all data types can be serialised & deserialised successfully, and that it can be done without any data loss.

```
#[test]
fn default_List() {
    let og = List::default();
    let new: List = deserialize(serialize(og).unwrap()).unwrap();
    assert_eq!(og, new);
}
```

Figure 6.17: A Rust test verifying that the IEEE 2030.5 List resource can be serialised & deserialised.

The logic for a test of this type can be succinctly expressed in Rust, as seen in Figure 6.17. This test takes advantage of the fact that all our resources are able to automatically implement the 'default' trait, and as such an instance of any resource can be instantiated with default values. In Figure 6.17, we first perform this default instantiation, convert the resource to an XML string, and then convert the XML string back into our internal List representation. Finally, we check if the newly deserialised resource is equal to it's default.

For each resource, the only part of this test that differs is the name of the type. As such, we auto-generate this test for each resource, and include it in `sep2_common`.

XML Correctness

The remainder of the common library will be tested by ensuring representations of resources are correct with respect to those defined in IEEE 2030.5. These tests use the example XML resources provided as part of IEEE 2030.5 as a starting point, as they are more likely to be correct than resources we would write by hand. However, the scope of these examples is narrow, and we will be required to write our own going forward.

6.2 Client Library

The client library, `sep2_client`, contains the remainder of our implementation, and provides an interface for interacting with IEEE 2030.5 servers, as well as interfaces for handling Event resources, the Time function set, and an implementation of the Subscription / Notification mechanism.

6.2.1 Asynchronous Programming

As mentioned, our client library is an I/O bound application, and uses async Rust to most efficiently make use of available resources. async Rust is runtime-agnostic, and must be connected to an executor of asynchronous tasks. In order to make the most of that runtime asynchronous functions capable of yielding are to be used in place of standard, blocking, functions, such as for I/O, or creating mutual exclusion.

Our client library is therefore implemented using the `Tokio` runtime, the most popular Rust async runtime [Tok23a]. Tokio uses `epoll` on Linux, and provides asynchronous wrappers for Rust standard library I/O. Furthermore, being the most popular, it is the only runtime to be extended and provide the functionality we require for our Subscription / Notification function set. This is the `tokio_openssl` crate.

The Tokio runtime operates on the basis that we as developers create 'tasks' that yield to one as required, in order to make progress, even on a single OS thread. The Tokio runtime is then able to distribute these tasks across available operating system threads to best handle the current workload. To use the runtime effectively we need only use async operations in place of blocking operations, and spawn tokio tasks to handle I/O bound operations. Spawning a Tokio task is not dissimilar from spawning a thread in Rust, but as an asynchronous runtime, does not necessarily lead to an OS thread being created.

6.2.2 Application Support

In IEEE 2030.5, the Application Support function set includes:

- "RESTful HTTP/1.1 application data exchange semantics".
- 'XML and/or EXI encoding as the data payload of RESTful operations'.
- "Authentication and encryption as HTTP over TLS " (Security Function Set).

[oEE18]

As such, this function set relies on a correct implementation of other standardised functionality. We choose to implement these standards by leveraging well-tested & open-source implementations of HTTP, XML, and TCP.

XML

IEEE 2030.5 resources are serialised to XML via our `sep2_common` crate, which depends on our `YaSerde` fork, `SEPSerde`. Internally, this uses the `xml-rs` crate, an XML parser & writer [Mat23]. This library was chosen to due it's emphasis on correctness.

In our research we found many competing libraries sacrifice correctness and completeness for performance, or are built for a specific use case, and then no longer maintained [Kra23]. Going forward, it may be worthwhile to explore alternative XML libraries that can be substituted into `SEPSerde` with the goal of improving XML parsing and writing performance.

HTTP

To communicate with IEEE 2030.5 servers using HTTP we use the very popular Rust `hyper` library, as it currently integrates best with our Security function set requirements, and provides us with adherence to HTTP/1.1. [Hyp23].

In section 7.3.1 we discuss alternatives to this, and why we'd like to move away from `hyper` in the future.

TCP

Clients and servers using HTTP/1.1 must use TCP. The most sensible approach to implementing TCP is to wrap the target operating system's sockets library. This is done for us by the Rust standard library. Additionally, when instantiating the client, users can set a `SO_KEEPALIVE` duration value, which will then be passed to the underlying Linux sockets library.

6.2.3 Security

The security function set of IEEE 20305 involves "securing transactions between clients and servers" via HTTP over TLS, or more commonly referred to as HTTPS. IEEE 2030.5 mandates a variation of TLS, mTLS, be used. mTLS provides a mechanism for mutually authenticating clients and servers.

The major constraint implementing the security function set is that "All devices SHALL support the `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` cipher suite", and that the "ECC cipher suite SHALL use elliptic

curve secp256r1" [oEE18].

IEEE 2030.5 does allow for additional cipher suites to be used, provided they "provide a cryptographic strength at least equivalent to the mandatory cipher suite" [oEE18]. As it stands, users wishing to add additional cipher suites are required to modify the source code of our library, though we do not rule out the possibility of providing an interface for setting additional cipher suites in the future.

Additionally, IEEE 2030.5 defines six classes of X509 certificates, each with different restrictions on how they are to be signed, and what X509 certificate extensions they contain, and how they are to be set.

Our client library provides developers with a generic interface for meeting the requirements of the Security function set.

TLS via OpenSSL

To implement TLS, we use the publicly audited, and commonly used library OpenSSL. OpenSSL is a C library, for which there exist popular Rust bindings that provide a safe foreign function interface to the underlying C code [Ope23a].

The HTTP library `hyper` doesn't natively support OpenSSL, but accepts a generic connector interface for use with any conforming external TLS configuration. To accommodate for this, we have OpenSSL generate a TLS config and use the `hyper_openssl` library to wrap that configuration in a connector that can be understood by `hyper` [Fac22].

An OpenSSL TLS configuration is generated when a IEEE 2030.5 Client, capable of making HTTPS requests, is instantiated. To create an IEEE 2030.5 TLS configuration for use in a Client device, library users must provide:

- Path to a IEEE 2030.5 "Device Certificate" or "Self-Signed Client Certificate" `.pem` file.
- Path to the certificate's corresponding private key `.pem` file.
- Path to a IEEE 2030.5 "Root certificate" (SERCA) `.pem` file.

IEEE 2030.5 places one additional restriction on what certificate can be used in the Subscription / Notification mechanism, namely that "all hosts implementing server functionality SHALL use a device certificate".

The relevant details on these certificate types are in IEEE 2030.5 `6.11.8.3.3`, `6.11.8.4.3` and `6.11.8.2`. We require the path to the CA to be supplied as to not require developers to install the CA on the system.

Client instances restricted to performing requests via HTTP do not need to supply any certificates, they need only be supplied an absolute URI of the target server. `hyper` handles DNS resolution, and as such, a hostname can be used.

Figure 6.18 shows a HTTPS client being instantiated with the server's address, the necessary certificates and key, the default `SO_KEEPALIVE` value, and the default poll 'tickrate' (see section 6.2.5)/

```
let client = Client::new_https(
    "https://127.0.0.1:1337",
    "client_cert.pem",
    "client_private_key.pem",
    "serca.pem",
    None, /* Default SO_KEEPALIVE */
    None, /* Default polling tickrate */
)
```

Figure 6.18: Example IEEE 2030.5 Client instantiation.

```
config.set_cipher_list("ECDHE-ECDSA-AES128-CCM8")?;
config.set_certificate_file(cert_path, SslFiletype::PEM)?;
config.set_private_key_file(pk_path, SslFiletype::PEM)?;
config.set_ca_file(rootca_path)?;
```

Figure 6.19: Steps necessary to create an IEEE 2030.5 TLS configuration using OpenSSL.

Figure 6.19 shows how we use the client supplied certificates and key to create a TLS configuration using OpenSSL. We note that OpenSSL supports the `ECDHE-ECDSA-AES128-CCM8` cipher suite.

X509 Parsing & Validation

Additionally, we provide an interface for library users to verify that the provided X509 certificates adhere to the specification - that they contain the necessary extensions and fields. OpenSSL provides this functionality, however the Rust FFI bindings we use currently do not expose the corresponding API. Rather than write unsafe Rust FFI, and possibly introduce undefined behaviour to our library, we simply defer to the `X509-Parser` library, which lets us check these conditions efficiently [Rus23c].

We distinguish between IEEE 2030.5 'device certificates', and 'self signed client certificates', as they have different requirements. The Subscription / Notification mechanism, as discussed later, forbids the use of the latter.

```
fn check_device_cert(cert: impl AsRef<Path>) -> Result<()>;
fn check_self_signed_cert(cert: impl AsRef<Path>) -> Result<()>;
fn check_ca(cert_path: impl AsRef<Path>) -> Result<()>;
```

Figure 6.20: IEEE 2030.5 certificate validation interface.

6.2.4 Core Client Functionality

The core functionality of our IEEE 2030.5 client library is providing users with an interface for interacting with IEEE 2030.5 servers - creating and sending HTTP requests. Our interface uses Rust's type system to ensure the contents of POST and PUT requests contain valid Resources, and that the contents of Responses to client GET requests also contain valid XML resources.

For all outgoing requests, our library also ensures spec adherence in terms of required HTTP headers.

- GET requests contain an `Accept` header, specifying that the body of the response should be `application/sep+xml`.
- PUT and POST requests contain appropriately set `Content-Length` and `Content-Type` headers.
- All outgoing requests also include the required date header, formatted as specified in RFC 7231, with appropriate accommodations for the Time function set [FR14].

```
async fn get<R: SEResource>(&self, path: &str)
    -> Result<R>;
async fn post<R: SEResource>(&self, path: &str, resource: &R)
    -> Result<SEPResponse>;
async fn put<R: SEResource>(&self, path: &str, resource: &R)
    -> Result<SEPResponse>;
async fn delete(&self, path: &str)
    -> Result<SEPResponse>;
```

Figure 6.21: Core client functionality interface.

Once a client is instantiated, Figure 6.20 shows the functions operating on an instance of a client (`&self`) for making requests. A generic `<R: SEResource>` parameter on these function signatures requires that the library user specify the type provided, as to determine how to serialise and deserialise. This need not be explicit, as Rust is able to infer this from the context.

For all these functions we automate error handling on behalf of the library user. Without knowing exact client use cases it's difficult to determine what level of error transparency they will require. As such, we make some assumptions on what information is and isn't useful.

- For GET requests, we return an error if the request could not be made for any reason OR if the returned Resource body could not be deserialised into the provided type successfully.
- For POST, PUT and DELETE we only return an error if the actual request failed to send, or if there was no response from the server. In all other cases a value of the SEPResponse enum is returned.

```
enum SEPResponse {
    /// HTTP 201 w/ Location header value, if it exists
    /// 2030.5-2018 - 5.5.2.4
    Created(Option<String>),
    /// HTTP 204
    /// 2030.5-2018 - 5.5.2.5
    NoContent,
    /// HTTP 400
    /// 2030.5-2018 - 5.5.2.9
    BadRequest(Option<Error>),
    /// HTTP 404
    /// 2030.5-2018 - 5.5.2.11
    NotFound,
    /// HTTP 405 w/ Allow header value
    /// 2030.5-2018 - 5.5.2.12
    MethodNotAllowed(String),
}
```

Figure 6.22: Definition of the SEPResponse enum.

Of special note, in Figure 6.21, is the `HTTP 400 Bad Request` SEPResponse. Per IEEE 2030.5, servers responding with `HTTP 400` will return an XML representation of an Error resource in the response body. When this occurs, our library will attempt to deserialise the error resource, and return it to the user. If that fails, the SEPResponse enum indicates it's absence using the Rust option type.

The IEEE 2030.5 successful request cases that can be returned are `201 Created` and `204 No Content`. Since the specification requires that `HTTP 201` responses contain an appropriate location header, we include that in the SEPResponse Created enum variant. The sam can be said for `HTTP 405 Method Not Allowed`, where servers must include an `Allow` header as part of the response, which we also include in our `MethodNotAllowed` variant.

Currently, this isn't exhaustive, though it includes all the successful request cases. For the vast majority of use cases this abstraction is sufficient. As part of our code release, we'll make library users aware of these deficiencies, and provide them with a plan for future enhancements.

In section 7.8 we discuss a design flaw with this current implementation, and why this interface might need to be adjusted going forward. In section 7.4 we discuss how we'll be able to improve the core client functionality, as a result of improvements to libraries in the Rust ecosystem.

Event Response Interface

Our library provides convenience functions for creating and sending response resources when given an event resource. This interface includes checks to ensure the provided response status matches up with that of the `ResponseRequired` field, and the requirements of the specific function set. As we'll discuss, the use of these are primarily automated via our Event schedules. We provide them directly to library users for completeness.

Each function sends the appropriate `SEResponse` resource to the server, returning an error if the inputs are invalid.

```
async fn send_der_response(
    &self,
    lfdi: HexBinary160,
    event: &DERControl,
    status: ResponseStatus,
    time: SEPTime,
) -> Result<SEPResponse>;

async fn send_drlc_response(
    &self,
    device: &SEDevice,
    event: &EndDeviceControl,
    status: ResponseStatus,
    time: SEPTime,
) -> Result<SEPResponse>;

async fn send_pricing_response(
    &self,
    lfdi: HexBinary160,
    event: &TimeTariffInterval,
    status: ResponseStatus,
    time: SEPTime,
) -> Result<SEPResponse>;

async fn send_msg_response(
    &self,
    lfdi: HexBinary160,
    event: &TextMessage,
    status: ResponseStatus,
    time: SEPTime,
) -> Result<SEPResponse>;
```

Figure 6.23: Interface for creating and sending response resources.

6.2.5 Resource Polling

IEEE 2030.5 defines two methods by which clients can receive updates to resources. One of those methods is via polling, where clients make GET requests on a regular interval.

Interface

We provide users with an abstraction for easily creating these regularly occurring poll events.

This abstraction is accessible from any instantiated client capable of making requests.

```
async fn start_poll<T: SEResource>(
    &self,
    path: impl Into<String>,
    poll_rate: Option<Uint32>,
    callback: impl PollCallback<T>,
);
async fn force_polls(&self);
async fn cancel_polls(&self);
```

Figure 6.24: Interface for creating and sending response resources.

Figure 6.23 shows this interface. `start_poll` is given a URI relative to the absolute server URI and a callback, and polls the resource at the given poll rate, or the IEEE 2030.5 default rate of 15 minutes. Each interval, it performs a GET request, and provides the deserialised resource to the callback. Since this is an automated background task users will only be notified of request failures via the (later discussed) logging we provide. Ideally, they would only start polling a route once they know it's valid.

We also provide a function for forcibly polling all tasks in the queue, and a function for cancelling all resource polling. Library users may want to forcibly poll all resources in the event they were operating without internet, and have regained connection. Library users may want to cancel all poll tasks when attempting a graceful shutdown, no longer aggregating on behalf of a specific device.

One flaw with the current design is that users cannot implement any form of error handling logic for when a retrieval fails. This is discussed further in section 7.8.

Ergonomics

When implementing this interface, we take into consideration it's ergonomics. Thus, we provide two ways for users to implement a polling callback.

```
trait PollCallback<T: SEResource>: Clone + Send + Sync + 'static {
    async fn callback(&self, resource: T);
}
```

Figure 6.25: A Rust trait representing the behaviour of a Poll callback.

The first method allows user to implement the trait in Figure 6.24 on a type, and then pass an instance of that type when starting the poll. This is useful when users have some thread-safe mutable state they want to share between all poll tasks, as they can implement handlers for multiple resources on a type, and then pass the same underlying instance of the type to multiple poll handlers, and thus access that same state in each handler.

```
impl<F, R, T: SEResource> PollCallback<T> for F
where
    F: Fn(T) -> R + Send + Sync + Clone + 'static,
    R: Future<Output = ()> + Send + 'static,
{
    async fn callback(&self, resource: T) {
        Box::pin(self(resource)).await
    }
}
```

Figure 6.26: Rust code generating implementations of the PollCallback trait for all applicable functions.

The second method uses the Rust language feature that allows us to generate implementations of a Rust trait for all types that meet certain conditions. In Figure 6.25 we therefore implement the PollCallback trait for all async, single argument functions that accept a valid Resource, and return the unit type (synonymous for no return value). This allows both function pointers to be used, and any thread-safe closure that does not capture any references to it's environment that last less than the length of the program.

The outcome of this is that we've produced an abstraction that places as few restrictions on the caller as is reasonably possible.

Internals

When implementing this interface, we must address the performance of the client, and it's ability to operate at scale. Client aggregators must be able to handle polling many resources on behalf of many clients.

To implement this we use the asynchronous sleep functions provided by `Tokio`, which yields a task until the next scheduled poll time. Tokio tasks, while slept, virtually very minimal overhead. Unfortunately, Tokio sleeps, much like Rust's threaded sleeps, do not make progress while the device itself is slept.

Whilst our client library is primarily for use on servers operating under the client aggregation model, we do not want to limit use of the library to that model. IEEE 2030.5 refers to particular end-user energy devices as "sleepy devices", that spend the majority of their time in modes of low power consumption. Should our implementation rely on asynchronous sleeps, we risk that, under specific cases, clients experience unnecessarily elongated intervals between polling.

A naive implementation that addresses this would have each individual Tokio task (one for each polled resource) wake intermittently, and check if it's time to poll their resource. This would be introduce an unnecessary overhead of multiple tasks repeatedly waking and sleeping.

Our final implementation uses a poll job queue, implemented using a binary heap, that sorts jobs by their next scheduled poll time. Then, we use a single Tokio task that sleeps, and wakes, intermittently to check if it is time to poll the resource at the top of the queue. Once that time arrives, the resource is retrieved and the stored user callback is given the result. The next time to poll on the poll task is then updated before pushing the job to the back of the queue. This process is entirely asynchronous, if it is time to poll multiple resources, each request will be sent off without waiting for the previous to complete.

For the sake of flexibility, giving end users control over this process, and for the sake of testing, we allow users to define the length of these intermittent sleep intervals when they instantiate the client, we refer to this duration as the 'tickrate' of the polling task, defined using the Rust `Duration` type. If not supplied, a default of a 10 minute interval is used.

```
type PollHandler =
    Box<dyn Fn()
        -> Pin<Box<dyn Future<Output = ()>
        + Send + 'static>>
        + Send + Sync + 'static>;

struct PollJob {
    handler: PollHandler,
    interval: Duration,
    next: Instant,
}

type PollQueue = Arc<Mutex<BinaryHeap<PollJob>>>;
```

Figure 6.27: Rust types used in implementing asynchronous resource polling.

Figure 6.26 shows the data structures used in our implementation. A `PollHandler` is simply an async function that takes zero arguments, and returns nothing. This function contains the user-supplied callback within. A `PollJob` couples this function with the poll rate duration and the timestamp at which the next poll should occur. `PollQueue` is a thread-safe Binary Heap of these `PollJob` instances, sorted by their next poll timestamp.

6.2.6 Subscription / Notification

The second method IEEE 2030.5 defines for receiving updates to Resources is via Subscription / Notification. Using this mechanism, a client runs it's own lightweight HTTP server, with predefined routes that servers

POST updated resources to. Clients notify servers of these routes by first creating a Subscription resource on the server, where they set the `notificationURI` field to an absolute URI.

Interface

This mechanism is optional, and runs irrespective of an instance of a client capable of making HTTP requests. Therefore, we provide users with the `ClientNotifServer` type, which can be instantiated much like a client.

To instantiate a HTTPS `ClientNotifServer`, a developer must provide:

- The IP address and port the server will listen on for TCP connections.
- Path to a IEEE 2030.5 "Device Certificate" `.pem` file.
- Path to the certificate's corresponding private key `.pem` file.
- Path to a IEEE 2030.5 "Root certificate" (SERCA) `.pem` file.

To instantiate a HTTP `ClientNotifServer`, only the first is required.

Once created, library users can add routes to the server by specifying the relative path to listen on, and an appropriate callback. In this case, a callback is a function that accepts one argument, a `Notification<T>`, where `T` is some resource, and returns a `SEPResponse`, the same type discussed in 6.2.4. The SEPResponse returned is constructed into an appropriate HTTP response, and then returned.

```
async fn incmg_dera(notif: Notification<DERAvailability>)
    -> SEPResponse {
    println!("DERAvailability Received: {:?}", notif);
    SEPResponse::NoContent
}
```

Figure 6.28: Example callback function for use by the `ClientNotifServer`.

Figure 6.27 shows an implementation of a function that can be used as a route callback. In this case, the callback will have the notiification server respond with `HTTP 204 No Content`. Figure 6.28 shows this function used as the callback for HTTP requests on the "/dera" route. This figure also shows how a closure with an appropriate function signature can be used, and also how the server can be run. The run function accepts a generic argument, an event that completes in the future, that can be used to end the server. In this example we provide it with a Tokio `SIGINT` (CTRL+C) signal handler. Upon receiving that signal, the server will attempt a graceful shutdown.

```
let server = ClientNotifServer::new(
    "127.0.0.1:1338",
)?
.with_https("client_cert.pem",
"client_private_key.pem",
"serca.pem")?
.add("/dera", incmg_dera)
.add("/txtmsg,| notif: Notification<TextMessage> | {
    println!("Message Received: {:?}", notif);
    async move { SEPResponse::NoContent }
}).run(signal::ctrl_c()).await;
```

Figure 6.29: Example instantiation and route creation on a `ClientNotifServer`.

Ergonomics

We place as few restrictions on the user of our Subscription / Notification mechanism as possible. Just as is done with resource polling, we allow users to implement a callback via a manual trait implementation, or by any function or closure with a matching function signature.

Our abstraction handles invalid HTTP requests, and responds accordingly, as per IEEE 2030.5, without any intervention.

Internals

To serve requests over HTTPS we once again use OpenSSL to generate a TLS configuration using the provided certificates and private key. To merge our TCP stream, created using a Tokio `TcpListener`, and our TLS configuration, into a single TLS Stream we require the `tokio_openssl` crate [Tok23c]. This stream can then be passed to `hyper` to serve the request.

From this, we are required to route incoming HTTP requests to the correct user-defined callback. The simple nature of the server, and lack of parametrized route names results in a straight-forward implementation.

Figure 6.29 shows that our implementation uses a `HashMap` from strings (routes) to stored callbacks.

The default hashing algorithm chosen by the Rust standard library sacrifices performance in order to defend against `HashDoS` attacks, as it uses an implementation of `SipHash`. Our router hashmap is never modified once the server is started, and therefore any DOS attacks on a client notification server instance would not be mitigated by a more secure hashing algorithm.

```
type RouteHandler = Box<
dyn Fn(&str) -> Pin<Box<dyn Future<Output = SEPResponse>
 + Send + 'static>>
 + Send
 + Sync
 + 'static
 >;

struct Router {
    routes: HashMap<String, RouteHandler, ahash::RandomState>,
}
```

Figure 6.30: Rust types used in implementing a concurrent HTTP router.

Therefore, we use the hashing algorithm provided by `ahash`, which is significantly faster, yet less resistant to `HashDoS` attacks [Kai23] [Rus23a].

6.2.7 Time

All IEEE 2030.5 clients must implement the Time function set. They must be able to use the current time to set HTTP date headers, update last modified timestamps, and check if a given Event resource has started or ended. Clients must be able to synchronise their device time with that of an IEEE 2030.5 server. This is done by having Servers expose Time resources. In the event there are multiple servers exposing time resources, the specification allows clients to choose just one Time resource, with the most accurate time (accuracy is specified within the resource itself), and synchronise using that.

The exception to this are event schedules: "... devices SHALL use the Time resource from the same FunctionSetAssignments when executing the events from the associated Event-based function set." which implies clients must be able to synchronise time differently, depending on the event-based function set. [oEE18].

Interface

Our client library must provide a generic abstraction that allows developers to synchronise time globally, and per event-based function set.

Throughout IEEE 2030.5 Client development, timestamps are required in a variety of formats. To provide a convenient interface for converting between these formats, we provide users with a `SEPTime` abstraction, that wraps the existing Rust native `SystemTime`, and represents a single timestamp that can be compared. `SEPTime` can be converted into:

- A Rust u64 for internal comparisons.
- An IEEE 2030.5 Int64 for Resource fields and attributes.
- An RFC 7231 HTTP Timestamp String via SystemTime and the `httpdate` crate [pyf23].

```
fn current_time() -> SEPTime;

fn current_time_with_offset() -> SEPTime

fn update_time_offset(time: Time);
```

Figure 6.31: Time function set interface.

Figure 6.30 details the interface we provide. Users can supply a Time resource to `update_time_offset` and update the global time offset. All future calls to `current_time_with_offset` will then be synchronised with that specific Time resource.

For event-based function set time synchronisation, a similar interface is present on Event schedules (Figure 6.31). Schedules can be provided a Time resource that will be used to determine the start and end time stamps of all events for that function set, as well as the HTTP Date header of automated schedule HTTP requests.

```
fn update_time(&mut self, time: Time);
```

Figure 6.32: Function operating on an instance of a Schedule for synchronising time.

6.2.8 Event Schedules

In IEEE 2030.5 some function sets employ the use of Event resources, resources that include a start timestamp, and a duration for which they are active. These event resources are exposed to clients with the intent the device take specific action during that interval. Servers have the ability to cancel those events by updating them, and supersede these events by exposing new ones before the event finishes. Clients communicate with servers about these Event resources by creating and sending Response resources (those that implement the `SEResponse` trait).

An exhaustive list of function sets utilising Event resources are:

- Distributed Energy Resources.
- Demand Response and Load Control.

- Messaging.
- Pricing.
- Flow Reservation.

All Schedules share very similar implementations. They differ in:

- What response status codes can be sent (defined in IEEE 2030.5 Table 27).
- Under what circumstances events supersede one another.
- Whether or not event start times and durations can be randomised.

Interface

For each of these function sets (excl. Flow Reservation) we produce a `Schedule` abstraction that acts as a black-box handler of events. It accepts instances of Event resources and whenever an event starts, ends, or is cancelled or superseded whilst active, the schedule calls the defined callback such that the client can apply the event to the relevant device(s), and determine the Response resource status to return to the server. In many cases, such as when resources are cancelled before starting, the client need not be informed, and the schedule is instead capable of determining the correct response status itself - creating and sending the response resource automatically.

```
trait Scheduler<E: SEEvent> {
    type Program;

    fn new(
        client: Client,
        device: Arc<RwLock<SEDevice>>,
        handler: impl EventCallback<E>,
        tickrate: Duration,
    ) -> Self;

    async fn add_event(&mut self,
                       event: E, program:
                       &Self::Program,
                       server_id: u8);
}
```

Figure 6.33: Definition of the Scheduler trait.

Figure 6.32 and 6.33 show the interface common to all schedules. Each schedule is associated with the type of the Event resource, and the type of the associated Event program. The type of the associated program in our interface is inferred from the type of the Event resource.

```
fn update_time(&mut self, time: Time);
fn shutdown(&mut self);
```

Figure 6.34: Interface common to all schedules.

To instantiate a schedule library users must supply a previously instantiated client, capable of making HTTP requests. Schedules must support multiple servers, whereas a `Client` instance pertains to a specific server. Therefore, we override the base URI in the `Client` instance when creating and sending automated responses from the schedule. For that reason, any `Client` instance can be used.

A thread-safe reference to our `SEDevice` abstraction is also required, as well as a callback to be called when the schedule needs to inform the client of event updates.

Schedules also take a duration value, determining the length of intermittent sleeps, addressing the same problem we discussed in 6.2.5. If not supplied, a default interval of 10 minutes is used.

One instantiated, events are added to the schedule via the `add_event` function, which takes a copy of the event, a reference to the event's program, and some unique ID pertaining to the server that the event was sourced from. This function is designed to be called every time a copy of the event is retrieved, via polling, or as a notification. Every event contains a unique MRID, and successive calls for the same MRID will simply update the status of the event, if the server has cancelled it.

If an event retrieved from a server has a status that differs from the schedule copy of the event, and that status is not cancelled, the client will log accordingly. In some cases this is expected, such as when working with events where the start-time is randomised on the client. In other cases such as when an event is marked as superseded on the server and is not locally, the client will log a warning.

We require that a reference to the Event's program is included in order to mark which program an Event was added from. This allows the schedule to send the "Event aborted due to alternate program event" response status, in Table 27 of IEEE 2030.5.

Similarly, we require that some unique 8 bit integer is provided as a form of server ID, as to mark internally which events were sourced from different servers. The, when an event from one server supersedes an event from another, we can send the "Event aborted due to alternate server event" response status, also in Table 27 [0EE18].

Finally, we provide a function for cleaning up the Tokio tasks spawned by the Schedule, freeing all allocated resources. This currently requires the shutdown function to be manually called. In the future, we'd like to have

it automatically be called when the Schedule is dropped, which is a straight-forward refactor.

SEDevice Abstraction

Responses to Events, auto-generated or otherwise, contain device-specific information. For all event function sets this is the device's short form identifier, and long form identifier. In the event of the Demand Response and Load Control function set, devices also return data representing their state, such as their applied load reduction value. Furthermore, events should only be applied when the category of device the Event targets matches the category of the device.

To inform a given event schedule of all these variables, we provide library users with an abstraction that represents a singular IEEE 2030.5 end-user energy device. This is the `SEDevice` struct. In addition to the data already mentioned, this struct contains an instance of an EndDevice resource, such that all information a developer may require when referring to a singular device is available under the singular type.

```
pub struct SEDevice {
    pub lfdi: HexBinary160,
    pub sfdi: SFDIType,
    pub edev: EndDevice,
    pub device_categories: DeviceCategoryType,
    #[cfg(feature = "drlc")]
    pub appliance_load_reduction: Option<ApplianceLoadReduction>,
    #[cfg(feature = "drlc")]
    pub applied_target_reduction: Option<AppliedTargetReduction>,
    #[cfg(feature = "drlc")]
    pub duty_cycle: Option<DutyCycle>,
    #[cfg(feature = "drlc")]
    pub offset: Option<Offset>,
    #[cfg(feature = "drlc")]
    pub override_duration: Option<Uint16>,
    #[cfg(feature = "drlc")]
    pub set_point: Option<SetPoint>,
}
```

Figure 6.35: Definition of the SEDevice struct.

Figure 6.34 shows the contents of this struct. For the sake of modularity, we use Rust compile flags to hide the DRLC function set specific fields when the DRLC schedule is not being used.

Ergonomics

Once again, we place as few restrictions on the library user as possible. Users can implement a callback via a manual trait implementation, or by any function or closure with a matching function signature.

```
pub trait EventHandler<E: SEEvent>: Send + Sync + 'static {
    async fn event_update(&self, event: &EventInstance<E>) ->
        ResponseStatus;
}
```

Figure 6.36: Definition of the Scheduler trait.

Figure 6.34 defines the trait containing this function signature. A single argument function that takes in an EventInstance, containing the corresponding event resource, and returns a ResponseStatus.

The `EventInstance<E>` type provides users with a way to inspect the event, providing users with:

- The current status, as a Rust enum.
- The underlying event resource.
- The primacy of the program.
- The start time and end time of the event, as a Unix timestamp.
- The corresponding program MRID.
- The user-defined server ID.

When determining the ResponseStatus to return, library users need only to refer to Table 27 of the specification.

If the returned status implies the client device is opting out of the event, the scheduler also knows to mark the event as cancelled internally. In the event an event is superseded whilst active, our scheduler would have more information on what response to send to the server, and in that case the provided response status is ignored.

Since these cases are non-trivial, we provide an explanation of the behaviour of the callback as part of our library documentation.

Internals

All implemented schedules share the same core implementation. `EventInstance`s are stored in a hashmap, and the unique event MRID is used as a key into the hashmap. This gives us constant time lookups for events, and ensures a schedule cannot store duplicate events. Notably, unlike the hashmap used in the Subscription / Notification mechanism, our `EventsMap` uses the default, more HashDoS resistant, hashing algorithm. We

make this decision as this hashmap does grow as the schedule runs, and MRIDs are technically provided as external input, making the possibility of a `HashDoS` attack possible, such as if resources were not retrieved (or sent) securely. Realistically, this is perhaps an overly cautious measure; `aHash` still provides resistance to `HashDoS` attacks.

```
type EventsMap<E> = HashMap<MRIDType, EventInstance<E>>;

struct Events<E>
where
    E: SEEvent,
{
    map: EventsMap<E>,
    next_start: Option<(i64, MRIDType)>,
    next_end: Option<(i64, MRIDType)>,
}
```

Figure 6.37: Internal Events data structure.

Figure 6.36 shows our wrapper around this hashmap that provides two values that act as hashmap invariants. These invariants store the time, and the MRID of the next event to start and the next event to end, or `None` if there are no events waiting to start or end. When instantiated, a schedule creates two Tokio tasks that sleep intermittently until it's time for an event to start.

This `Events` wrapper is designed with the intention of minimizing the overhead when a Tokio task wakes from sleep and checks if it's time for the next event to start or end, it would only need to read a single value.

When these tasks find that an event should be started or ended, it updates it's status, notifies the client via the callback and uses the output of the callback to send a response to the server. When constructing a response, the schedule uses the data stored in the `SEDevice` instance it was passed, and uses the last Time resource provided to it via `update_time`.

Finally, as to avoid our hashmap of events growing endlessly as the program runs, we have a third Tokio task act as a form of garbage collection, deleting cancelled or superseded events if they have not been updated for an extended period of time. Currently, this is configured at a week, but should be adjusted in the future as necessary, or provided as a configuration option.

The remainder of the schedule implementation differs for each function set. We've attempted to eliminate as much duplicate code as possible, however, we've been somewhat unsuccessful. Each schedule behaves sufficiently different to each other such that any further improvements would be pushing the boundaries of what is currently possible in Rust.

DER

The DER function set extends our core schedule implementation with events that can supersede one another, and a set of rules defining how and when this occurs. Therefore when an event is added to a DER schedule, it compares it with all other events in the schedule, and determines if it supersedes, or if it is superseded by another.

We determine if two `DERControl` instances supersede by checking if hey both target the same hardware controls (the fields of DERControlBase) and if their active time periods overlap. If this is the case, the superseding event is the one with the lower program primacy, or if their primacies are the same, the event with the latest creation timestamp.

If an event is superseded and active, the user-defined callback is called, and the appropriate response to the server created and sent, as per Table 27, including the necessary comparisons between program MRIDs and server IDs.

We are also required to handle the case where an event is superseded by another, and then the superseding event is cancelled before the superseded event would start. To handle this we store, for each event, a list of MRIDs corresponding to events that supersede it. When an event is cancelled, it's removed from all those lists. Any superseded events that would not have started yet with an empty list can therefore be un-superseded. This behaviour isn't explicitly mentioned in the specification, but follows naturally from the given rules. Additionally, this behaviour is implemented by EPRI C in their library.

DER events are 'randomizable' events. Therefore, when adding them to our Schedule we first determine their start time and interval with a random offset within the given range.

Since DERControl events apply only to specific device categories, we check if there is an intersection between the event targeted categories, and the categories set in `SEDevice`.

Finally, the interface we defined in Figure 6.22 is responsible for checking that a valid response can be constructed from a ResponseStatus and Event, and that it adheres to Table 27. If this is not the case, and an automated response cannot be made for any reason, the schedule will log accordingly.

DRLC

The DRLC function set schedule operates near identically to the DER schedule, with different types, and one difference in behaviour. They both exhibit 'direct control' over a device, and as such, only one event can be active at any given point in time.

Unlike DER, `EndDeviceControl` instances do not target specific hardware controls. To check if two events supersede we need only check if they are active at the same time, and then use their primacy and creation time to determine which supersedes which.

Once again, the interface in Figure 6.22 ensures created responses are valid, where created responses require additional information from the supplied `SEDevice` instance.

Messaging

The Messaging function set schedule differs greatly from the others, in that multiple `TextMessage` events can be active at once. For that reason, the implementation remains the same, but without any tracking of superseded events, and slightly different possible response status values.

Pricing

The Pricing function is very much similar to DER and DRLC. One difference is that a Pricing event requires both the `TariffProfile` resource, and the `RateComponent` resource. Together, these form the "Program" of the event. "Pricing clients shall support at least one RateComponent instance for each TarifProfile." and "Pricing servers SHALL provide at most one active TimeTariffInterval per rate component." indicate this [oEE18].

Therefore, we use the provided `RateComponent` resource to determine the program MRID, and the provided `TariffProfile` to determine the primacy of the `TimeTariffInterval` resource.

By that measure, a `TimeTariffInterval` event supersedes another if their active times overlap, and they belong to the same rate component.

Flow Reservation

The Flow Reservation function set contains an Event resource (`FlowReservationResponse`), and it contains a response for that Event, `FlowReservationResponseResponse`. However, it does not contain an Event program, nor does the specification provide instructions for under what circumstances clients should send `FlowReservationResponseResponse`, and with what response status.

IEEE 2030.5 Table 27 details response types by function set, and the Flow Reservation function set is not at all present. Furthermore, IEEE 2030.5 is unclear on under what circumstances `FlowReservationResponse`

events are permitted to overlap, and when they should be superseded.

Whilst we could implement a Flow Reservation schedule with minimal assumptions, it's interface and behaviour would differ greatly from all the other schedules. It would defer to the client supplied callback in all cases, and require them to manually specify every response. Unlike every other schedule, it cannot some program resource type.

If one were to implement the FLow Reservation schedule with a specific use case in mind, it would be a straight forward implementation, and a great deal of existing code in our client library could be reused. For that reason, we do not provide an implementation for this schedule, and instead leave it to library users to implement as required. We have, nonetheless, provided them with an ample resource for developing an IEEE 2030.5 event schedule, our codebase.

6.2.9 Logging

Our client library exposes log messages under a generic logging facade. We use industry standard log levels, `FATAL, ERROR, WARN, INFO, DEBUG` and `TRACE`. To accomplish this we use the `log` crate, developed by the Rust foundation [Rus23b]. The logs we create can therefore be attached to any given log implementation or logging frontend, whatever is most suitable for the users of the library.

As many parts of the client library run in the background (Polling, notification server, event schedule) logging is, in many cases the only way to discern that an error has occurred, and that the client has recovered.

Situations that result in a `WARN`, or an `ERROR` log include:

- When a regularly polled resource could not be retrieved.
- If there is a mismatch between the status of an event in the schedule, and a copy retrieved from a server (and then fed into the schedule).
- If an incoming Notification server request or connection could not be handled.

Situations that result in a `INFO` log include:

- When a resource is retrieved, from polling or otherwise.

- When a POST or PUT request is successful.
- When jobs that run in the background shutdown / cleanup (Schedules, Servers).

6.2.10 Testing

This client library has been built alongside tests with the goal to maximise test coverage. When running LLVM source-based coverage on `sep2_client`, the `grcov` report claims we have 71.11% coverage, or 1297 /1824 lines of code covered by tests [Moz23] [Fou23a].

In section 7.6 we discuss future improvements to testing.

Mock Server

With the goal of testing the client's ability to make HTTP requests, we required a way to mock an IEEE 2030.5 server. For this, we implemented a 'dumb' IEEE 2030.5 server that produces hardcoded resources and responses for specific inputs. This test server follows a near identical implementation to that of the Subscription / Notification mechanism. We use this test server to ensure functions that make HTTP requests fail and succeed as expected, and that resource polling retrieves the correct number of resources over a given time period.

We also use this server to conduct system tests. These system tests follow the examples outlined in IEEE 2030.5 Annex C, currently only testing the core client functionality.

Subscription / Notification

In order to send notifications to clients, IEEE 2030.5 need to operate a client capable of making HTTP POST requests. In our client library, a `Client` instance is separate from a `ClientNotifServer` instance, meaning our testing process is straight-forward - we make requests to the notification server using our client.

Our tests ensure that a client interacting with the server is capable of receiving HTTP POST requests, and that the expected response is received. We also implicitly test that our notification server does not crash during regular use.

Event Schedules

As each schedule for each function set behaves differently, they are tested separately. For each schedule we create a set of of unique events that each last several seconds, and compare the output on the given callback with the expected output.

Cases we test include:

- Basic sequential execution of events.
- Cancelling & superseding events in progress.
- Cancelling & superseding events before they start.
- Ensuring events superseded by cancelled events are rescheduled.
- Ensuring Events with different primacy values supersede as required.
- Ensuring DER events with differing controls do not supersede.
- Ensuring TextMessages do not supersede at all.
- Ensuring TimeTariffIntervals from different RateComponents do not supersede one another.

Security

During development, we maintained a local certificate authority via the tool `mkcert` and used it for testing [Val22]. However, this tool is not capable of producing certificates that match those required by IEEE 2030.5

In order to properly test our Security function set, we required TLS certificates as close as possible to those used in the real world; certificates that include all the relevant X509 extensions, and are generated using the specified elliptic curve, and are compatible with the mandatory cipher suite. For this reason, we consult the SunSpec Alliance, who provide free IEE 2030.5 test certificates upon request. As part of this thesis we have requested certificate packages for both our client, and test server, and have tested them when communicating resources, and our interface for parsing and validating X509 certificates.

However, these certificates do not provide us a way with interacting with any publicly accessible IEEE 2030.5 test servers. For that, we'll require test certificates from Energy Queensland, as discussed in section 7.6.

Chapter 7

Evaluation & Future Work

Despite all we've accomplished during this thesis, there are many features and improvements we haven't been able to implement purely due to time constraints. Furthermore, we've been able to identify some flaws in our existing design - potential pain points for developers using our library.

In this chapter we'll compare our implementation to an existing solution, and discuss it's weaknesses and how we plan to improve them. Through this, and our released project, we provide a plan for any and all future work.

7.1 Comparison with the EPRI IEEE 2030.5 Client

We'll first provide a reference point for our progress on this thesis project by comparing it to that of the EPRI IEEE 2003.5 client library, implemented in C. This library is the oldest, and most popular, open-source implementation of the client protocol.

Table 7.1 shows a comparison of features and functionality between our implementation, and EPRI's.

Of note is that the EPRI implementation uses minimal third-party libraries, where the security and reliability of required standards (HTTP, XML) is unlikely to have been tested as extensively as the popular open-source libraries used in our implementation.

Of note is that our common and client library do not include implementations for DNS-SD and EXI resource serialisation & deserialisation. In sections 5.2 and 5.3 we discuss the reasoning behind this.

In terms of Event function sets, our client has a more complete implementation. We provide library users with

Feature	sep2_common + sep2_client	EPRI 2030.5
XML Resource Serialisation & Deserialisation	Yes	Yes
EXI Resource Serialisation & Deserialisation	No	Yes
DNS-SD	No	Yes
TLS via OpenSSL	Yes	Yes
Resource Scheduled Polling	Yes	Yes
Subscription / Notification Mechanism	Yes	No
DER Event Scheduler	Yes	Yes
DRLC Event Scheduler	Yes	No
Pricing Event Scheduler	Yes	No
Messaging Event Scheduler	Yes	No
Flow Reservation Event Scheduler	No	Yes
Scheduler Time Offset	Yes	No
Global Time Offset	Yes	Yes
CSIP-AUS Extensions	Yes	No
QualityLogic Formal IEEE 2030.5 Testing	No	Yes

Table 7.1: Table comparing `sep2_common + sep2_client` and the EPRI IEEE 2030.5 Client.

black-box schedulers handling DER, Messaging, Pricing, and Demand Response and Load Control events. The EPRI C implementation only includes a schedule for Distributed Energy Resources. The same goes for applying Time resources at a schedule level, EPRI provides no way to have different schedules use different time resources.

Importantly, EPRI's implementation does NOT include a subscription / notification mechanism, the resource retrieval method mandated as part of CSIP. [All17].

However, EPRI's implementation has the benefit of having undergone formal testing provided by QualityLogic, which is a paid service. Due to the cost of the service, it is unlikely our implementation will have the opportunity to undergo the same testing.

Finally, our implementation has the benefit of being written in a memory safe language, safe Rust. The EPRI implementation is written in C, where memory safety is not proven by the compiler.

7.2 EXI Serialisation & Deserialisation

Going forward, we must explore how EXI support can be added to the common library. It's likely that it would be implemented alongside a rewrite / redesign of `SEPSerde`, such that that the sole library could wrap an EXI library, and an XML library, and produce both as required. Though this EXI library could be implemented

using FFI to a C library, there are complete implementations of EXI parsers and writers in modern languages, and it would be more appropriate to use those as reference implementations for a memory safe fully Rust EXI library. [Pei17]

7.3 DNS-SD

Although we've deemed this feature not wholly necessary for this iteration of the client, it is required for full IEEE 2030.5 adherence, and as such we must explore ways to implement it going forward.

Whilst we could implement this from scratch using just Rust standard library UDP sockets, the open-source tool `Avahi` provides DNS-SD on Linux. There also exists a tool `xmDNS-Avahi` that provides an `xmDNS` server, required by IEEE 2030.5, by modifying `Avahi`. Furthermore, `xmDNS-Avahi` cites IEEE 2030.5 as a motivation for requiring an `xmDNS` server. It's therefore likely wrapping this library using Rust FFI would be the most appropriate approach. [Poe] [Cha15]

7.4 Rust Async Traits

As of this report, the stable branch of the Rust compiler does not support 'Return Position Impl Trait' (RPIT) in traits (RPITIT). It is simply not supported by the type system [Gou23].

RPIT is Rust's implementation of existentially quantified types. Functions using RPIT simply return an opaque type that implements a specific trait. e.g. `fn foo() -> impl Display` is a function that returns some type that can be displayed.

This functionality is necessary in expressing async Rust at the type-level. In Rust, async functions return state machines, types that implement the 'Future' trait. However, these state machines are auto-generated by the compiler and therefore require the use of RPIT to return. Therefore, the async keyword is simply syntactic sugar for `impl Future<Output = T>`, where T is the final value of the state machine.

Our implementation defines a trait for each of the possible callback types. We want library users to be able to supply asynchronous callbacks, and as such, we require RPITIT.

Currently, there is a popular workaround to this that involves utilizing Rust's dynamic dispatch to heap allocate, and then perform type erasure on the future before returning it, as to allow the function to return a concrete

type. This workaround is achieved by having the `async_trait` crate generate the necessary code, although it could be accomplished by hand also [Tol23].

This has the obvious downside of incurring an unnecessary heap allocation every time an async function in a trait is called. In our library this is whenever a user defined callback is called, or when an event is added to a schedule.

Fortunately, with the release of the Rust compiler v1.75 on the 28th of December 2023, RPITIT will be stable, and therefore so will async functions in traits [Rus].

This comes with a caveat. When async trait functions are declared but not defined, there is no way to infer what bounds on the return type can be relaxed. More specifically, the compiler cannot infer that all implementations of that trait function return a type that can be sent or referenced across thread boundaries (the Rust Send & Sync traits). The solution to this is in active development, and is referred to as "minimal associated return type notation" [Gro23].

The Tokio runtime is a 'work-stealing' executor. In practice, this means Rust futures we return must implement the `Send` trait, this means they can be sent across thread boundaries. Since we cannot express this using an async function in a trait, we would fallback on what would be the desugared syntax alternative, `impl Future<Output = T> + Send`. This results in a marginally less ergonomic interface, as implementers of the trait now need to wrap their code in a async block themselves. However, this is still an improvement on the performance overhead and additional dependency that we require for the current workaround.

7.5 Native Rust TLS

In section 6.2.3 we discussed how make use of the available OpenSSL Rust bindings to implement the security function set, and so far this has been sufficient. However, we note the 19 reported CVEs in OpenSSL in 2023 alone as a slight cause for alarm [Ope23b].

However, there exists a project called `rustls` that eventually aims to compete with OpenSSL by providing a Rust native implementation of TLS for server and client verification, and bulk encryption [rus23d], leveraging the memory safety guarantees of Rust. `rustls` uses the `ring` crate, which provides (almost) native Rust cryptography algorithms [Smi23]. In early benchmarks, `rustls` is shaping up to be more performant and more memory efficient than OpenSSL. [BP19]

Currently, the `ring` crate does not support Authenticated Encryption using CCM, as required by IEEE 2030.5, stalling progress on being able to use `rustls`. As part of our library release, we will inquire with the status of this implementation, and what further work is required to add the functionality to `ring`. Currently, progress on this feature in ring has stalled, and there may be avenues for us to contribute to ring directly [Cra22a] [Smi15].

7.6 HTTP Implementation Improvements

Whilst we use the `hyper` client as part of our implementation, and not the more commonly used wrapper around it, `reqwest`, we lose a great deal of functionality that is necessary for both ergonomics, and IEEE 2030.5 requirements [McA23].

One such crucial example is the ability to handle HTTP redirects without having to implement it ourselves. With the recent release of `hyper 1.0`, this problem is exacerbated, with the library requiring more of the lower level details be handled in our implementation, as it aims to provide a more generic, less opinionated, HTTP implementation.

`reqwest` supports TLS using either OpenSSL (via the `native_tls` crate), or `rustls`. However, when used with OpenSSL it does not expose a way to specify the cipher-list used. We require this functionality to force the use of the IEEE 2030.5 cipher suite. One workaround to this would be to have users of our library modify the OpenSSL source-code, and compile a version that has our cipher suite as the default. This is far from ideal from a usability perspective [Cra22b].

Aside from this, we do not require any of the extra low-level functionality provided by `hyper`. Outside of security, `reqwest` is the library better suited to our use case.

Therefore, to use `reqwest` we require that `native_tls` and `hyper` expose a way to modify the cipher-suite used. As `native_tls` is designed to be cross-platform, and OpenSSL is only used on Linux, this seems like an unlikely addition.

Alternatively, to use `reqwest` we require that `rustls` support the IEEE 2030.5 cipher suite, which first depends on progress on `ring`.

In the meantime, we will continue to use `hyper`, and make users of our library aware of the current deficiencies in the library as a result, using GitHub issues. Furthermore, we are very much motivated to look into the

possibility of contributing to `rustls`, `ring`, `reqwest` or `native_tls`, and implementing the required functionality ourselves.

7.7 Maintainability

With the goal of this thesis to produce an IEEE 2030.5 client library implementation that can be released under open-source licenses, we develop our implementation with a focus on ensuring it can be maintained into the future.

As of present, this takes the form of:

- Generated 'rustdoc' documentation, as is preferred by the Rust open-source community.
- Thorough internal documentation, with explicit references to IEEE 2030.5 to justify behaviours.
- Example client binary source-code, showing the DER function set and event scheduler in use under the direct / individual model.
- GitHub project management, detailing planned enhancements and broken functionality as GitHub issues.
- The modular nature of our implementation, with responsibilities delegated to different Rust crates, and compile flags.

In order to ensure our client is maintainable into the future, we will:

- Produce additional full client binary examples, showing usage of the subscription / notification model, and other event schedules.
- Develop function usage examples, and include them as part of the rustdoc.
- Respond to and engage with library users, and their feedback. Working with users to fix issues they report.
- Upload our implementation to the `crates.io` package manager, which will host our rustdoc, and handle version releases. This will also allow us to have a singly sourced README for both github and rustdoc.

7.8 Testing

As it stands, our client and common library are reasonably well-tested, with code coverage of 63.79% when measured by lines of code. Keeping in mind, a large portion of this is compile-time generated code that has already been tested, we would nonetheless like to improve this going forward by:

- Testing our resource serialisation against all remaining example XML in Annex C.
- Testing our implementation against all remaining example system tests in Annex C.
- Testing automated schedule responses are sent as required (only client output is tested currently).
- Improving coverage wherever possible.

Furthermore, we would like to test our implementation against production-ready IEEE 2030.5 servers. There are currently two candidates against which this testing can be done, the `GridAPPS` IEEE 2030.5 server implementation, and the publicly accessible IEEE 2030.5 Test API hosted by Energy Queensland [Que23] [GD]. Both of these servers are poorly advertised, and were discovered very late into this thesis.

Going forward, we will investigate if and how these servers can be used to further test our implementation. Of note, is that the Energy Queensland test server accepts certificate signing requests, and thus gaining access to that server is very much possible.

(A previous figure of 73% code coverage was previously given, however this figure included the tests as part of the coverage, and was therefore incorrect).

7.9 Authenticating Notification / Subscription

As of present, a `ClientNotifServer` can be instantiated without TLS. This is done as to not restrict library users to performing TLS termination at the application, and allowing them to run the notification server behind a reverse proxy or load balancer, such as `nginx` or `Apache`. However, this poses some difficulties when paired with `mTLS`. It is not immediately obvious how client notification servers will perform authentication of incoming requests. This was an issue also encountered by BSGIP, and discussed in their report "On the implementation and publishing of operating envelopes" [BSG21].

It's likely that library users will want to define this authentication procedure themselves. For that reason, we will likely need to discuss with them how they wish to implement this functionality, and use that to determine if and how it can be implemented into our client in a generic way.

7.10 Error Handling

One potential pain point in our current implementation is our usage of opaque error types. All errors returned by our interface resolve to a single Error type, provided by the `anyhow` crate. The exception to this is are parts of the core client functionality interface, as we discussed in 6.2.4.

This design simplifies our error handling, but limits the potential error cases users of our library can handle. Since we are unable to envisage exactly how users of our library might want to handle errors, it would be more appropriate to use transparent error types that provide more insight into why some operations failed.

Fortunately, our interfaces expose few very functions that return an error. Many error cases occur running in the background. Consequently, improving error handling would require minimal changes, yet careful consideration.

One case where improvements could be easily made are user-defined callbacks. Currently our implementation only calls the poll and notification callbacks when a valid resource could be deserialised, it does not allow the library user to specify a specific behaviour when an error occurs. We could change this to provide users with the `Result` enum directly, and allow them to handle the error case however they wish. This change makes more sense to implement once we provide more transparent error types.

7.11 Benchmarking & Profiling

We currently have no data, or frame of reference, for the performance of the client.

Going forward, it would be extremely beneficial to perform some form of profiling on an instance of the client in a production context. Particularly, we are interested in what, if any parts of the code are responsible for excessive heap allocations, or uncessecarily complex computations that may impact performance. We are also interested in the memory footprint of the client over an extended period of time, to ensure that memory is being freed appropriately. For example, that the schedule garbage collection process is occurring.

Furthermore, we would like to know what parts of our implementation run most frequently and take the longest time to complete, the 'hot' function calls that we would need to optimise further.

It's likely these will be the functionality responsible for serialising & deserialising resources, and will further incentivise us to rewrite `SEPSerde` from scratch.

It would also be sensible to conduct benchmarks of the client when completing a fixed workload, such as several completing several requests simultaneously, and then compare those benchmarks to that of the EPRI C implementation.

Chapter 8

Conclusion

To conclude, we have designed and implemented Rust libraries for developing IEEE 2030.5 clients with an emphasis on safety, correctness, reliability, and interface ergonomics. Furthermore, our work paves the way for IEEE 2030.5 servers to be implemented in Rust. In this thesis we have discussed the high-level design decisions, and the technical considerations that have guided our implementation. We have also evaluated our progress, and explored future improvements.

With the core functionality implemented, our libraries have been released under open-source licenses, and those interested in building IEEE 2030.5 clients can begin learning and experimenting with the API we provide. In the future, we will work with users and potential contributors to address issues, and implement additional functionality, such that we can produce a fully compliant, high quality IEEE 2030.5 client implementation.

Bibliography

[Age23] Australian Renewable Energy Agency. Common smart inverter profile - australia. https://arena.gov.au/assets/2021/09/common-smart-inverter-profile-australia.pdf, January 2023.

[All13] ZigBee Alliance. Zigbee specification faq. https://web.archive.org/web/20130627172453/http://www.zigbee.org/Specifications/ZigBee/FAQ.aspx, June 2013.

[All17] SunSpec Alliance. Ieee 2030.5/ca rule 21 foundational workshop. https://sunspec.org/wp-content/uploads/2019/08/IEEE2030.5workshop.pdf, June 2017.

[All21] SunSpec Alliance. Sunspec modbus. https://sunspec.org/sunspec-modbus-specifications/, 2021.

[Arl19] Ember Arlynx. Relicense assistant. https://github.com/emberian/relicense-assistant/blob/master/single-issue-template.md, 2019.

[Bit23] Bitflags. Bitflags documentation. https://docs.rs/bitflags/latest/bitflags/, 2023.

[BP19] Joseph Birr-Pixton. Tls performance: rustls versus openssl. https://jbp.io/2019/07/01/rustls-vs-openssl-performance.html, jul 2019.

[BSG21] BSGIP. envoy-client. https://github.com/bsgip/envoy-client, 2021.

[Cha15] ChargePoint. xmdns-avahi. https://github.com/ChargePoint/xmDNS-avahi, 2015.

[Cra22a] Adam Crain. Add aead aes 128 ccm support. https://github.com/briansmith/ring/pull/1501, June 2022.

[Cra22b] Adam Crain. Tls 1.2 ccm modes. https://github.com/sfackler/rust-native-tls/issues/227, May 2022.

[Cut22] CutlerMerz. Review of dynamic operating envelope adoption by dnsps. https://arena.gov.au/assets/2022/07/review-of-dynamic-operating-envelopes-from-dnsps.pdf, August 2022.

[Fac22] Steven Fackler. hyper-openssl. https://github.com/sfackler/hyper-openssl, February 2022.

[Fou23a] Rust Foundation. Instrumentation-based code coverage. https://doc.rust-lang.org/rustc/instrument-coverage.html, 2023.

[Fou23b] Rust Foundation. Platform support. https://doc.rust-lang.org/nightly/rustc/platform-support.html, 2023.

[FR14] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014.

[GC23] Pacific Gas and Electric Company. Electric rule no.21. https://www.pge.com/tariffs/assets/pdf/tariffbook/ELEC_RULES_21.pdf, February 2023.

[GD] GridAPPS-D. Gridapps-d ieee 2030.5 server. https://pypi.org/project/gridappsd-2030-5/.

[Gou23] Michael Goulet. https://github.com/rust-lang/rust/pull/115822, September 2023.

[Gro16] California Smart Inverter Implementation Working Group. Ieee 2030.5 common california iou rule 21 implementaton guide for smart inverters. https://sunspec.org/wp-content/uploads/2017/02/CSIPImplementationGuide-v1-0.pdf, August 2016.

[Gro23] Rust Async Working Group. Minimal associated return type notation. https://hackmd.io/@wg-async/H1qTVUMgn, May 2023.

[Hei08] Bob Heile. Zigbee smart energy. https://www.altenergymag.com/article/2008/10/zigbee-smart-energy/468/, January 2008.

[HM19] Daniel Henry-Mantilla. inheritance-rs. https://github.com/danielhenrymantilla/inheritance-rs, October 2019.

[Hyp23] Hyperium. hyper. https://github.com/hyperium/hyper, November 2023.

[Ins18] Electric Power Research Institute. Electric power research institute ieee 2030.5 client user's manual. https://www.epri.com/research/products/000000003002014087, July 2018.

[Kai23] Tom Kaitchuck. ahash. https://github.com/tkaitchuck/aHash, 2023.

[Kra23] Conrad Kramer. Performance is not comparable to other xml parsing libraries. https://github.com/netvl/xml-rs/issues/126#issuecomment-1542888016, 2023.

[Lum16] Gordon Lum. California use case for ieee2030.5 for distributed energy renewables. https://smartgrid.ieee.org/bulletins/december-2016/california-use-case-for-ieee2030-5-for-distributed-energy-renewables, December 2016.

[Lum22] Lumeo. xsd-parser-rs. https://github.com/lumeohq/xsd-parser-rs, August 2022.

[Lum23] Lum::Invent. Yaserde. https://github.com/media-io/yaserde, February 2023.

[Mat23] Vladimir Matveev. xml-rs. https://github.com/netvl/xml-rs, September 2023.

[McA23] Sean McArthur. reqwest. https://github.com/seanmonstar/reqwest, November 2023.

[Mil19] Matt Miller. A proactive approach to more secure code. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/, July 2019.

[Moz23] Mozilla. grcov. https://github.com/mozilla/grcov, 2023.

[oEE11] Institute of Electrical and Electronics Engineers. Ieee guide for smart grid interoperability of energy technology and information technology operation with the electric power system (eps), end-use applications, and loads, 2011.

[oEE18] Institute of Electrical and Electronics Engineers. Smart Energy Profile Application Protocol (2030.5-2018), December 2018.

[Ope23a] OpenSSL. Openssl. https://www.openssl.org/, 2023.

[Ope23b] OpenSSL. Vulnerabilities. https://www.openssl.org/news/vulnerabilities.html, 2023.

[Pei17] Daniel Peintner. Exificient.js. https://github.com/EXIficient/exificient.js, July 2017.

[Poe] Lennart Poettering. Avahi. https://www.avahi.org/.

[pyf23] pyfisch. httpdate. https://github.com/pyfisch/httpdate, August 2023.

[Que23] Energy Queensland. Sep2 client handbook. https://www.energex.com.au/__data/assets/pdf_file/0007/1072618/SEP2-Client-Handbook-13436740.pdf, September 2023.

[Rus] Rust. Rust 1.75. https://releases.rs/docs/1.75.0/.

[Rus23a] Rust. hashbrown performance. https://github.com/rust-lang/hashbrown#performance, 2023.

[Rus23b] Rust. log. https://github.com/rust-lang/log, 2023.

[Rus23c] Rusticata. x509-parser. https://github.com/rusticata/x509-parser, 2023.

[rus23d] rustls. rustls. https://github.com/rustls/rustls, November 2023.

[Smi15] Brian Smith. Add ccm aead. https://github.com/briansmith/ring/issues/25, 2015.

[Smi23] Brian Smith. ring. https://github.com/briansmith/ring, November 2023.

[SVC+20] Partha S. Sarker, V. Venkataramanan, D. Sebastian Cardenas, A. Srivastava, A. Hahn, and B. Miller. Cyber-physical security and resiliency analysis testbed for critical microgrids with ieee 2030.5, 2020.

[Tan21] Tom Tansy. Get on the modbus: An explanation of ieee 1547 and why it matters for solar installers. https://solarbuildermag.com/news/get-on-the-modbus-an-explanation-of-ieee-1547-and-why-it-matters-for-solar-installe March 2021.

[Tok23a] Tokio. Tokio. https://github.com/tokio-rs/tokio, 2023.

[Tok23b] Tokio. Tokio documentation. https://docs.rs/tokio/latest/tokio/, 2023.

[Tok23c] Tokio. tokio-openssl. https://github.com/tokio-rs/tokio-openssl, 2023.

[Tol23] David Tolnay. async trait. https://github.com/dtolnay/async-trait, 2023.

[Tri] Josh Triplett. https://news.ycombinator.com/item?id=21566968.

[Val22] Filippo Valsorda. mkcert, April 2022.

[Wei21] Ben Weise. On the implementation and publishing of operating envelopes. https://arena.gov.au/assets/2021/04/evolve-on-the-implementation-and-publishing-of-operating-envelopes.pdf, april 2021.