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 also 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 previously.

I would also like to acknowledge the efforts of all contributors to free and open-source software, especially in the Rust ecosystem. This thesis simply 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

CPUC California Public Utilities Commission

CPU Central Processing Unit

CSE Computer Science and Engineering

CSIP-AUS Common Smart Inverter Profile Austrai

CSIP Common Smart Inverter Profile

CVE Common Vulnerabilities and Exposures

DER Distributed Energy Resources

DNS-SD Domain Name System - Service Discovery

DNSP Distribution Network Service Provider

DNS Domain Name System

ECC Electric Curve Cryptography

EPRI Electric Power Research Institute

EXI Efficient XML Interchange

FS Function Set

HAN Home Area Network

HTTP Hypertext Transfer Protocol

IANA Internet Assigned Numbers Authority

I/O Input / Output

IEEE Institute of Electrical and Electronics Engineers

IP Internet Protocol

IoT Internet of Things

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

RSA Rivest-Shamir-Adleman

SEP Smart Energy Profile

SIP Smart Inverter Profile

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

UNSW University of New South Wales

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 that exists 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 export AC power 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, 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.

The 2030.5 protocol 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 IEEE 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, the protocol was 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. The specification 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, where both allow for clients & servers to be deployed under a different model from that describe in the specification, whilst CSIP-AUS extends the possible structured data that can be communicated between clients & servers to better fit the requirements 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 schema, an XSD. Despite the client-server model, IEEE 2030.5 purposefully does not make distinctions between clients and servers, as to avoid resources 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. Device manufacturers or electric utilities implementing IEEE 2030.5 need only communicate resources from function sets relevant to the purpose of the device.

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, whether a resource can be subscribed to can be further refined by the server exposing the resource. For that reason, it is often 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, and all encryption done, using ECC cipher suites, with the ability to use RSA cipher suites as a fallback. Unlike regular TLS, these 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.

The standardised resource schema is defined in a 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, and discuss how they may influence our open-source implementation.

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 make these considerations to 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. For that reason, we let the implementation models for the Californian IEEE 2030.5 implementation guide influence our own development of the protocol, whilst of course still adhering to the specification.

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 this model there is a 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.

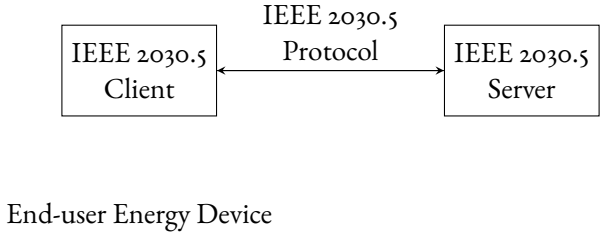

Figure 4.1: The Individual/Direct IEEE 2030.5 Model, as defined by California SIP

Aggregated Clients Model

The aggregated clients model, outlined in the implementation guide, is one 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.

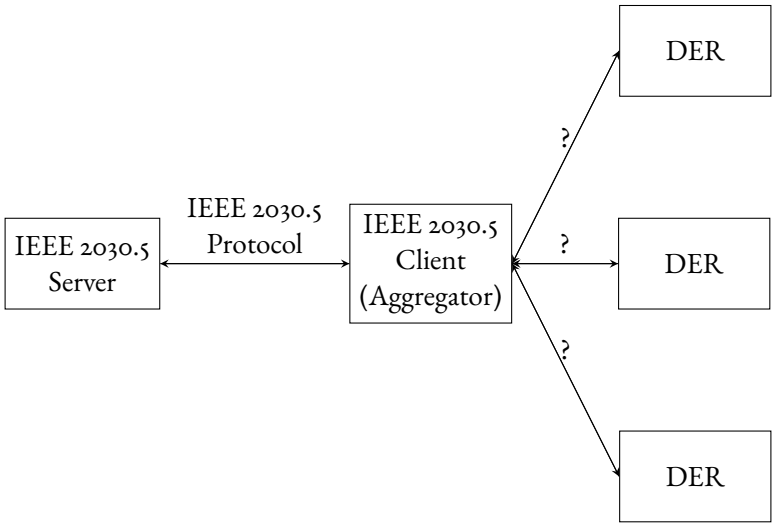

Figure 4.2: The Aggregated Clients IEEE 2030.5 Model, as defined by California SIP

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, as indicated in Figure 4.2. 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. This is mandated in order to reduce network traffic, and of course, allow for use of the protocol at scale.

Given the circumstances of this model, the aggregator 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" (Common SIP) (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, in addition to polling. 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, and identify the tradeoffs in each.

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 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, running as per user input.

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 API calls, 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.

The IEEE 2030.5 Client implementation by EPRI states, in it's User's 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, whilst universal, lacks many features present in more 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 could 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. For example, due to the lack of tagged unions (or algebraic sum types) in C, many functions exposed to users accept a typeless 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 it the process is capable of reading and/or modifying.

In contrast to this, 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 need 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 perform it's role operating under the client aggregation model. [Ins18,]

4.4.2 Battery Storage and Grid Integration Program: envoy-client

A newer 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 the IEEE 2030.5 Server library implementation `envoy`, and provides a very bare implementation of core IEEE 2030.5 Client functionality 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 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 parts of resources accessed are present, where unchecked accesses to data may lead to runtime errors.

With minimal dependencies, and without an interface for TLS, the library is as portable as Python itself.

Under these circumstances, the library is an ideal tool for quickly testing an IEEE 2030.5 Server implementation, but would likely struggle in real-world use, aggregating on behalf of clients.

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 consumer environment. Therefore, where possible, we design an implementation capable of operating under both. One example of this we'll discuss is 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-prioritised 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 libraries 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 must adhere to many other standards. In 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, TCP.

An exception to this is our usage of `openssl`, which in 2023 alone has had 19 reported CVEs [Ope23b]. In section TODO we discuss alternatives to this going forward.

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 series of Rust libraries (called 'Crates'), for use by developers to allow them to develop IEEE 2030.5 compliant software, providing them with an abstraction for interacting with IEEE 2030.5 servers and resources.

XML Serialisation & Deserialisation Library: SEPSerde

As we've established, resources can be communicated between clients and servers as their XML representations. This means we require the ability to serialize & deserialize 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. Note that we are yet to complete the renaming of this library, in all code snippets the library will still be referred to as `YaSserde`.

Common Library: sep2_common

IEEE 2030.5 resources are required for use in both clients and servers. For that reason, we've developed a common library with a Rust implementation of the IEEE 2030.5 XSD, whilst including our `SEPSerde` crate as to allow these resources to be serialized and deserialized to and from XML.

This common library, `sep2_common`, can then be easily integrated and implemented in a future 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.

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. For that reason, we have 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 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.

5.1.4 Open-source Software

Our implementation will be open-sourced as it aims to address the lack of open-source IEEE 2030.5 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.

As it stands, our client and common library are reasonably well-tested, with code coverage of 73% when measured by lines of code.

Furthermore, included alongside our implementation are inline comments that are used to generate 'rustdoc' documentation from our source code. Once generated, they can be distributed as a stand-alone website documenting the features and interfaces of each crate. We place an emphasis on ensuring that this documentation can be easily understood by those who are already familiar with the Rust programming language.

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.

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 primarily develop clients operating under the client aggregation model, it is necessary that our client 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 enagage in a specific behaviour over a given internal.
- 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, providing us with a zero cost abstraction for asynchronous programming.

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

Per the nature of the protocol, all software used must be reliable and all expected errors are to be recovered from gracefully. Client instances must run autonomously for extended periods of time, particularly under the client aggregation model. Failure to do so could possibly lead to a denial of service for electricity.

For that reason, we leverage Rust's compile-time guarantees on 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, such as in the standard library. C++ does not require programmers to handle these exceptions at compile-time.

As a result of Rust's type system, safe Rust 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 great deal of operating-system-specific functionality, and as such will need to 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 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, due to it's prevalence in server operating-systems.

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 further portability 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. [Fou23] [Tok23b]

In the case this portability is still not sufficient, our libraries are open-source and allow users to fork our implementation and modify it for their use case, with the vast majority of our code being as portable as Rust itself.

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. These assumptions have determined what functionality we have and have not implemented.

5.2.1 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 single HTTP route on their notification server that handles all incoming notifications. Instead, each notification containing a different resource would be sent to a different route. Each subscription would have a different `notificationURI` field on each subscription made.

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. Under a single route, each incoming HTTP request would first need to parse the body of the request to determine how to handle it. Using multiple routes simplifies this logic, requiring only the `Host` header be inspected.

If the client 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 the 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.

5.2.2 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 local 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 primarily 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 a nice-to-have for 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. We have no real measure, or way to determine how one might want to use our library.

Fortunately, we have the aforementioned 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 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.

5.3.3 Rust Usability

A major constraint for potential users of our library is that they will need to have a strong understanding of the Rust programming language. 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 how to write single-threaded Rust code.

It will be unreasonably difficult for a programmer to develop an IEEE 2030.5 client as their first complete Rust program.

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 largest aspect of the common library is the internal representation of resources, the data communicated between clients & servers. These resources 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 both specification and schema, these data structures are separated into packages for each of the function sets.

Whilst the 2030.5 specification makes no mention of the object oriented programming paradigm, OOP inheritance underpins the design of all resources, including both multi-level and hierarchial inheritance. As such, for the purpose of code reuse, many base types appear in the 'common' package; data structures extended by many others.

Representing Resources in Rust

Rust, despite being influenced by OOP, 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 type. Rather shared behaviour is defined through traits, where 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 inheritance. One can have traits that require other traits to 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 5.2, we have a Rust trait that describes the behaviour of the List base type. All lists are resources, and thus we have a trait restriction 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 still do not have have polymorphism using the base-types of resources. To allow for polymorphism a trait must be defined for each base-type, and then those trait functions need be implemented for every type that extends that base type. This is unavoidable duplicate code, although Rust provides mechanisms 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 SEResourceObj {
    href: Option<String>
}

impl SEResource for ResourceObj {
    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 SEList {
    #[inherits(SEResource)]
    res: SEResourceObj,
    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, however, we are not the first to require this 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 particular implementation supports 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 from 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 currently produces for the List resource. 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 true OOP inheritance, 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, in that 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 a concrete data type serialized into XML.

Further advocating for the use of an XSD parser is the fact that the 2030.5 XSD is updated more often than revisions of the specification itself, even if those updates are relatively infrequent. Using xsd-parser-rs will allow us to better maintain the client, by way of simply running updated schemas through the parser, should it change drastically as a result of a major revision (although this is unlikely).

Of course, `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, and modify it to suit our needs. As part of our implementation we've:

- Instructed it to use the Rust Option type where `minOccurs=0` and `maxOccurs=1`.
- Instructed it to provide us with all types in a type's inheritance tree.
- Manually implemented Rust enums for all types that are integer enumerations.
- Manually implemented Rust `bitflags` for all integer bitmap types.
- Modified the Derive output on all data types to automatically implement traits where useful.

Integer Bitmaps

IEEE 2030.5, for the sake of efficiency, has many fields that encode many boolean values together in a single integer, a bitmap of booleans, or 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. [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, such as by allowing them to store all "Subscribable" resources in a single container.

Therefore, as part of the common library, we define a set of Rust traits that mirror common fields in 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 into our traits. For example, `RespondableSubscribableIdentifiedObject` inherits from `RespondableSubscribableObject`, providing it with three new fields that allow it to be 'Identified'. Similarly, `RespondableIdentifiedObject`

inherits from `RespondableObject`, also providing it with the 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.

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

Figure 6.7: Our 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 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 `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 serialize and deserialize resources to and from the appropriate XML. This means HTTP requests containing resources are sent with `Content-Type` set to `application/sep+xml`.

On the public Rust crates registry, there exists a popular XML serialisation and serialization 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`, and as such, is to be used in our client implementation.

Referring again to Figure 6.5, we see `xsd-parser-rs` has qualified our struct with the appropriate namespace, such that YaSerde serializations will include it, and deserializations 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 serialized 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:

- Serialize Rust enums as their underlying integer representations.
- Allow for Rust trait objects (dynamic dispatch) to be created on the serialize and deserialize traits.
- Serialize and deserialize IEEE 2030.5 `HexBinary` primitives as hexadecimal with an even number of digits.
- Allow `bitflags` generated bitmaps to be serialized and deserialized.
- Allow for generic resources to be serialized and deserialized (Notification)

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 monomorphization, and therefore has no runtime overhead in order to make the Resource type generic.

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 resources, but we also need a way to instruct it 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 responsible for generating the serialisation code also 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: Our Notification resource implementation

Figure 6.11 this in use, we append the additional namespace as required, and include the `generic` annotation. 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 are 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 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.

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 it is now.

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 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 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 'vtable' or '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 typess. 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 lookup mapping the names of resources to code that deserialize based off the type of the resource - which we could likely auto-generate.

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 run-time 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.

Unfortunately, this is not described in any standardised 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: Our 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.

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 intended 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. 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: Our 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 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 (crate 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]

```
#[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: The DERCapability annotated with optional CSIP-AUS extensions.

```
<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 can exist contain a "Master Resource Identifier" or an "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 of the MRID.

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 provides the requirement that MRIDs are sufficiently cryptographically unique, where 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 never collide, by use of a global, mutable, atomic integer that increments each ID. 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: Our 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 is is sufficiently cryptographically unique, however, this is out of scope of 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.

MRID Generation

We provide a test verifying that the PEN ID can be obtained from a given MRID, 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 serialized & deserialized successfully, and that it can be done without any data loss.

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

Figure 6.17: A Rust test verifying that the IEEE 2030.5 List resource can be serialized & deserialized

The logic for a test of this type can be succinctly expressed in Rust, as seen in Figure 5.7. 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 5.7, 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 deserialized 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 are required to write more 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 Event-driven Architecture

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 placed of standard, blocking, functions, such as for I/O, or concurrency primitives.

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 another 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.

6.2.2 Application Support

In IEEE 2030.5, the Application Support function set is defined to include: [oEE11]

- "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)

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 specification adherence. 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 Rust `hyper` library, as it currently integrates best with our Security function set requirements, and provides us with adherence to HTTP/1.1. [Hyp23]

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. Additional, when instantiating the client, users can set a TCP keepalive duration value, that will then be passed to `hyper`.

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 `TLS_ECDHE_ECDSA_WITH_AES` 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 an (approximately) safe foreign function interface to

the C library. [Ope23a]

Our 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:

- The IP address of the target IEEE 2030.5 server for this client instance.
- 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 can be instantiated with just a server address.

```
let client = Client::new_https(
    "https://127.0.0.1:1337",
    "client_cert.pem",
    "client_private_key.pem",
    "serca.pem",
    /* Additional configuration ... */
)
```

Figure 6.18: Example IEEE 2030.5 Client instantiation

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 specific extensions and fields. `OpenSSL` provides this functionality, however

the Rust FFI bindings we use 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. [Rus23b]

```
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 6.19: Our IEEE 2030.5 certificate validation interface

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.

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 6.20: Our interface for the core client functionality

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, but Rust can 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 6.21: 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 most importantly 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 TODO we discuss a design flaw with this current implementation, and why this interface might need to be adjusted going forward. In section TODO 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

Additionally, our library provides convenience functions for creating and sending response resources when given an event resource, with 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, however we provide them to library users for completeness.

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

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 provided by any instantiated client capable of making requests.

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 logging API we interact with. Ideally, they would only start polling a route once they know it's valid. Additionally, we provide a function for forcibly polling all tasks in the queue, and a function for cancelling all resource polling. Library users may want to

```
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 6.22: Our interface for creating and sending response resources.

forcibly poll al 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, or when acting as a client aggregator, no longer aggregating on behalf of a specific client.

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

Ergonomics

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

The first method allows user to implement the trait in Figure 6.24 on a type, and then pass an instance of taht ype when starting the poll. This is useful when users have some thread-safe mutable state they want to share

```
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 6.23: Our interface for creating and sending response resources.

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

Figure 6.24: A Rust trait representing the behaviour of a Poll callback.

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 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 6.25: 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 (no 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 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.

One way to implement this would be to use the asynchronous sleep functions provided by `Tokio`, yielding a task until the next scheduled poll time. Tokio tasks, while slept, have very minimal overhead. Unfortunately, Tokio sleeps, much like Rust's threaded sleeps, do not make progress while the device is slept.

Whilst our client library is primarily for use on servers operating under the client aggregation model, again, 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 have intermittent internet connections, and 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 an unnecessary overhead of 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 polled and the stored user call-back is called with 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, and to give end user control over this process, but also for the sake of testing, we allow users to define the length of these intermittent sleep intervals when the instantiate the client.

```
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 6.26: Rust types used in implementing asynchronous resource polling

Figure 6.26 shows the data structures guiding 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 POSTing a Subscription resource, 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 `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.

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 6.27: Example callback function for use by the `ClientNotifServer`

```
let server = ClientNotifServer::new(
    "127.0.0.1:1338",
    "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 6.28: Example instantiation and route creation on a `ClientNotifServer`

Figure 6.27 shows an implementation of a function that can be used to as a route callback. In this case, the callback will have the server respond with `HTTP 204 No Content`. Figure 6.28 shows this function used as the callback for HTTP requests on the "/dera" route. The same figure shows how a closure with an appropriate function signature can be used, and also how the server can be run. The run function accepts an optional 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.

Ergonomics

Using the same exact method used in implementing Polling Callbacks, we place as few restrictions on the user of our Subscription / Notification mechanism. 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 config 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.

The rest of the implementation requires us to route incoming HTTP requests to the correct user-defined call-

back. The simple nature of the server, and lack of parametrized route names results in a straight-forward implementation.

```
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 6.29: Rust types used in implementing a concurrent HTTP router

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

The default hashing algorithm chosen by 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.

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, and having clients use that Time resource to create all timestamps. In the case there are multiple servers exposing time resources, the specification allows Clients to choose just one Time resource, with the most accurate time, 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." [oEE18].

Interface

Our client library must provide a generic abstraction that allows developers to synchronise time globally, and per event 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

```
fn current_time() -> SEPTime;

fn current_time_with_offset() -> SEPTime

fn update_time_offset(time: Time);
```

Figure 6.30: 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 per event 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 6.31: 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 intention they

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, at any time. Clients communicate with servers about these Event resources by creating and sending Response resources.

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, and creating and sending it automatically.

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.

```
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 6.32: Definition of the Scheduler trait

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

Figure 6.33: 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 pertain 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 alert 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.

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, such as if the server has cancelled it.

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. [oEE18]

Finally, we provide a function for cleaning up the Tokio tasks spawned by the Schedule, freeing all allocated resources. This currently requires shutdown 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 usesr 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 also contains an instance of an EndDevice resource, such that all information a developer may require when referring to a singular device is available.

```
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 6.34: 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 6.35: 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 in-depth 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. The unique event MRID is the key into this hashmap. This gives us constant time lookups for events, and ensures a schedule cannot store duplicate events.

```
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 6.36: 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 needs only 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.

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 is superseded by another.

We determine if two `DERControl` instances supersede if they 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, and one is newer than the other.

If an event is superseded and active, the 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 remaining 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 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.

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.

Pricing

The Pricing function is very much similar to DER and DRLC. One difference is that a DER 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.

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 mysteriously not 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, and also unlike every other schedule, not accept some program resource.

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 coe in our client library could be reused. For that reason, we do not provide an implementation for this schedule, and insead leave it to library users if they require it. Instead, we provide them with an ample resource for developing an IEEE 2030.5 event schedule, our codebase.

6.2.9 Logging

6.2.10 Testing

Security

During development, we maintained a local certificate authority via the tool `mkcert`. 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 TODO.

Chapter 7

Evaluation

Chapter 8

Future Work

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.

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

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

[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.

[Fou23] 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.

[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, 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.

[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.

[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.

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

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

[Rus23b] Rusticata. x509-parser. https://github.com/rusticata/x509-parser, 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.

[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.