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 Common Smart Inverter Profile

CSIP-AUS Common Smart Inverter Profile Austrai

CVE Common Vulnerabilities and Exposures

DER Distributed Energy Resources

DNS Domain Name System

DNSP Distribution Network Service Provider

DNS-SD Domain Name System - Service Discovery

ECC Electric Curve Cryptography

EPRI Electric Power Research Institute

EXI Efficient XML Interchange

FS Function Set

HAN Home Area Network

HTTP Hypertext Transfer Protocol

IEEE Institute of Electrical and Electronics Engineers

I/O Input / Output

IoT Internet of Things

IP Internet Protocol

mTLS Mutual TLS

OOP Object-Oriented Programming

OS Operating System

REST Representational State Transfer

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] [0EE11,]

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] [0EE18]

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. [0EE18]

- 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.

```
                        IEEE 2030.5
                          Protocol
  ┌──────────────┐                    ┌──────────────┐
  │ IEEE 2030.5  │ ←───────────────→  │ IEEE 2030.5  │
  │   Client     │                    │   Server     │
  └──────────────┘                    └──────────────┘
```
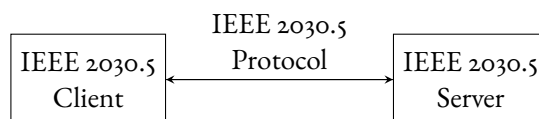
End-user Energy Device

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.

```
                                                    ┌──────────┐
                                                    │   DER    │
                                                    └──────────┘
                                                 ?
                        IEEE 2030.5
                          Protocol   ┌──────────────┐
  ┌──────────────┐                   │ IEEE 2030.5  │        ┌──────────┐
  │ IEEE 2030.5  │ ←─────────────→   │   Client     │ ? ──→  │   DER    │
  │   Server     │                   │ (Aggregator) │        └──────────┘
  └──────────────┘                   └──────────────┘
                                                 ?
                                                    ┌──────────┐
                                                    │   DER    │
                                                    └──────────┘
```
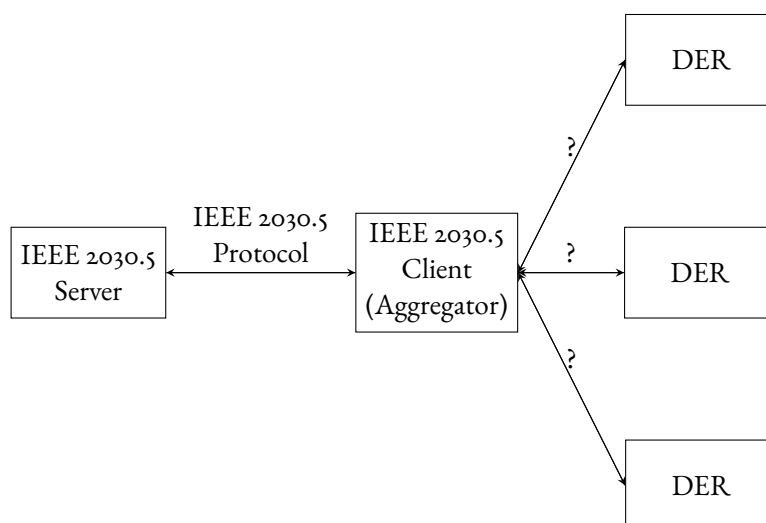
Figure 4.2: The Aggregated Clients IEEE 2030.5 Model, as defined by California SIP

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,model and use of subscription/notification retrieval. 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. [All,]

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 work is that we are developing a client library that can be used to build client aggregators, the IEEE 2030.5 model preferred by electric utilities.

5.1.2 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.

5.1.3 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.

5.1.4 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 [Ope23]. In section TODO we discuss an in-progress alternative to this.

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] [Tok23]

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.

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.

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.

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

6.1.1 Resource Data Types

The largest aspect of this 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 List must first implement the Resource 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 Resource {
    fn get_href(&self) -> Option<&str>;
}

pub struct ResourceObj {
    href: Option<String>
}

impl Resource 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 List {
    #[inherits(Resource)]
    res: ResourceObj,
    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 Resource 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 Resource trait. This generated code simply calls the underlying ResourceObj 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,]

```rust
#[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.

```xml
<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 Resource 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.

```
/// Implemented by all types whose base type is ['IdentifiedObject']
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. Once again, it is 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 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`.

29

```
#[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

6.1.3 Testing

The goal for this thesis is to produce a IEEE 2030.5 client that is well-tested for correctness, and conformance to the IEEE 2030.5 specification. As such, our common library will be developed alongside a suite of tests, written with the goal to maximise code coverage, measured by lines of code. All testing for the Common Library will be done via unit testing.

Testing YaSerde

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.

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.

```
#[test]
fn list_serde() {
    let orig = List::default();
    let new: List = from_str(&to_string_with_config(&orig, &
        YASERDE_CFG).unwrap()).unwrap();
    assert_eq!(orig,new)
}
```

Figure 6.12: A Rust test verifying that the IEEE 2030.5 List resource can be serialized & deserialized

For each resource, the only part of this test that differs is the name of the type, making this test suite a very reasonable automation. Not only will this inform of us any bugs or issues in YaSerde that lead to data loss, it will also inform us of any limitations of YaSerde, in terms of things it can and cannot serialize/deserialize. Furthermore, as we make changes to our YaSerde fork, this test suite will inform us of any regressions.

However, this test suite is not capable of testing for resource serialization correctness; whether or not the outputted XML conforms to the specification.

Unit Tests

The remainder of the common library will be tested using manually written unit tests. A set of these unit tests will test whether resource serialization matches that of the specification using the provided example outputs, whilst others will ensure the network and security interfaces are correct. These unit tests are paramount when working with our Rust crate dependencies, as they can be used to test for regressions if and when we choose to update them.

6.2   Client Library

Despite the claim that we are developing an IEEE 2030.5 client, in reality, the goal of this thesis is to produce a client library, such that manufacturers of IEEE 2030.5 complaint devices will extend upon our client. These device manufacturers would simply, in Rust or otherwise, call upon the documented interface of the client library to communicate with a IEEE 2030.5 server.

The exact details of this client library interface are yet to be determined, but what a developer should be able to achieve with that interface is fairly straight forward. They'll require:

- A mechanism for discovering or connecting to a specific server.

- The ability to retrieve resources from a specific URI, and have it deserialized into a local data type.

- The ability to create or update a resource, and have it included as a PUT or POST request to a connected server.

- A mechanism to automate this process, using event-driven architecture, to handle timer events, such as for polling, or network events for resource subscription.

6.2.1 Server & Connection

Connecting to a server is, otherwise, as expected. We use the cipher suite interface and the TCP interface, as built in the common library, to connect to the server using TCP/TLS. This of course requires certificates signed by a common library cipher suite. This certificate exchange will determine the parameters by which the cipher suite will be used to encrypt traffic.

6.2.2 Event-driven Architecture

With the IEEE 2030.5 Client being an I/O bound application, we are to implement concurrency using an event-driven architecture, such that many operations can performed concurrently on a single threaded machine when waiting on I/O.

Although the client does not connect to multiple servers, we require event-based architecture to handle:

- Input from the client device, such as the creating or updating of resources.

- Timed events, such as to poll for a resource update on a given schedule.

- 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.

An event-based architecture furthermore enables us to take advantage of multiple OS threads, and therefore multiple CPU cores, as to best accommodate the scale required by the client aggregator model. As mentioned previously, Rust supports asynchronous programming, with runtime-agnostic support for await and async syntax, of which there are multiple Rust async runtime available.

The specifics of this asynchronous programming architecture are yet to be learned, and as such we can look to the EPRI client implementation, where they've chosen to use the Linux native `epoll syscall`. Fortunately,

`epoll` is the driver behind the two most popular async runtimes for Rust, `Tokio` and `std-async`. As such, we'll need to determine which is most suitable for our client.

6.2.3 Function Sets

As established, function sets are logical groupings of IEEE 2030.5 resources by their functionality, of which there are twenty-five. A subset of these function sets are for general or 'Common' use, such as the time function set, for synchronizing a device clock, whilst others have a very specific scope, such as the Electric Vehicle function set.

QualityLogic, a quality assurance company specialising in testing standardised software, has grouped these function sets further than the specification itself.

Figure 5.8 shows these groupings, and which of these function sets are required for California SIP. These groupings provide us with a starting point for determining the priority of function sets, where by those required for California SIP will be implemented first. An exception to this is the Demand Response and Load Control function set, which has been deemed relevant for DER in Australia by electric utilities in Queensland, and as such it's implementation will be prioritised. [?,]

Regardless, the groupings in Figure 5.8 further indicate the dependencies between function sets. For example the Device Capabilities function set describes the function sets available on a server, and is therefore used to discover the URI of resources for other function sets.

The conclusion to draw from this is that function sets on the left side of Figure 5.8 are very much depended upon by function sets on the right side, as they describe the core functionality on the client. Our development will begin on the Security, Certificate Management, and Application Support function sets as part of the common library, where the Application Support function set refers to the use of prescribed technologies, such as TCP, HTTP/1.1 and XML.

6.2.4 Testing

As expected, our client library will be developed alongside a suite of tests. Unlike the common library, these tests will be comprised of both unit and system tests. System tests will primarily be used to check for conformance to the IEEE 2030.5 specification, using the example system outputs in the specification. The actual software

framework to be used for conducting system tests is yet to be determined, but the requirements to use such a framework are relatively straightforward, obviously requiring both a test client binary, and a mock server.

Test Client Binary

In order to perform system tests we will need to wrap our client library and produce a binary that will connect to a server, simulating a potential use case. This binary client will be a demonstration of all implemented functionality. Furthermore, this client binary will act as documentation for device manufacturers using our library. Likewise, EPRI produced a test client binary for demonstration purposes in their IEEE 2030.5 client implementation.

Mock Server

When it comes to writing system tests, our testing philosophy is hindered by a lack of open-source IEEE 2030.5 implementations, in-development or otherwise. As such, we'll need to create our own mock IEEE 2030.5 server to test both the sending and receiving of resources from our client. Neel Bhaskar, as part of UNSW CSE, developed a portion of a working IEEE 2030.5 Server in 2022, and with changes to support hardcoded resources and routes, will be used as the starting point for this mock server. We will also need to make changes to this server in order for it to use our common library, where we will therefore gain a better idea of what functionality is common to both client & server.

Specific client functionality, such as the subscription/notification method of resource retrieval, may require extensive work on this mock server. As a result, we won't rule out the possibility of writing code that could be reused in a future server implementation.

Bibliography

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

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

[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.

[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.

[Fou23]   Rust Foundation. Platform support. `https://doc.rust-lang.org/nightly/rustc/platform-support.html`, 2023.

[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.

[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.

[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.

[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.

[Ope23]   OpenSSL. Vulnerabilities. `https://www.openssl.org/news/vulnerabilities.html`, 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.

[Tok23]   Tokio. Tokio documentation. `https://docs.rs/tokio/latest/tokio/`, 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.