A Modern IEEE 2030.5 Client Implementation

Thesis A - Trimester 1, 2023

By Ethan Dickson (z5309251)

Supervised by Jawad Ahmed

Assessed by Nadeem Ahmed

School of Computer Science & Engineering
University of New South Wales, Sydney

Outline

1. Background
    - Smart Grid & End User Energy Devices
    - System Architecture: Clients, Servers, and Utilities
    - Existing implementation
    - Product considerations
2. Proposed Design
    - Assumptions & Constraints
    - Rust programming language
    - Client & Server Common Library
    - Event-driven Client design
    - Service Discovery
    - Function Sets
    - Testing
3. Timeline

Background:

Context

- Electric Grid:
  - A network enabling electricity delivery between utilities & individuals.
- Smart Grid:
  - An electric grid assisted by computers to enable communication and control between utilities & individuals on the network.
- End User Energy Devices
  - Devices existing within the *'end user energy environment'*.
  - e.g. Smart Meters, Electric Vehicle Chargers, Water Heaters, Distributed Energy Resources
- Distributed Energy Resources (DER)
  - End user energy devices that are able to *'deliver active AC power for consumption in the residence or the grid' (IEEE, 2018)*
  - e.g. Solar Inverters, Solar Batteries

Background:

IEEE 2030.5

- An application layer protocol for facilitating communication between end user energy devices and electrical utility.
- Iterations published in 2013 and 2018.
- Follows a RESTful architecture, ‘built using Internet of Things (IoT) concepts’.
- Came to be via:
  - IEEE 2030: “Guidelines for Smart Grid Interoperability”
  - SunSpec Modbus
  - ZigBee Alliance Smart Energy Profile

Background:

System Architecture

Resource

- Any data object manipulated by the Rest API, addressable via Uniform Resource Identifier (URI)
- Resources transferred as \`sep+xml\` or \`sep-exi\`
- Normatively defined using XML Schema Definition (XSD).

Server

- Exposes resources
- No open source server implementation exists as of present
- Implementation started by CSE Research Assistant Neel Bhaskar
- Currently not being developed.

Client

- Creates, requests and deletes resources on the server.
- Receives resources from server via polling or *'subscription/notification'*
- There is one fully open source implementation of the client, written in C, by the American Electric Power Research Institute (EPRI)

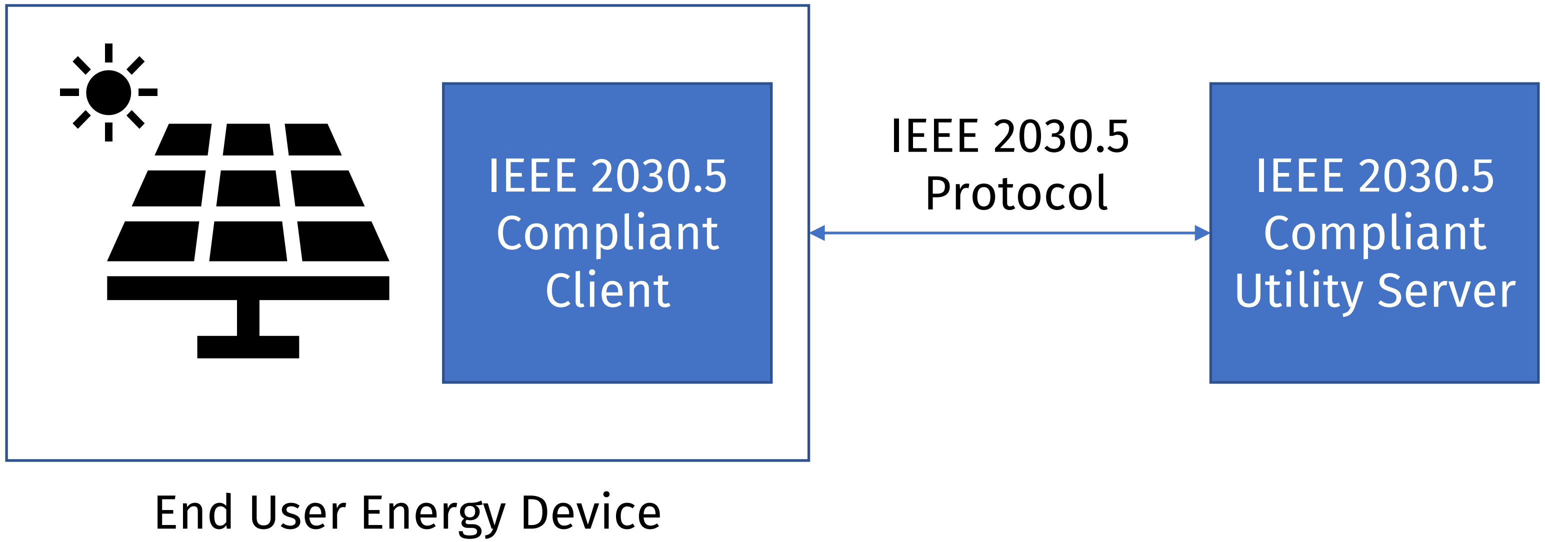
Background:

Product

- California Public Utilities Commission Rule 21
  - Primarily defines Smart Inverter usage in the State of California
    - California Smart Inverter Profile (CSIP)
  - Default Application-level protocol: IEEE 2030.5
  - Enabling the use of IEEE 2030.5 at scale, meeting the requirements of utilities.
  - Requires a subset of all function sets to be implemented.

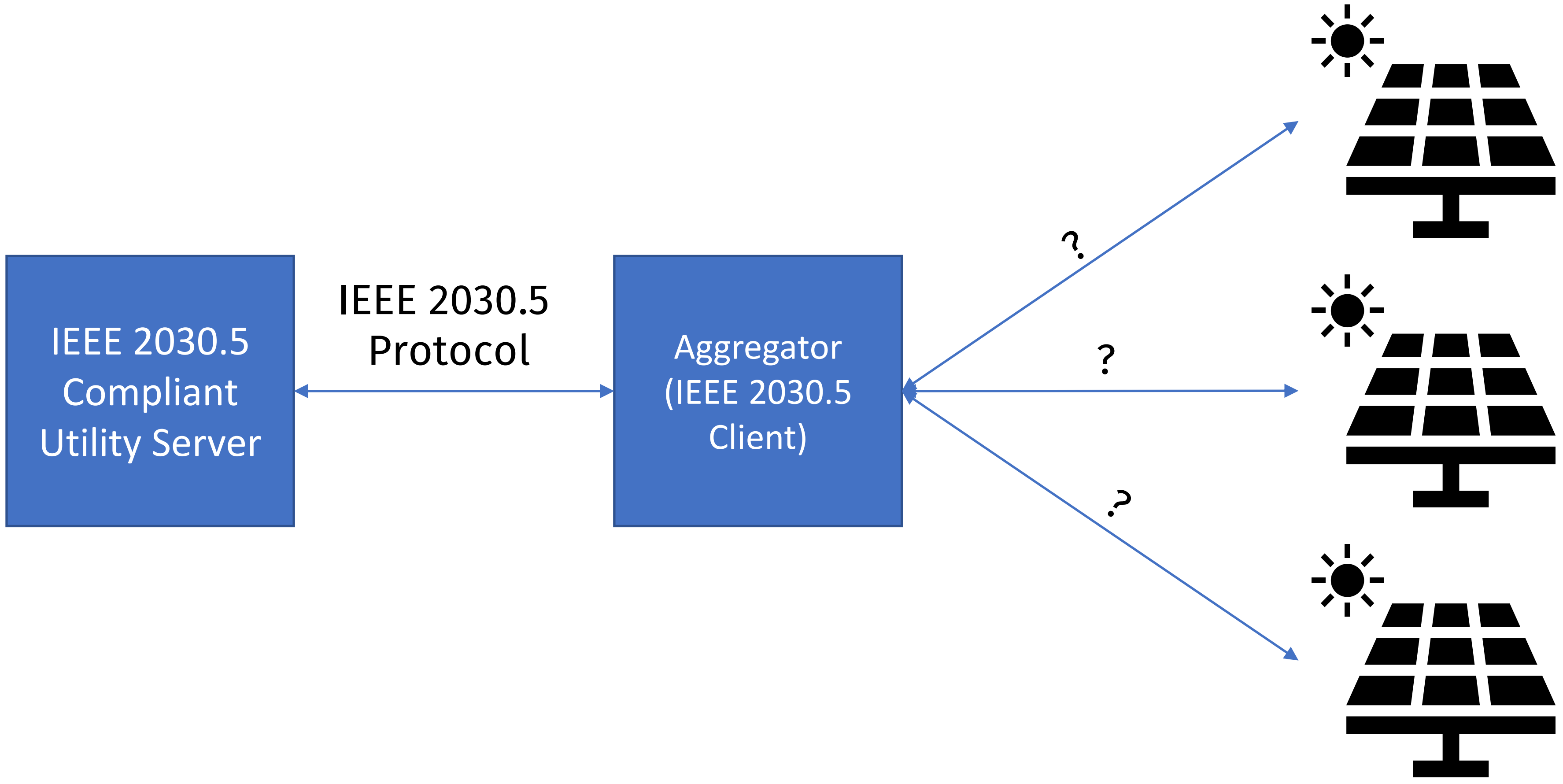
Background: Product

Individual/Direct Model



Background: Product

Aggregated Clients Model



Background:

Existing Research

- SunSpec MODBUS
  - “Semantically identical” to and “fully interoperable” with IEEE 2030.5
- IEEE 2030.5 Implementation By EPRI:
  - Developed an IEEE 2030.5 client library without a corresponding server implementation.
  - Designed for both individual, and aggregated clients
  - Written in C for it’s “*universality*”.
  - Defined a platform-specific interface, such that it can be ported.
  - Chose to auto-generate internal representations of 2030.5 resources from the XSD.
  - Periodically polls servers for resource retrieval, does not support servers ‘*subscription/notification*’.
  - Optionally supports RSA-based cipher suites.

Design:

System Considerations

- Operating System Considerations
    - IoT device prevalence >Target operating systems
    - Server prevalence >Target client aggregators
- Programming Language Considerations:
    - Device resource constraints.
    - Cross-platform capabilities
    - Security
    - Support for event-driven architecture.
    - Available high-level abstractions
    - Available libraries under non-restrictive licenses

Design:

Rust

- High-level, general-purpose programming language.
- Strongly influenced by other programming languages, both industrial and academic.
  - C++, Haskell, ML family languages
- Includes elements of both functional and object oriented programming paradigms.
- Provides programmers with control over guarantees on software safety, and performance.
- IEEE 2030.5 client is to be implemented using ‘Safe’ Rust.

Design:

Common Library

- Internal representations of resources
    - Specified in XSD *(sep.xsd)*
    - Auto-generated or otherwise
    - Organised by packages, function sets
- Serializing & Deserializing those internal representations
- Network & Security (Application Support, Security FS)
    - TCP
    - UDP
    - HTTP 1.1
    - Elliptic Curve TLS (IETF RFC 7251)
    - Optionally, an RSA cipher suite

Design: Common Library

Resource Data Types

- Types are exclusively extensions of other types.
- 700~ different types defined (across all function sets).
- Featuring multi-level, hierarchical inheritance.
- Rust uses Traits to define shared behaviour, not object inheritance.

```
pub trait Resource {
    fn get_href(&self) -> Option<String>;
}
```

```
pub trait List : Resource {
    fn all(&self) -> UInt32;
    fn results(&self) -> UInt32;
}
```

Design: Common Library

Emulating Inheritance

- Two Approaches:
  - Duplciate individual inherited members in data structures
  - Composite base types
- Traits need to be implemented individually, regardless of inheritance implementation.
- An ‘inheritance-rs’ crate (library) automates inheritance via composition, to a point.

Design: Common Library

Inheritance via Composition

- Inheritance can be substituted with composition
- An ‘inheritance-rs’ crate (library) automates inheritance via composition.
- Ultimately, not ideal.

```rust
#[inheritable]
pub trait Resource {
    fn href(&self) -> Option<String> {
        None
    }
}

pub struct ResourceObj {
    href: Option<String>,
}

impl Resource for ResourceObj {
    fn href(&self) -> Option<String> {
        if let Some(output) = &self.href {
            return Some(output.to_owned());
        }
        None
    }
}
```

```rust
#[derive(Inheritance)]
pub struct List {
    #[inherits(Resource)]
    res: ResourceObj,
    all: UInt32,
    results: UInt32,
}
```

Design: Common Library

XSD to Rust Types

- XSD is self-contained, pseudo-standardised.
- Several existing Rust libraries for converting XSD to Rust types.
- Lumeo, producing an ONVIF client created `xsd-parser-rs`
- We’ll maintain a fork to meet our 2030.5 specific requirements, as needed.

Design: Common Library

Using `xsd-parser-rs`

```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>,
}
```

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

Design: Common Library

Resource Serialization

- Communicated via HTTP/1.1 with content-type:
  - application/sep+xml
    - Extension of `application/xml` (Extensible Markup Language)
    - SEP semantics
    - Parameter for schema version
  - application/sep-exi
    - Extension of `application/exi` (Efficient XML Interchange)
    - Binary representation of XML
    - SEP semantics
    - Parameter for schema version

Design: Common Library

XML Serialization

- “YaSerde”: Serializing & Deserializing common Rust types.
  - YaSerde Serialization structure derived by `xsd-parser-rs`
  - We will maintain a fork of YaSerde for our use case.

```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>,
}
```

Design: Common Library

Testing YaSerde

- No compile time guarantees that serialization or deserialization are successful
- Two Test Suites
  - Spec adherence / Correctness – Handwritten tests
  - Data Loss – Compile-time generated

```
#[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);
}
```

Design:

Client Library

- We are building a client library, not a binary client.
- Our standalone client will exist purely for demonstration & testing purposes.
- Includes:
  - Service (Server) discovery & connection
  - Resource creation
  - Resource retrieval
  - Event based architecture
    - Resource updating via polling (Timer events)
    - Resource Subscription/Notification

Design: Client Library

Server Discovery & Connection

- 2030.5 specifies clients may
  - Use DNS-SD to discover 2030.5 servers
  - Use DNS to resolve hostnames
  - Connect to a server using an IP address and port
- DNS Service Discovery
  - Local network discovery only
  - Not a high priority feature, although required.
  - Many DNS-SD Rust implementations exist
    - Which wrap ‘Avahi’ on Linux.
- Connection
  - TCP/TLS
  - Certificate Exchange determines cipher suite usage

Design: Client Library

Event-Driven Architecture

- Our client is I/O bound, waiting for:
  - Client device input
  - Timers (Scheduled Polling)
  - Server notifications
  - HTTP request responses
- Rust async/await native support
  - Async runtime agnostic
  - Popular libraries: \`Tokio\`, \`std-async\`
- EPRI Client uses epoll to drive it's asynchronous model
  - As do \`Tokio\` and \`std-async\`

Design: Client Library

Function Sets

- Grouping resources by functionality
- Dependencies between function sets
- Meta Function Set: Device Capabilities
- Already Covered Today:
  - Application Support
  - Design Pattern
  - Security
  - Certificate Management
  - All function set internal representations

QualityLogic. (2017). IEEE 2030.5 Workshop

Design: Client Library

Testing

- Prioritising coverage: >80%
- Unit Tests
  - Generated Serde test suite
  - Manually Written:
    - Network Interface
    - Cipher Suite Interface
- System Tests
  - Mock Server
    - Sample resources, and responses.

Timeline:

Thesis A

Start of Term 1 – Start of Term 2
~150 hours

- Background Research
- Thesis A Seminar
- Common library
  - Generating Resource data structures
  - Generating Resource ➔ XML Unit Tests
  - Cipher Suite Interface + Tests (Security)
  - Network Interface + Tests (Application Support)
- Mock Server &
- Client Library
  - Event-based architecture
  - FS: Security, Application Support
- Test Client Binary for Testing
- Thesis A Report

Timeline:

Thesis B

Start of Term 2 – Start of Term 3
150-200 hours

- Client Library
  - Time FS, Scheduled events, Server polling
  - End Device FS, Client & Server relationship
  - Subscription/Notification mechanism
  - Metering, LogEvent, File Download, Distributed Energy Resources FS
  - More Unit tests
- Mock Server & Test Client Binary
  - Updated to support more system tests
- Thesis B Demonstration
  - All implemented functionality

Timeline:

Thesis C

Start of Term3 – End of Term 3
150-200 Hours

- Client Library
  - Metering Mirror
  - Power + Network Status
  - Configuration
  - Demand Response and Load Control
  - Pricing
  - Messaging
    Billing
  - Prepayment
  - Flow Reservation
- Test Client Binary & Mock Server Updates
- Stretch Goals:
  - Efficient XML interchange library for Rust
  - Discovery using DNS-SD
- Thesis C Seminar
- Thesis C Report

References

- Institute of Electrical and Electronics Engineers. (2018).
  Standard for Smart Energy Profile Application Protocol (IEEE 2030.5-2018)
  https://standards.ieee.org/ieee/2030.5/5897/
- Electric Power Research Institute. (2018).
  EPRI IEEE 2030.5 Client User's Manual
  https://www.epri.com/research/products/000000003002014087
- Electric Power Research Institute. (2018).
  EPRI IEEE 2030.5 Client
  https://github.com/epri-dev/IEEE-2030.5-Client
- SunSpec. (2019).
  IEEE 2030.5 Workshop
  https://sunspec.org/wp-content/uploads/2019/08/IEEE2030.5workshop.pdf
- California Energy Commission. (2021).
  Smart Inverter Interoperability Standards and Open Testing Framework to Support High-Penetration Distributed Photovoltaics and Storage
  https://www.energy.ca.gov/sites/default/files/2021-05/CEC-500-2020-056.pdf
- Pacific Gas and Electric Company. (2021)
  Electric Rule No. 21
  https://www.pge.com/tariffs/assets/pdf/tariffbook/ELEC_RULES_21.pdf

References

- IEEE. (2011).
  IEEE Guide for Smart Grid Interoperability of Energy Technology and Information Technology Operation with the Electric Power System (EPS), End-Use Applications, and Loads
  https://standards.ieee.org/ieee/2030/4593/
- Solar Builder Magazine. (2021).
  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-installers/
- QualityLogic. (2019).
  IEEE 2030.5 Takes Off
  https://www.qualitylogic.com/knowledge-center/ieee-2030-5-takes-off/
- Sunspec. (2016).
  IEEE 2030.5 Common California IOU Rule 21 Implementation Guide for Smart Inverters
  https://sunspec.org/wp-content/uploads/2017/02/CSIPImplementationGuide-v1-0.pdf
- Gordon Lum. (2016).
  California Use Case for IEEE 2030.5 for Distributed Energy Renewables
  https://smartgrid.ieee.org/bulletins/december-2016/california-use-case-for-ieee2030-5-for-distributed-energy-renewables

References

- Lum::Invent. (2023).
  YaSerde
  https://github.com/media-io/yaserde
- Lumeo. (2023).
  xsd-parser-rs
  https://github.com/lumeohq/xsd-parser-rs