A Modern IEEE 2030.5 Client Implementation

Thesis C - Trimester 3, 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
   - IEEE 2030.5 Architecture
2. Implemented Design
   - Assumptions, Constraints, Considerations
   - Client & Server Common Library
   - Core Client Functionality + Application Support + Security
   - Subscription / Notification Mechanism
   - Event Schedules (DER, DRLC, Pricing, Messaging)
   - Testing
   - CSIP-AUS
3. Conclusion
   - Future
   - Evaluation
4. Q&A

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 Architecture

IEEE 2030.5

- An application layer REST API for facilitating communication between end user energy devices and electrical utility.

Resource

- Any data object manipulated by the Rest API, addressable via Uniform Resource Identifier (URI)
- Resources transferred as `sep+xml` or `sep-exi`.
- Grouped into function sets.

Server

- Exposes resources to clients.
- Currently being developed by Matthew Kokolich as part of Thesis A.

Client

- Creates, requests and deletes resources on the server.
- Retrieves resources from server via polling or *'subscription/notification'* (Publish-Subscription)

Event

- A category of resource exposed by servers, instructing clients of a certain behaviour over a specific time interval.

Implemented Design:

Considerations

- The use cases for IEEE 2030.5 Clients are broad, therefore we've:
  - Developed a library/framework, not a single binary application.
  - Developed a modular codebase.

- There is a great deal of overlap between client and server implementations
  - Therefore, we've developed a Common library.

- We can't precisely determine how our library will be integrated into devices:
  - Therefore, we must provide a highly generic interface and minimise restrictions on users.

- The security of end user energy devices is a major motivator for the IEEE 2030.5 specification. Therefore we've:
  - Developed our library in a memory-safe programming language, Safe Rust
  - Used publicly audited & widely used implementations of other required standards, TLS, HTTP, and SSL

Implemented Design:
Considerations

- Our library will be distributed under an open-source license:
  - Our implementation is well-documented, and reasonably well-tested.
  - Current coverage by LOC: 73%

- IEEE 2030.5's Presence in California & Australian Smart Inverter Profile means we need:
  - The Subscription / Notification mechanism
  - The ability for the client to scale when running on more powerful hardware.
  - The ability for users to easily add their own extension resources
  - To design a client for use on Linux Operating Systems.
  - To leverage async Rust.

Implemented Design: Common Library

Overview: sep2_common

- Name: sep2_common
- Functionality:
  - Rust data types for all IEEE 2030.5 resources, types & primitives.
  - Integer types that represent an enumeration of states are implemented as Rust enums.
  - Integer types used as bitmaps are implemented as Rust ‘bitflags’ for ergonomics.
  - Resources for specific function sets are available behind compile flags.
  - All data types serialize & deserialize to and from XML using our ‘SEPSerde’ crate.
  - Compile-time code-generation implementing traits depending on XSD base types.
  - Gracefully handles bad parses, as to ensure reliability.
  - All specified list ordering implemented.
  - CSIP-AUS extensions

Implemented Design: Common Library

Future

- Finish implementing example resource tests.
- EXI Serialising & Deserialising all resources:
  - Requires a Rust EXI library
  - Would likely lend itself to a SEPSerde rewrite
- Fixing the currently broken/unclear NotificationList
  - Notifications contain some generic resource.
  - NotificationList definition ambiguous.

Implemented Design: Client Library

Overview: sep2_client

Functionality:

- Application Support Function Set (TCP + HTTP + XML)
- Security Function Set (TLS/HTTPS + Certificate Verification)
- Core REST Client Functionality (GET, POST, PUT, DELETE)
- Asynchronous Resource Polling
- Notification / Subscription - Client as a Server
- Time Function Set (Global + Per Schedule)
- Event Schedules
  - DER
  - DRLC
  - Messaging
  - Pricing
- Logging
- Unit & System Tests

Implemented Design: Client Library:

Application Support

- XML
  - xml-rs: XML Parser & Writer with an emphasis on spec adherence
  - SEPSerde: Our fork of YaSerde for serialising & deserialising
  - xsd-parser-rs: XSD to Rust Structs, forked for our use case.
- HTTP
  - hyper: HTTP Rust library built for performance and correctness
- TCP
  - Rust Standard Library

Implemented Design: Client Library:

Security

- We are required to use ‘ECDHE-ECDSA-AES128-CCM8’ & EC secp256r1
- rust-openssl: Rust bindings for OpenSSL
- hyper_openssl: HTTPS using OpenSSL TLS
- Users provide:
  - Device/Self-Signed Certificate (As per Spec)
  - Corresponding Private Key
  - Corresponding Certificate Authority
- X509-Parser: Validate Certificates
- SunSpec Alliance: IEEE 2030.5 Test Certificate Generation

Implemented Design: Client Library:

Core Functionality

- POST & PUT
  - Validates Request Body is Resource
  - Headers: Content-Length, Content-Type, Date
- GET
  - Validates Response Body is a Resource
  - Headers: Accept, Date
- DELETE
  - Headers: Date

Implemented Design: Client Library:

Core Functionality

```rust
pub 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),
```

Implemented Design: Client Library:

Resource Polling

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

async fn force_poll(&self);

async fn cancel_polls(&self);
```

Implemented Design: Client Library:

Resource Polling

- Sleepy Devices require asynchronous sleeps to wake periodically
- Significant overhead when many background tasks are repeatedly waking.

```rust
type PollQueue = Arc<RwLock<BinaryHeap<PollTask>>>;

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

struct PollTask {
    handler: PollHandler,
    interval: Duration,
    next: Instant,
}
```

Implemented Design: Client Library:

Resource Polling

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

```rust
client
    .start_poll("/dcap", Some(Uint32(4)), move |r: DeviceCapability| {
        println!("{:?}",r);
    })
    .await;
```

Implemented Design: Client Library:

Subscription/Notification

- Client runs a light-weight HTTP server.
- Servers are informed of Notification routes via Subscription resources.
- Client library provides a `ClientNotifServer` type.
- Requires a IEEE 2030.5 “Device Certificate”.
- We assume Client developers will not require a catch-all Notification route.
- Examples show Notifications sent to “/ntfy”
- Specification does not mandate a single route.
- Leverages ‘dashmap’, a highly-concurrent hashmap for route lookups.

Implemented Design: Client Library:

Subscription/Notification

```rust
async fn incoming_dcap(notif: Notification<DeviceCapability>) -> SEPResponse {
    println!("Notif Received: {:?}", notif);
    SEPResponse::Created(None)
}
```

```rust
let server = ClientNotifServer::new(
    "127.0.0.1:1337",
    "path/to/cert.pem",
    "path/to/key.pem",
    "path/to/ca.pem"
)?
.add("/dcap", incoming_dcap)
.add("/edev", incoming_edev);
server.run(signal::ctrl_c()).await;
```

Implemented Design: Client Library:

Time Function Set

- Clients require the current date & time for:
  - Setting HTTP Date headers
  - Marking ‘changed_time’ resource timestamps
  - Checking Event start and end timestamps.
- Clients must synchronise their time with servers:
  - Globally
  - For each Event Schedule
- SEPTime abstraction
  - u64 for internal comparisons
  - IEEE 2030.5 Signed Int64 for resources
  - RFC 7231 HTTP Timestamp String

Implemented Design: Client Library:

Event Schedules

- Function Sets with Event Resources:
  - DER
  - DRLC
  - Messaging
  - Pricing
  - Flow Reservation
- Schedules inform clients when Events start and end.
- Schedules support multi-program & multi-server events.
- Schedules automatically send appropriate responses to servers.
- Schedules defer Events to client as required.

Implemented Design: Client Library:

Schedule Interface

```rust
pub trait EventHandler<E: SEEvent>: Send + Sync + 'static {
    /// Called whenever the state of an event is updated
    /// such that a response to the server cannot be automatically determined.
    /// Type is bound by an [`SEEvent`] pertaining to a specific function set.
    ///
    /// Allows the client to apply the event at the device-level,
    /// and determine the correct response code.
    ///
    /// When determining the ResponseStatus to return,
    /// refer to Table 27 of IEEE 2030.5-2018
    async fn event_update(&self, event: &EventInstance<E>) -> ResponseStatus;
}
```

Implemented Design: Client Library:

Schedule Interface

- Schedules require:
  - Short (& Long) Form Device Identifier of the specific device.
  - The IEEE 2030.5 Categories of the Device.
  - DRLC specific state (Applied Target Reduction, etc.)

```rust
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>,
}
```

Implemented Design: Client Library:

Schedule Interface

- Creating a Schedule requires:
  - The type of the Event
  - The type of the Event Handler
  - Any valid `Client` instance.
  - A mutable thread-safe `SEDevice`
  - A thread-safe Event Handler
  - A rate at which the Schedule should intermittently wake from sleep.

```rust
pub trait Scheduler<E: SEEvent, H: EventHandler<E>> {
    type Program;
    fn new(
        client: Client,
        device: Arc<RwLock<SEDevice>>,
        handler: Arc<H>,
        tickrate: Duration,
    ) -> Self;

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

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

Implemented Design: Client Library:

Schedule Interface

- Adding an Event to a Schedule requires:
  - A copy of the Event itself
  - A reference to the Event's Program
  - Any unique ID pertaining to the source server.

```rust
pub trait Scheduler<E: SEEvent, H: EventHandler<E>> {
    type Program;
    fn new(
        client: Client,
        device: Arc<RwLock<SEDevice>>,
        handler: Arc<H>,
        tickrate: Duration,
    ) -> Self;

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

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

Implemented Design: Client Library:

Differing Schedules

- All Schedules share very similar implementations.
- Differing in:
  - What Response status codes can be sent (IEEE 2030.5 Table 27)
  - How events supersede one another
  - Support for Randomizable start times & durations

Implemented Design: Client Library:

Schedule Internals

```rust
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)>,
}
```

```rust
/// A wrapper around an [`SEEvent`] resource.
pub struct EventInstance<E>
where
    E: SEEvent,
{
    // Event start time, after randomisation
    start: i64,
    // Event end time, after randomisation
    end: i64,
    // Event primacy
    primacy: PrimacyType,
    // The SEEvent instance
    event: E,
    // The MRID of the program this event belongs to
    // In the pricing function set,
    // multiple TimeTariffIntervals can be active
    // for a specific TariffProfile (program)
    // In that case, we store the MRID of the Rate Component,
    // of which there can only be one
    // TimeTariffInterval active at a time
    program_mrid: MRIDType,
    // The current status of the Event,
    status: EIStatus,
    // When the event status was last updated
    last_updated: Instant,
    // The event(s) this supersedes, if any
    superseded_by: Vec<MRIDType>,
    // Which server this event was sourced from,
    // different values indicate a different server
    server_id: u8,
}
```

Implemented Design: Client Library:

Flow Reservation Schedule

- Contains an Event Resource: FlowReservationResponse
- Contains an Event Response: FlowReservationResponseResponse
- Does NOT contain an Event Program.
- Does NOT include a column in Table 27.
- Does NOT specify any specific Event behaviour.

Table 27 —Response types by function set

Enumeration value	Description	Why sent	When sent	*response Required* bit position	DRLC and DER	Pricing	Messaging
0	*Reserved*						
1	Event received	To notify response server that client has initially received event	When the device first receives the event, either via a GET or a Notification	0	X	X	X
2	Event started	To notify response server that client has begun	At *EffectiveStartTime* (see 10.2.3.2)	1	X	X	X

Implemented Design: Client Library:

Testing

- Produced a reasonably well-tested Client & Common library
- We use Rust Built-in Cargo Instrumentation-based Code Coverage
- grcov claims 73% coverage by lines-of-code.
- Common Library
  - Auto-generated test-suite for default instantiations of resources
  - IEEE 2030.5 Example XML Representation
- Client Library
  - sep2_test_server: Mock IEEE 2030.5 Server Binary
  - Module-specific Unit tests
  - System Tests as per IEEE 2030.5 Examples
- MRID Generation collision probability needs calculating.

Implemented Design: Client Library:

Logging

- Expose log messages under a generic façade
- Use standard logging levels:
  - FATAL, ERROR, WARN, INFO, DEBUG and TRACE
- Logging Background Task Errors (WARN & ERROR)
  - Scheduled Poll failing
  - Client + Server Event Status Mismatch
  - Notification Server connection failure
- Logging status updates (INFO & DEBUG)
  - Event cancellation
  - Outgoing requests
  - Incoming Notifications

Implemented Design: Common Library:

CSIP-AUS

- Defines a new resource and new types on existing DER & End Device Resources.
- Convenient implementation using SEPSerde (Our fork of YaSerde)
- We handle 2030.5 extension namespace rule adherence.

Struct sep2_common::packages::conn_point::ConnectionPoint source · [-]

```
pub struct ConnectionPoint {
    pub connection_pointid: String32,
    pub href: Option<String>,
}
```

Available on **crate feature** `csip_aus` only.

[-] The intent of the Connection Point registration extension is to provide the capability for SEP2 servers to offer a consistent mechanism to provide additional information about the existence and location of a newly registered device on the network.

`op_mod_watt_var: Option<DERCurveLink>`

Specify DERCurveLink for curveType == 14. The Watt-Var function varies vars as a function of delivered active power. The Watt-Var curve is specified as an array of Watt-Var pairs that are interpolated into a piecewise linear function with hysteresis. The x value of each pair specifies a watt setting in %setMaxW, (0-100). The y value specifies a target var output interpreted as a signed percentage (-100 to 100). The meaning of the y value is determined by yRefType and must be one of %setMaxW, %setMaxVar, or %statVarAvail.

`op_mod_imp_lim_w: Option<ActivePower>`

Available on **crate feature** `csip_aus` only.

This is the constraint on the imported active power at the connection point.

`op_mod_exp_lim_w: Option<ActivePower>`

Available on **crate feature** `csip_aus` only.

This is the constraint on the exported active power at the connection point.

`op_mod_gen_lim_w: Option<ActivePower>`

Available on **crate feature** `csip_aus` only.

This is a constraint on the maxium allowable discharge rate, in Watts, specifically for a single physical device (or aggregation of devices, excluding uncontrolled devices) such as an EV charge station.

`op_mod_load_lim_w: Option<ActivePower>`

Available on **crate feature** `csip_aus` only.

This is a constraint on the maximum allowable charge rate, in Watts, specifically for a single physical device (or aggregation of devices, excluding uncontrolled devices) such as an EV charge station.

Implemented Design: Client Library

Future

- Unachieved Stretch Goals:
  - DNS-SD
- Future Improvements
  - Improved error handling
  - Performance & Memory Profiling
  - Testing with a production-ready server
    - Energy Queensland SEP2 Test Server
    - GridAPPS-D IEEE 2030.5 Server
- Awaiting being able to:
  - Leverage Security library improvements in the Rust ecosystem
  - Incorporate new Rust Language Features

Implemented Design: Client Library: Future

Rust TLS

- Currently, we use OpenSSL.
- rustls: Native Rust TLS
  - Powered by ring, cryptography in Rust
- Reqwest supports OpenSSL & Rustls
  - However, does not expose a way to modify the used cipher list.
  - Would require library users to modify & compile OpenSSL themselves.
- Use rustls to switch from Hyper to Reqwest.
- To perform AES_128_CCM_8 Bulk Encryption in RustLS.
  - Requires Ring to support CCM bulk encryption.

Implemented Design: Client Library: Future

Rust Async Traits

- Rust has async functions return state machines – a “Future”
- Every async function call returns a different state machine, with a different concrete type.
- async function:
  - Trait bound generic return type (‘Return Position Impl Trait’ / RPIT)
  - `async` keyword is simply syntactic sugar for Future RPIT.
- Return Position Impl Trait in Trait (RPITIT) NOT stable.
- Workaround uses dynamic dispatch to create a concrete type
  - Incurs an unnecessary heap allocation every function call.
- RPITIT will be stable in December this year.

Conclusion: Evaluation:

EPRI C Client Library Comparison

Feature	sep2_common + sep2_client	EPRI IEEE 2030.5 Client
XML Resources	✓	✓
Resource GET, POST, PUT, DELETE	✓	✓
Resource Scheduled Polling	✓	✓
DER Event Schedule	✓	✓
Time Global Offset	✓	✓
DRLC Event Schedule	✓	✗
Subscription/Notification	✓	✗
Pricing Event Schedule	✓	✗
Messaging Event Schedule	✓	✗
Time Schedule Offset	✓	✗
DNS-SD	✗	✓
EXI Resources	✗	✓
Flow Reservation Event Schedule	✗	✗

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/
- Lum::Invent. (2023)
  YaSerde
  https://github.com/media-io/yaserde
- Lumeo. (2023)
  xsd-parser-rs
  https://github.com/lumeohq/xsd-parser-rs
- Kornel. (2023)
  xml-rs
  https://github.com/kornelski/xml-rs
- Hyperium. (2023)
  Hyper
  https://github.com/hyperium/hyper
- OpenSSL. (2023)
  OpenSSL
  https://www.openssl.org/
- Steven Fackler. (2023)
  rust-openssl
  https://github.com/sfackler/rust-openssl
- Steven Fackler. (2023)
  hyper-openssl
  https://github.com/sfackler/hyper-openssl

References

- Rusticata. (2023)
  X.509 Parser
  https://github.com/rusticata/x509-parser
- SunSpec Alliance. (2019)
  SunSpec Test PKI Certificates
  https://sunspec.org/wp-content/uploads/2019/02/SunSpecTestPKICertificates.pdf
- Michael Tilli. (2023)
  httpdate
  https://github.com/pyfisch/httpdate
- Tokio. (2023)
  Tokio
  https://github.com/tokio-rs/tokio
- Xacrimon. (2023)
  Dashmap
  https://github.com/xacrimon/dashmap
- Mozilla. (2023)
  grcov
  https://github.com/mozilla/grcov
- Australian Renewable Energy Agnecy (ARENA)
  Common Smart Inverter Profile – Australia
  https://arena.gov.au/assets/2021/09/common-smart-inverter-profile-australia.pdf
- Energy Quensland, Et al. (2023)
  SEP2 Client Handbook
  https://www.energex.com.au/__data/assets/pdf_file/0007/1072618/SEP2-Client-Handbook-13436740.pdf

References

- Energy Queensland, Et al. (2023)
  Sep2 Test Server
  https://sep2-test.energyq.com.au/api/v2/
- GridAPPS-D
  IEEE 2030.5 Server
  https://pypi.org/project/gridappsd-2030-5/0.0.2a12/
- David Tolnay. (2023)
  anyhow
  https://github.com/dtolnay/anyhow
- Rust Team. (2023)
  log
  https://github.com/rust-lang/log
- rustls. (2023)
  rustls
  https://github.com/rustls/rustls
- Sean McArthur. (2023)
  Reqwest
  https://github.com/seanmonstar/reqwest
- Brian Smith. (2023)
  ring
  https://github.com/briansmith/ring
- Rust Team. (2023)
  Stabilize `async fn` and return-position `impl Trait` in trait
  https://github.com/rust-lang/rust/pull/115822
- Electric Power Research Institute
  IEEE 2030.5 Client
  https://github.com/epri-dev/IEEE-2030.5-Client

Q&A