SZTPD Design Notes

Watsen Networks

July 8, 2025

Abstract

Some design notes for SZTPD.

Contents

1 Introduction

This documentation describes the SZTPD design.

2 End User Goals

Some end user goals that greatly influenced the design.

2.1 Easy Installation

Many opportunities are missed due to onerous installation procedures. Keeping installation as simple as possible seems good. Note, this is why SZTPD doesn't use any configuration files, preferring instead to put all data into the database supplied on the command line.

2.2 High Portability

Deployment environments vary, with many variations of Linux, BSD, and Windows to choose from, not to mention some niche systems. In the same desire to simplify installation, a highly portable product seems like goodness.

2.3 Low Footprint

There seems to be a sweet spot of having a minimal footprint, e.g., on CPU, memory, etc.. A low footprint allows it to run as a micro-service within a larger framework, or as an ephemeral daemon in an SDN context.

2.4 Programmatic API

Having everything available via a programmatic API enables many integration options. Not only can a deployment-specific GUI be layered on top of it, but it can also be called into by controller / NMS applications.

2.5 RDBMS Support

The ability to persist all SZTPD data into a RDBMS is needed to enable the use of a host of RDBMS management tools enabling, for instance, backup, recovery, and encryption.

2.6 Multi-tenant

The immediate use-case is the network equipment vendors and, from their perspectives, they would need to expose a multi-tenant service to their customers. Supporting tenants with isolated data views is critical.

DISCLAIMER: The multi-tenancy implementation was temporarily removed in late 2024. It is planned to be added back as a post-1.0 feature.

3 High-level Decisions

3.1 Use Python

Using Python as the programming language was made based on past experience. Python is known for its easy installs, high portability, and low footprint.

3.2 Use Asyncio

Previous experience suggests that a single process can handle the expected load with cycles to spare. The simplicity goal is further achieved by not having to worry about virtual machines or Docker containers.

3.3 Use SQLAlchemey

Using SQLAlchemy was selected for its ability to work with multiple database backends, including in-memory, file-based, and RDBMSs such as MariaDB, MySQL, Postgres, Oracle, etc.

A few "no SQL" databases were looked at. In particular, document and graph based databases, but all required setting up an external server (no built-in in-memory or file-based options. Optional to install plugins may be made available to access no-SQL backends.

3.4 YANG-driven RESTCONF API

YANG RFC 7950 has been shown to be well suited to defining APIs for networking equipment. The RESTCONF RFC 8040 protocol has been shown to provide an easy to use YANG-driven API.

4 Low-level Decisions

4.1 Data Abstraction Layer

YANG models define arbitrary N-ary trees. It was needed to map arbitrary N-ary trees to SQL tables and rows. Having a "DAL" layer for this seemed prudent.

4.2 Distinct Validation Layer

The options for how to validate YANG datastore were unclear. Decision to use Yangson. Integrating into DAL didn't seem right. Currently a separate layer so can be swapped out with another option if ever need be.

4.3 Native View vs. Facades

The database itself only ever persists the 'native' view. This is why it is called the "native" view. The other views only define "facades" on top of the native view. The 'rfc8572' view facade only presents two RPCs, though it internally writes to both the audit log and bootstrapping log, which are seen in the 'native' view.

4.4 JSON-native

Python objects are nearly indistinguishable from JSON objects - they both print the same tree structures. Thus using JSON natively made sense. This decision was/is further constrained by the fact that Yangson initially only supported JSON, though now it supports XML as well.

4.5 Dynamic Callouts

SZTPD supports "dynamic callouts" in order to interact with external systems (except an RDBMS). Notable uses are:

- to deliver audit log records
- to deliver notification log records
- to deliver bootstrapping event notifications
- to verify device ownership
- to get a dynamic bootstrapping response

At this time, dynamic callouts must be implemented by a plugin-based callback but, in a future release, may be implemented as via a webhook (i.e., an HTTP POST request).

5 Module Diagram

The following diagram illustrates how the various Python modules within SZTPD are composed.

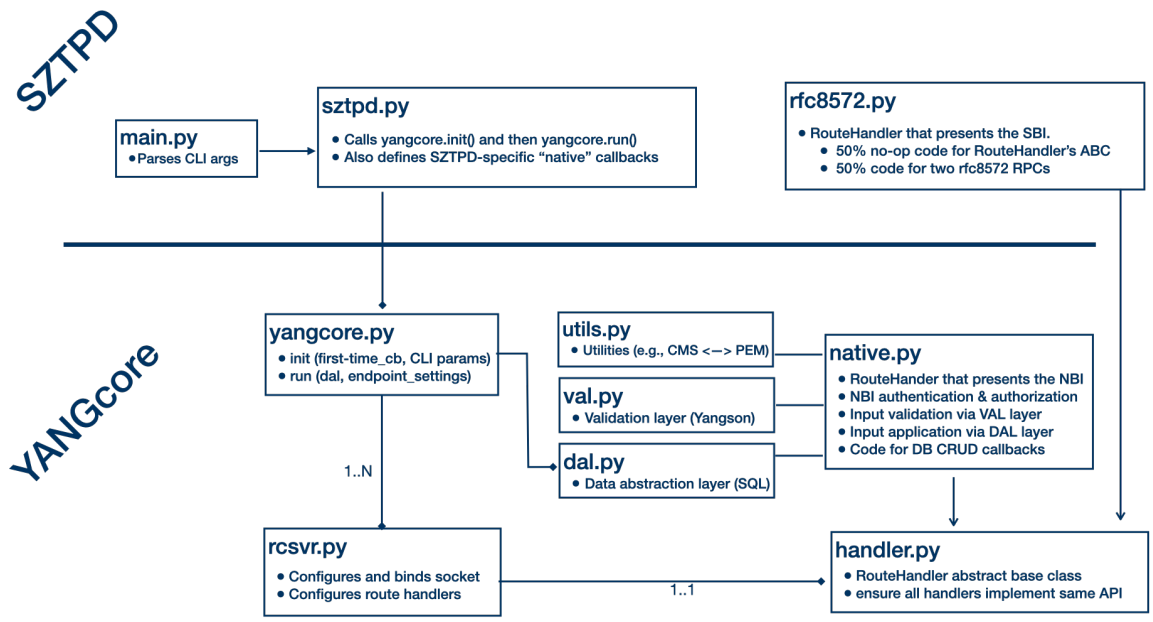

Figure 1: Module Diagram

