

Software for Open Networking in the Cloud (SONiC) Architecture

Table of Contents

[1 Abstract 3](#_Toc444854936)

[2 High Level Architecture 3](#_Toc444854937)

[3 Definitions 3](#_Toc444854938)

[3.1 Switch Hardware 3](#_Toc444854939)

[3.2 Platform Abstraction Layer (PAL) 3](#_Toc444854940)

[3.3 Switch Abstraction Interface (SAI) 4](#_Toc444854941)

[3.4 Switch State Service 4](#_Toc444854942)

[3.5 Network Applications 4](#_Toc444854943)

[4 Switch State Service (SwSS) 4](#_Toc444854944)

[4.1 Overview 4](#_Toc444854945)

[4.2 Key value database 5](#_Toc444854946)

[4.3 Network applications 5](#_Toc444854947)

[4.4 Orchestration Agent 5](#_Toc444854948)

[4.5 syncd 5](#_Toc444854949)

[5 Database Implementation 5](#_Toc444854950)

[5.1 Table Operations 6](#_Toc444854951)

[5.2 Transactions 6](#_Toc444854952)

[6 Switch Data Service Database Schema 7](#_Toc444854953)

[6.1 Overview 7](#_Toc444854954)

[6.2 Schema 7](#_Toc444854955)

[6.3 Database 7 – ASIC\_DB 7](#_Toc444854956)

[7 Switch state service Layer 3 Implementation 7](#_Toc444854957)

[7.1 Overview 7](#_Toc444854958)

[8 Appendix 8](#_Toc444854959)

[9 References 8](#_Toc444854960)

# Abstract

Software for Open Networking in the Cloud (SONiC) is a collection of software packages that are installed on Linux running on a network hardware switch which make it a complete, functional router in the Azure physical data center network.

# High Level Architecture

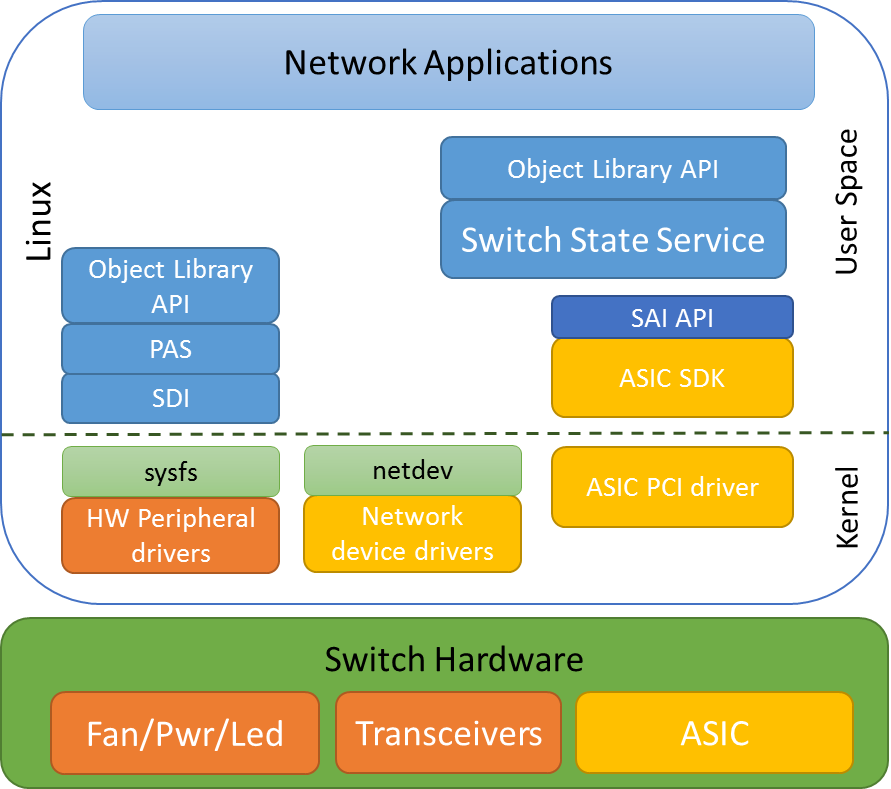


Figure – SONiC High Level Software Design

# Definitions

## Switch Hardware

Switch hardware refers to all the physical components inside the network switch enclosure (chassis). This includes fans, power supplies, status LEDs and network transceivers. In SONiC terminology, these are called “system devices”.

## Platform Adaptation Services (PAS)

The platform abstraction layer is an interface to network switch hardware peripherals such as transceivers, fans, power supplies and leds. SONiC exposes these details via the platform abstraction services.

The PAS uses an abstract low-level platform-independent abstraction for all types of system devices, called System Device Interface (SDI) API. Therefore, only system device drivers (either user space or kernel device drivers) which implement the SDI API are hardware-specific, while the API itself is hardware-independent.

## Switch Abstraction Interface (SAI)

The switch abstraction interface is a standardized C API to the switching ASIC. This API is normally implemented by an SDK specific to the Switch ASIC. More information on SAI is available at the [SAI GitHub](https://github.com/opencomputeproject/SAI) repository.

## Switch State Service

Switch state service (SwSS) is a collection of software components that provide an API and database for storing and synchronizing network switch data with the Switch ASIC. This includes initialization, configuration and current status of the switch ASIC.

## Network Applications

Network applications, such as a BGP routing protocol, may use the switch state service API’s to get and set the state of the Switch State Service.

# SONiC Object Library

As of this writing, the object library and switch state service designs are being integrated. As this occurs, this document will be updated. A high level description of the Object library follows.

The SONiC Object Library (Object Library) mediates interactions between SONiC applications and external applications. The Object Library infrastructure defines two types of application roles: clients and servers. Client applications execute create, set, get, and delete operations on objects. Server applications execute operations requested by clients.

In addition, the Object Library infrastructure supports a publisher/subscriber model. Server applications publish relevant events; client applications can subscribe (register) for specific events and objects. Client applications can register for events generated when objects are created, modified, or deleted.

The publisher/subscriber approach and object-centric operations allow for the completely independent operation of client and server applications.

Custom-written applications use the Object Library API to communicate with the SONiC components.

Object model definitions (written in YANG) are used to generate C header files included by client and server applications. The SONiC C/C++ representation of objects and their attributes is designed to ensure compatibility between multiple versions of the object model.

SONiC provides both C/C++ and Python programming interfaces for the Object Library.

# Switch State Service (SwSS)

## Overview

The Switch State Service (SwSS) is a collection of software that provides a database interface for communication with and state representation of network applications and network switch hardware.

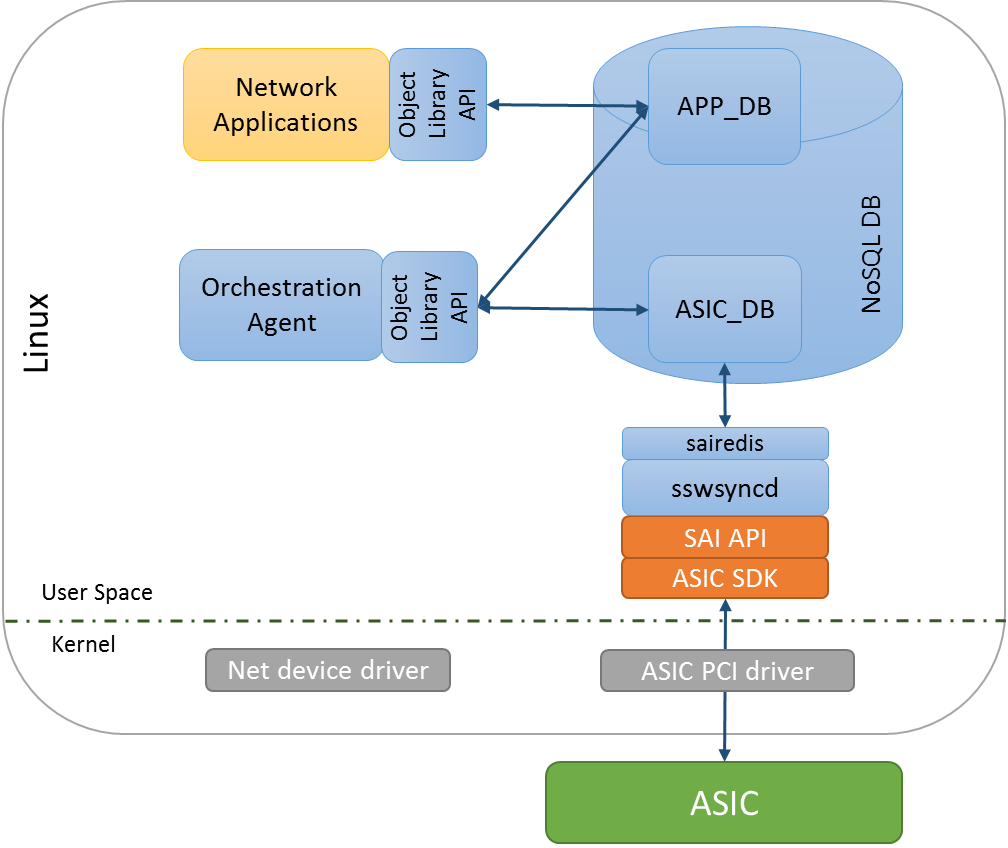


Figure 3 – Switch State Service High Level Design

Network applications read and write to APP\_DB. Example applications include a netlink route syncer, quagga FPM route syncer, access control list (ACL) control, QoS control, load balancer, telemetry control and so on.

The Orchestration agent reads and writes data between APP and ASIC databases. The Orchestration agent is responsible for any necessary logic to verify and transform the data into SAI objects which can be processed by syncd.

The syncd process reads and writes SAI objects between the ASIC\_DB and the SAI SDK.

## Key value database

A key value database was chosen to provide a language independent interface, a method for data persistence, replication and multi-process communication. An API wrapper is implemented in swss/common which implements transactions, convenience methods and allows the database storage engine to be changed in the future if necessary. Redis was chosen as the underlying database engine, but this could be changed in the future.

## Network applications

Using the SwSS API, SONiC network applications are written to be entirely independent of the lower layer communication details to the hardware. Applications subscribe only to the data views they require and avoid implementation details that are not specific for their functionality. Examples of applications that intended to interface with SwSS include: Layer 3 routing, Layer 2 bridging, Access control lists (packet filtering), Quality of service, Telemetry streaming, tunneling, link aggregation, load balancing and policy based routing to name a few.

## Orchestration Agent

This process contains logic for transforming and copying data between the APP tables and the ASIC tables.

There must only be one producer for each ASIC table. Currently there is just one orchestration agent, although others could be added over time.

Only a single Orchestration Agent may write to an ASIC\_DB table.

## syncd

The switch sync daemon syncd copies data between the ASIC\_DB tables and a SAI compliant ASIC SDK. There must only be one syncd process per SAI SDK instance.

# Database Implementation

SwSS implements the concept of a table in redis by naming keys with prefixes. A producer / consumer design is implemented to ensure integrity of data.

APP\_ tables are designed for each use case. For example, ROUTE\_TABLE and NEIGH\_TABLE.

ASIC\_ tables are created from the SAI header files. For example ASIC\_sai\_unicast\_route\_entry\_t and ASIC\_sai\_neighbor\_entry\_t.

## Table Operations

[TODO: link to github, .h files, API’s for common table operations]

Producer

SET – insert or update a key -> fields and values.

DEL – deletes a key

Consumer

POP – get a table change notification, the key name and the key->fields and values and operation [SET, DEL].

SELECT – check if a table notification exists.

[TODO: Elad to add examples]

## Transactions

SwSS implements transactions internally so producers and consumers can to stay in sync with the database using a queue-like method.

For each ‘TABLE’, there are QUEUE keys used for internal implementation of notifications. Here is an example of how it works.

The intfsyncd process performs a SET to the APP.INTF\_TABLE using the swss producer API. The producer API SETs the key/value in the TABLE and SETs an equivalent entry for each of the QUEUE keys.

The orchestration agent (OA) is a CONSUMER of the APP.INTF\_TABLE. OA will receive a notification from the swss consumer API that there is a data change on APP.INTF\_TABLE. The consumer API will POP the KEY, VALUE and OP from the QUEUE keys. The data the intfsyncd wrote to the APP.INTF\_TABLE remains untouched.

See the code for more details.

Tablename+”\_KEY\_QUEUE”

Tablename+”\_VALUE\_QUEUE”

Tablename+”\_OP\_QUEUE”

# Switch Data Service Database Schema

## Overview

Two databases are defined, APP and ASIC. Applications outside of SwSS are expected to store data by adding keys with well-defined names into the APP database. The ASIC database stores data used by hardware sync agents. Keys in the ASIC database are expected named strictly following SAI attributes.

Keys must be prefixed with a string that looks like a table name. The allowed keys are “[a-z][A-Z][0-9]\_” and end with “\_TABLE:”

In redis, databases are only defined my numbers:

Database 0 = APP\_DB

Database 1 = ASIC\_DB

Database 7 = TEST (used for unit testing)

## Schema

The SwSS schema is defined at: <https://github.com/Azure/sonic-swss/wiki/SwSS-Schema>

## Database 1 – ASIC\_DB

The ASIC database stores data used by hardware sync agents. Keys in the ASIC database are named strictly following SAI attributes. See <https://github.com/opencomputeproject/SAI>

# Switch state service Layer 3 Implementation

## L3 Route learning example

This section provides an example of how a BGP route is learned and propagated to the ASIC. Quagga is used as an example, but other routing applications could be used.

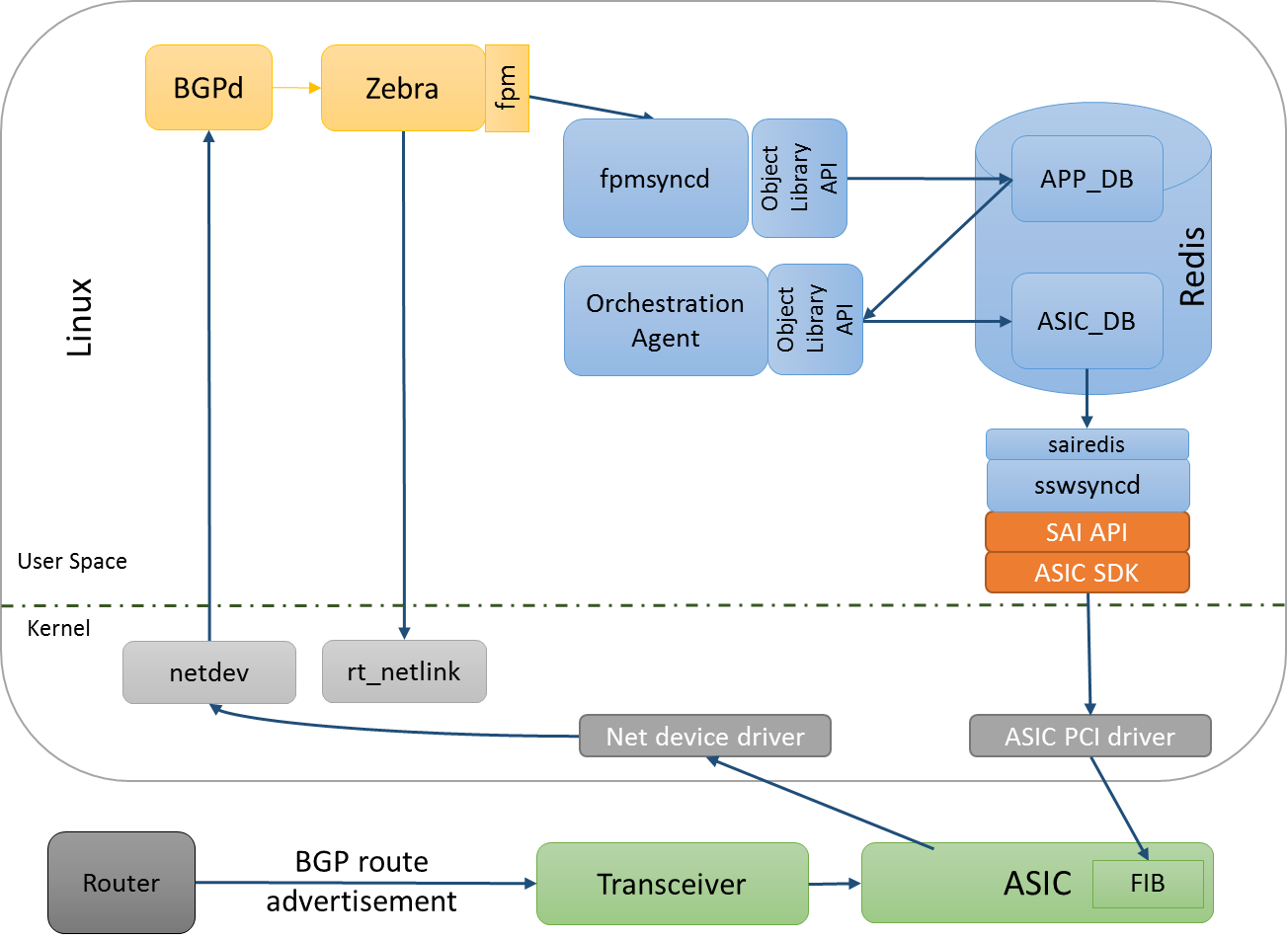


Figure 4 – Learn bgp route

# Appendix

# References