University of Amsterdam

# White-label open-source networking

*Authors*
Łukasz Makowski makowski@uva.nl
Paola Grosso p.grosso@uva.nl

January 22, 2018

# Table of Contents

# 1 Motivation for open-networking

Traditionally, networking equipment used to be a locked-in hardware with vendor's proprietary software installed. However, recently the trend of opening this ecosystem is becoming more and more popular.

Networking vendors allow the installation of other operating systems and software on their devices. This gives more power to network administrators, allowing the customization of the device according to their specific needs. Secondly, it also opens the possibility of running unified network operating system and networking daemons across the devices from various vendors.

There are a couple of commercial NOS distributions targeting white-label switches (e.g. Cumulus Linux[1] or PicaOS[2]). Generally, they are based on Linux operating system and for the purposes of control-plane also open-source software could be used. Nevertheless, the NOS itself remains proprietary and closed-source. In the opposition to this trend, there are also open-source NOS projects available. They constitute an alternative for users wanting to exchange and adjust the system to their requirements. Moreover, allowing to avoid the licensing costs of using commercial NOSes.

In this research we aim to answer the following questions:

1. Is it feasible to use a white-label, open-source based switching stack for a traditional deployment requiring basic L2 and L3 functionalities?

2. Are there any performance related differences between open-source NOS implementations?

[1] https://cumulusnetworks.com/products/cumulus-linux/
[2] http://www.pica8.com/products/picos

# 2 Architecture background

In this section, we discuss the components which can be identified in the white-label network stack, where Figure 1 gives the overview on this. Starting from the bottom, there is a hardware layer representing the actual device i.e. Network Processing Unit (NPU), interfaces, sensors, etc. It later gets discovered and handled by a lower-level layer, which signifies the drivers and other components residing in the space of network operating system. Next, for the actual interaction with the NPU device, the abstraction layer is placed. Lastly, to realize the network devices features (e.g. routing protocol operations, configuration) the management and control-plane applications are used.

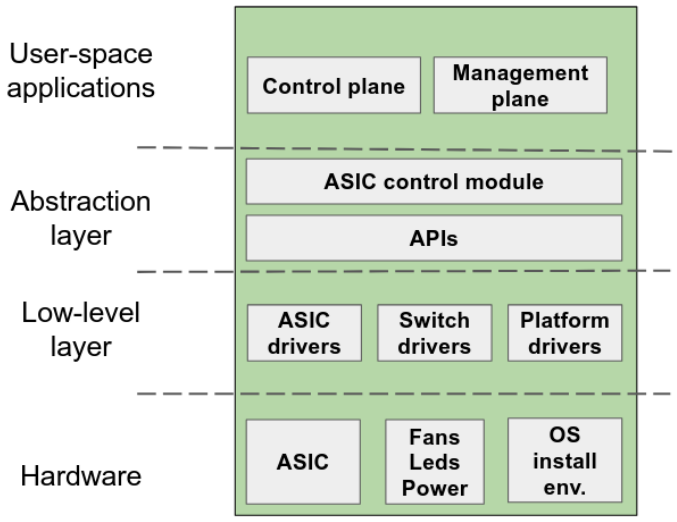


Figure 1: White-label networking stack logical structure

## 2.1 NPU communication

To program the NPU, there can be two generic approaches possible: (1) using NPU's SDK, (2) using a "proxy" component which will handle the NPU communication on behalf of the application. Generally, the former is not present in the design of open-source white label stacks, as it would require re-adjusting the applications to every new change in SDK. Following the second approach, there have been two ways established:

- Switch Abstraction Interface (SAI)
- switch-dev

### SAI

SAI develops an abstracted interface over SDKs provided by NPU vendors used in white-label switches (e.g. Broadcom, Mellanox). This approach allows the NOS developers to focus on a compatibility with SAI's API without the need to adjust to SDK changes or even a completely new NPU vendor.

### switch-dev

Switch-dev is a project aiming to contribute open versions of NPU drivers to the Linux kernel code. The vision is to perform all NPU communication inside the kernel space, so that the standard Linux interfaces (CLI, netlink API) can be used to change NPU state.

## 2.2 ASIC control module

The role of ASIC control module could be characterized as three activities: (1) interfacing with control-plane apps, (2) maintaining the state of the desired configuration, (3) communicating with an NPU (for example using SAI). On the second page of SAI specification [1] it can be seen that authors explicitly define the requirement for such component, additionally specifying rudimentary features it should implement. The examples of ASIC control module implementations can be found in open-networking systems such as OpenSwitch (OPX) and SONiC. Those are implemented as sets of processes cooperating with each other according to the publish/subscribe communication scheme.

### Message passing architecture

The concept of message-based asynchronous process communication is a key design principle for the analyzed network operating systems. It is also one of the attributes of Erlang[1] programming language which has been designed with a telecom-grade high reliability in mind. Erlang provides language level paradigms for message passing, enforcing

[1] http://www.erlang.org/

process communication to be realized only in this manner. Such approach allows for high process isolation and as a consequence, it results in a fact that "a software error in a concurrent process should not influence processing in the other processes in the system" [2]. A similar solution appears to be empowering Arista EOS[2] network operating system. A central database component (Sysdb) is used for inter-process message exchange between EOS individual components [3]. The analysis of middleware subsystems of Azure SONiC and OpenSwitch reveals that Redis[3] message-queue is used to realize similar objectives as outlined for Erlang and EOS.

[2]https://eos.arista.com/
[3]https://redis.io/

# 3 Network Operating Systems (NOS)

## 3.1 Open Networking Linux (ONL)

ONL is a project hosted by the Open Compute Project[1]. As the name implies, it is a Linux distro suited for installation on the network devices. Authors add the support for a specific hardware used in a particular model. This provides the foundation for starting to use the given platform, however, it lacks higher-level components which are the key part of fully usable network device. Most importantly, it is left to the user how to handle the communication with the NPU.

## 3.2 OpenSwitch (OPX)

The current version of OpenSwitch is actually the second generation of this project. The first one, sponsored mainly by HPE (Hewlett-Packard Enterprise) has been discontinued. The two main contributors are Dell (providing the actual NOS layer) and SnapRoute (FlexSwitch control-plane). At the moment of our evaluation, there were no official project images available. However, the images offering Dell S6000-ON switch support were available at one of the repository pages[2]. The inspection of the images revealed that the control-plane stack was not included as the project work was wrapping up. However, it was possible to access FlexSwitch source code (opx-flxl* repositories in OPX github project).

[1] http://www.opencompute.org/

[2] https://github.com/open-switch/opx-docs/wiki/Install-OPX-Base-on-Dell-S6000-ON-platform

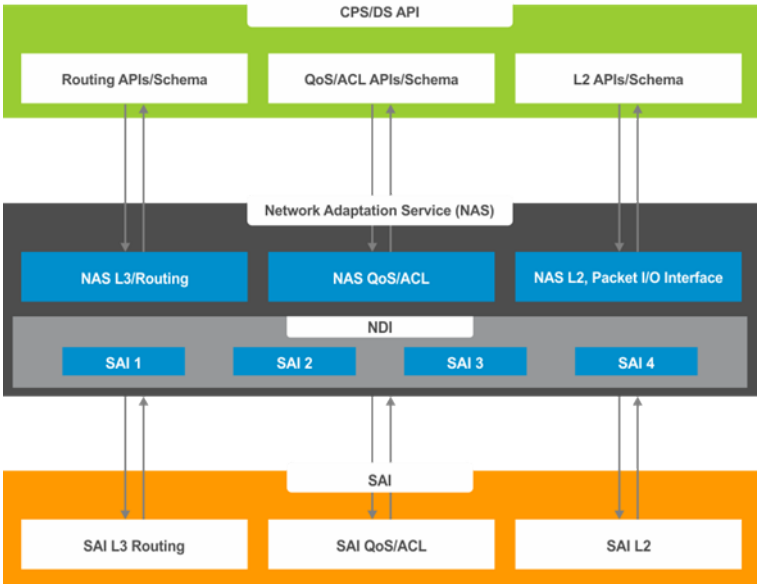


Figure 2: NAS design[3].

OPX is based on Debian operating system combined with the abstraction layer components and utilities developed by OPX project participants. As illustrated in Figure 2, the component ingesting the intended configuration is Control Plane Service (CPS). Next, Network Adaptation Service (NAS) is responsible for translating the configuration to the set of NPU commands and communicating those using SAI interface. Table 1 (taken from [4]) lists the available features. The important fact is that not all of them are configurable with the means of using standard Linux network utilities. The operations which are simply not present there need to be performed through CPS API.

[3]https://github.com/open-switch/opx-docs/raw/master/images/nas_design.png

| Networking Feature | Configure with Linux Commands or Open Source Application | Configure with CPS API |
|---|---|---|
| Interfaces | | |
| Physical | Yes | Yes |
| Link Aggregation (LAG) | Yes (Bond) | Yes |
| VLAN | Yes | Yes |
| Fanout (4x10G) | No | Yes (script) |
| Layer 2 Bridging | | |
| LLDP | Yes | No |
| MAC address table | No | Yes |
| STP | Yes | Yes |
| VLAN | Yes | Yes |
| Layer 3 Routing | | |
| ECMPv4 | Yes | Yes |
| ECMPv6 | Yes | Yes |
| IPv4 | Yes | Yes |
| IPv6 | Yes | Yes |
| Unicast routing | Yes | Yes |
| QoS | No | Yes |
| ACLs | No | Yes |
| Monitoring | | |
| Mirroring | No | Yes |
| sFlow | No | Yes |
| Port and VLAN statistics | No | Yes |

Table 1: Selected supported Dell OS10 networking features.

## 3.3 SONiC

SONiC is an abbreviation for Software for Open Networking in the Cloud. It is a NOS by Microsoft Azure used to empower the network operations, however, the scale of its deployment is non-public. Table 2 outlines the features listed in the official SONiC's roadmap [5]. Based on that, it can be concluded that SONiC aims at building a BGP empowered Layer 3 fabrics than traditionally switched Ethernet network.

| Networking Feature |
|---|
| BGP |
| ECMP |
| LAG |
| LLDP |
| QoS - ECN |
| QoS - RDMA |
| Priority Flow Control |
| WRED |
| COS |
| SNMP |
| Syslog |
| Sysdump |
| NTP |
| COPP |
| DHCP Relay Agent |
| SONiC to SONiC upgrade |
| Multiple Images support |
| One Image |
| ACL permit/deny |
| IPv6 |
| Tunnel Decap |
| Mirroring |
| Post Speed Setting |
| BGP Graceful restart helper |
| BGP MP |

Table 2: SONiC supported networking features

Similarly to OPX, SONiC is also working on top of Linux. SONiC's architecture is depicted in Figure 3. Its core design principle is to isolate each component into a separate Docker container. The center of a system is Orchestration Agent (OrchAgent) which fetches the unified configuration and converts it to particular SAI API commands. Those are further picked up by syncd daemon, which directly handles the NPU programming process.

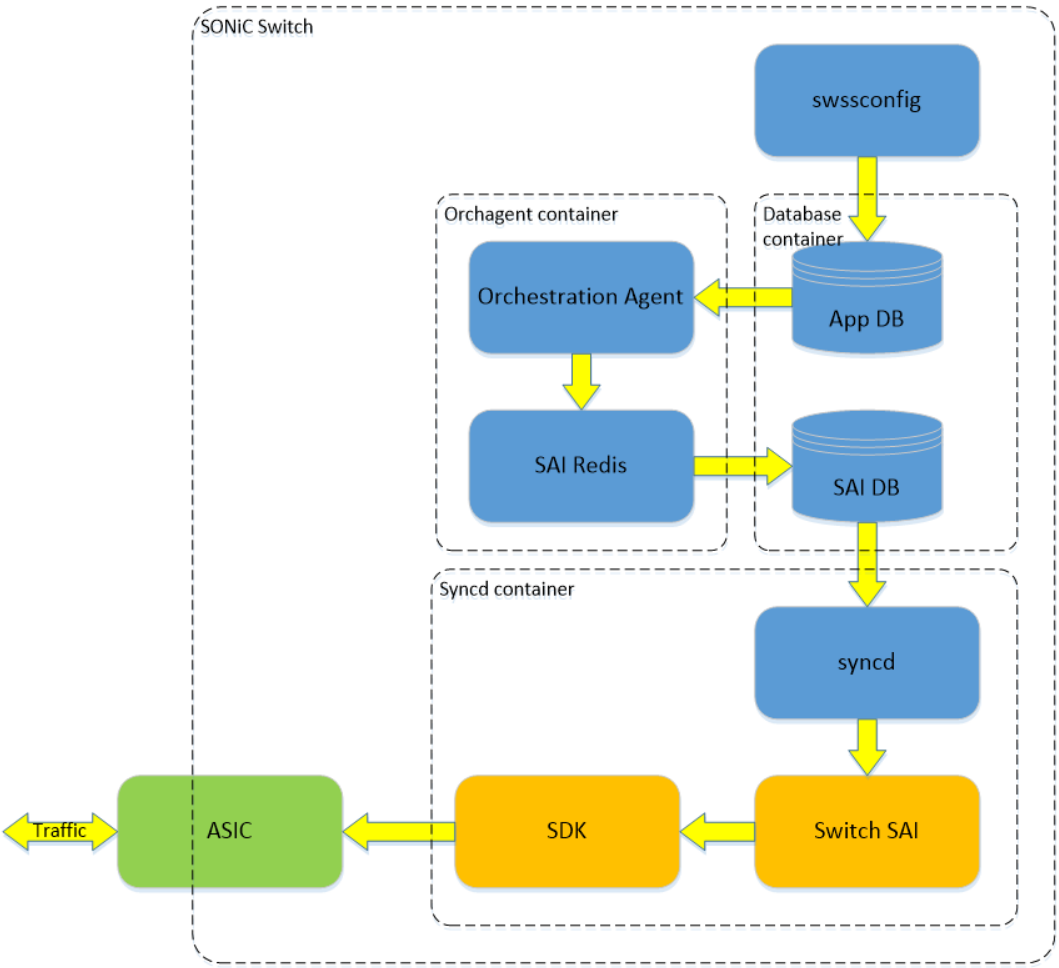


Figure 3: SONiC Architecture[4]

[4]https://github.com/Azure/SONiC/wiki/Everflow-High-Level-Design

## 3.4 Broadcom NPU interface access

The SAI specification assumes the ability to expose the NPU's internal interface to a switch user. This feature is available on SONiC and OPX.

The utilities allowing the switch communication are effectively opening a UNIX socket to the SAI's kernel module interface, proxying all the commands to the NPU. The OPX version of the tool is called `opx-switch-shell`, whereas SONiC one is `bcmcmd`.

This feature is particularly useful for the verification of the control-plane operations and observing if an intended state is actually reflected in the NPU state. The downside of it is that there is no publicly available manual for all available commands. Therefore, a user is limited to the brief descriptions visible in the NPU help menus.

# 4 Control-plane stacks

## 4.1 Quagga/FRR

Quagga[1] is a routing suite implementing a handful of routing protocols. Specifically, OSPF and BGP for both the IPv4 and IPv6 protocol versions.

Additionally, it is also worth to note Quagga's fork called Free Range Routing[2] (FRR). Among sharing the features of its predecessor, FRR focuses on faster development pace and new features. In specific, BGP improvements for the data center fabric environments and new protocols (i.e. EIGRP or LDP). FRR is a routing suite shipped with Cumulus Linux, commercial NOS for network devices.

## 4.2 BIRD

BIRD Internet Routing Daemon (BIRD[3]) aims to be an alternative to the Quagga suite. As its competitor, it is capable of running BGP and OSPF protocols for both versions of the IP protocol. Its modular design enables the possibility to run multiple instances of the same routing protocol making it better suited for multitenant deployments requiring a higher-level of isolation.

## 4.3 Flexswitch

FlexSwitch is the suite of network protocol implementations from SnapRoute company. SnapRoute contributed with a former versions of FlexSwitch utilities to OPX project[4]. Among many features, it claims to support STP, LACP as well as routing protocols such as OSPF or BGP. As of the moment of writing FlexSwitch has not been integrated into OPX system image.

[1] http://www.nongnu.org/quagga/index.html
[2] https://frrouting.org/
[3] http://bird.network.cz/
[4] https://github.com/open-switch/opx-flxl2, https://github.com/open-switch/opx-flxl3

# 5 Method

Our research consisted of three main parts:

- Preliminary phase
- Feature tests
- FIB (Forwarding Information Base) installation latency test

## 5.1 Preliminary phase

This part involved analyzing the available open-source networking software (described in Chapters 3 and 4) and establishing the testbed allowing to perform the evaluation. The criteria were that the NOS sources should be open-source and offer the support for white-label switches available in SNE OpenLab testbed. Ultimately, we decided to focus on two NOS distributions — OpenSwitch (OPX) and SONiC, followed by BIRD and Quagga for control-plane routing features. The final configurations are outlined in Table 3.

| Vendor | Model | NOS | Build version |
|---|---|---|---|
| Dell | S6000-ON | SONiC | 201705 |
| Dell | S6000-ON | OPX | 2.1.0(0) |
| Dell | S4048-ON | OPX | 2.1.0(0) |

Table 3: Used hardware and NOS versions installed

## 5.2 Feature tests

We have synthesized the list of rudimentary features which we consider important for a regular data-center switch:

- Configuration and management (CLI/API, LLDP, DHCP relay)
- Layer 2 features (VLAN, STP, LAG)
- Layer 3 features (OSPF)

In the following subsections, we discuss our approach regarding each test. In order to perform the evaluation, the switches were interconnected as illustrated in Figure 4. This was the topology which allowed us to evaluate all of our established scenarios, without the need to re-adjusting it for every test.

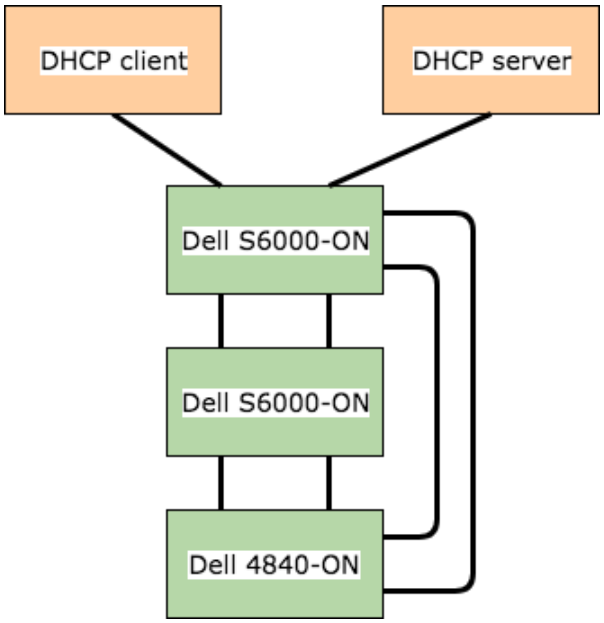


Figure 4: Physical topology used in feature test phase.

### Configuration and management

#### CLI & API

We looked at the methods allowing to deploy configuration on a switch. First, we analyzed CLI, with the focus on what can be achieved with it i.e. configuration, verification.

Moreover, we also looked at the methods for applying a configuration in a persistent manner. Lastly, we examined configuration interfaces on the switch allowing for programmatic device configuration supporting tasks such as automation.

### Link Layer Discovery Protocol (LLDP)

We checked if LLDP messages are properly exchanged between a pair of the devices. Additionally, verifying if the information contained in LLDP messages corresponds to the actual config.

### Dynamic Host Configuration Protocol (DHCP) relay

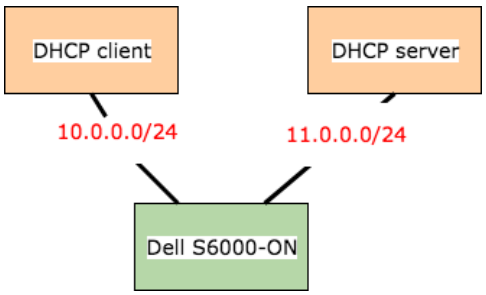


Figure 5: DHCP relay test topology setup

Similarly to LLDP, the actual DHCP functionality resides entirely in the control-plane layer. SONiC claims officially to support it (internally it uses `isc-dhcp-relay`), in OPX documentation we did not encounter any DHCP-relay related notes. However, we manually installed ISC's DHCP relay agent and evaluated it.

In the simple set-up (Figure 5) we used three components: DHCP client (`isc-dhcp-client`) residing in 11.0.0.0/24 network, switch acting as DHCP relay agent, and DHCP server (`dnsmasq`) placed in 10.0.0.0/24 network.

## Layer 2 (L2)

### VLAN

While testing VLAN functionality we were generally focusing on two rudimentary features:

- ability to set VLAN on a port (so-called “access port”)
- IEEE 802.1Q protocol support (VLAN “trunking”)

As shown in Figure 6, the approach was to create VLAN trunk between two OPX switches, as well as between OPX and SONiC switch. Moreover, we also configured an untagged port (VLAN 20) between OPX and SONiC devices.

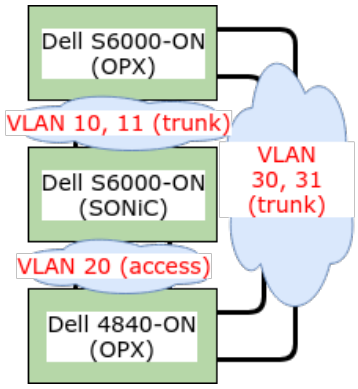


Figure 6: VLAN test topology setup consisted of two tagged VLAN trunks and one untagged VLAN port

### Spanning-Tree Protocol (STP)

We tested STP operations between two switches connected with two links. First, between two OPX switches, secondly between SONiC and OPX switches. The setup is illustrated in Figure 7. The version of SONiC used in this research had no STP support integrated. Regardless of this fact, we decided to see what will be the behavior of linux-bridge STP operating without the NPU support.

Using the topology defined earlier, we tested the STP operation between two switches interconnected with two cables. The expected result (due to the nature of STP) was having one of those two links in a `blocking` state, and one of the switches operating as an STP root bridge.

Additionally, we looked at Spanning-Tree Group (STG) mechanism support. It allows grouping of VLANs in order to assign them to different port and bridge priorities. This enables more effective link usage i.e. for STG1 the port can be `blocking`, whereas for STG2 it will be in a `forwarding` state.

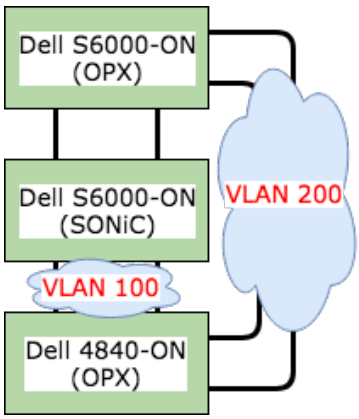


Figure 7: STP test topology setup

### Link aggregation (LAG)

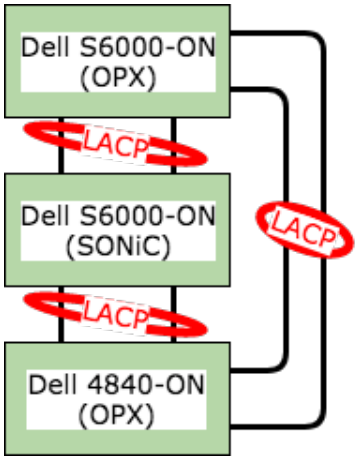


Figure 8: Link aggregation (LAG) setup

To evaluate LAG we attempted to create link bundles between all devices in the test setup (Figure 8). This was in the essence aggregating two physical links between each switches pair as a single virtual port. Link aggregation feature was handled by Linux's `bonding` driver in case of OPX, and `libteam`[1] for SONiC. To create LAG in OPX we were using `ifenslave` Linux command, whereas in SONiC it is a matter of specifying the configuration file's `PortChannel` option.

[1] http://libteam.org/

### Layer 3 (L3)

#### Open Shortest Path First (OSPF)

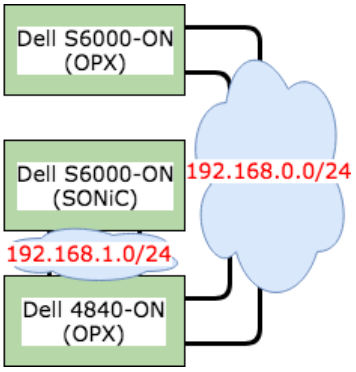


Figure 9: OSPF setup

To test OSPF we set-up three device network, where particular S6000-ON devices were placed in the different IP segments. Whereas, S4048-ON resided in the middle, effectively having full knowledge about all configured IP networks (Figure 9). To be able to achieve the communication between them (i.e. with `ping` commands), there had to a routing information exchanged. For that purpose we setup basic, single area OSPF neighbourship between all devices.

## 5.3 FIB installation latency

In order to quantitatively compare the performance of OPX and SONiC we measured the latency of FIB route installation. Which effectively is the time it takes from receiving a route update and offloading it to the hardware. As previously outlined in Chapter 2, the information about a new route has to traverse multiple components before it ends as a FIB entry in the NPU. In this test we were using two configurations:

- Dell S6000-ON running OPX+Quagga
- Dell S6000-ON with SONiC and its stock Quagga installation

As outlined in Figure 10, the switch was running BGP protocol and it was receiving updates from an external exaBGP[2] speaker combined with super-smash-brogp[3] tool.

[2]https://github.com/Exa-Networks/exabgp
[3]https://github.com/spotify/super-smash-brogp

The latter was responsible for generating test BGP network updates which were sent to the switch. The detailed configuration of super-smash-brogp is provided in Listing 1. To measure the amount of time (latency) of route installation, for each route update we were capturing two timestamps:

- t1 - route update arrival as logged by Quagga
- t2 - route installation time as reported by NPU control module

Later on, t1 and t2 were subtracted resulting in a relative latency score.

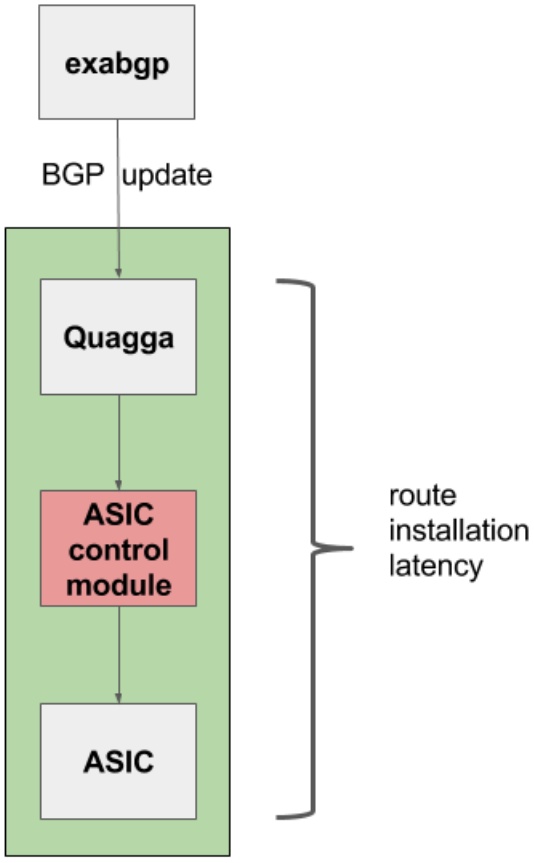


Figure 10: Route installation latency setup

```
---
PREFIXES_FILE: "data/v4_full_table_and_default"

NEXT_HOP: "10.0.0.34"

WAITING_TIME: 55 # Time to wait between iterations

INITIAL_WARMUP: 100 # Number of prefixes to send initially
INITIAL_WAIT: 10 # Seconds to wait initially

MAX_TOTAL: 30000

# On every iteration we will install randint(MIN_PREFIXES, MAX_PREFIXES)
# prefixes
MIN_PREFIXES: 10
MAX_PREFIXES: 50

REMOVE_PREFIXES: 1 # Percentage of prefixes to remove on each iteration

# Some parameters to add random AS PATHs to each advertisement
NUM_DIFFERENT_AS_PATHS: 100
MIN_AS_LENGTH: 0
MAX_AS_LENGTH: 5
```

Listing 1: Configuration file of super-smash-brogp. It was executed directly by exaBGP (`process add-routes` configuration parameter) and was responsible for route generation

# 6 Results: feature tests

In the following sections we discuss details regarding performed feature related evaluation. We structure the results in relation to both OPX and SONiC to clearly mark our findings regarding each NOS.

## 6.1 Configuration and management

In Table 4 we present the overview of the performed tests.

| Feature name | OPX | SONiC |
|---|---|---|
| CLI/API | pass[1] | pass[1] |
| LLDP | pass | pass |
| DHCP relay | pass | supported, but not evaluated |

Table 4: Configuration and management tests summary

### CLI & API interfaces

#### OPX

OPX offers three general approaches to its configuration:

- using standard Linux utilities and config files (e.g. `/etc/network/interfaces`) supplemented with extra commands (prefixed with `opx-`)
- interacting with Control Plane Services (CPS) API
- XML files read by OPX subsystems during the OS startup (detailed information can be found in [4])

[1]Only subset of operations supported in the CLI

While configuring the devices, our first approach was using the default Debian Linux networking configuration files. Listing 2 shows a sample configuration snippet.

```
iface vlan100 inet static
   bridge_ports none
   address 2.2.2.1
   netmask 255.255.255.0

allow-hotplug e101-001-0
iface e101-001-0 inet manual
    post-up bash -c ". /etc/opx/opx-environment.sh;opx-ethtool -s e101-001-0 speed 40000 duplex full autoneg off"
    post-up ifconfig e101-001-0 up
    post-up brctl addif vlan100 e101-001-0 || true
    post-down ifconfig e101-001-0 down
```

Listing 2: OPX configuration with the use of Debian configuration syntax

The switch interface required setting non-default duplex or speed values that implied the need to use the `post-up` Debian configuration stanza. Even if specified correctly, this was still not operating as expected as a persistent configuration. The NPU attached interfaces are actually brought up several seconds after the startup of a NOS, and that was causing edge conditions which effectively were leaving the device in an unconfigured state. Therefore, the end configuration we ended up with was adjusting port duplex and speed values in the `phy_media_default_npu_setting.xml` configuration file, complemented by the configuration in the `interfaces` file.

### SONiC

SONiC could be interacted with in the following manners:

- Similarly to OPX, with standard Linux tools or SONiC provided `show` and `configure` commands[1]. Whereas the latter implements only a subset of the available options
- Minigraph configuration file. It serves as a single point of configuration for all switch related options. Additionally, the configuration of Quagga (providing control-plane features) is also integrated here
- SwSS API, however is not meant for user originating configuration

In our experiments, we configured the device with the Minigraph file (`/etc/sonic/config_db.json`, the configuration snippet is presented in Listing 3).

[1] https://github.com/Azure/SONiC/wiki/Command-Reference

```
"PORTCHANNEL_INTERFACE": {
"PortChannel04|FC00::7D/126": {},
"PortChannel04|10.0.0.62/31": {},
"PortChannel01|10.0.0.56/31": {},
"PortChannel01|FC00::71/126": {},
"PortChannel03|FC00::79/126": {},
"PortChannel03|10.0.0.60/31": {}
},
"onie_config_version": "1",
"PORTCHANNEL": {
"PortChannel03": {
"members": [
"Ethernet120"
]
},
"PortChannel01": {
"members": [
"Ethernet0",
"Ethernet4"
]
},
"PortChannel04": {
"members": [
"Ethernet124"
]
}
},
```

Listing 3: The fragment of SONiC's JSON configuration file

In the recent SONiC release the configuration is expressed in a JSON format (previously XML), which in turn gets translated to a particular daemon's configuration file (Figure 11).

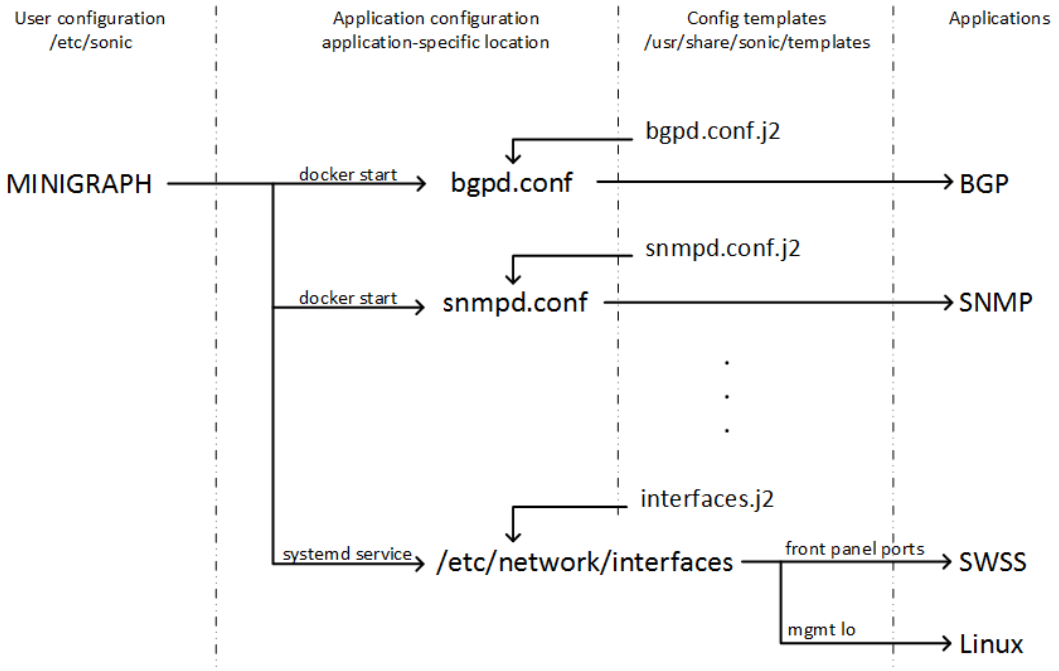


Figure 11: Sonic Minigraph configuration file workflow[2]

[2]https://github.com/Azure/SONiC

At the moment of testing, the dynamic configuration with a `config load` command seemed not to be operating correctly. As a result, device restart was required to make a new state of configuration effective.

## LLDP

Both LLDP and OPX, support LLDP protocol by using `lldpad` application. We were able to successfully inter-exchange LLDP information between SONiC and OPX switches.

## DHCP relay

### OPX

We were able to successfully relay DHCP messages with the applied configuration. Listing 4 presents the output of DHCP relay daemon forwarding incoming DHCP requests to a remote DHCP server answering the request.(Listing 5).

```
root@OPX-S6000-ON:/var/log# /usr/sbin/dhcrelay -d -i e101-009-0 -i eth0 10.0.0.2
Internet Systems Consortium DHCP Relay Agent 4.3.1
Copyright 2004-2014 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/
Listening on LPF/eth0/ec:f4:bb:fb:d2:f8
Sending on LPF/eth0/ec:f4:bb:fb:d2:f8
Listening on LPF/e101-009-0/ec:f4:bb:fb:d3:19
Sending on LPF/e101-009-0/ec:f4:bb:fb:d3:19
Sending on Socket/fallback

Forwarded BOOTREQUEST for 00:1c:73:7b:f7:5d to 10.0.0.2
Forwarded BOOTREPLY for 00:1c:73:7b:f7:5d to 11.0.0.14
Forwarded BOOTREQUEST for 00:1c:73:7b:f7:5d to 10.0.0.2
Forwarded BOOTREPLY for 00:1c:73:7b:f7:5d to 11.0.0.14
Forwarded BOOTREQUEST for ec:f4:bb:fc:1c:69 to 10.0.0.2
```

Listing 4: ISC DHCP relay agent running on OPX. DHCP requests arriving at e101-009-0 interface, were forwarded to 10.0.0.2 DHCP server through eth0 interface.

```
[root@chicken src]# ./dnsmasq -d -i eth2 -F 11.0.0.10,11.0.0.20,255.255.255.0

dnsmasq-dhcp: DHCPOFFER(eth2) 11.0.0.14 00:1c:73:7b:f7:5d
15:44:51.179315 IP (tos 0x0, ttl 64, id 5615, offset 0, flags [DF], proto UDP (17), length 328)
  10.0.0.44.67 > 10.0.0.2.67: [udp sum ok] BOOTP/DHCP, Request from 00:1c:73:7b:f7:5d, length 300, hops 1, xid 0xa6f0977d,
     secs 22, Flags [none] (0x0000)
  Gateway-IP 11.0.0.1
  Client-Ethernet-Address 00:1c:73:7b:f7:5d
#output omitted for brevity
15:44:51.179435 IP (tos 0xc0, ttl 64, id 48792, offset 0, flags [none], proto UDP (17), length 328)
  10.0.0.2.67 > 11.0.0.1.67: [bad udp cksum f384!] BOOTP/DHCP, Reply, length 300, hops 1, xid 0xa6f0977d, secs 22, Flags [
     none] (0x0000)
  Your-IP 11.0.0.14
  Server-IP 10.0.0.2
  Gateway-IP 11.0.0.1
#output omitted for brevity
```

Listing 5: DHCP server listening for upcoming requests on eth2 interface and serving addresses out of 11.0.0.10-20 range.

## 6.2 Layer 2 (L2)

| Feature name | OPX | SONiC |
|---|---|---|
| VLAN | pass | VLAN trunking not supported |
| STP | pass | not supported |
| LAG | pass | pass |

Table 5: L2 tests summary

In Table 5 we outline the results of performed tests. In the following subsections we present listings corresponding to VLAN, STP and LAG tests.

### VLAN

#### OPX

OPX satisfied both VLAN requirements we had. We were able to configure a port as an untagged interface, as well as, create VLAN trunk hosting multiple VLANs. In Listing 6 we present how the VLAN trunk configuration was reflected in the NPU.

```
root@OPX-4048-ON:~# opx-switch-shell "l2 show"
mac=ec:f4:bb:fb:d3:62 vlan=30 GPORT=0x45 modid=0 port=69/xe68
mac=ec:f4:bb:fb:d3:62 vlan=31 GPORT=0x45 modid=0 port=69/xe68
```

Listing 6: Broadcom NPU state showing VLANs 30 and 31 configured on the trunk port

#### SONiC

In SONiC configuration we were able to configure ports as untagged, however VLAN trunking did not work. In Listing 7 it can be observed that regardless having VLANs 10,11 in the trunk configuration, those were not present in the form of NPU entries.

```
root@sonic:/var/log/swss# bcmcmd 'l2 show'
l2 show
mac=34:17:eb:f3:2b:05 vlan=1 GPORT=0x20 modid=0 port=32/xe31 Hit
mac=ec:f4:bb:fb:d2:f9 vlan=1 GPORT=0x2 modid=0 port=2/xe1 Hit
mac=ec:f4:bb:fc:1c:69 vlan=1 GPORT=0x0 modid=0 port=0/cpu0 Static CPU
drivshell>
```

Listing 7: VLAN tagging configuration was not reflected in the NPU state (all ports are associated with VLAN 1)

# STP

## OPX

In the STP domain between two OPX devices we were able to achieve successful STP operations. Listing 8 presents the STP state on a device which has elected another device as a STP root (S4048-ON switch). Listing 9 presents issued verification commands.

```
root@OPX-S6000-ON:~# brctl showstp vlan200
vlan200
 bridge id 8000.ecf4bbfbd35f
 designated root 8000.3417ebf32b05
#output omitted for brevity

e101-031-0 (1)
 port id 8001 state forwarding
#output omitted for brevity

e101-032-0 (2)
 port id 8002 state blocking
#output omitted for brevity
```

Listing 8: STP bridge state on S6000-ON it elected a remote STP root bridge as a consequence inerface e101-032-0 is in the `blocking` state

```
root@OPX-4048-ON:~# brctl showstp vlan200
vlan200
 bridge id 8000.3417ebf32b05
 designated root 8000.3417ebf32b05
#output omitted for brevity

e101-053-0 (1)
 port id 8001 state forwarding
#output omitted for brevity

e101-054-0 (2)
 port id 8002 state forwarding
#output omitted for brevity
```

Listing 9: STP enabled bridge at S4048-ON (STP root), both its ports (e101-053-0 and e101-053-0) are in `forwarding` state

In Listing 10 we show the state of STG after a bridge with VLAN 30 was configured on S6000-ON. As can be observed, this configuration is not reflected in STG. It appears that linux-bridge STP implementation does not implement STG. However, as stated in OPX documentation, similar goals might be achieved by having multiple linux-bridge instances performing separate STP cost calculations.

```
root@OPX-S6000-ON:~# opx-switch-shell 'stg show'
STG 0: contains 0 VLANs
  Disable: xe
STG 1: contains 1 VLAN (1)
  Disable:
xe1-xe3,xe5-xe7,xe9-xe11,xe13-xe15,xe17-xe19,xe21-xe23,xe25-xe27,xe29-xe31,xe33-xe35,xe37-xe39,xe41-xe43,xe45-xe47,xe53-xe55
      ,xe57-xe59,xe61-xe63,xe65-xe67,xe69-xe71,xe73-xe75,xe77-xe79,xe81-xe83,xe85-xe87,xe89-xe91,xe93-xe95,xe97-xe99
  Forward:
xe0,xe4,xe8,xe12,xe16,xe20,xe24,xe28,xe32,xe36,xe40,xe44,xe48-xe52,xe56,xe60,xe64,xe68,xe72,xe76,xe80,xe84,xe88,xe92,xe96,
      xe100-xe103
STG 255: contains 1 VLAN (4095)
  Disable:
xe1-xe3,xe5-xe7,xe9-xe11,xe13-xe15,xe17-xe19,xe21-xe23,xe25-xe27,xe29-xe31,xe33-xe35,xe37-xe39,xe41-xe43,xe45-xe47,xe53-xe55
      ,xe57-xe59,xe61-xe63,xe65-xe67,xe69-xe71,xe73-xe75,xe77-xe79,xe81-xe83,xe85-xe87,xe89-xe91,xe93-xe95,xe97-xe99
  Forward:
xe0,xe4,xe8,xe12,xe16,xe20,xe24,xe28,xe32,xe36,xe40,xe44,xe48-xe52,xe56,xe60,xe64,xe68,xe72,xe76,xe80,xe84,xe88,xe92,xe96,
      xe100-xe103
```

Listing 10: Even though VLAN 30 was configured, it has not been reflected in the NPU's STG configuration

### SONiC

For SONiC-OPX setup we did not manage to achieve operational STP configuration. The packet capture on SONiC switch (Listing 11) showed that NPU was dropping incoming STP frames. It can be observed that the switches did not properly detect itself (Listings 12 and 13). Interestingly, regardless of root function, OPX device blocked one of its ports. That effectively was preventing network loops from occurring.

```
root@sonic:~# tcpdump -i Ethernet0 -nnn -vvv
tcpdump: listening on Ethernet0, link-type EN10MB (Ethernet), capture size 262144 bytes
15:24:17.878647 STP 802.1d, Config, Flags [none], bridge-id 8000.ec:f4:bb:fc:1c:69.8012, length 35
  message-age 0.00s, max-age 20.00s, hello-time 2.00s, forwarding-delay 15.00s
  root-id 8000.ec:f4:bb:fc:1c:69, root-pathcost 0
15:24:19.878662 STP 802.1d, Config, Flags [none], bridge-id 8000.ec:f4:bb:fc:1c:69.8012, length 35
  message-age 0.00s, max-age 20.00s, hello-time 2.00s, forwarding-delay 15.00s
  root-id 8000.ec:f4:bb:fc:1c:69, root-pathcost 0
15:24:21.878654 STP 802.1d, Config, Flags [none], bridge-id 8000.ec:f4:bb:fc:1c:69.8012, length 35
```

Listing 11: Tcpdump capture shows that neighbour device STP frames were not delivered to the control-plane. We could only observe those originated by SONiC switch itself

```
root@OPX-S6000-ON:/etc/network/interfaces.d# brctl showstp vlan20
vlan20
 bridge id 8000.ecf4bbfbd2f9
 designated root 8000.ecf4bbfbd2f9
#output omitted for brevity

e101-001-0 (1)
 port id 8001 state forwarding
#output omitted for brevity

e101-002-0 (2)
 port id 8002 state blocking
#output omitted for brevity
```

Listing 12: OPX-S6000 has elected itself as a root bridge, while surprisingly keeping one of its ports in `blocking` state.

```
root@sonic:~# brctl showstp Vlan20 | grep -wE 'Ethernet(0|4)' -A 6
Ethernet0 (18)
 port id 8012 state forwarding
 designated root 8000.ecf4bbfc1c69 path cost 100
 designated bridge 8000.ecf4bbfc1c69 message age timer 0.00
 designated port 8012 forward delay timer 0.00
 designated cost 0 hold timer 0.00
 flags
--
Ethernet4 (20)
 port id 8014 state forwarding
 designated root 8000.ecf4bbfc1c69 path cost 100
 designated bridge 8000.ecf4bbfc1c69 message age timer 0.00
 designated port 8014 forward delay timer 0.00
 designated cost 0 hold timer 0.00
 flags
```

Listing 13: SONiC switch detected itself as the only device in the STP domain. Therefore all its ports are in `forwarding` state.

Regarding STG functionality we observed that VLAN 20 configured in the tests was actually reflected in the STG NPU status (Listing 14).

```
root@sonic:~# bcmcmd 'stg show'
stg show
STG 0: contains 0 VLANs
  Disable: xe
STG 1: contains 4 VLANs (1,20,4093-4094)
  Forward: xe
```

Listing 14: SONiC STG state

### LAG

We have successfully created all planned LAG bundles. Below we present the verification of SONiC-OPX LAG. The LACP protocol status as reported on SONiC indicated the LAG bundle is functioning (Listing 15). Additionally, atop virtual interfaces we set-up IP addresses and the communication was verified with the `ping` command (Listing 16).

```
root@sonic:~# teamdctl PortChannel01 state
setup:
  runner: lacp
ports:
  Ethernet0
  link watches:
  link summary: up
  #output omitted for brevity
  Ethernet4
  link watches:
  link summary: up
  #output omitted for brevity
```

Listing 15: LACP bundle status verified on SONiC device

```
root@OPX-S6000-ON:~# ping 10.0.0.56
PING 10.0.0.56 (10.0.0.56) 56(84) bytes of data.
64 bytes from 10.0.0.56: icmp_seq=1 ttl=64 time=2001 ms
#(output omitted for brevity)
--- 10.0.0.56 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9004ms
rtt min/avg/max/mdev = 0.502/300.147/2001.968/640.093 ms, pipe 2
```

Listing 16: Verification of LAG bundle operations by pinging the IP address configured on the other side of the link

## 6.3 Layer 3 (L3) : OSPF

| Feature name | OPX+BIRD | OPX+Quagga | SONiC+Quagga |
|---|---|---|---|
| OSPF | pass | pass | pass |

Table 6: L3 tests summary

As shown in Table 6 we successfully established an OSPF domain consisting of three distinct platforms. First, there were a couple of problems related to OSPF specifics.

Namely we discovered that hence SONiC uses a fixed MTU size of 9100 bytes, we had to adjust this parameter on the neighbour OPX switch.

SONiC+Quagga switch (attached to `192.168.1.0/24` network) received a route update about `192.168.0.0/24` and installed it in his routing table (Listing 17).

```
root@sonic:/home/admin# vtysh -c 'show ip ospf rout'
============ OSPF network routing table ============
N 192.168.0.0/24 [20] area: 0.0.0.0
  via 192.168.1.11, PortChannel02
N 192.168.1.0/24 [10] area: 0.0.0.0
  directly attached to PortChannel02
```

Listing 17: The route to 192.168.0.0/24 remote network is received to SONIC+Quagga switch

Additionally, it can be seen that OPX+Quagga switch detected both its neighbours (Listing 18).

```
root@OPX-4048-ON:/etc/network/interfaces.d# vtysh -c 'show ip ospf nei'
Neighbor ID Pri State Dead Time Address Interface RXmtL RqstL DBsmL
0.0.0.10 1 Full/DROther 39.206s 192.168.0.10 bond0:192.168.0.11 0 0 0
10.1.0.32 1 Full/DROther 39.633s 192.168.1.12 bond1:192.168.1.11 0 0 0
```

Listing 18: OSPF neighbours reported by OPX+Quagga. It shows the visibility of SONiC+Quagga and OPX+BIRD devices.

In Listing 19 we present NPU FIB entries on all devices used in the test. As can be observed the routing table entries from the NOS were offloaded to the hardware.

```
root@sonic:/home/admin# bcmcmd 'l3 defip show'
Unit 0, Total Number of DEFIP entries: 393216
# VRF Net addr Next Hop Mac INTF MODID PORT PRIO CLASS HIT VLAN
#output omitted for brevity
262152 0 192.168.1.0/24 00:00:00:00:00:00 100003 0 0 0 1 n
262153 0 192.168.0.0/24 00:00:00:00:00:00 100004 0 0 0 0 n

root@OPX-4048-ON:/etc/network/interfaces.d# opx-switch-shell 'l3 defip show'
Unit 0, Total Number of DEFIP entries: 16384
# VRF Net addr Next Hop Mac INTF MODID PORT PRIO CLASS HIT VLAN
2048 0 192.168.0.0/24 00:00:00:00:00:00 100002 0 0 0 0 n
2048 0 192.168.1.0/24 00:00:00:00:00:00 100002 0 0 0 0 n

root@OPX-S6000-ON:~# opx-switch-shell 'l3 defip show'
Unit 0, Total Number of DEFIP entries: 16384
# VRF Net addr Next Hop Mac INTF MODID PORT PRIO CLASS HIT VLAN
2816 0 192.168.0.0/24 00:00:00:00:00:00 100002 0 0 0 0 n
2816 0 192.168.1.0/24 00:00:00:00:00:00 100004 0 0 0 0 n
```

Listing 19: NPU level verification of FIB entries. The required routes corresponding to our test were installed in particular switch FIB.

Lastly, we show the verification of the connectivity issued from the SONiC device (Listing 20).

```
root@sonic:/home/admin# traceroute 192.168.0.10 -n -q 1
traceroute to 192.168.0.10 (192.168.0.10), 30 hops max, 60 byte packets
 1 192.168.1.11 0.494 ms
 2 192.168.0.10 1.468 ms
```

Listing 20: Traceroute from SONiC+Quagga, traversed OPX+BIRD device (192.168.1.11) and reached OPX+Quagga (192.168.0.10)

# 7 Results: FIB installation latency

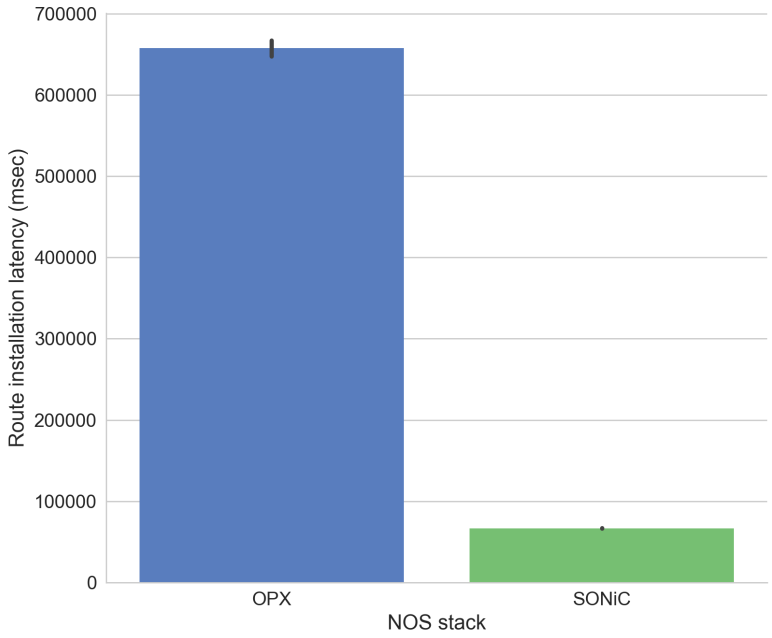


Figure 12: OPX and SONiC route installation latency comparison

Figure 12 illustrates the route installation latency for two different configurations we created. According to our measurements, it appears that SONiC had noticeably lower FIB installation latency than corresponding OPX configuration.

# 8 Discussion

## 8.1 Preliminary phase

The ecosystem of white-label NOSes and hardware is in constant development and therefore it undergoes dynamic changes. During our project, we were blocked a couple of times due to deletion or move of documentation (FlexSwitch) or simply the lack of supported hardware. Whereas, in the span of a few weeks the situation was suddenly changing i.e. HW support added, new features included in the project.

### Flexswitch

We initially aimed to include FlexSwitch in the tested control-plane stacks. First, we attempted to test FlexSwitch's container-based test setup[1]. With that, we managed to perform basic Ansible configuration including IP addressing and OSPF. Secondly, we have built FlexSwitch from the source code committed to OPX project. However, we were unable to reproduce OSPF configuration which has worked in a container-based setup. This was mostly due to the fact that that FlexSwitch's version and setup method we used in the container setup differed from the one we have compiled. Due to time constraints, therefore, we considered FlexSwitch to be unfeasible to be used.

## 8.2 Feature test

### CLI usability

Generally, the CLI of SONiC was more consistent and appeared well-thought. It definitely was resembling the functionality known from commercial NOSes. OPX on the other hand offered rather cumbersome CLI experience. We believe that considering usability, CPS API should be the only way of configuring OPX device. However, that

[1] https://github.com/SnapRoute/dockerLab

would require a layer of extra code interacting with CPS and translating the configuration to CPS calls. Yet, additional configurations e.g. IP address of a management interface or Quagga configuration, would require separate means to configure.

### SONiC development focus

In the future release SONiC.201712 there are some important features to be supported such as VLAN trunking, MAC aging or port breakout. As the project itself focuses on creating NOS for a cloud datacenter fabric, it does not emphasize on a feature set one would expect from a general purpose network device. Therefore, it seems to be no active development in regard to features such as OSPF or Spanning-Tree Protocol support.

## 8.3 FIB installation latency accuracy

There were couple of weak points in the approach we took to measure the route installation latency. First, to explain the result we reviewed the configuration once more and noticed that a stock Quagga installation in SONiC is configured with Forwarding Plane Manager (FPM). This is an optional component designed specifically for the cases of forwarding plane being distinct than Linux kernel. This effectively provides a shortpath to NPU communication (Figure 13).

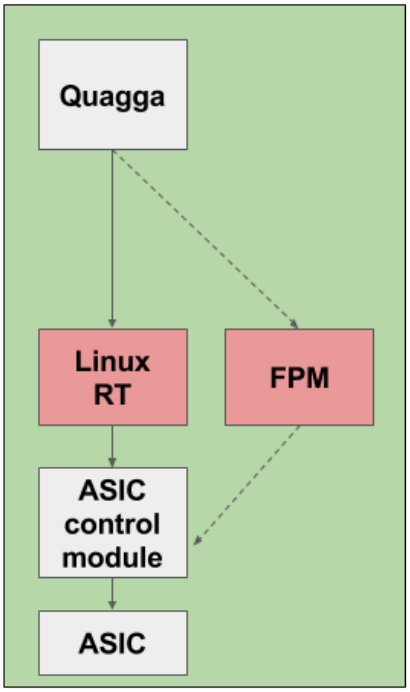


Figure 13: FPM component enables bypassing the kernel FIB and direct communication with NPU interface component

Secondly, both of the timestamps we recorded were gathered by parsing relevant log files and extracting the required information. We have not estimated what could be the accuracy of those timestamps and whether it is reliable to use them. Lastly, the t2 timestamp was obtained from two different sources: (1) OPX's CPS and (2) SONiC's SwSS logs. Although functionally similar, it is possible that those two components report successful route installation at different execution stages effectively falsifying the results.

# 9 Conclusions

First, we find it feasible to create an open-source white-label switching stack with both OPX and SONiC. When a NOS is combined with an open routing protocol software (e.g. Quagga, BIRD) it is possible to provide a basic set of L2 and L3 functionalities one would require from a data-center switch. Nonetheless, neither of evaluated NOS projects should be considered perfect. Looking at the CLI usability, we find SONiC to have a more unified approach to the NOS configuration than OPX. We believe that having a single configuration file, which in addition has automation-friendly structure is a more robust solution than OPX's approach. However, considering following feature tests, OPX covered the wider scope of functionalities (i.e. VLAN, STP) than SONiC.

Secondly, in the performed evaluation we encountered noticeable differences in regard to the FIB route installation latency between OPX+Quagga and SONiC+Quagga setups. The results suggest that message passing in SONiC operates significantly faster than in OPX. However, we find it crucial to adjust Quagga's configuration with FPM (and redoing the tests) before aiming to draw any further conclusions.

# 10 Future work

As both NOS projects are in ongoing development we believe it will be beneficial to reassess their new versions. Moreover, the test scenarios could be extended and complemented with new approaches (i.e. testing API, BGP protocol, commercial NOS interoperability). Lastly, it is interesting to evaluate non-Broadcom based devices e.g. Mellanox switches.

# 11 Acknowledgments

We would like to thank the following individuals:

- Ramon Semmekrot from Dell for providing us with open networking switches we used in our evaluation
- Bachelor student Mateusz Sadowski for his work on routing agents performance comparison
- Attilla de Groot from Cumulus Linux for the demo license of their NOS (used for the preliminary phase of this project)

# Bibliography

[1] Open Compute Project, Switch Abstraction Interface v0.9.1, (2014). https://github.com/opencomputeproject/SAI/blob/master/doc/SAI-v0.9.1.pdf.

[2] Armstrong, Joe, Making reliable distributed systems in the presence of software errors, PhD thesis, Mikroelektronik och informationsteknik, 2003.

[3] Arista Networks, Understanding EOS and Sysdb, (2016). https://github.com/aristanetworks/EosSdk/wiki/Understanding-EOS-and-Sysdb.

[4] S. Devireddy, Dell Networking OS10 Open Edition Administration and Programmability, 2016. http://en.community.dell.com/techcenter/extras/m/white_papers/20442852/download.

[5] Liu, Xin, SONiC: Features and Roadmap, (2017). https://github.com/Azure/SONiC/wiki/Features-and-Roadmap.