Exploring SONiC's True Potential

The Full Story of Building a Custom Functionality: Design & Walkthrough

- SONiC and Switch Abstraction Interface (SAI) key components and architecture
- Development execution description
- PoC tests performed in lab environment



codilime

Executive summary

The problem

The purpose of this ebook is to share our experience gained during our journey to use SONiC for a non-standard use case.

CodiLime's engineers checked how much time and labor this system would require to deliver a custom, natively unsupported, network functionality.

Use case description

For case purposes, it is assumed that GTP-U traffic is to be distributed by a SONiC switch among a set of endpoints. Packet distribution is based purely on the GTP-U TEID field value and no other action is required (meaning the switch's "distribution logic" only sets the next hop address). The endpoints' functionality may take different forms, like physical devices, VMs (as a VNF) or Kubernetes pods (as a CNF) - it does not impact the SONiC implementation.

Both endpoints have the same IP address for their loopback interface (which is used as the destination IP in packets carrying GTP-U). As such, the switch does not need to change the destination IP, only sets/chooses the appropriate next hop IP (there are multiple paths to the destination and the desired one is chosen based on the TEID field value).

The proposed approach

The GTP TEID field has a length of 32 bits (4 bytes). Depending on the number of endpoints, the actual number of bits taken into account can be reduced.

The considered use case assumes that there are two endpoints, each connected with a SONiC device using a specific connectivity subnet. For the scenario, the value of the last bit of TEID field (from GTP-U header) is enough to set the next hop for the packet.

The outcome

The conclusion of the PoC, although positive in terms of the outcome, shows that some steps can be more complex, but we proved that it is possible and worth our time and effort. While developing a custom feature, some dependencies have to be considered.

As SONiC and SAI's further development is proceeding, we will be more than happy to leverage new opportunities, effectively building other custom features.

Table of Contents

Executive summary 2
Introduction 4
What is SONiC – a solution overview 5
Why did we decide to test out SONiC? 5
SONiC and SAI - architecture and key components 6
Use case description 8
The implementation – what were our possible options? 13
Description of development plan 15
Development execution 16
Setup of development environment 20
The plan 22
The PoC execution steps 23
What may come next – summary and our conclusions 30
About the authors 32
About CodiLime 33

Introduction

Market analyses claim that in the future, networking will shift more and more towards so-called Open Networking. Open Networking relies mainly on hardware and software disaggregation, i.e. with open standard bare-metal equipment and a choice of independent NOS (Network Operating System). Such an approach fits with the main industry trends, like Software-Defined Networking, virtualization, containerization and of course, cloud computing.

To respond to these needs, our CodiLime's experts took a closer look at SONiC. They tested it inside out to check how much time and labor this system would require to deliver a custom, natively unsupported, network functionality for SONiC. Did SONiC meet the expectations? Take this journey with us and find the answer!

What is SONiC – a solution overview

SONiC, which stands for **Software for Open Networking in the Cloud**, is an open-source, Linux-based NOS dedicated to running network equipment. Originally developed by Microsoft and the Open Compute Project (OCP), and currently hosted by the Linux Foundation, SONiC is steadily gaining recognition in the networks and open-source communities as a driver of innovation for which industry reports forecast further rapid revenue growth in upcoming years.

The solution was primarily developed to equip Microsoft's own Azure data centers' switches, and proved its value in such a role. It was then open sourced and is now freely available on GitHub. Today's releases already offer out of the box a strong portfolio of network functions, which can be extended independently from the hardware vendors - and this is what our work aims to cover.

The ecosystem is already jaw-dropping, involving 60+ global software and hardware giants and will definitely grow further. On the development side, while the Linux Foundation primarily focuses on the software component, it has partnered with OCP to align with the hardware part, in particular the SAI specification (Switch Abstraction Interface), which is assumedly the key enabler to provide SONiC developers with a vendor-independent way of controlling forwarding elements like ASIC.

Why did we decide to test out SONiC?

Observing the recognition which SONiC has gained since its initial release in 2016, we at Codilime decided to conduct our own experiment and gain some hands-on experience in order to form our own opinion, and potentially seed a new competence in our portfolio. Our goal was to find out how difficult and labor- and time-intensive it would be to deliver a relatively uncomplicated, custom (natively unsupported) network functionality for SONiC.

After checking the portfolio of supported features in SONiC (at that time the latest release was SONiC.202205), we decided to design and build a proof of concept of a simple load balancing functionality, allowing to distribute packets based on the ISO OSI layer 4 payload (chosen fields from the payload of layer 4). As an example use case, we've chosen the GTP-U protocol (which is carried as a layer 4 payload) with the distribution to be based on the Tunnel Endpoint IDs (TEID) field value. GTP-U stands for GPRS Tunneling Protocol - User plane; it is one of the key protocols in the mobile networks of all generations, from 2G through 5G.

The example, even though quite specific, has the potential for relatively wide generalization, making our work useful for a larger set of use cases involving other upper layer protocols and various load balancing strategies. Our case uses GTP's TEID field to make load balancing decisions, but may also very well be based on a completely different field or fragment belonging to a completely different network unit. The real question we wanted to address was how tedious it would be to develop a feature, not necessarily this particular one.

SONiC and SAI - architecture and key components

The SONiC system has a modular architecture built on independent Docker containers interacting with each other through a central Redis database engine.

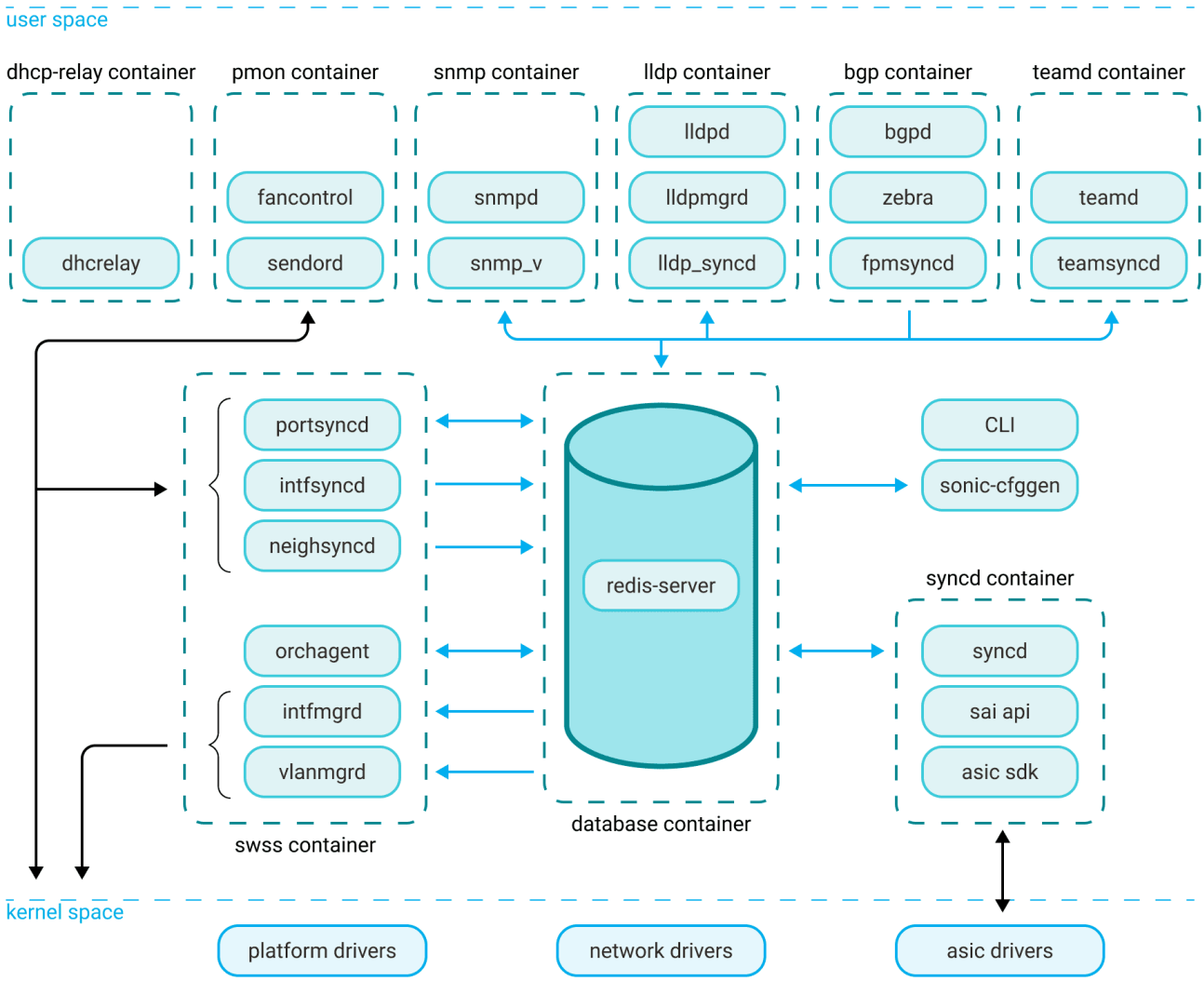


Fig. 1 SONiC system architecture, source

The main components include:

- Network applications, responsible for higher-level management of network functionalities; for example, BGP or SNMP.
- A central Redis container. This is used both as runtime storage and as a means of communication for other parts of the system. It consists of multiple databases, responsible for system configuration, system state, statistics etc. A unique database worth mentioning is the ASIC_DB, which is used for both steering and reflecting the underlying dataplane (ASIC) configuration.

- SwSS container. The Switch State Service (SwSS) is a collection of processes essential for communication among all SONiC modules. It coordinates communication between the network applications, the database and the system kernel. A unique component worth mentioning is the Orchestration Agent, responsible for interpreting the control configuration and setting the dataplane config accordingly.
- SyncD container. This is meant to provide synchronization between the ASIC database (ASIC_DB) representation and the switch's actual hardware/ASIC using SAI.

This is just a high-level overview of the parts we find important because we needed to actively deal with them in this project. For a detailed description of the architecture, please refer to the official SONiC Wiki page.

More about Switch Abstraction Interface (SAI)

The Switch Abstraction Interface is at the heart of the SONiC dataplane configuration. It's an open-source set of tools for interfacing with general vendor-agnostic switch functionalities. It's used by SONiC as an isolation layer between the generic SONiC functionalities and the hardware-specific platform details.

Practically speaking from the user's (developer's) perspective, **SAI is a collection of definitions** of various data structures and function signatures used to represent and configure generic switch capabilities, put in C header files. There are also auto-generated tools for SAI attribute validation, serialization and more in the meta directory.

It's the responsibility of the ASIC vendor to provide a specific **SAI implementation** for their hardware, in order to support a given SAI version:

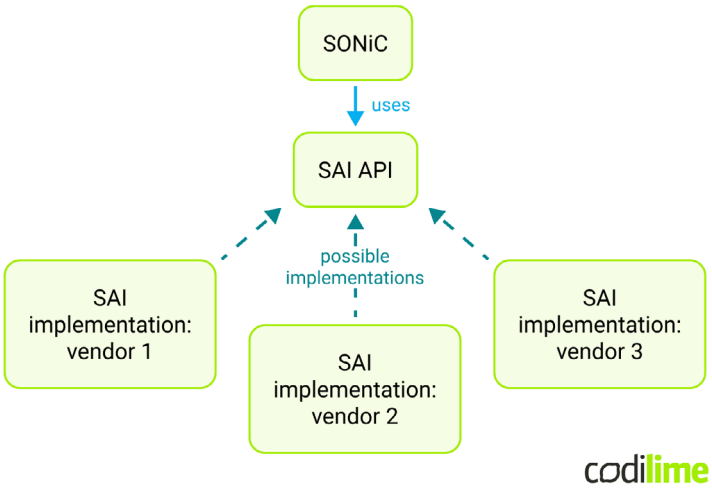


Fig. 2 SONiC & SAI relationships

SAI is versioned. It is divided into features with mandatory and optional support from the implementation provider (ASIC vendor). It's worth mentioning that a SAI implementation provider is allowed to add custom SAI features and document that functionality for the user.

Given that the SAI specification has to be implemented by the hardware chip vendors, adding changes to SAI isn't really an option in the case of small teams like ours. It's much easier to experiment with SONiC control plane features using the existing SAI, rather than customizing SAI itself.

Use case description

In our use case it is assumed that GTP-U traffic is distributed by a SONiC switch among a set of endpoints. Packet distribution is based purely on the GTP-U TEID field value and no other action is required (meaning the switch's "distribution logic" only sets the next hop address). The endpoints' functionality may take different forms, like physical devices, VMs (as a VNF - Virtual Network Function) or Kubernetes pods (as a CNF - Cloud-Native Network Function) - it does not impact the SONiC implementation.

Our example use case (in its simplest form with only two endpoints) is presented in the picture below:

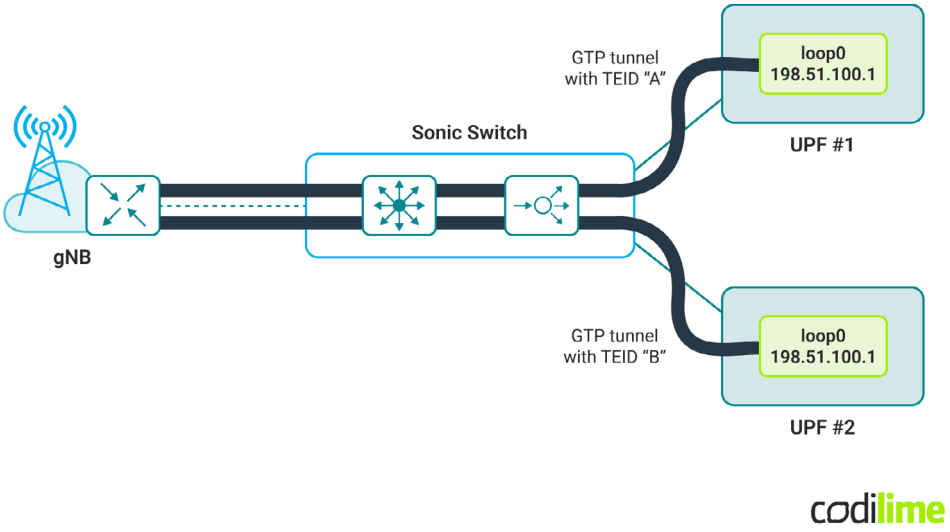


Fig. 3 SONiC use case graphic description

In this simple case, both endpoints have the same IP address for their loopback interface (which is used as the destination IP in packets carrying GTP-U). As such, the switch does not need to change the destination IP, only sets/chooses the appropriate next hop IP (one could say there are multiple paths to the destination and the desired one is chosen based on the TEID field value).

What is worth noting is that such load balancing must be predictable in the sense that it must be possible to predict/calculate (based on the TEID value) to which endpoint a packet will be sent - i.e. the endpoint providing the GTP-U tunnel termination functionally must be configured before receiving the GTP-U packet. This is done by a control plane function that must know the endpoint responsible for the particular TEID.

Before we dig into the solution, let's focus for a moment on the GTP-U protocol. It is used in mobile networks and according to the 3GPP specification:

"The GTP-U protocol entity provides packet transmission and reception services to user plane entities in the RNC, SGSN, GGSN, eNodeB, SGW, ePDG, PGW, TWAN, MME, gNB, N3IWF, and UPF. The GTP-U protocol entity receives traffic from a number of GTP-U tunnel endpoints and transmits traffic to a number of GTP-U tunnel endpoints. There is a GTP-U protocol entity per IP address.

The TEID in the GTP-U header is used to de-multiplex traffic incoming from remote tunnel endpoints so that it is delivered to the user plane entities in a way that allows multiplexing of different users, different packet protocols and different QoS levels. Therefore no two remote GTP-U endpoints shall send traffic to a GTP-U protocol entity using the same TEID value except for data forwarding as part of mobility procedures."

In other words GTP-U is used for carrying user data within the mobile core networks and between the radio access networks and the core networks. GTP-U is a part of the GTP protocol, which additionally contains GTP-C and GTP'. GTP-C is responsible for carrying the signaling messages (between Gateway GPRS Support Nodes and Serving GPRS Support Nodes) while GTP' handles charging data.

GTP can be carried in UDP or TCP datagrams, version 1 is UDP only and hereafter we assume usage of version 1. IANA assigned port numbers are: 2123 for GTP-C, 2152 for GTP-U and 3386 for GTP'.

The protocol stack for GTP-U packets (carried as UDP payload) is shown in the picture below:

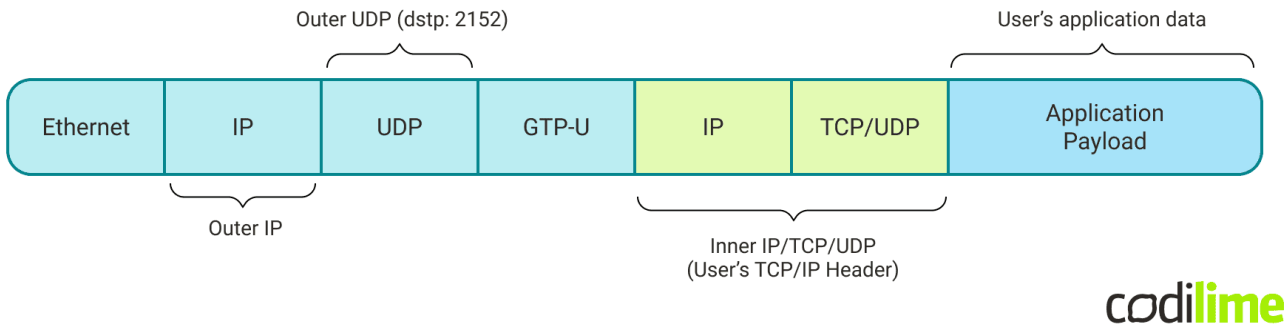


Fig. 4 The protocol stack for GTP-U packets

The implementation logic

The intended functionality is to extract TEID field values from GTP-U packets. To be able to do that, first it has to identify the GTP-U packets. Looking at the protocol stack, GTP-U packet identification can be done by filtering for IP packets carrying UDP datagrams with destination port 2152.

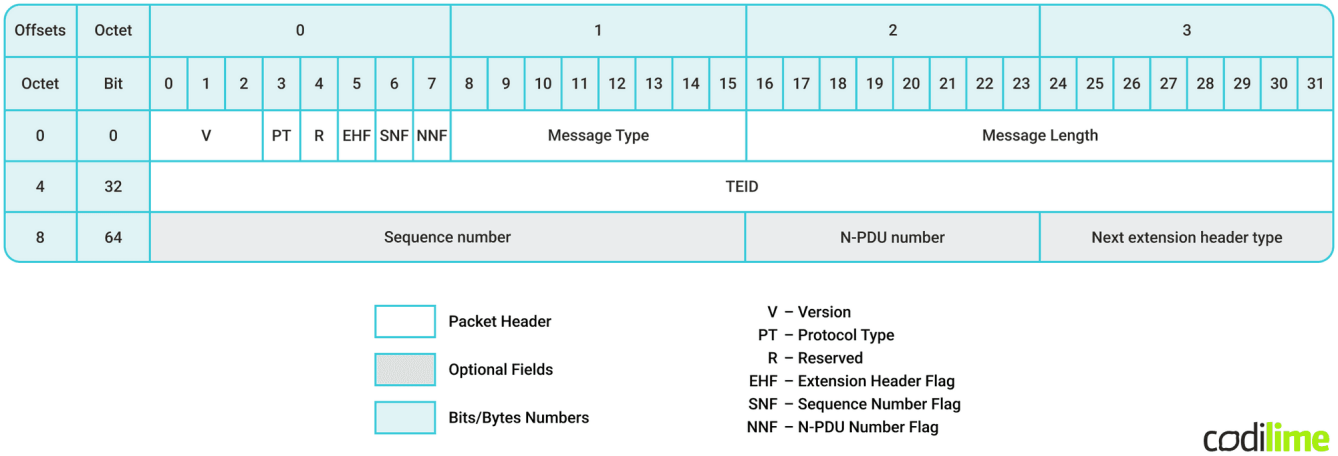


Fig. 5 GTPv1 Header

Looking at the GTP v1 packet header, one can see that the TEID value is held between the bits number 32 and 63 of the header (which itself is the payload of the UDP packet). In terms of bytes, the TEID field starts at the 4th byte of the GTP header and is 4 bytes long.

Choosing the right approach

In general, any functionality which is supposed to be performed by an ASIC chip, must be supported by SAI (included in SAI specification) and implemented by the ASIC vendor of choice (vendors may implement different versions of SAI spec). Because of that, our first task was to look for available features in SAI which could fulfill the needs of our use case. GTP protocol as such is not supported (so it is not possible to get a TEID value in a simple, built-in way; e.g. referencing this field as is done for TCP destination port number). On the other hand, SAI supports the concept of a User-Defined Field (UDF), which allows the extraction of arbitrary field(s) from selected packets. UDF can be used in the context of:

- hash value calculation - e.g. hash for ECMP or LAG,
- Access Control Lists (ACL) - with the extracted field value as qualifier.

At first glance, the use case seems to fit into the ECMP (Equal-Cost MultiPath) scenario with hash calculation based on TEID (extracted by UDF) since, from SONiC's perspective (working in L3 mode), there are multiple paths to the destination and it has to forward to one of them. However, there is a second requirement in the use case - predictability in terms of which endpoint is selected.

This selection depends on the algorithm used for hash calculation. At the time of writing, the SAI specification allows a choice of hash algorithm for the switch from the following (source: SAI Switch Interface definition):

```
/**
 * @brief Attribute data for #SAI_SWITCH_ATTR_ECMP_DEFAULT_HASH_ALGORITHM
 * and #SAI_SWITCH_ATTR_LAG_DEFAULT_HASH_ALGORITHM
 */
typedef enum _sai_hash_algorithm_t
{
    /** CRC based hash algorithm */
    SAI_HASH_ALGORITHM_CRC = 0,

    /** XOR-based hash algorithm */
    SAI_HASH_ALGORITHM_XOR = 1,

    /** Random-based hash algorithm */
    SAI_HASH_ALGORITHM_RANDOM = 2,

    /** Lower 16-bits of CRC32 based hash algorithm */
    SAI_HASH_ALGORITHM_CRC_32LO = 3,

    /** Higher 16-bits of CRC32-based hash algorithm */
    SAI_HASH_ALGORITHM_CRC_32HI = 4,

    /** CRC using CCITT polynomial based hash algorithm */
    SAI_HASH_ALGORITHM_CRC_CCITT = 5,

    /** Combination of CRC and XOR based hash algorithm */
    SAI_HASH_ALGORITHM_CRC_XOR = 6,

} sai_hash_algorithm_t;
```

Though the algorithms are predefined, they still have parameters which are chosen by the vendor implementing SAI. In consequence, even when the same algorithm is used for the packet, the resulting hash can differ on different platforms. In a particular switch or even in a given ASIC chip this should be consistent (the hash for a given packet is always the same) though it would not be possible to calculate it by hand, unless the implementation details are known (which is unlikely as most ASIC vendors do not publish their SAI implementation, which includes hash functions). Additionally, vendors could change or update the algorithm's parameters leading to different hash values after the SAI layer is updated on the switch. While for ECMP or LAG this is perfectly acceptable behavior, it could be problematic in the considered use case.

Fortunately, the second option - making use of ACLs (Access Control Lists) with UDF is better suited for setting the target endpoint based on the TEID value. In this approach, UDF is responsible for TEID value extraction and ACL for packet filtering (IP packets carrying GTP-U protocol) and forwarding to specific endpoint. This approach is able to give full control over specifying the target endpoint for a given packet (based on TEID value), thus it fulfills the requirement of the use case.

As a result, we have chosen to utilize the concept of UDF-based ACLs.

The solution's assumptions and specifications

The GTP TEID field has a length of 32 bits (4 bytes). Depending on the number of endpoints, the actual number of bits taken into account can be reduced. For example, one bit is enough to load-balance between two endpoints, two bits to four endpoints, etc. Generally, one can use the bits of least significance. However this assumes that TEID values are assigned uniformly (meaning it is equally probable for each value to be used, the assignment is not based on chosen ranges, etc.)

Note: a number of endpoints (to which load balancing is done) being a power of 2 (2^n) makes the distribution easier to implement.

The considered use case assumes that there are two endpoints, each connected with a SONiC device using a specific connectivity subnet. For the scenario, the value of the last bit of TEID field (from GTP-U header) is enough to set the next hop for the packet; i.e.:

- if last bit of TEID == 0 -> set next hop IP: IP_ADDRESS_A
- if last bit of TEID == 1 -> set next hop IP: IP_ADDRESS_B

In reality one needs to ensure that only the intended GTP-U packets are matched for redirection (match based on inbound port, IP Protocol type = UDP, UDP destination port = 2152, additionally it might be required to match IP source and/or destination address). Such processing, for a given packet, could look like this (symbol description):

```
IF inbound_port != Port0 OR ip_protocol != udp OR udp_dst_port != 2152
    action(back_to_standard_processing)
ELSE IF last_bit_of_teid_field == 0
    action(set_next_hop_ip: IP_ADDRESS_A)
ELSE IF last_bit_of_teid_field == 1
    action(set_next_hop_ip: IP_ADDRESS_B)
```

Note: the switch port number and IP addresses are examples (values depend on the setup configuration). The IP Protocol, UDP destination port and TEID last bit values are use case-specific.

ACL handling by SAI and SONiC

The SAI ACL Table defines the stages at which ACL rules are performed in the SAI pipeline:

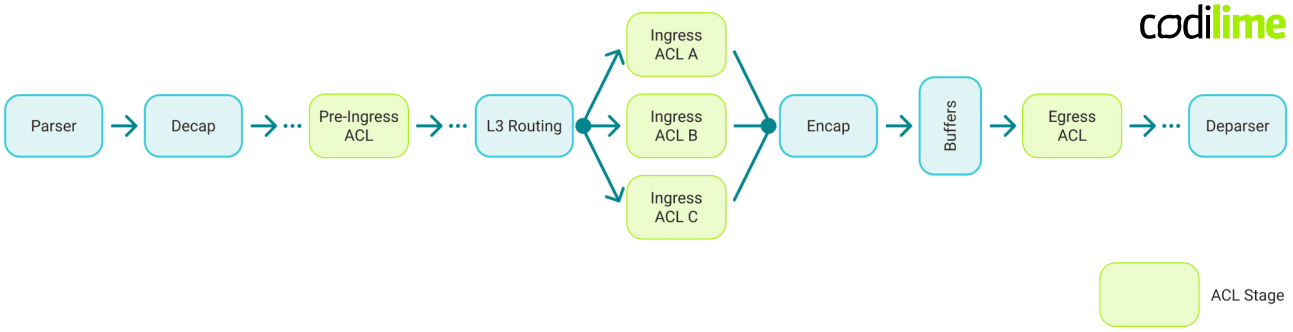


Fig. 6 SAI pipeline

For each stage there is a defined set of actions which can be performed. In the context of the load balancing use case, one can use:

- Pre-Ingress ACL stage with an action “set VRF” (packet is set to a particular VRF based on ACL match rules).
- Ingress ACL stage with an action “REDIRECT” (packet is redirected to an interface, a next hop, or a next hop group, based on ACL match rules).

As the SONiC project currently only supports the Ingress and Egress ACL stages - the Ingress stage has been selected for this use case.

The implementation – what were our possible options?

In general, there are two ways: to find and configure or adapt a suitable existing solution, or to develop it from scratch. It is reasonable to use available features (whenever possible) and if necessary to add only the missing parts. Initial research led us to two possible solutions: P4 Runtime (part of PINS) or adding support for UDF-based ACLto the SONiC code.

First approach - SONiC PINS and P4 Runtime

PINS (P4 Integrated Network Stack) is a relatively new initiative, (at the time of writing) which aims at bringing SDN capabilities and P4 programmability to traditional routing devices. It uses P4 Runtime (P4RT) as a new control plane interface. The P4 Runtime API is a control plane specification for controlling the data plane elements (in our case in SONiC switch) defined or described by a P4 program.

PINS was first introduced as part of the SONiC release 202111. Among other aspects, it added the P4 Runtime application to SONiC. The role of this application is to listen to a remote SDN controller (which communicates over the P4RT protocol) and create entries in SONiC databases.

After checking the documentation and usage examples, it seemed that UDF-based ACLs should be supported (at least to some extent), which would essentially fill the SONiC gap in the context of our use case. So, we decided to give it a try.

What we can say today is that employing P4RT for a use case like ours proved to be neither fast nor effortless. The entry threshold turned out to be relatively high as we encountered some challenges. For example, the default options for the SONiC build do not include PINS and in consequence the SONiC upstream builds do not contain PINS components either. It required a little effort to build our own virtual switch with PINS support. Another issue was the lack of working examples, as the ones found in the SONiC PINS repository required some modifications to work.

From our perspective, the official PINS tutorial was oriented around PINS usage in a physical switch. Given that, there was no working p4info file for us to use with the software switch in the first place. Generally speaking, a software switch would be a better fit (more robust, replicable) for an entry-level tutorial, which would allow beginners to “learn by doing” and then adapt this knowledge to their use case. So, we wish there was an “out-of-the-box” working configuration of a virtual switch available, as this would have greatly supported our work from day one.

Given the limited time frame we had for the project and the challenges we faced, we decided not to use PINS in our solution.

Second approach - developing a tailored solution

Consequently, the development of the required features was the only option left (meaning we had to implement support for UDF and UDF-based ACLs). In a nutshell, in order to instruct a SONiC switch to perform the desired logic, we need to feed it with the desired configuration through CONFIG_DB (Redis database), from where it is pulled by the Orchestration Agent (Orchagent), processed and pushed to ASIC_DB (another Redis database). Finally, it is handed over to the switch’s ASIC by SyncD and SAI API.

To achieve that, we modified Orchestration Agent (i.e. we added support to create UDF objects, then ACL tables and ACL rules were modified to support UDF as a qualifier). Another modified element was *saimetadata* library (it enriches the SAI API with runtime metadata). As a result, additional changes were required in components dependent on saimetadata: Orchagent and SyncD.

From the very beginning, our focus was on the use case in order to keep things simple and efficiently utilize the time allotted for the project. As a result, our implementation is not intended for generic usage (e.g. UDF implementation is intended to be used with ACLs only and as such does not support hashes).

Description of development plan

The intended solution was to extend ACLs with UDFs. How was that approached?

Having to deal with ACLs, the first place we checked was the doc/acl directory, specifically ACL High Level Design. It showcases the flow diagram of the creation of ACL objects in terms of the logical architecture of the control plane (figure modified):

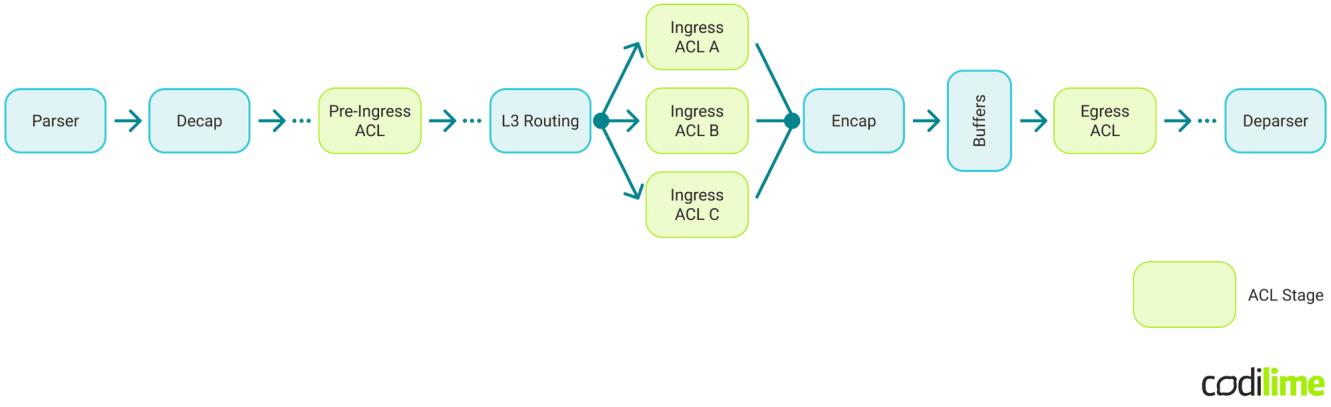


Fig. 7 Flow diagram of the creation of ACL objects (source: SONiC GitHub)

The logical architecture does not map 1-to-1 to the physical software architecture. In terms of the processes involved, ACL creation needs to engage the following physical components:

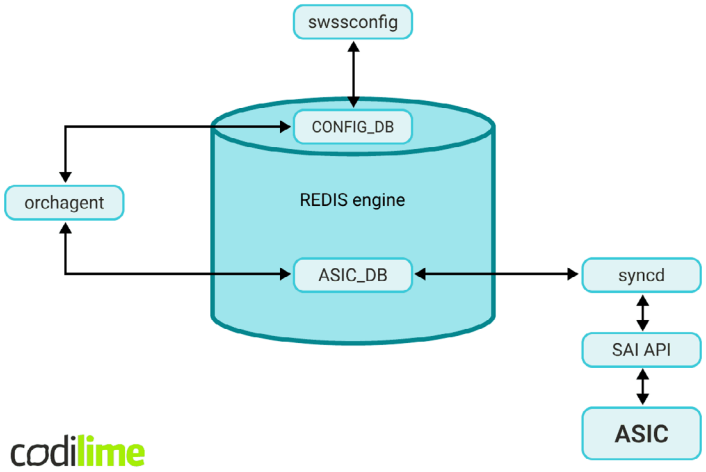


Fig. 8 Physical components involved in ACL creation

Our plan was to update the control flow and verify that the ACL configuration based on user-defined fields is handled correctly throughout the entire stack.

SAI specification

First, based on the SAI documentation UDF-based ACL and the actual SAI definitions, we've specified the proper SAI API usage for our use case.

SAI, for ACL and UDF mechanisms, defines several object types which must be used. Those objects and their relationships are presented in the picture below:

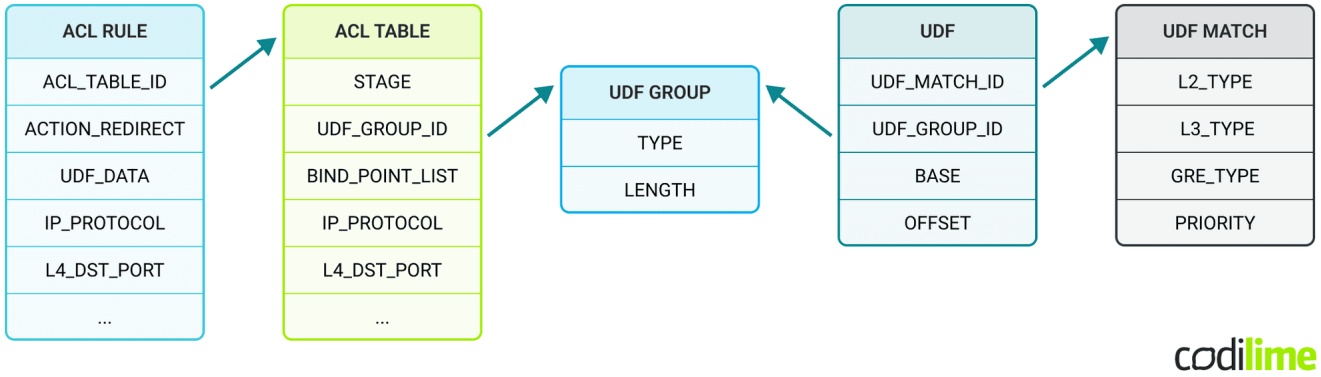


Fig. 9 SAI objects' relationships for ACL and UDF mechanisms

Note: for ACL RULE and ACL TABLE objects not all attributes are shown (only those important / used for the use case)

In general, a UDF Group may contain multiple UDF objects (in fact a UDF object references the UDF Group to which it belongs). Multiple UDF objects may point to the same or different UDF Match objects (when a packet goes through a UDF group object, one UDF is selected based on its UDF match object). For the purposes of our use case, single UDF Match and UDF objects are enough.

The initial research has shown that most of the development work will be done in Orchestration Agent (aka Orchagent). It's the component responsible for interpreting the user configuration and calling the SAI API (through a database proxy). We needed to implement two kinds of change in orchagent:

1. add new logic for creating UDF objects with SAI,
2. modify the existing logic to use the UDF in ACL tables and ACL rules with SAI.

Development execution

We could separate the two phases of development:

1. Core logic changes in Orchagent - implementing the core logic for configuring UDF with SAI API and connecting the UDFs with ACLs.
2. Integrating the Orchagent changes through SONiC - fixing any follow-up bugs or problems.

Phase 1: Core logic changes - Orchagent

Knowing what new data is required by SAI from the user, we've updated the Configuration Database schema, which serves as an input for Orchestration Agent.

Having specified the required input and expected output for Orchagent, it was enough to implement the test case for configuring UDF-based ACLs using the existing Orchagent unittest infrastructure:

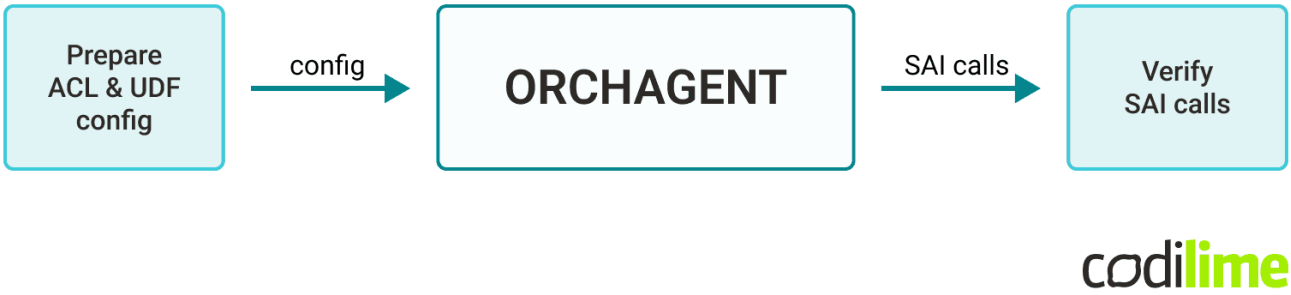


Fig. 10 Orchagent unit test scenario visualization

Finally, using the test-first approach, we implemented and verified the required logic in Orchagent. Looking at the ACL flow diagram, the central piece was working as expected:

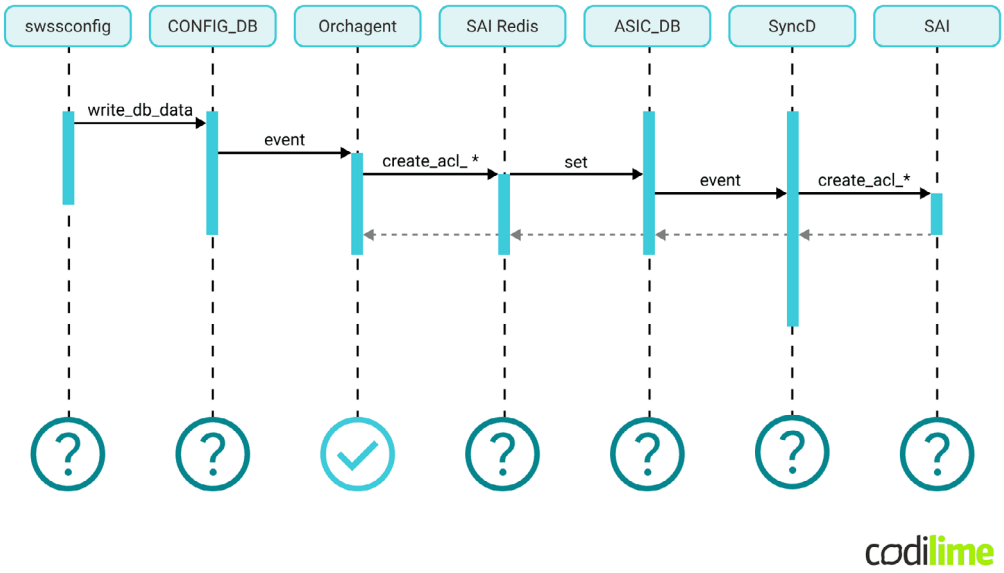


Fig. 11 ACL configuration phase 1 work-in-progress

Phase 2: Breaking further through this jungle (watch out for the bugs)

At that point, we were not in possession of a physical switch with SONiC installed and were thus limited by the capabilities of the SONiC virtual (software) switch. We performed multiple trials, but could not get the software switch to respect the ACL configurations. That made it impossible for us to test ACLs on the data plane.

We could only verify if the control plane worked as expected. A valid approach to test that in SONiC is to check the ASIC_DB database, which reflects the current state of the underlying SAI-compliant ASIC.

The scenario was simple:

- pass a simple UDF-based ACL config to swssconfig,
- verify that the content of ASIC_DB Redis database resembles our input.

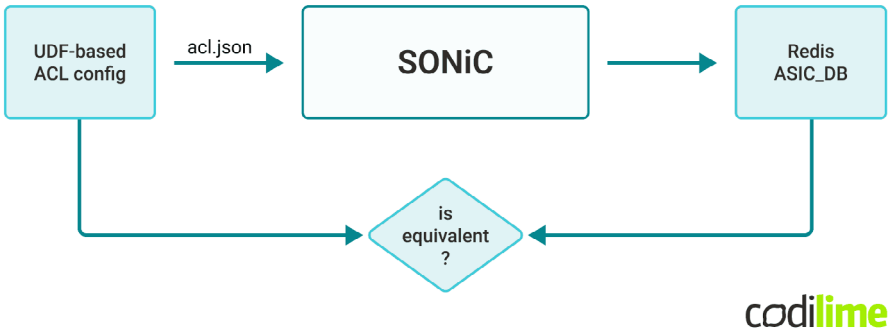


Fig. 12 The principle of end-2-end test scenario

The first results were negative. The ASIC_DB database showed no valid UDF-based ACLs configured. Orchagent syslog analysis gave us the reason.

There is a library in SONiC (saimetadata) responsible for enriching SAI API with additional runtime metadata. We configured a SAI object not used by SONiC before and some part of the SAI metadata logic for that object was not implemented yet.

Moreover, there are two components dependent on saimetadata: Orchagent (via the SAI Redis proxy) and SyncD (responsible for interfacing with the vendor ASIC according to the state of ASIC_DB). So now we knew that both Orchagent and SyncD needed a fix.

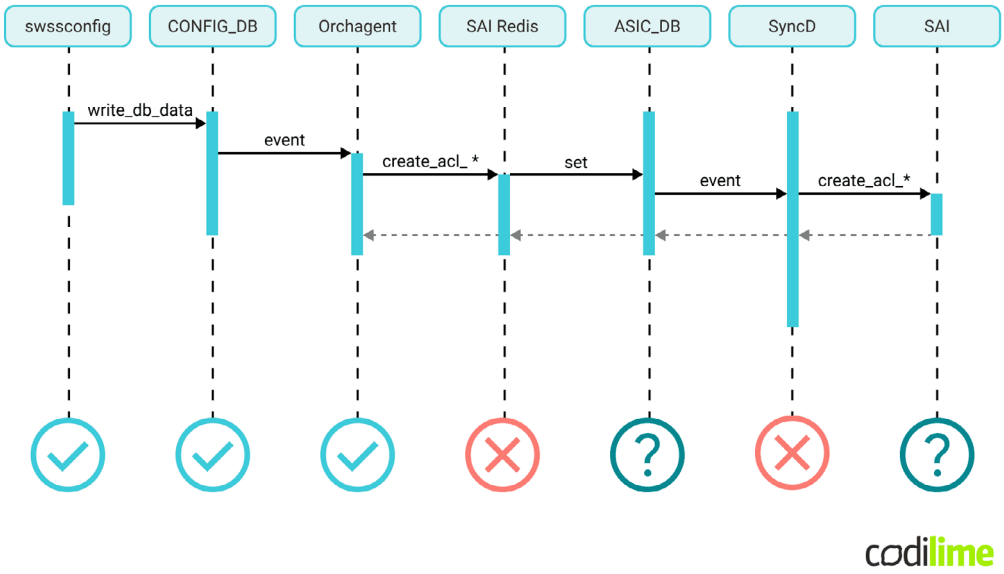


Fig. 13 ACL configuration phase 2 work-in-progress

The saimetadata fix required adding missing validation of the data and turned out to be rather simple. We rebuilt and redeployed the swss and syncd containers.

At that point, the control plane test was successful: a UDF-based ACL configuration was correctly digested by the system, which could be verified by the ASIC state represented in ASIC_DB.

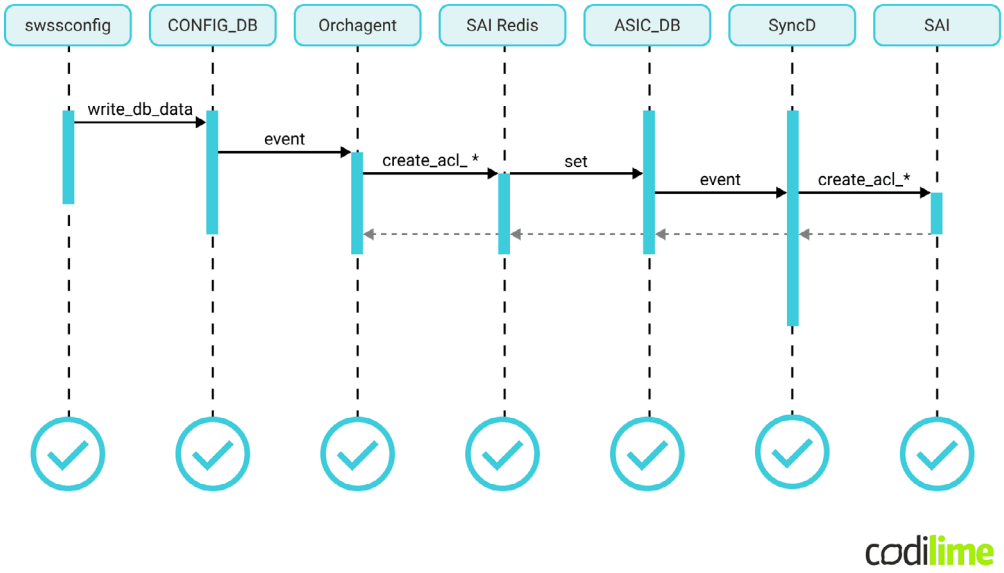


Fig. 14 ACL+UDF successful configuration applied

Setup of development environment

Now we would like to share some of our experience with development setup.

Because of the large number and variety of components in the SONiC system, the build system is rather complicated. There is a convenient Makefile available for the user. The actual build of targets, i.e. .deb packages, Docker images and binary images, is dockerized in multiple layers of containers and governed by autotools, aka GNU Build System. That gives a useful separation between the host machine and the build process, resulting in a robust build toolchain working out-of-the-box.

That however, does not play well with the smart modern IDEs used for development. Accurate index search, test framework integration, integrated debugger, code completion, and compilation errors shown in the editor before the code is compiled - all these increase one's productivity, but require the development tools to be fully aligned with the build process.

Remote development, but hosted locally

Our use case required most of the work to be performed within the Orchagent component, written in C++. Practically speaking, the simplest setup for a C++ project would be to have the dependencies and build tools natively on the host system, alongside the IDE (CLion by JetBrains in our case):

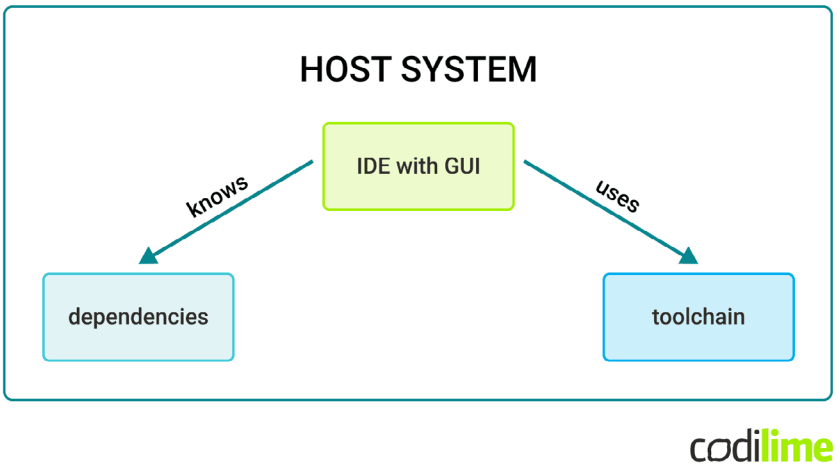


Fig. 15 Regular IDE development overview

Given the system dependencies and package versions required, it is not possible to build Orchagent natively on a regular Ubuntu machine using the preferred IDE in a simple manner.

To work around that problem we used a tool called JetBrains Gateway. Orchagent build is configured inside a Debian Docker container hosted locally, with all the specific required .deb packages installed by hand. Then, the JetBrains client connects via ssh to the Docker builder instance and runs a CLion backend there. The host system is just used for rendering and interacting with the IDE frontend. The Gateway supports any ssh-enabled system, but hosting a Docker instance locally was just the most convenient solution:

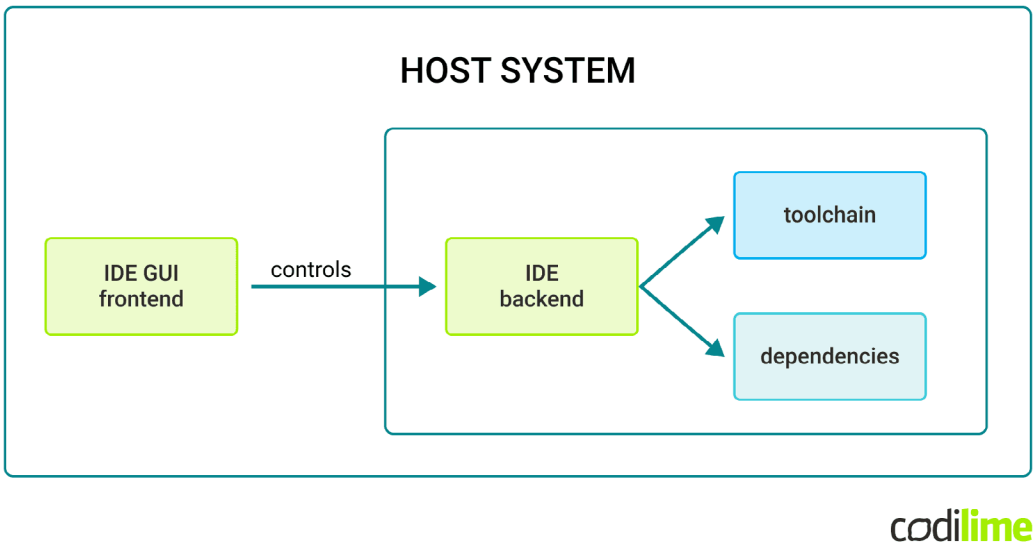


Fig. 16 Remote IDE development overview

Judging from the information we got from the SONiC community via the sonicproject Google Group, such an approach does not seem to be widespread among SONiC developers.

There are of course alternatives to the JetBrains Gateway. For example, an analogous approach is available for VS Code in the Visual Studio Code Remote Development extension.

Lessons learned - developer's perspective

- This specific project required flexibility and trial-and-error methods. We never knew if we were chasing a dead end or a silver bullet. One can never be certain about the time estimates for research tasks.
- The SONiC control plane is divided into functional containers. The data plane is abstracted behind the SAI layer. That results in a flexible and relatively easy-to-learn architecture, open for extension.

- From the developer's perspective, SONiC has a rather complicated build system with knowledge stored in multiple spots:
  a. sonic-buildimage README
  b. sonic-buildimage buildsystem README
  c. SONiC build system documentation
- "Remote" development on a local container can be a lifesaver when working on projects with strict or complicated dependency requirements.

The plan

In order to conduct this PoC we needed to:

- Choose an appropriate technique, which is reliable and can be expected to give consistent results across all potential deployments. Its job is to be able to look into the payload of a UDP datagram, extract the TEID field and distribute (forward) these datagrams to the appropriate endpoints – user defined fields (UDF) based ACLs were chosen here.
- Check what needed to be implemented and where:
  a. Modification of two SONiC modules: Orchagent and SyncD.
  b. The modification added the logic necessary for them to handle UDFs and UDF - based ACLs.
  c. The modified modules needed to be installed on the test SONiC switch, substituting the original.
- Choose a hardware platform able to run the test scenario in order to conclude the PoC.
- Propose and build the lab setup.
- Run the test to verify proper distribution.

The PoC execution steps

Step 1: Choosing the hardware platform

As explained, ACL tables are not being effectively programmed into the SONiC virtual switch (ACLs are inserted into the SONiC's internal database, but the data plane is not programmed to reflect that ACL configuration). In order to finalize and test our proof of concept, it was necessary to acquire a real piece of hardware capable of running SONiC.

When looking for a candidate, first we checked the list of SONiC-supported platforms. The next step was to check the specific version of SAI (Switch Abstraction Interface) a particular ASIC vendor supports in a specific chipset - and that turned out to be surprisingly hard to figure out. This was essential since our development depended on SAI support of specific features i.e. UDF and UDF-based ACLs (earlier versions of SAI did not yet define/support UDFs and UDF-based ACLs were added even later). Even particular SAI version support is not enough as ASIC vendors may implement a specific SAI version but with some exceptions (e.g. a feature does not support a particular parameter).

In short, this is definitely an important factor you need to consider when choosing your hardware. There is no "one SAI fits all" principle here.

Considering those factors, we decided to employ the Mellanox/NVIDIA Spectrum SN2010 switch for our proof of concept. Mellanox/NVIDIA publishes its SAI implementation for Spectrum ASICs and it supports SAI version 1.10 (the latest at that time, although with some limitations).

Step 2: building for the chosen hardware platform

This is where the SONiC container-based architecture shines. Even though we implemented a new SONiC functionality, there was no need to build the whole system from scratch.

SONiC images (and other build artifacts) for various platforms are publicly available in the SONiC Image Azure Pipelines. Following the SONiC User Manual, we downloaded a regular SONiC image for the Mellanox platform and installed it on our switch.

Having a regular SONiC system in place, we needed to update the two services (docker images) that contained the changes we prepared in the development phase:

- Orchestration Agent, aka. **orchagent**
- Synchronization Daemon, aka. **syncd**

The SONiC build system requires specifying the target ASIC platform. We used a VS (Virtual Switch) during the development phase, but then we needed images for the physical Mellanox switch. The docker images for Mellanox were built without problems, at the first attempt.

Then we just uploaded the updated images to the switch's Docker daemon, removed the old images, rebooted the switch, and voilà. The updated services were up and responsive. Based on that experience, we can agree that SONiC keeps its promise of being modular in terms of deployment.

Step 3: Testing in the lab environment

To test our implementation on the data plane level, i.e. to check if sample GTP-U packets are load balanced according to the configured rules, we built a simple setup emulating our use case scenario, consisting of:

- Mellanox/NVIDIA Spectrum SN2010 switch with SONiC NOS (upstream build with changed swss and syncd components).
- A BMS (Bare Metal Server) running Linux (Ubuntu 20.04), connected to the switch by three links.

A base configuration (allowing to forward packets from a traffic generator to two traffic endpoints) for the SONiC switch was simple: IP addresses assigned to interfaces connected to the server and a routing entry for the target IP address. Next, we added the configuration for UDFs (UDF Match, UDF Group and UDF) as well as for ACLs (ACL Table and ACL rules). As a result, the switch was expected to forward GTP-U packets to traffic endpoint No.1 if the TEID field's last bit was equal to 0 or to traffic endpoint No.2 if the TEID field's last bit was equal to 1. This is depicted in the figure below:

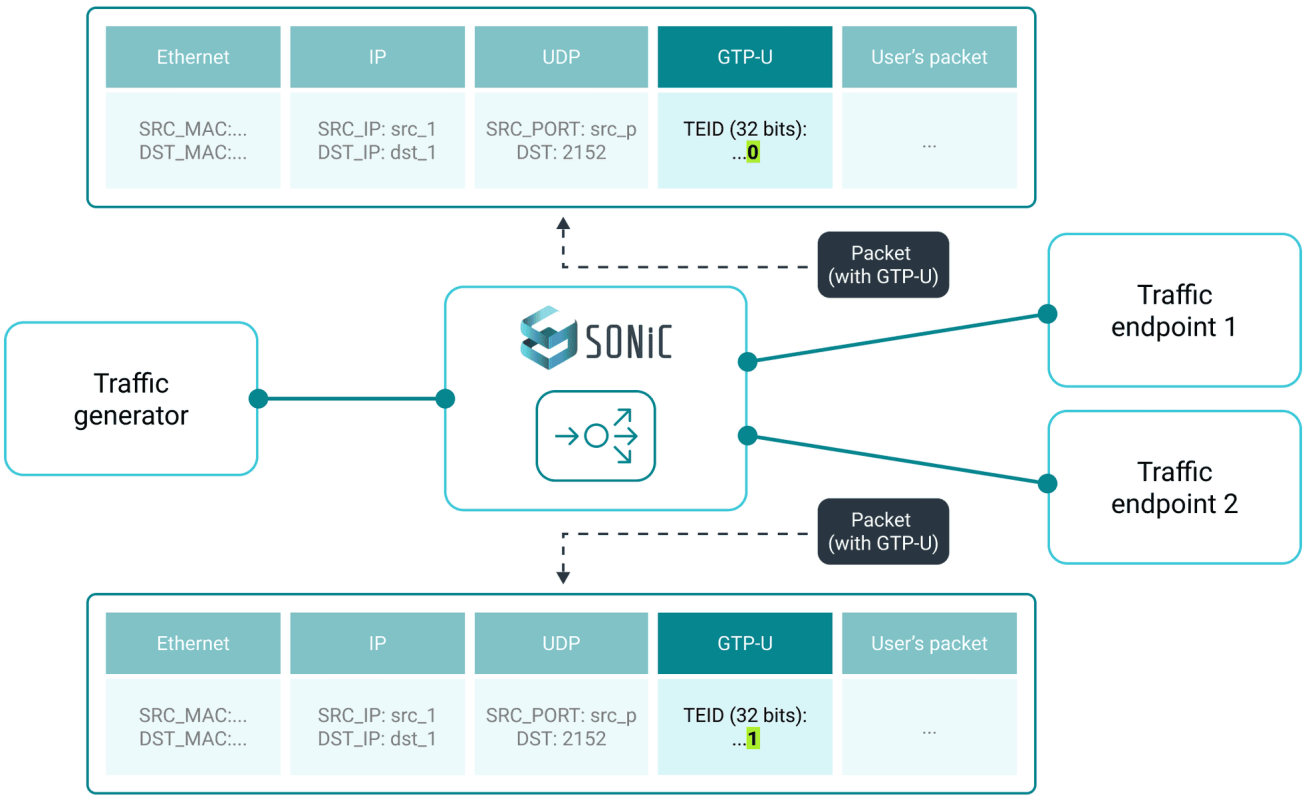
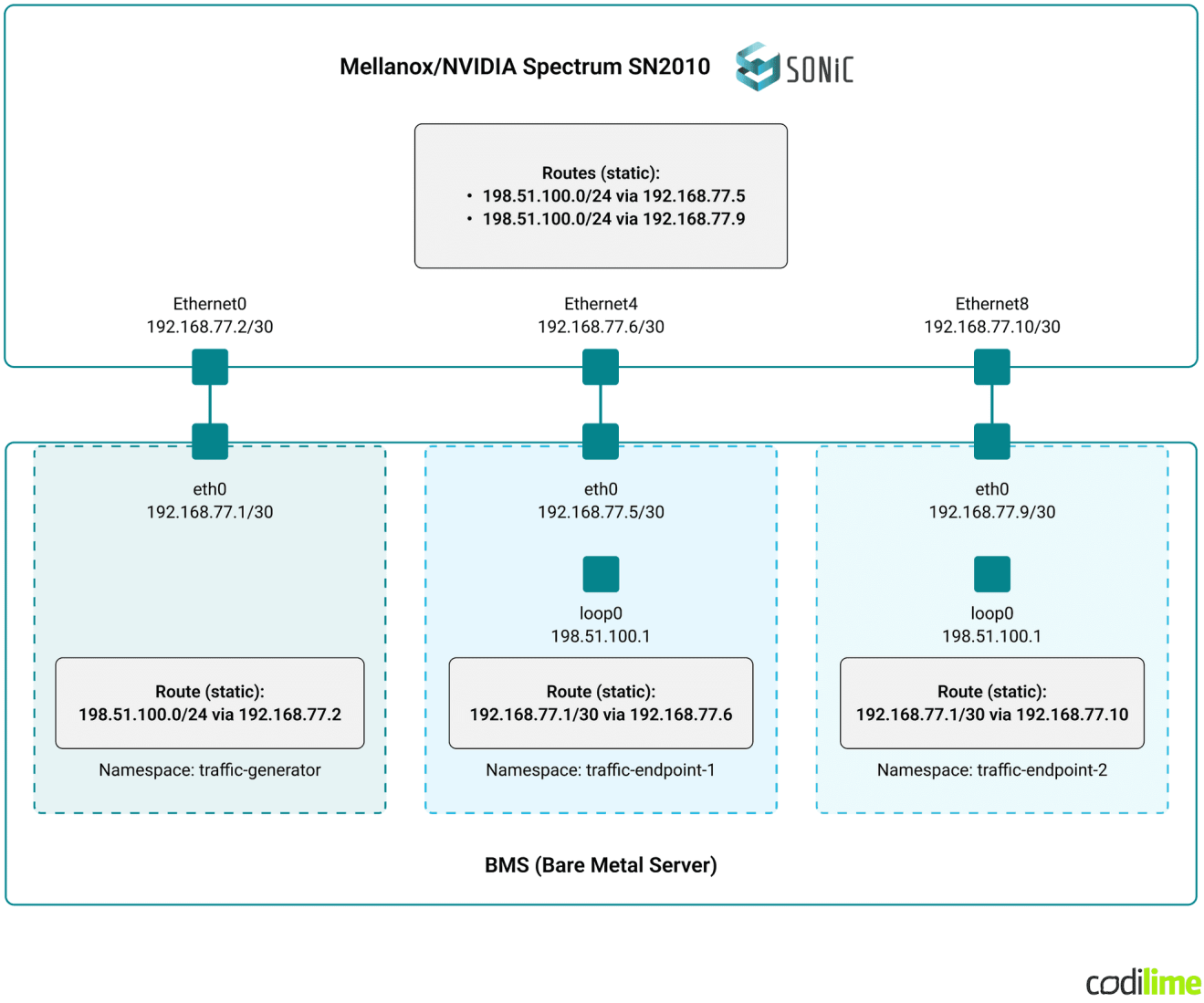


Fig. 17 Lab setup with SONiC switch configured to forward GTP-U packet based on TEID field

Sample GTP-U packets have the same source and destination IPs, and the same UDP ports (as is often the case in real-life implementations) but they differ by the GTP-U TEID field values and based on that are forwarded to the desired endpoint.

BMS with Ubuntu was used to simulate GTP-U traffic source (e.g. 5G gNB aka gNodeB), depicted as traffic generator, and endpoints (e.g. 5G UPFs - User Plane Function). Linux Network Namespaces were employed to separate the traffic generator and the endpoints with relevant physical network interfaces being moved to those namespaces (so that IP addresses assigned in namespaces were not directly reachable from other namespaces nor from the host OS).

The connectivity between the switch and the BMS as well as layer 3 configuration (IP addressing and routing entries) is shown in the diagram below:



codilime

Fig. 18 Layer 3 configuration in lab setup

The Python Scapy library was used to prepare a simple GTP-U packet generator to be used on the source side as well as to capture (sniff) and count the incoming packets in the endpoints' namespaces.

A sample GTP-U packet, generated using the Python Scapy library, captured at a traffic endpoint's interface and displayed by Wireshark looks like this:

```
▸ Frame 1: 99 bytes on wire (792 bits), 99 bytes captured (792 bits)
▸ Ethernet II, Src: 2a:0e:a4:88:8e:08 (2a:0e:a4:88:8e:08), Dst: 1e:6a:38:06:18:c7 (1e:6a:38:06:18:c7
▸ Internet Protocol Version 4, Src: 192.168.77.1, Dst: 198.51.100.1
▸ User Datagram Protocol, Src Port: 33333, Dst Port: 2152
▾ GPRS Tunneling Protocol
  ▾ Flags: 0x30
      001. .... = Version: GTP release 99 version (1)
      ...1 .... = Protocol type: GTP (1)
      .... 0... = Reserved: 0
      .... .0.. = Is Next Extension Header present?: No
      .... ..0. = Is Sequence Number present?: No
      .... ...0 = Is N-PDU number present?: No
    Message Type: T-PDU (0xff)
    Length: 49
    TEID: 0x00000001 (1)
▸ Internet Protocol Version 4, Src: 10.88.0.101, Dst: 18.200.150.181
▸ Transmission Control Protocol, Src Port: 60000, Dst Port: 50000, Seq: 1, Ack: 1, Len: 9
▸ Data (9 bytes)
```

To create, within the source namespace, GTP-U packets with TEID=0 - 19 (0, 1, 2, ...,19) and to send them through the SONiC switch to endpoints' namespaces, the following commands were used:

```
$ sudo ip netns exec source scapy -H
Welcome to Scapy (2.4.5) using IPython 7.13.0
>>>
>>> from scapy.contrib.gtp import *
>>>  pkts=IP(src="192.168.77.1",dst="198.51.100.1")/UDP(sport=33333,dport=2152)/
GTP_U_Header(teid=[(0,19)],gtp_type=0xff)/IP(src="10.88.0.101",dst="18.200.150.181")/
TCP(dport=50000,sport=60000,flags="PA",seq=12,ack=8)/"Test data"
>>> send(pkts)
....................
Sent 20 packets.
>>>
```

One can see that a set of 20 IP packets was defined (with specific source and destination IP addresses), carrying UDP (with the destination port of 2152, which is assigned to GTP-U) and GTP-U header. The TEID field contained a list of values (from 0 to 19) and the GTP-U payload (emulating end user traffic) was set to sample IP addresses with TCP ports.

On the endpoints' side, a simple Python script (Scapy based), which captured and filtered traffic (only GTP-U was of interest), was used. Information about the intercepted GTP-U packets was displayed - the total number and TEID from received GTP-U packets.

Traffic endpoint 1:

```
$ sudo ip netns exec endpoint-1 python listener-endpoint-1.py

Packet #1: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 0
Packet #2: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 2
Packet #3: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 4
Packet #4: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 6
Packet #5: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 8
Packet #6: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 10
Packet #7: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 12
Packet #8: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 14
Packet #9: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 16
Packet #10: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 18
```

Traffic endpoint 2:

```
$ sudo ip netns exec endpoint-2 python listener-endpoint-2.py

Packet #1: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 1
Packet #2: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 3
Packet #3: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 5
Packet #4: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 7
Packet #5: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 9
Packet #6: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 11
Packet #7: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 13
Packet #8: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 15
Packet #9: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 17
Packet #10: 192.168.77.1:33333 -> 198.51.100.1:2152 TEID: 19
```

Packets were forwarded to traffic endpoint No.1 or traffic endpoint No.2 depending on TEID field value - custom load balancing implemented on the SONiC switch worked as expected. The test was repeated several times with different combinations of TEID values and additionally other fields of IP packet were changed (e.g. UDP source port, GTP-U payload) - the end result was the same.

Step 4: SAI implementation - real-life check

We've already mentioned the hardware platform selection challenges related to SAI versions. Having the SONiC switch and testing the implementation for our use case allowed us to verify support for specific options and parameters which we initially planned to use. Checking the release notes (of Mellanox/NVIDIA SAI) helped to figure out the real status of the SAI API implementation. Below we present our observed limitations (related to the use case we built).

Relevant (to our use-case) limitation based on release notes:

1. UDF Match:
    a. Only L2 matches which can be ARP/IPv4/IPv6 ethertype or empty are supported
    b. An empty match is only supported for SAI_UDF_BASE_L2
    c. Non-empty matches (ARP/IPv4/IPv6) are only supported for SAI_UDF_BASE_L3

This means:

a. Only L2 (EtherType field from Ethernet header) matches are supported. SAI specification also defines L3 (Protocol field from IP header) matches but they were not implemented in the Mellanox SAI - below are the logs from SONiC when the L3 match was specified:

```
sonic ERR syncd#SDK: [SAI_UDF.ERR] mlnx_sai_udf.c[1829]- mlnx_sai_create_udf_match:
SAI_UDF_MATCH_ATTR_L3_TYPE is not supported
```

b. If there is no EtherType specified in UDF Match (empty match), layer 2 is used as a base (UDF offset base from the start of L2 header)

c. If the EtherType is specified (supported types: ARP/IPv4/IPv6) layer 3 is used as a based (UDF offset base from the start of L3 header)

Initially, we also wanted to filter packets based on IP protocol type/number (to match only UDP packets) but it turned out not to be possible (this was eventually not an issue as filtration was done by ACLs).

2. UDF Base:
    a. If L2 – Support only one UDF in the UDF group
    b. If L3 – Support up to three UDFs in the UDF group. UDF matches should be unique within the UDF group

Where:

a. L2 - SAI_UDF_BASE_L2 (UDF offset base from the start of L2 header)

a. L3 - SAI_UDF_BASE_L3 (UDF offset base from the start of L3 header)

Note: "SAI_UDF_BASE_L2" and "SAI_UDF_BASE_L3" are supported, but "SAI_UDF_BASE_L4" is not. As a consequence, we had to count the UDF OFFSET from the beginning of the layer 3 header and not layer 4 (as initially planned).

3. *ACL VLAN bind point only works for ingress stage, SAI_VLAN_ATTR_EGRESS_ACL isn't supported*

This means that the ACL Table can be bound to the ingress stage, but not the egress stage. However when checking the configuration using VLANs (a single link between the BMS and switch with three VLANs to separate traffic), it turned out that ACL Table could not bind to the VLAN interface (even though it was set to ingress stage):

```
sonic ERR syncd#SDK: [SAI_ACL.ERR] mlnx_sai_acl.c[21006]- mlnx_acl_table_bind_
point_list_fits_group: ACL Group's bind point type (SAI_ACL_BIND_POINT_TYPE_VLAN)
is not supported for ACL Table (0)
```

In our use case, the switch port bind point was used instead for ACL Table ("SAI_ACL_BIND_POINT_TYPE_PORT") and additional filtering criteria in ACL rules were added (matching the given VLAN ID).

Additionally, we observed surprising behavior related to the UDF value set in the ACL rule (to compare with that extracted by UDF objects). This value is internally kept as a list of 1-byte fields. It turned out that if the UDF length is equal to 2 bytes (meaning a list consisting of two values, each of length of 1 byte), each part of the value has to have different values:

```
sonic ERR syncd#SDK: [SAI_UTILS.ERR] mlnx_sai_utils.c[842]- sai_attribute_value_
list_is_unique: Attribute SAI_ACL_ENTRY_ATTR_USER_DEFINED_FIELD_GROUP_MIN contains
equal elements at indexes 0 and 1
```

We have only checked ACL rules with UDF lengths of 1 and 2 - behaviors for longer UDF fields were not verified.

PoC - the most important notes in the nutshell

The proof of concept demonstrated that once we have migrated the modified modules to the physical switch, a few vendor-specific problems arose in the process. To address these, we needed to apply two types of accommodations:

- invasive, i.e. the ones which forced us to waive some less crucial parts of the solution (specifically, the step of making sure that only packets containing UDP are processed),
- non-invasive, i.e. the ones for which we needed to develop a workaround, but the functionality was retained (e.g. the base for counting the offset for UDF).

After adjusting to the limitations we could finally run our solution using an appropriate lab setup to verify that the test traffic was being balanced according to our expectations. Eventually we managed to implement a working solution using the adapted SONiC modules in a physical switch.

All of that got us a chance to get our hands on custom SONiC feature development and learn how to do it, what the limitations are and how to circumvent them, plus what to leverage and what to avoid.

What may come next – summary and our conclusions

Since the PoC has proven the feasibility of the concept, the work may be taken further, leading to releasing a fully-fledged, production-ready piece of software.

So what would we do next, to make that happen?

1. Performance tests - because one of the key benefits expected from a hardware solution designed to process high loads of traffic is its performance (scalability and throughput).
2. Redundancy - our implementation does load balancing with a static (configured) number of endpoints, but a real deployment would be required to handle situations when one or more endpoints fail.
3. Developing YANG models for UDF and updating them for ACL (to include UDF) - because SONiC uses YANG models to expose its features to the user (by means of a CLI, gNMI or REST API).
4. Endpoint capacity (i.e. number of served endpoints) - because a production deployment will not be that simple. It is important to provide support for a higher number of served endpoints (any reasonable number, also not being a power of 2).
5. Universal UDF implementation - our UDF implementation was simplified and tailored to support only our use case. A generic SONiC UDF implementation would require a more flexible UDF configuration for both ACLs and hashes (used by ECMP or LAG to load balance the traffic).

We have recapped our plan of work, chosen a hardware platform, installed the modified SONiC modules and finally tested our solution. We are happy to share this experience: both the process as well as our findings. This is a PoC, so we intentionally leave the development at this point, which is conclusive but far from being a finished product.

The conclusion of the PoC, although positive in terms of the outcome, is rather moderate in terms of what could have been expected based on the promises of SONiC. Developing a custom feature is not a piece of cake, as there are some dependencies to be considered and several adjustments to the solution design might be necessary to fit them.

Most importantly, building custom functionality for SONiC requires taking into account the limitations of both hardware and SAI. As we have seen, while SONiC and SAI are separate entities, the former relies heavily on the latter when it comes to data plane configuration, i.e. we may “teach” SONiC to do new things, but at the end of the day SAI must support them (understand and be able to convey to the ASIC). On top of that, SAI implementation (version, additional features) varies from vendor to vendor, and some vendors document it publicly while others do not.

On the upside, SONiCs modular architecture really makes it flexible for development, so you do not need to rebuild the whole system for every change. And developers can focus on particular modules only.

All in all, as SONiC and SAI’s further development is proceeding, more possibilities will open, which we will be more than happy to leverage, effectively building other exciting custom features.

About the authors



Łukasz Drożdż – Software Engineer

Łukasz is a CodiLime Senior Software Engineer. He is experienced in the architecture and development of backend systems for the telco and aviation industries. Also, he enjoys being up to date with C++ evolution.



Michał Pawłowski – Senior Network Engineer

Michał is a Senior Network Engineer at CodiLime. He is involved in projects related to automation and the performance of network applications. His other fields of expertise are virtualized and cloud-native network solutions.



Tomasz Sikorski – Engineering Manager

Tomasz Sikorski is an Engineering Manager at Codilime. He is experienced in leading network engineering teams specializing in network security, automation, monitoring and custom network software development.



About CodiLime

Since 2011, CodiLime has been the engineering partner of choice for semiconductor companies, networking vendors, telecom services, and software solution providers. Our experts help explore their data assets better by creating scalable solutions that make storing, processing, and analyzing data a breeze.

We appreciate long-term collaborations above all, as well as our partners. Below are some of the names that have already trusted us:

CodiLime aims to **link network engineering talent with business domain expertise** – we focus on five N.E.E.D.S. - Networks, Equipment, Environment, Data and Security.

Do you need any further information regarding our services? Don't hesitate to contact us!

contact@codilime.com