Grant Agreement No.: 101015857
Research and Innovation action
Call Topic: ICT-52-2020: 5G PPP - Smart Connectivity beyond 5G

Secured autonomic traffic management for a Tera of SDN flows

TeraFlow

D3.2: Final evaluation of Life-cycle automation and high performance SDN components

Deliverable type	R
Dissemination level	PU
Due date	31/12/2022
Submission date	30/12/2022
Lead editor	Georgios P. Katsikas (UBITECH)
Authors	Dimitrios Klonidis, Panagiotis Famelis , Dimitrios Manolopoulos (UBI), Ricard Vilalta, Lluis Gifre, Ricardo Martinez (CTTC), Juan Pedro Fernandez-Palacios, Juan Carlos Caja, Oscar González-de-Dios, Pablo Armingol, Antonio Pastor (TID), Stanislav Lange (NTNU), Alberto Mozo, Luis de la Cal, Amit Karamchandani (UPM), Carlos Natalino (CHAL), Sebastien Andreina, Konstantin Munichev, Giorgia Mason (NEC), Min Xie, Jane Frances Pajo, Håkon Lønsethagen, Hanne Kristine Hallingby (Telenor), Achim Autenrieth, José Juan Pedreño Manresa (ADVA), Mika Silvola (Infinera), Michele Milano, Nicola Carapellese (SIAE), Javier Moreno, Sergio González, Esther Garrido (ATOS), Sebastien Merle, Peer Stritzinger (Stritzinger)
Reviewers	Ricard Vilalta (CTTC), Georgios P. Katsikas (UBI)
Quality check team	Adrian Farrel, Daniel King (ODC)
Work package	WP3

Abstract

This deliverable leverages the preliminary evaluation of life-cycle automation and high performance SDN components reported in D3.1, the final architecture design provided in D2.2, and the second release of the core TeraFlowSDN components reported in MS3.3 to provide the final evaluation of the core components of TeraFlowSDN. For each core component of the TeraFlowSDN architecture, this deliverable provides (i) a short list of new features added in release v2, (ii) the final component design, (iii) the final interfaces exposed to other TeraFlowSDN components or external entities, (iv) detailed workflows highlighting key interactions of each component, and (v) evaluation of each component's essential functions, mainly focusing on performance and scalability aspects. In addition, this deliverable is vital for validating TeraFlowSDN in WP5 for the remaining period of the project.

[End of abstract]

Disclaimer

This report contains material which is the copyright of certain TeraFlow Consortium Parties and may not be reproduced or copied without permission.

All TeraFlow Consortium Parties have agreed to publication of this report, the content of which is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License[1].

Neither the TeraFlow Consortium Parties nor the European Commission warrant that the information contained in the Deliverable is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using the information.

CC BY-NC-ND 3.0 License – 2021 - 2023 TeraFlow Consortium Parties

Acknowledgment

The research conducted by TeraFlow receives funding from the European Commission H2020 programme under Grant Agreement No 101015857. The European Commission has no responsibility for the content of this document.

Revision History

Revision	Date	Responsible	Comment
0.1	19.01.2022	Editor	Initial document structure
0.2	21.11.2022	Editor	Final design and interfaces per component
0.3	02.12.2022	Editor	Final operational workflows per component
0.4	12.12.2022	Editor	Final evaluation results per component
0.5	23.12.2022	Editor	Ready for revisions
0.6	26.12.2022	Q/A	Q/A Review by Daniel King
1.0	30.12.2022	Editor	Submitted
1.1	15.10.2023	Editor	After EC approval

[1] http://creativecommons.org/licenses/by-nc-nd/3.0/deed.en_US

EXECUTIVE SUMMARY

This deliverable summarizes the activities of WP3 throughout the entire TeraFlow project. The objective of this document is to provide: (i) a detailed design overview of each core TeraFlowSDN component, including internal architecture and adopted technologies; (ii) a set of interfaces per component with precise interactions both with other TeraFlowSDN components and external entities; (iii) detailed workflows highlighting interactions between TeraFlowSDN components and external systems and stakeholders, and (iv) evaluation results per component, focusing on performance and scalability aspects. A list of new features per component is also provided to highlight the delta between release v1 (early 2022) and the latest release, v2.

This document begins with an introductory section highlighting the purpose of this deliverable and overall structure; and the relationship with other deliverables. Section 2 maps partners to core TeraFlowSDN components and presents a taxonomy of these core components across key WP3 aspects detailed in Sections 3-6. Section 3 tackles components related to performance (T3.1), Section 4 describes components related to heterogeneous hardware and multi-layer service integration (T3.2), Section 5 addresses components related to SDN automation and policy management (T3.3), and Section 6 presents the slicing and multi-tenancy component (T3.4).

This document concludes in Section 7, while Section 8 serves as an annexe with technical details about various components, such as data models and XML templates.

Table of Contents

List of Figures

List of Tables

Abbreviations

5G	Fifth Generation
5G-PPP	5G Infrastructure Public Private Partnership
ABNO	Application Based Network Optimization
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
B5G	Beyond 5G
BGP	Border Gateway Protocol
BSS	Business Support System
DB	Database
E2E	End-to-End
ECA	Event-Condition-Action
FPGA	Field-Programmable Gate Array
FRR	Free-Range Routing
gNMI	gRPC Network Management Interface
gNOI	gRPC Network Operations Interface
gRPC	gRPC Remote Procedure Call
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
I/O	Input-Output
IP	Internet Protocol
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
L1	Layer 1
L2	Layer 2
L3	Layer 3
L3NM	Layer 3 Network YANG Model
LSP	Label Switched Path
MS	Milestone
MW	Microwave
NBI	North-Bound Interface
NOS	Network Operating System
OAS	OpenAPI Specification
OC	OpenConfig
OLS	Open Line System
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OS	Operating System
OSPF	Open Shortest Path First
OSS	Operation Support System
P4	Programming Protocol-independent Packet Processors
PCE	Path Computation Element
PCEP	Path Computation Element Protocol
QoS	Quality of Service
REST	Representational State Transfer
RPC	Remote Procedure Call
SBI	South-Bound Interface
SDN	Software-Defined Networking
SDO	Standards Development Organization

SLA	Service-Level Agreement
SLE	Service-Level Expectation
SLI	Service-Level Indicator
SLO	Service-Level Objective
SQL	Structured Query Language
SR	Segment Routing
TAPI	Transport API
TE	Traffic Engineering
TED	Traffic Engineering Database
TFS	TeraFlow SDN
VLAN	Virtual Local Area Network
VPN	Virtual Private Network
WP	Work Package
XML	eXtensible Markup Language
ZTP	Zero-Touch Provisioning

1. Introduction

TeraFlowSDN is a novel software-defined networking (SDN) controller architecture aiming at capabilities for beyond fifth generation (B5G) network deployment. In this context, TeraFlowSDN bridges critical gaps in state-of-the-art SDN controllers in four distinct areas organized as WP3 tasks:

- **Focus Area 1 (T3.1):** high-performance control plane operations through a revolutionary cloud-native network operating system (NOS) design, based on distributed and fully disaggregated microservices;
- **Focus Area 2 (T3.2):** native support for key transport technologies, such as Internet Protocol (IP), optical, and microwave (MW), as well as emerging next-generation SDN technologies, such as the programmable protocol-independent packet processors (P4);
- **Focus Area 3 (T3.3):** automated, zero-touch provisioning (ZTP) of network services and service-level and device-level policy management;
- **Focus Area 4 (T3.4):** multi-tenant network slicing as a service coupled with service-level agreement (SLA) requirements.

1.1. Objectives

The purpose of this deliverable (D3.2) is threefold. The first objective of the deliverable is to provide core TeraFlowSDN components for addressing the four areas introduced above. This is done by mapping all core TeraFlowSDN components to the various WP3 tasks (each task corresponds to a section between Section 3 and Section 6 of this document), while providing fundamental concepts and a detailed design overview per component. The second objective is to position all core TeraFlowSDN components within the same ecosystem, thus prescribing how they communicate with each other and communication with external entities and systems. This is achieved by associating each component description with a dedicated sub-section describing its interfaces and another sub-section detailing essential workflows. Finally, this deliverable's third objective is to evaluate the features of the core TeraFlowSDN components through another sub-section per component outlining performance and scalability tests.

1.2. Relation with Other Tasks and Deliverables

This deliverable complies with the latest changes in the TeraFlowSDN architecture as per D2.2 “Final requirements, architecture design, business models and data models” [4] to present an extended version of D3.1 [5]. In D3.1, a preliminary version of the core TeraFlowSDN components was presented along with a preliminary evaluation for some of these components, based on the first TeraFlowSDN source code release as per MS3.2 “Code freeze for TeraFlow OS components (v1)” [2].

In this final WP3 deliverable, the final version of each core TeraFlowSDN component is documented, based on the latest source code developments that comprise the second TeraFlowSDN release in MS3.3 “Code freeze for TeraFlow OS release components (v2)” [3].

This final core TeraFlowSDN release will act as a basis for accommodating the TeraFlowSDN components reported in D4.2 “Final evaluation of TeraFlow security and B5G network integration” [6] expected to be delivered on January 2023. Moreover, the final TeraFlowSDN platform-as-a-whole (combined WP3 and WP4 components) will be used to validate the various scenarios reported in WP5 in the context of D5.2 “Implementation of pilots and first evaluation” [7] and D5.3 “Final demonstrators and evaluation report” [8].

1.3. Deliverable Structure

In the rest of this deliverable, Section 2 presents an overview of the core TeraFlowSDN components that comprise the entire WP3. Sections 3, 4, 5, and 6 highlight the design overview, interfaces, workflows, and evaluation results of the various core TeraFlowSDN components across the four tasks in WP3, respectively. Section 7 concludes this work. Finally, Section 8 (Annex) reports data models and XML templates for specific TeraFlowSDN components.

2. Core TeraFlow OS Components' Overview

This section provides an overview of the core TeraFlow OS components in the context of WP3. Table 1 shows how these components are mapped to the various WP3 tasks and the corresponding partners carrying out their design, implementation, and preliminary evaluation during the project's first year.

Table 1: Mapping core TeraFlow components to WP3 tasks and the contributing partners.

WP3 Task	Component Name	Involved Partners
T3.1	Context Management	CTTC, TID
	Monitoring	ATOS
	Traffic Engineering	STR
	Path Computation	CTTC
	Auto Scaling	Features covered by Kubernetes Orchestrator (see MS3.2)
	Load balancing	
T3.2	South-bound Interface (SBI)	CTTC, TID, UBI, SIAE, INF, ADVA
	Service	CTTC, TID, UBI, SIAE, INF
	Forecaster	CTTC
T3.3	Automation or ZTP	UBI
	Policy Management	UBI, ODC
T3.4	Slice Management	ADVA, CTTC

In the following technical sections:

- Section 3 deals with components related to SDN performance, context management (Section 3.1), monitoring (Section 3.2), traffic engineering (Section 3.3), and path computation (Section 3.4). As noted in Table 1 and MS3.2 [2], auto-scaling and load-balancing components are provided by the Kubernetes orchestrator framework, which serves as a deployment engine for the TeraFlow microservices;
- Section 4 presents hardware and multi-layer service integration components, namely the SBI component with various driver plugins for different SDN devices (Section 4.1), the Service component with multiple service handlers for key service types (Section 4.2), and the Forecaster component (Section 4.3);
- Section 5 introduces the zero-touch device provisioning (ZTP) component (Section 5.1) as well as the policy management component (Section 5.2); and
- Section 6 presents the slice management component.

3. High-Performance SDN Framework

This section provides the design overview, interfaces, operational workflows, and evaluation results of the core TeraFlowSDN components of T3.1, i.e., the Context Management component (see Section 3.1), the Monitoring component (see Section 3.2), the Traffic Engineering (TE) component (see Section 3.3), and the Path Computation component (see Section 3.4).

3.1. Context Management Component

The context component is the single entry point to the necessary operations for reading, updating, or removing elements from the TeraFlowSDN controller database. Context handles objects such as topologies, devices, links, services, and connections. The TeraFlowSDN components require access to these objects to provide the necessary functionalities.

3.1.1. New Features/Extensions

A complete internal re-design of the component has been necessary to include self-replication and scalability of the component's database. Also, novel topics are included to serve as a state database for several components. The new main features in release 2 are the following:

- Context Component replication. The internal database has been modified to allow distribution. We also include a NewSQL database: CockroachDb.
- We have reviewed and extended protocol buffers definitions that were required by other components:
    - Add support for constraints in Service and SBI Components;
    - Add service constraints to path computation;
    - Add support for Inventory;
    - Added Policy definitions for permanent storage of policies from Policy component;
    - Added support for ACL in Service.

3.1.2. Final Design

The Context micro-service is where the NewSQL database resides. It is based on a distributed architecture, where the key spaces of the tables are split into continuous ranges, called shards. The ranges are then replicated on at least two nodes. This way, if a node stops functioning, there is at least another copy of the data stored on other nodes and reads/writes can still be served.

A consensus algorithm is used to spread the changes and synch changes across nodes before the final commit to maintain consistency across all database shards.

Depending on the structure, we can identify two consensus algorithms, leader-less or leader-based architecture, which most distributed databases rely on. Paxos and Raft consensus algorithms are the two most used. Paxos has been one of the first algorithms to provide consistent and fault-tolerant distributed consensus, although Raft is usually more used due to its simplicity. Fundamentally, at high level, both algorithms base their functioning on electing a leader for each of the shards of the database. All changes in the shard must be taken out by the leader, and most followers must acknowledge this change before the change is finally committed. When a leader goes down the other nodes that take part in the shard must achieve a quorum to elect a new leader.

3.1.3. Final Interfaces

Context offers many interfaces to all other TeraFlowSDN components. We have divided the interfaces into the following topics or categories:

- Context
- Topology
- Device
- Link
- Service
- Slice
- Connection
- Policies
- ACL

3.1.3.1. Context

The final list of RPC methods exported by the context service to handle context objects are defined in its data model and widely reported in the deliverable D2.2 [4]. In Table 2, we expose a summary of the available RPCs in this interface.

RPC Method Name	Parameters	Results
ListContextIds		ContextIdList
ListContexts		ContextList
GetContext	ContextId	Context
SetContext	Context	ContextId
RemoveContext	ContextId	
GetContextEvents		stream ContextEvent

Table 2 Context component interfaces for Context instances

3.1.3.2. Topology

The final list of RPC methods exported by the context service to handle Topology objects are defined in its data model and widely reported in the deliverable D2.2 [4]. In Table 3, we expose a summary of the available RPCs in this interface.

RPC Method Name	Parameters	Results
ListTopologyIds	ContextId	TopologyIdList
ListTopologies	ContextId	TopologyList
GetTopology	TopologyId	Topology
SetTopology	Topology	TopologyId
RemoveTopology	TopologyId	
GetTopologyEvents		stream TopologyEvent

Table 3 Context component interfaces for Topology instances

3.1.3.3. Device

The final list of RPC methods exported by the context service to handle Device objects are defined in its data model and widely reported in the deliverable D2.2 [4]. In Table 4, we expose a summary of the available RPCs in this interface.

RPC Method Name	Parameters	Results
ListDeviceIds		DeviceIdList
ListDevices		DeviceList
GetDevice	DeviceId	Device
SetDevice	Device	DeviceId
RemoveDevice	DeviceId	
GetDeviceEvents		stream DeviceEvent

Table 4 Context component interfaces for Device instances

3.1.3.4. Link

The final list of RPC methods exported by the context service to handle Link objects are defined in its data model and widely reported in the deliverable D2.2 [4]. In Table 5, we expose a summary of the available RPCs in this interface.

RPC Method Name	Parameters	Results
ListLinkIds		LinkIdList
ListLinks		LinkList
GetLink	LinkId	
SetLink	Link	LinkId
RemoveLink	LinkId	
GetLinkEvents		stream LinkEvent

Table 5 Context component interfaces for Link instances

3.1.3.5. Service

The final list of RPC methods exported by the context service to handle Service objects are defined in its data model and widely reported in the deliverable D2.2 [4]. In Table 6, we expose a summary of the available RPCs in this interface.

RPC Method Name	Parameters	Results
ListServiceIds	ContextId	ServiceIdList
ListServices	ContextId	
GetService	ServiceId	Service
SetService	Service	ServiceId
UnsetService	Service	ServiceId
RemoveService	ServiceId	
GetServiceEvents		stream ServiceEvent

Table 6 Context component interfaces for Service instances

3.1.3.6. Slice

The final list of RPC methods exported by the context service to handle Slice objects are defined in its data model and widely reported in the deliverable D2.2 [4]. In Table 7, we expose a summary of the available RPCs in this interface.

RPC Method Name	Parameters	Results

ListSliceIds	ContextId	SliceIdList
ListSlices	ContextId	SliceList
GetSlice	SliceId	Slice
SetSlice	Slice	SliceId
UnsetSlice	Slice	SliceId
RemoveSlice	SliceId	
GetSliceEvents		stream SliceEvent

Table 7 Context component interfaces for Slice instances

3.1.3.7. Connection

The final list of RPC methods exported by the context service to handle Connection objects are defined in its data model and widely reported in the deliverable D2.2 [4]. In Table 8, we expose a summary of the available RPCs in this interface.

RPC Method Name	Parameters	Results
ListConnectionIds	ServiceId	ConnectionIdList
ListConnections	ServiceId	ConnectionList
GetConnection	ConnectionId	Connection
SetConnection	Connection	ConnectionId
RemoveConnection	ConnectionId	
GetConnectionEvents		stream ConnectionEvent

Table 8 Context component interfaces for Connection instances

3.1.3.8. Policies

The final list of RPC methods exported by the context service to handle Policy objects are defined in its data model and widely reported in the deliverable D2.2 [4]. In Table 9, we expose a summary of the available RPCs in this interface.

RPC Method Name	Parameters	Results
ListPolicyRuleIds		policy.PolicyRuleIdList
ListPolicyRules		policy.PolicyRuleList
GetPolicyRule	policy.PolicyRuleId	policy.PolicyRule
SetPolicyRule	policy.PolicyRule	policy.PolicyRuleId
RemovePolicyRule	policy.PolicyRuleId	

Table 9 Context component interfaces for Policy instances

3.1.4. Final Operational Workflows

Figure 1 depicts the basic steps of the provisioning of a connectivity service. First, the OSS/BSS requests creating a connectivity service to the Service module. Then, the request is recorded on the database using the Context module. After it, the SBI module is asked to create the necessary connections to provision the connectivity service, and again, they are stored on the database by the Context module. The SBI module then provisions the connections to the ROADM to make the connectivity service effective. Finally, the SBI module responds to the Service module that all necessary connections have been created. It then responds to the OSS/BSS that the connectivity service has been effectively provisioned.

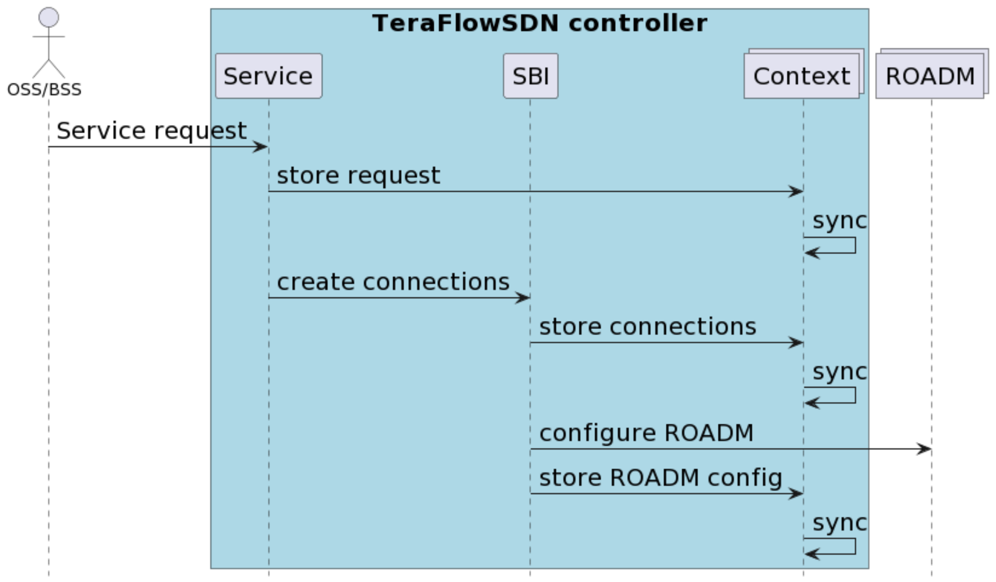


Figure 1: Context workflow

3.1.5. Evaluation

The TeraFlow controller runs on a virtual machine inside a computing node of the ADRENALINE cloud infrastructure, running Ubuntu 21.10 with an Intel(R) Xeon(R) CPU E5-2420 @ 1.90GHz CPU, 64GB of RAM, and 2TB of disk storage. As the cloud orchestrator, the software that manages and coordinates the micro-services, Kubernetes 1.23, has been deployed.

The Context micro-service includes a NewSQL fault-tolerant distributed database named CockroachDB version 21.2.4 with three database replicas deployed [18].

To validate the effectiveness and fault-tolerance of the system, the SDN controller has been loaded with an optical connectivity service request generator. Up to 3000 connections have been generated with an inter-arrival time of 0.1 seconds and a holding time of also 0.1 seconds, creating a system load of 1 Erlang (number of simultaneous connectivity services). Note that this high number of optical connectivity services is meant to prove the fault-tolerance in a worst-case scenario.

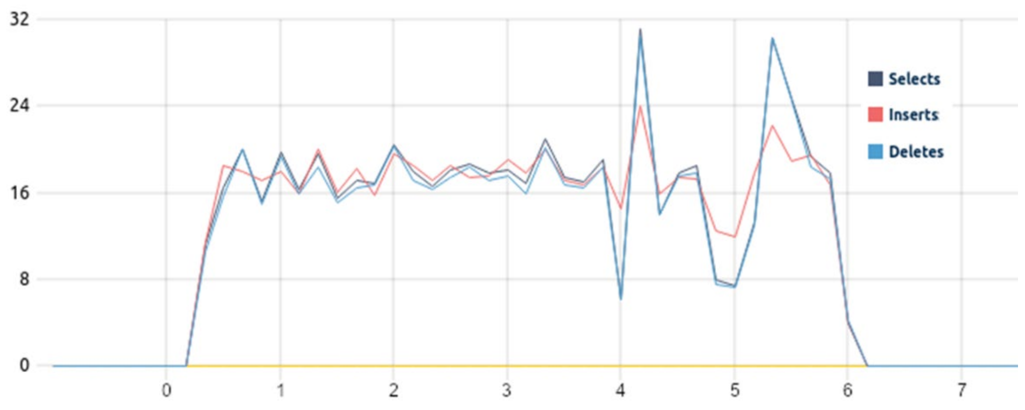


Figure 2: Timeseries (minutes) of queries on the database cluster

Figure 2 shows the time-series of the number of queries received on the aggregated database cluster (i.e., including the three database replicas). The load was sent just after minute 0 to the SDN controller and maintained the average number of queries (select, insert, and deletes) at approximately 20 per second during the test duration. Inserts are executed twice for each optical connectivity service request, as shown in Figure 1, one for the connectivity service and one for the physical connection. Likewise, in the release phase, each insert has a delete, and a select precedes each delete to search for the row to delete.

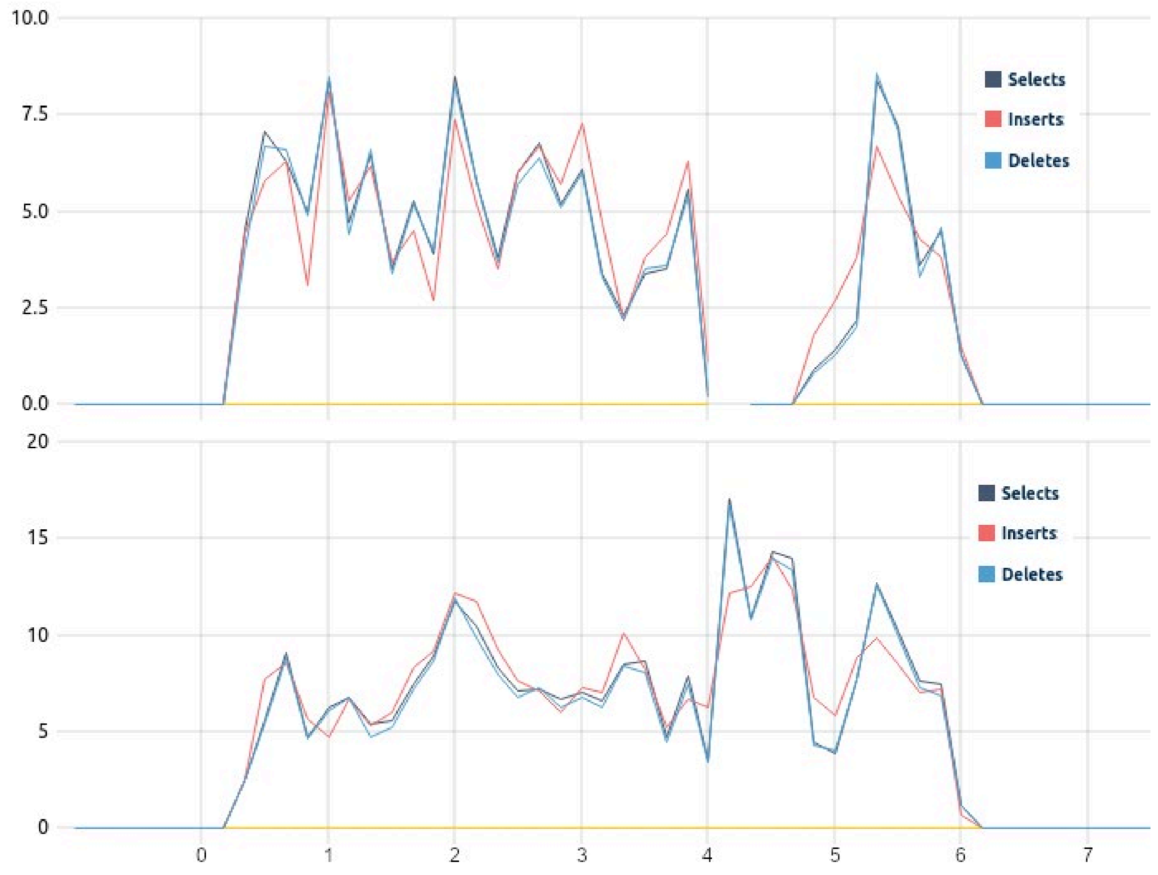


Figure 3: Timeseries (minutes) of queries on a) the faulty node b) a healthy node

Figure 3 shows the same time-series of queries on two of the three total nodes. Figure 3.a shows the node on which a network error has been found, so the cloud orchestrator has restarted the virtual machine. It can be seen that just before minute 4, the database node stopped serving requests, and after the node had restarted again, it continued to serve the requests. Figure 3.b shows the healthy node.

It can be seen that at the same time the faulty node went down, the healthy node experienced a sudden peak in requests. This is due to the load balancing of the database. As the data is replicated and distributed across different nodes, there will always be a node that can serve the requests that the problematic node cannot without compromising data integrity. Finally, when the faulty node resumes its operation, around 40 seconds after the reset, the load is again balanced between the three nodes until the end of the test.

3.2. Monitoring Component

The monitoring component is the core of the TeraFlow SDN (TFS) system devoted to the management, storage and access metrics in the shape of Key Performance Indicators (KPIs). It provides a gRPC API to export its related functionalities to be exploited by other components or agents within the TeraFlow SDN controller. KPIs drive the monitoring model where external actors can define their own generic, ad-hoc and vendor-independent KPIs, as well as alarms and subscriptions associated with those KPIs. The primary purposes to be addressed by the monitoring component are listed below:

- Generate and manage multiple metrics and KPIs;
- Enable external KPI subscriptions to timely serve monitoring data;
- Definition, management and provision of ad-hoc alarms attached to the KPIs;
- Automatic integration with external time-series visualization tools.

3.2.1. New Features/Extensions

The evolution of the monitoring component concerning what was reported in the last deliverable D3.2 [2] has focused on a three-fold approach:

- Expanding the set of functionalities to cover the monitoring requirements;
- Improve overall component stability and performance;
- Enhance horizontal scalability to support highly loaded scenarios.

Based on the roadmap, the monitoring component has evolved, providing new functionalities and features and are now integrated. The enhancements are summarized as follows:

- Evolution of the monitoring protobuf data model to support subscriptions and alarms;
- New RPC methods to export new functionalities according to the data model update;
- New distributed and scalable time-series database (QuestDB) to store the monitoring data aimed at improving overall performance;
- Definition and implementation of subscription subsystem;
- Definition and implementation of alarm subsystem;
- Expand management database structure and features to support the management of the subscription and alarm subsystems.

3.2.2. Final Design

The design is driven by key performance indicators (KPIs), where external actors can define their own generic, vendor-agnostic KPIs and the alarms and subscriptions associated with those KPIs. KPIs, alarms and subscriptions are registered internally and managed through an internal management Database. In addition, external sources can ingest monitoring data associated with the KPIs into a high-performance time-series database via the monitoring component. Finally, it supports integrating multiple third-party data visualisation tools, such as Grafana. This architecture considers two main blocks, the *Monitoring Core* and the *MetricsDB*, designed to be deployed in separate containers linked to the same Monitoring component, depicted in the Figure 4 below:

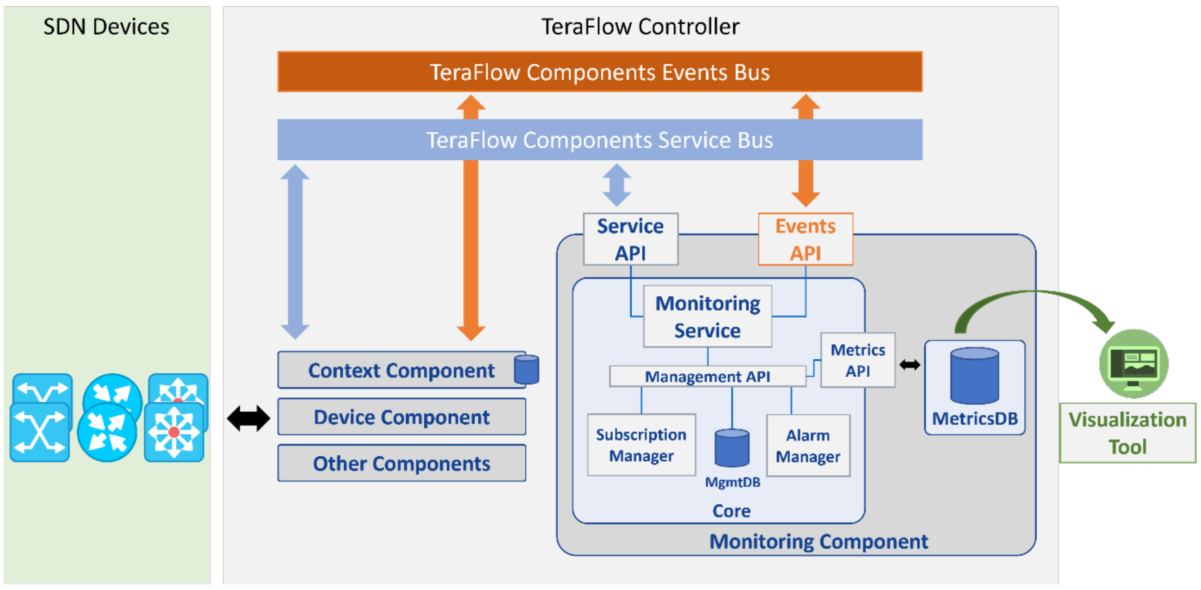


Figure 4: Architecture of the monitoring component

- **Monitoring Core**

The brain of the TFS platform is the Monitoring component. It implements the necessary logic for managing KPIs, subscriptions and alarms. Likewise, the Monitoring Core can be decomposed into six main sub-modules according to its functional role:

  - *Monitoring Service* is in charge of exporting the set of available RPC methods defined in the monitoring data model to be requested by external entities by using gRPC communication;
  - *Management API* is the sub-module responsible for linking all the internal sub-modules within the monitoring core. It can be seen as a gateway for the management DB;
  - *Management Database* supports internal management by storing information associated with the definition of subscriptions, alarms, and KPIs organized in separate SQL-based tables. Here, the models of the tables must have a concrete structure and fields to be in line with the monitoring data models;
  - *Subscription Manager* is responsible for managing the information of the subscribers, and for coordinating the actions inside the Monitoring Core among the different sub-components;
  - *Alarm Manager*, like the subscription manager*,* is the sub-module where the internal methods are implemented focused on alarm management;
  - *Metrics API* is oriented to handle internal service-level communication procedures between the Monitoring Core and the MetricsDB.

- **Metrics Database**

The MetricsDB is the other main functional block of the monitoring component, and it is designed to deploy in separate containers to leverage its inherent distributed nature. Moreover, the main goal of the MetricsDB is to store and record the information linked to the metrics/KPIs. To provide full interoperability with the core monitoring block, the stored samples' structure is mapped to the fields of the monitoring data model. In addition, the MetricsDB is directly integrated with external data visualisation tool (e.g., Grafana), thus not only reducing the complexity of the integration, but also minimising potential problems arising from low-level interoperability. Finally, as exposed in the previous bullet, it exchanges information with the Monitoring Core block via the Metrics API.

3.2.3. Final Interfaces

Regarding the monitoring connectivity, we consider three main interfaces in total: the monitoring service gRPC interface, which permits external entities (e.g., TFS components) to request a broad set of monitoring-related methods, and two internal interfaces, one focused on management interactions driven by the management API module, and the MetricsDB interface devoted to exchange information between the Monitoring Core and the MetricsDB blocks.

1. **Monitoring service gRPC methods**:

   The final list of RPC methods exported by the monitoring service is defined by its data model and widely reported in the deliverable D2.2 [4]. In the following Table 10, we expose a summary of the available RPCs in this interface:

Table 10: gRPC interface definition for Monitoring component.

RPC Method Name	Parameters	Results
SetKpi	KpiDescriptor	KpiId
DeleteKpi	KpiId	context.Empty
GetKpiDescriptor	KpiId	KpiDescriptor
GetKpiDescriptorList	context.Empty	KpiDescriptorList
IncludeKpi	Kpi	context.Empty
MonitorKpi	MonitorKpiRequest	context.Empty
QueryKpiData	KpiQuery	RawKpiTable
SetKpiSubscription	SubsDescriptor	stream SubsResponse
GetSubsDescriptor	SubscriptionID	SubsDescriptor
GetSubscriptions	context.Empty	SubsList
DeleteSubscription	SubscriptionID	context.Empty
SetKpiAlarm	AlarmDescriptor	AlarmID
GetAlarms	context.Empty	AlarmList
GetAlarmDescriptor	AlarmID	AlarmDescriptor
GetAlarmResponseStream	AlarmSubscription	stream AlarmResponse
DeleteAlarm	AlarmID	context.Empty
GetStreamKpi	KpiId	stream Kpi
GetInstantKpi	KpiId	Kpi

2. **Monitoring management API methods**:

   The available internal management API-oriented methods that can be used by the other submodules of the Monitoring Core are defined in the ManagementDBTools class. Table 11 below lists their methods:

Table 11: Monitoring management API methods.

Method Name	Parameters	Results

create_monitoring_table	-	-
create_subscription_table	-	-
create_alarm_table	-	-
insert_KPI	kpi_description, kpi_sample_type, device_id, endpoint_id, service_id, slice_id, connection_id	kpi_id
insert_subscription	kpi_id,subscriber, sampling_duration_s, sampling_interval_s, start_timestamp, end_timestamp	subs_id
insert_alarm	alarm_description, alarm_name, kpi_id, kpi_min_value, kpi_max_value, in_range, include_min_value, include_max_value	alarm_id
delete_KPI	kpi_id	Bool
delete_subscription	subs_id	Bool
delete_alarm	alarm_id	Bool
get_KPI	kpi_id	kpi_descriptor
get_subscription	subs_id	subs_descriptor
get_alarm	alarm_id	alarm_descriptor
get_KPIS	-	list kpi_descriptor
get_subscriptions	-	list subs_descriptor
get_alarms	-	list alarm_descriptor
check_monitoring_flag	kpi_id	Bool
set_monitoring_flag	kpi_id, flag	Bool

3. **Monitoring MetricsDB API methods**:

   The Monitoring-MetricsDB interface defines the potential interactions between the Monitoring Core block with the MetricsDB. In this version of the component the role of the MetricsDB is played by QuestDB [26] and the methods here presented are designed to exploit the by-default QuestDB APIs according to the needs of the monitoring component. The MetricsDBTools class and Monitoring-Metrics API methods are implemented and summarised below. In the following Table 12, we document several relevant methods:

Table 12: Monitoring Metrics API methods.

Method Name	**Parameters**	**Results**
create_table	-	-
write_KPI	time, kpi_id,	-

	kpi_sample_type, device_id, endpoint_id, service_id, slice_id, connection_id, kpi_value	
run_query	sql_query	kpi_dataset
run_query_postgre	postgre_sql_query	kpi_dataset
get_raw_kpi_list	kpi_id, monitoring_window_s, last_n_samples, start_timestamp, end_timestamp	kpi_dataset
get_subscription_data	subs_queue, kpi_id, sampling_interval_s	kpi_list
get_alarm_data	alarm_queue, kpi_id, kpiMinValue, kpiMaxValue, inRange, includeMinValue, includeMaxValue, subscription_frequency_ms	kpi_list

3.2.4. Final Operational Workflows

In this section, we present three exemplary operational workflows involving the most common monitoring features to be exploited by other TeraFlow components in a final shape. Those features are exposed below.

4. **KPI creation and data visualization**:

This first operational workflow aims to create (“registration”) a new KPI requested by a generic TFS component and its automatic data tracking from the data source to its display by an end-user in the time-series-based data visualization tool Grafana. For simplicity, it is worth mentioning that the example exposed in Figure 5 is only focused on a concrete workflow about creating KPIs associated with retrieving monitoring data from real or emulated network devices. The definition and monitoring of KPIs of other natures (e.g., based on services or connections rather than just devices) can be defined similarly as in the workflows presented in deliverables D2.2 [4] and D5.2 [6].

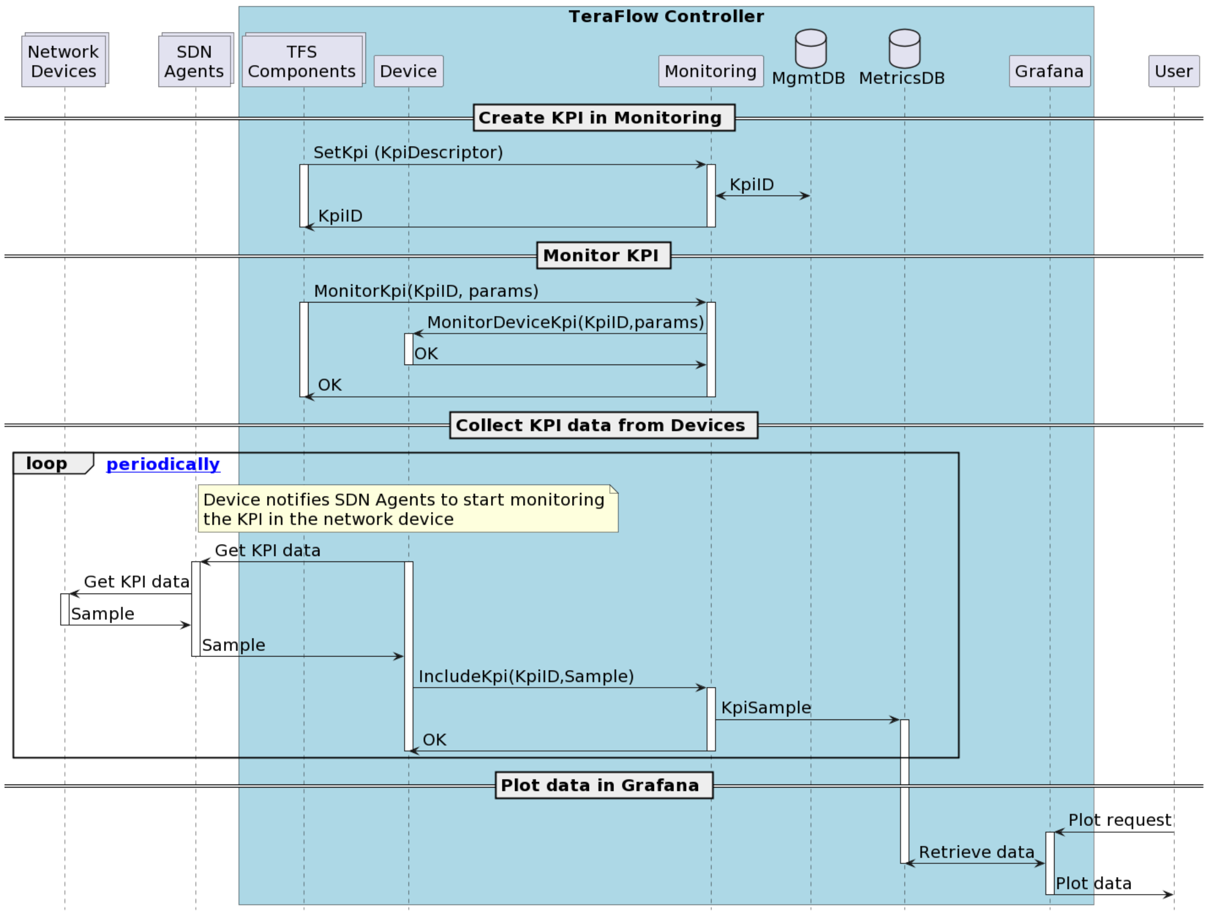


Figure 5: An example workflow of a generic definition of a KPI and its data visualization in Grafana

Thus, the workflow depicted in Figure 5 is divided in the following stages:

1. *KPI creation*: In this preliminary step a concrete TFS component defines its own KPI in a *KpiDescriptor* format. Based on its data model in a *KpiDescriptor* the requester can define the KPI according to a *KpiSampleType*, *DeviceId*, *ServiceId* or *ConnectionId* among others. For the sake of this example, we assume that the KPI is related to an existing *DeviceId*. The TFS component, by using the monitoring client, executes the gRPC *SetKpi* method passing the *KpiDescriptor* as a request message. Then the monitoring gRPC Service receives the requests and tries to store the descriptor in the MgmtDB which assigns a *KpiId* in case the combination of fields defined in the *KpiDescriptor* does not exist in the MgmtDB. Finally, the Monitoring Service returns the *KpiId* to the TFS Component.
2. *Request Monitoring the Kpi*: Once the *KpiID* reaches the corresponding TFS component side, it is enabled to request starting the monitoring in the corresponding device. To do so, it triggers the *MonitorKpi* RPC via the monitoring client by passing the generate *KpiId* and for a concrete monitoring time window and a sampling rate to get the data. Then, the request is received by the monitoring component and processes the data attached to such *KpiId*. After that, the monitoring service informs the device component about start monitoring the about the concrete metrics in the concrete device according to the defined in the *KpiDescriptor* and *KpiMonitorRequest*. Finally, the device component receives the KPI information and triggers a parallel workflow to retrieve the actual device information from the system.
3. *Collection KPI Data*: in this stage the device component uses the corresponding SBI to request the device associated to such DeviceId to start retrieving the given data and receives the data

periodically for every time iteration and for a given sampling rate. Then the device component forwards the data to the monitoring service by executing the *IncludeKpi* method. After that, the monitoring service ingests the monitoring data in the MetricsDB.

4. *Plot data in Grafana*: Finally, as the MetricsDB is automatically attached to the Grafana instance, once the monitoring data is available at the MetricsDB it can be visualized by a user in the Grafana dashboard.

5. **KPI Subscription and data provisioning**

The workflow presented in the next Figure 6 showcases a generic example of how a TFS component can subscribe to a concrete KPI, previously defined and is being monitored.

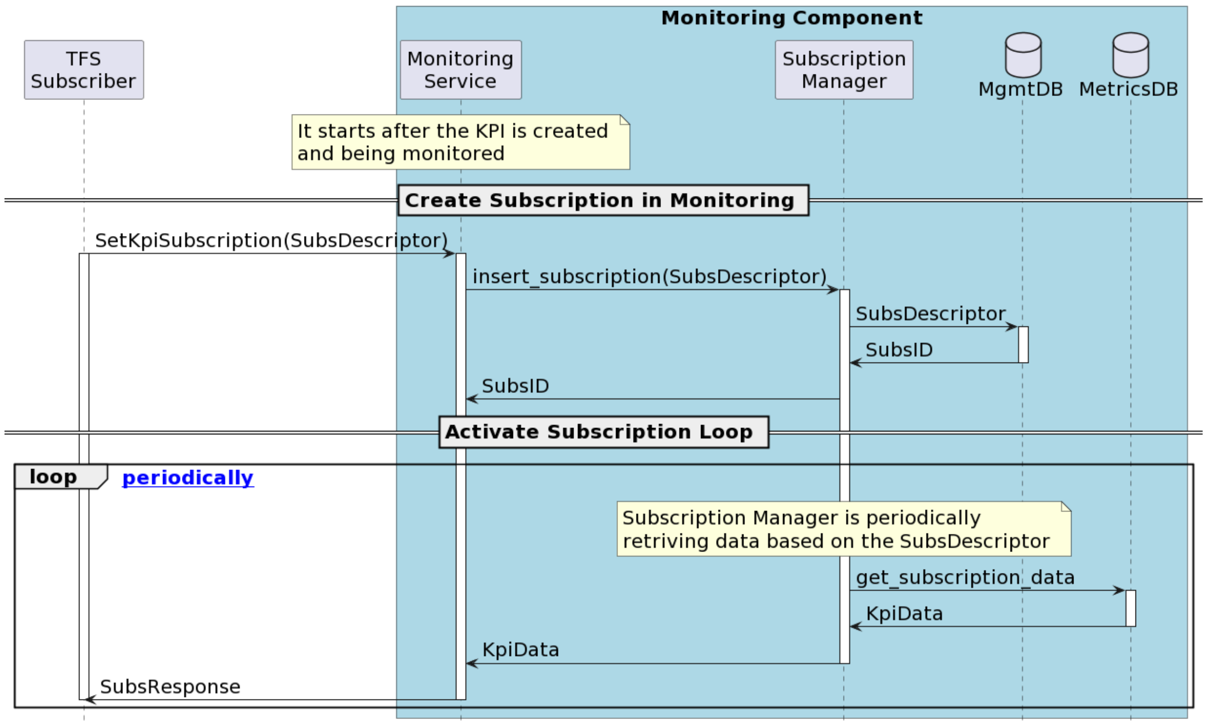

Monitoring Component
TFS Subscriber
Monitoring Service
Subscription Manager
MgmtDB
MetricsDB
It starts after the KPI is created and being monitored
Create Subscription in Monitoring
SetKpiSubscription(SubsDescriptor)
insert_subscription(SubsDescriptor)
SubsDescriptor
SubsID
SubsID
Activate Subscription Loop
loop
periodically
Subscription Manager is periodically retriving data based on the SubsDescriptor
get_subscription_data
KpiData
KpiData
SubsResponse

Figure 6: Exemplary workflow of a generic subscription loop

The workflow is defined as follows:

1. *Create Subscription*: This workflow is only triggered after registering the given KPI and is currently being monitored (see the first workflow). Like a KPI definition the TFS component can define its subscription in a *SubsDescriptor* shape by setting the existing *KpiId* for a concrete time interval and sampling rate compatible with the monitoring parameters or defining start and end timestamps and sending that request by using the *SetKpiSubscription* method. The request arrives at the monitoring component, where the Monitoring Service forwards the request to the Subscription Manager. After internal checks to see if the subscription fits the monitoring parameters, the MgmtDB assigns a unique *SubsId,* and the Monitoring Service receives it.
2. *Actvate Subscription Loop*: then, in parallel, the Subscription Manager starts with the monitoring loop to periodically query the MetricsDB about the monitoring data associated to that *KpiID* and for a concrete time window defined in the *SubsDescriptor*. Then, this *KpiData* is sent to the Monitoring Service to build the *SubsResponse* by attaching the *KpiData* and the

associated *SubsId* and replying to the subscriber. This stage is repeated periodically for the time window defined in the *SubsDescriptor*.

6. **Alarm definition and rule violation notification:**

Here, in Figure 7 we explain the final exemplary workflow to use the alarm functionality exposed by the monitoring service.

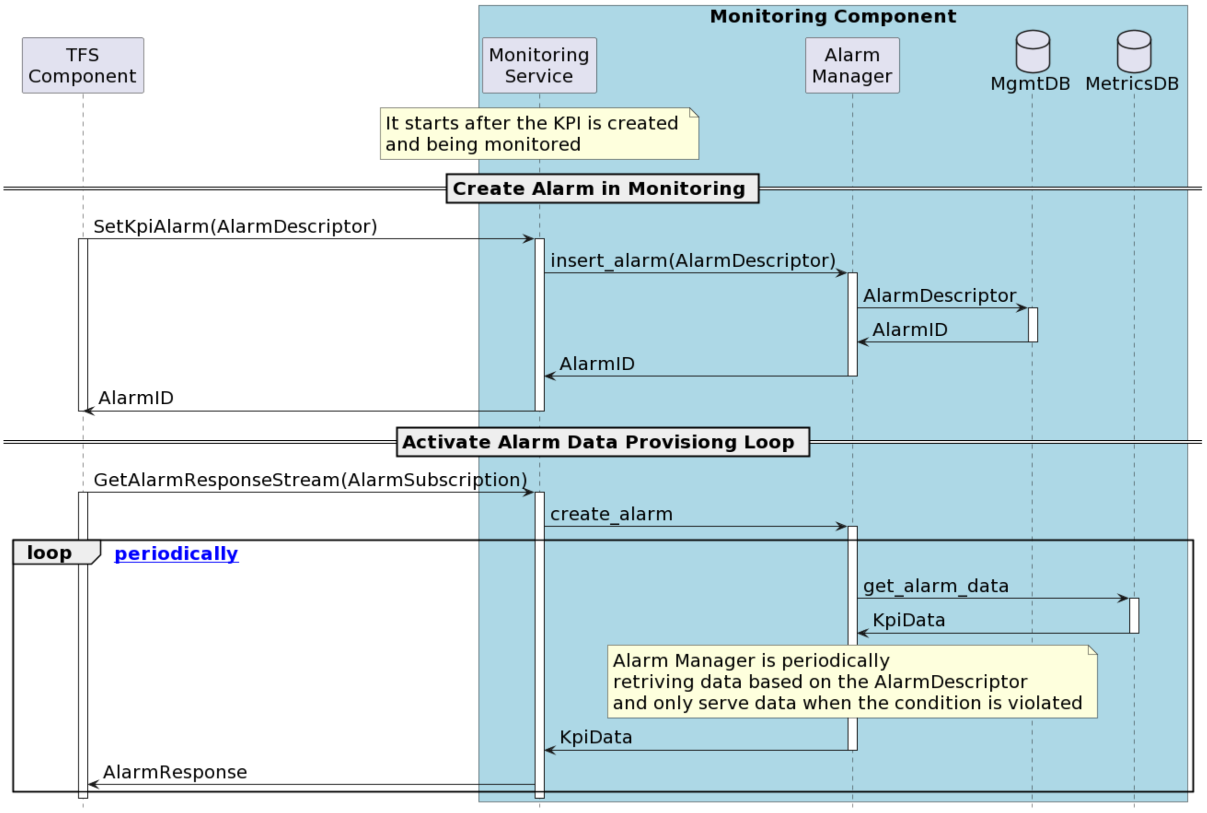

Monitoring Component
TFS Component
Monitoring Service
Alarm Manager
MgmtDB
MetricsDB
It starts after the KPI is created and being monitored
Create Alarm in Monitoring
SetKpiAlarm(AlarmDescriptor)
insert_alarm(AlarmDescriptor)
AlarmDescriptor
AlarmID
AlarmID
AlarmID
Activate Alarm Data Provisioning Loop
GetAlarmResponseStream(AlarmSubscription)
create_alarm
loop
periodically
get_alarm_data
KpiData
Alarm Manager is periodically retriving data based on the AlarmDescriptor and only serve data when the condition is violated
KpiData
AlarmResponse

Figure 7: Exemplary workflow of a generic alarm definition and its related loop

This workflow is like the subscription one but with some critical differences specific to alarms. The most obvious is that in subscriptions, the entire workflow relies on a single RPC method and in this workflow, it is based on two, one to define the alarm and one to trigger it, the rationale about this below:

- *Create alarm*: In this first stage a TFS component can define in an *AlarmDescriptor* an ad-hoc alarm associated to an existing KPI. In this alarm descriptor is where the alarm rules are set that the value thresholds or intervals are defined as valid for that concrete KPI. Typically, the TFS Component send a request to register the alarm (*SetKpiAlarm*) to the Monitoring Service and the Alarm Manager processes it. In this case, if the alarm is properly registered in the MgmtDB the *AlarmId* is returned to the TFS component in this first stage. This is done to provide more flexibility on where and when to activate the defined alarm and permitting to use that alarm not only to the same subscriber but multiple, thus, expanding the potential scenarios to use it;
- *Activate alarm*: Once the alarm is properly registered in the monitoring system, a TFS component can have some freedom to define when to check if a KPI is under the alarm conditions. To do so, the TFS component executes the RPC *GetAlarmResponseStream* by

passing the given *AlarmId* for a concrete time window and a concrete frequency (set in the *AlarmSubscription* message). This way, the Alarm Manager can periodically check the kpi values to see if the conditions in this time interval are violated. If so, the Monitoring Service responds to the TFS Component with the values and timestamps that overcome the alarm conditions. This process is repeated according to the *AlarmSubscription* for a concrete time window and frequency.

3.2.5. Evaluation

Based on the recent updates of the Monitoring component architecture, this performance evaluation is focused on the newer metrics database (QuestDB), aiming to evaluate its high performance and high scalability capabilities compared to the previous choice (InfluxDB). The detailed performance evaluation in this section has been assessed in a machine running Windows 10 and powered by an Intel Core i5-1145G7 and 16GB of RAM. For the databases, we have used the latest Docker containers available at that moment in the official repositories (QuestDB v6.5.5 and InfluxDB v2.5.1) running in the Windows official Docker Desktop app over the WSL 2 backend.

The initial tests of this performance evaluation aim to measure the time required by the databases to ingest the multiple samples. We will compare the ingestion time of InfluxDB and the multiple ingestion methods available in QuestDB, such as PostgreSQL, REST and Influx Line Protocol (ILP). This test will measure the time needed to ingest 10^2, 10^3 and 10^4 time ordered samples as well as out-of-order samples to measure the impact that ordering has in the data ingestion process.

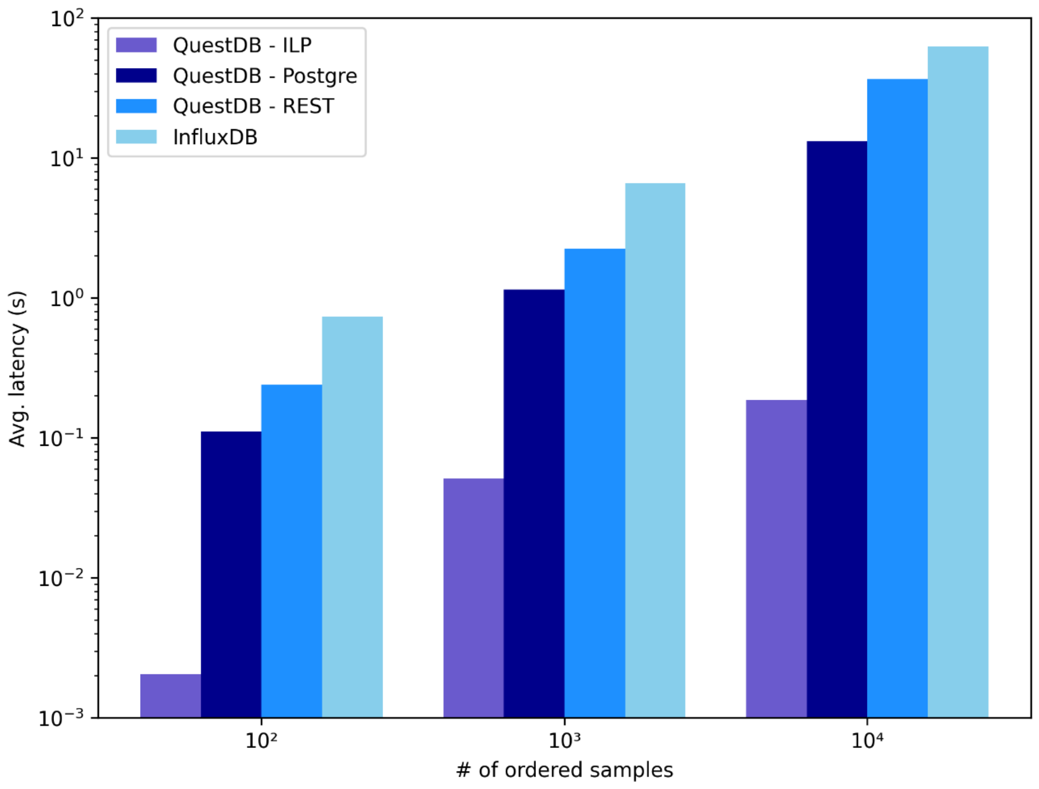


Figure 8: Average ingestion latency of time ordered samples

Figure 8 shows the average ingestion time of ordered samples after 100 executions of the test. The results confirm that QuestDB ingestion methods outperform InfluxDB, with PostgreSQL showing better performance than REST and achieving the best performance when using ILP, being more than two orders of magnitude faster than InfluxDB when ingesting 10^4 samples.

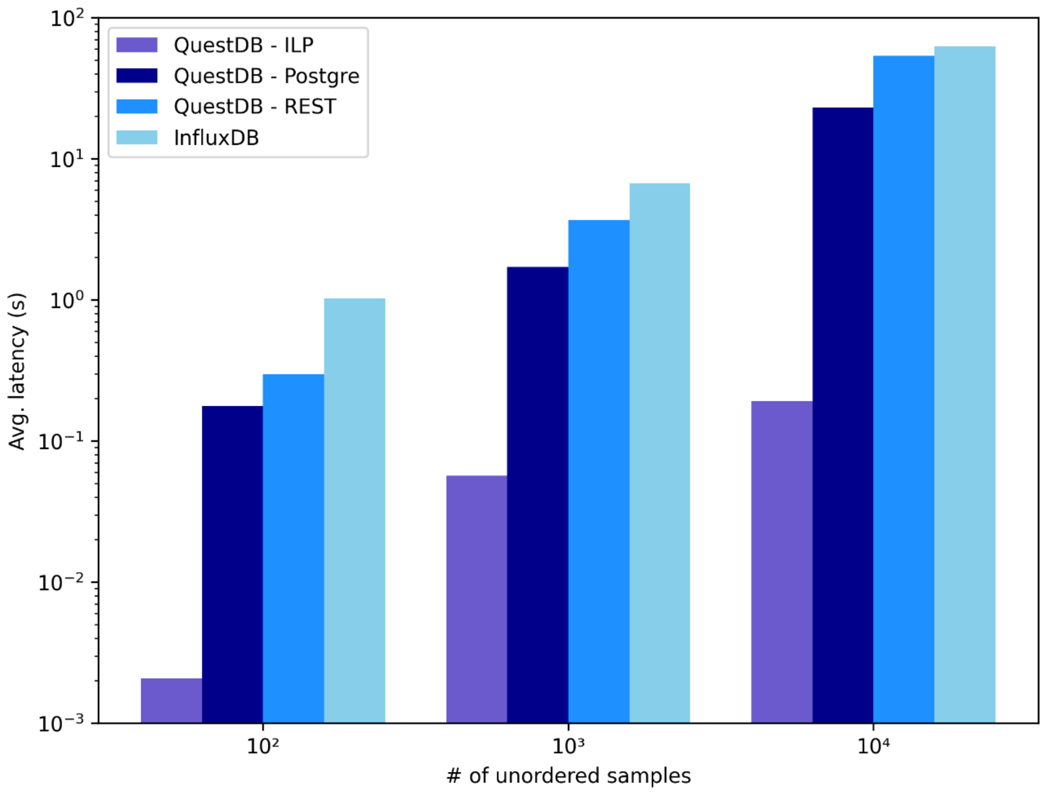


Figure 9: Average ingestion latency of time unordered samples

Figure 9 shows the second test results, focusing this time on ingesting unordered samples. For this test the table of the QuestDB has been partitioned by day, and the timestamp of the ingested samples has been randomly generated over the last 10^7 seconds, a timeframe big enough to force and stress the ingestion process to write data over a big number of table partitions. The results depicted in Figure 9 follow the same pattern than Figure 8, with QuestDB outperforming InfluxDB, especially when using ILP.

The next test will focus on query performance, measuring the time required to request 10^3, 10^4, 10^5 and 10^6 samples to the databases in a single query. We will compare the query latency of InfluxDB using the default query protocol of InfluxDB 2.X called Flux, and the supported query protocols by QuestDB, PostgreSQL wire protocol and REST. ILP will not be tested for this test since it is just an ingestion protocol and does not support queries.

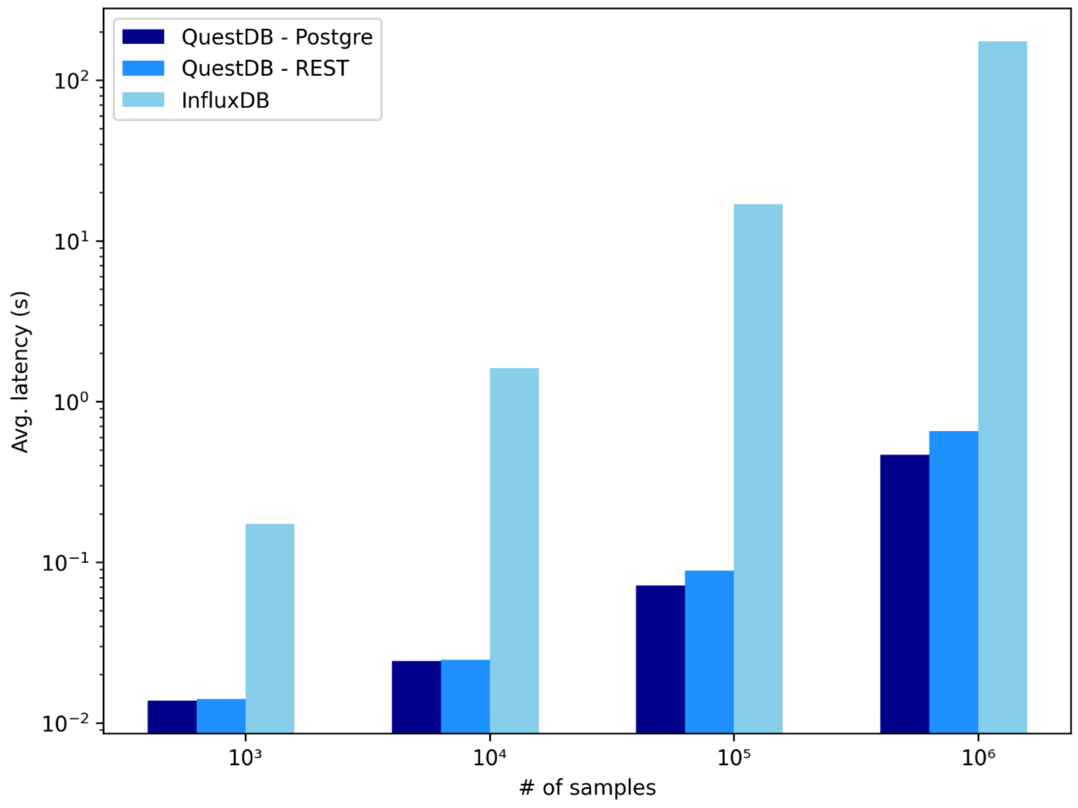


Figure 10: Average query latency

The results are depicted in Figure 10, showing the average query latency over 100 repetitions of the test. The performance is consistent with the results previously presented, with QuestDB having much better performance than InfluxDB, and PostgreSQL wire protocol being slightly faster than REST. Like in the previous evaluations, QuestDB shows more than two orders of magnitude better performance than InfluxDB for the bigger queries.

Once we tested the ingestion and query performance of isolated operations we decided to repeat the same tests but in this case with multiple simultaneous operations, testing the scalability of the databases when multiple concurrent access is required. The following concurrency tests are based on executing multiple processes (10 and 100), ingesting and querying multiple samples per process simultaneously to measure the scalability of the databases. The objective of these tests is to measure the total time needed to finalize all the operations of all the simultaneous processes.

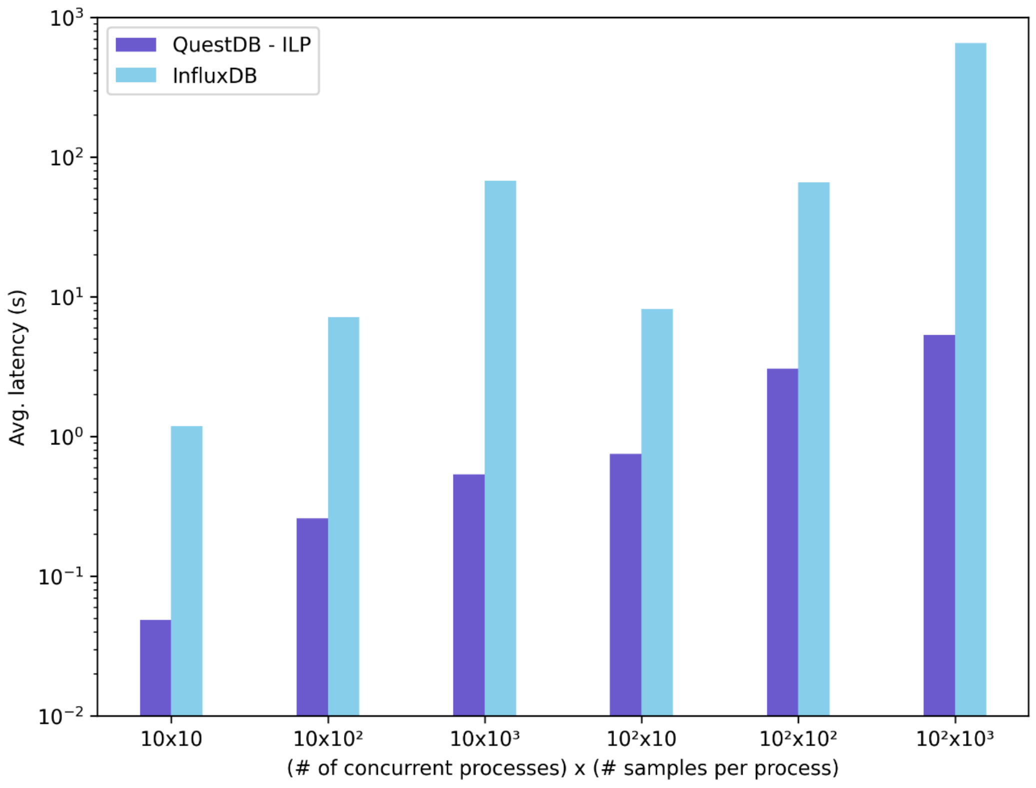


Figure 11: Average ingestion latency with multiple concurrent processes

Figure 11 shows the results of the concurrent ingestion tests. The X-axis represents the number of concurrent processes multiplied by the number of samples ingested per process, the Y-axis depicts the average latency over 100 repetitions needed to ingest all the samples from all the concurrent processes. Although, based on the results obtained from Figure 8 and Figure 9, we decided to ingest only unordered samples since it is the most stressing operation for the databases, we have also decided only to use ILP as the ingestion protocol for the QuestDB since it is by far the fastest ingestion method. The results again show that QuestDB outperforms InfluxDB, having a minimum difference higher than one order of magnitude, being even higher than two orders of magnitude when ingesting 10^3 samples per process.

Figure 12 depicts the results of the concurrent query evaluation. Analogously, the figure represents the number of concurrent processes multiplied by the number of queried samples per process in the X axis, while in the Y axis represents the average latency over 100 executions required to query the total number of samples from all the simultaneous processes. Similarly, to Figure 10 we have tested PostgreSQL wire protocol and REST for the QuestDB and Flux for the InfluxDB. The results are consistent with the previous ones, having by a little difference the best results with the QuestDB PostgreSQL wire protocol followed by QuestDB REST, with both protocols clearly outperforming InfluxDB. It is worth mentioning that InfluxDB could not handle the load of the last two tests, throwing multiple exceptions of ReadTimeoutError making it impossible to finalize the tests.

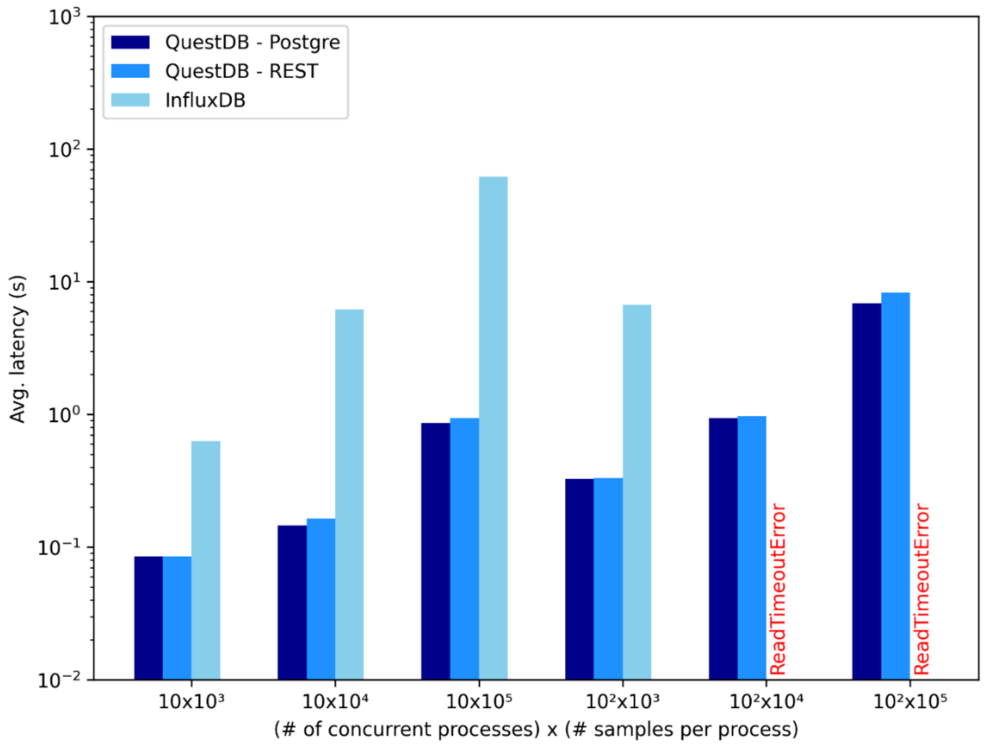


Figure 12: Average query latency with multiple concurrent processes

In light of the results mentioned above, it is fair to conclude that QuestDB offers far superior performance and scalability capabilities than InfluxDB, a more suitable option to fulfil the requirements of the TeraFlowSDN controller.

QuestDB offers multiple configuration parameters that can affect to the performance, in the following lines we will analyze some of these parameters to optimize the performance for the TeraFlowSDN controller.

As previously demonstrated, Influx Line Protocol (ILP) is the best ingestion method; it is the recommended method in the QuestDB documentation [26] for high-performance applications. ILP is a text protocol that runs over TCP, it is a one-way protocol just devoted to inserting data. ILP was originally created for InfluxDB, QuestDB has adopted it with a minor modification: QuestDB uses ILP as both serialization and the transport format, InfluxDB on the other hand uses HTTP as the transport and ILP as serialization format. For this reason, the InfluxDB client libraries do not work with QuestDB. QuestDB incorporates some mechanisms for ILP for reducing congestion and optimizing the insertion of out-of-order data. The data insertion process in ILP can be divided in two different phases:

- Ingestion: when the data is received it is firstly kept in memory only, invisible for the queries. Then, the QuestDB out-of-order algorithm detects, process and optimizes the data for the next insertion phase, the commit;
- Commit: once the data is ingested and ordered by the out-of-order QuestDB engine it is pushed to the database and becomes visible for the queries.

The most expensive task to perform in the insertion process is the commit, hence for optimal performance the data commits should be minimized as much as possible, accumulating the data for a time period that will allow sorting of the collected samples before they are committed to the database. In order to optimize the insertion of data and control this waiting period QuestDB provides some configurable parameters in the server side, such as the *commit interval*, the *commit timeout* and the *maximum number of uncommitted rows*.

The *commit lag* is a value defined in milliseconds that has a significant impact on the insertion timing. The *commit interval fraction* is just a constant that multiplies the *commit lag* value resulting in the *commit interval*, the *commit interval* is the total waiting time applied each time a commit is performed to the database. Therefore, after this waiting period the data samples older than the *commit interval* value are committed and become visible for queries. The commit interval parameter should always be considered in conjunction with the *maximum number of uncommitted rows*, which refers to the maximum number of samples that can be ingested in memory without being committed to the database. As a result, the commit delay will be defined by the value of these two parameters that is reached faster. Consequently, we have evaluated the performance of QuestDB with different values of the *commit interval* and the *maximum number of uncommitted rows* parameters, testing multiple commit strategies.

The first commit strategy is the row-based commit, with this strategy the commit is performed when the number of uncommitted samples reaches the value of the *maximum number of uncommitted rows* parameter. The second commit strategy is the idle table timeout, when there is no data ingested after a period of time the database force a commit after a timeout interval, this commit timeout interval can be configured with two parameters the *idle ms before writer release* and the *maintenance job interval*, the *commit timeout* is the result of the summation of these two parameters. Finally, the third commit strategy is the interval-based, that is defined by the *commit interval*, as it is mentioned previously the *commit interval* is a function of the *commit lag* and the *commit fraction*.

Figure 13 depicts the average insertion latency of multiple number of samples over 100 repetitions with multiple values of the *commit interval*. The insertion latency is divided in the figure in two stages, the ingest time (in solid dark color) and the commit time (in light color with crosshatching). For this test we have fixed the *maximum number of committed rows* parameter with a value higher enough to ensure that is never reached, 10^6. The *commit interval fraction* is by default configured to 0.5, in this evaluation we have tested three values for the *commit lag*, 1, 10 and 100 milliseconds resulting in a *commit interval* of 0.5, 5 and 50 milliseconds. The figure shows an interesting and at the same time intuitive result, for small data insertions (10 samples) the database performs better with lower values of the *commit interval*. On the contrary, for big data insertions (10^5 samples) the database performs better with bigger values of the *commit interval* parameter, allowing more time to process the data and therefore reducing the number of commits, offloading the database.

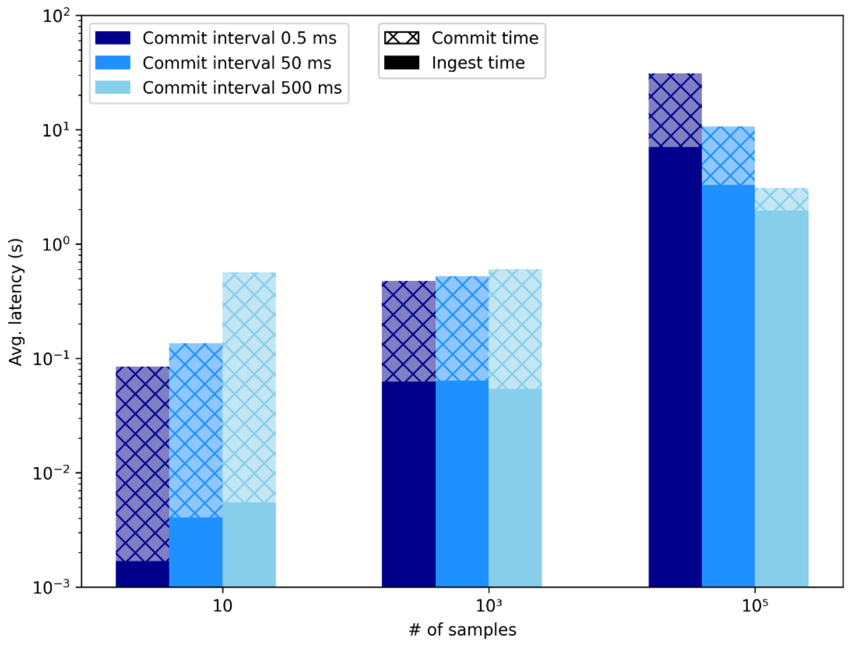


Figure 13: Insertion time with multiple commit intervals and a fixed maximum number of uncommitted rows

Finally, Figure 14 also shows the average insertion latency of the multiple numbers of samples over 100 repetitions, but in this case, varying the value of the *maximum number of uncommitted rows* parameter. Like in the previous figure 13, the insertion time is divided in ingestion time and commit time. For this test we have fixed the *commit lag* parameter with a value higher enough to ensure that is never reached, 10^5 milliseconds. The figure demonstrates that for the ingestion of a small number of samples (10) the commit strategy is based on the *commit timeout*, which is the reason why the three bars have approximately the same commit time. On the contrary, for big data insertions (10^5 samples) the database performs much better with bigger values of the *maximum number of uncommitted rows* parameter since the number of commits is minimized.

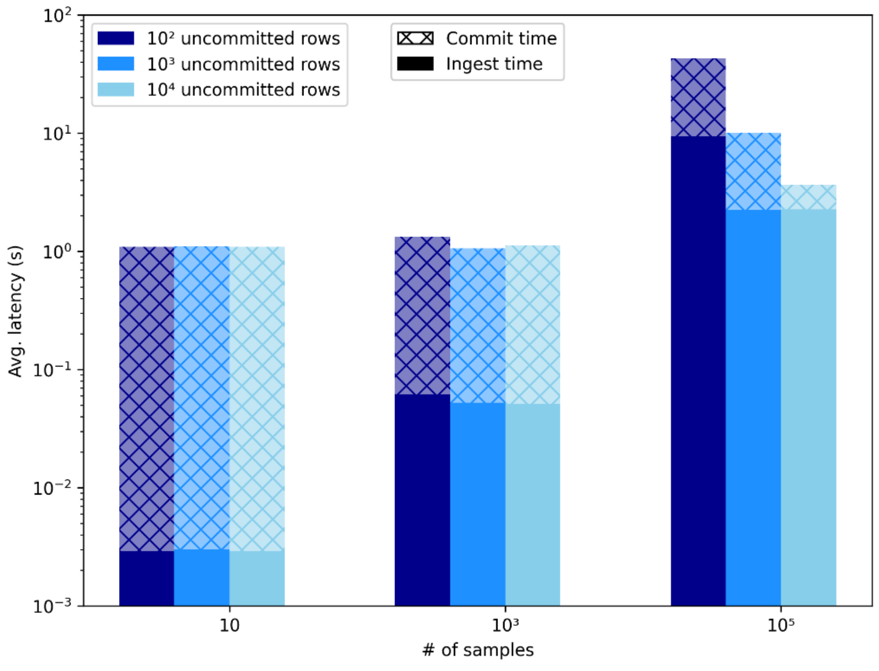


Figure 14: Insertion time with multiple maximum number of uncommitted rows values and a fixed commit Interval

In the light of the above results, the QuestDB have been configured with a *commit lag* of 1000 milliseconds and a *maximum number of uncommitted rows* of 10000, these are the default values adopted for TeraFlowSDN and the ones used in the tests performed for the first 5 figures of this subsection and also in the forthcoming test.

Once we evaluated the performance of QuestDB [26], verifying that is a much faster and scalable solution than InfluxDB, we aimed to evaluate the performance of the full Monitoring component, including the gRPC server. For the following experiments, we have executed the full Monitoring microservice and tested the gRPC methods using the Monitoring gRPC client from the same host machine.

Therefore, since the ingestion is the most stressing operation, we decided to evaluate the IncludeKpi gRPC method, which as can be seen in Section 3.2.3 is the method in charge of receiving the monitoring samples through gRPC and ingesting them into the MetricsDB (QuestDB). For this evaluation we decided to measure the latency overhead introduced by gRPC by measuring the ingestion time of 1 sample directly in the QuestDB through ILP and using the IncludeKpi gRPC method using the monitoring gRPC client. This experiment has been repeated 100 times and the results are depicted in Figure 15, as can be seen in the CDFs the QuestDB standalone ingestion takes an average time of 7.6 ms, while the IncludeKpi method (which involves the gRPC as well as the QuestDB) shows an average ingestion latency of 11.9 ms. This figure demonstrates that the overhead introduced by the gRPC server is minimum, around 4.3 ms on average.

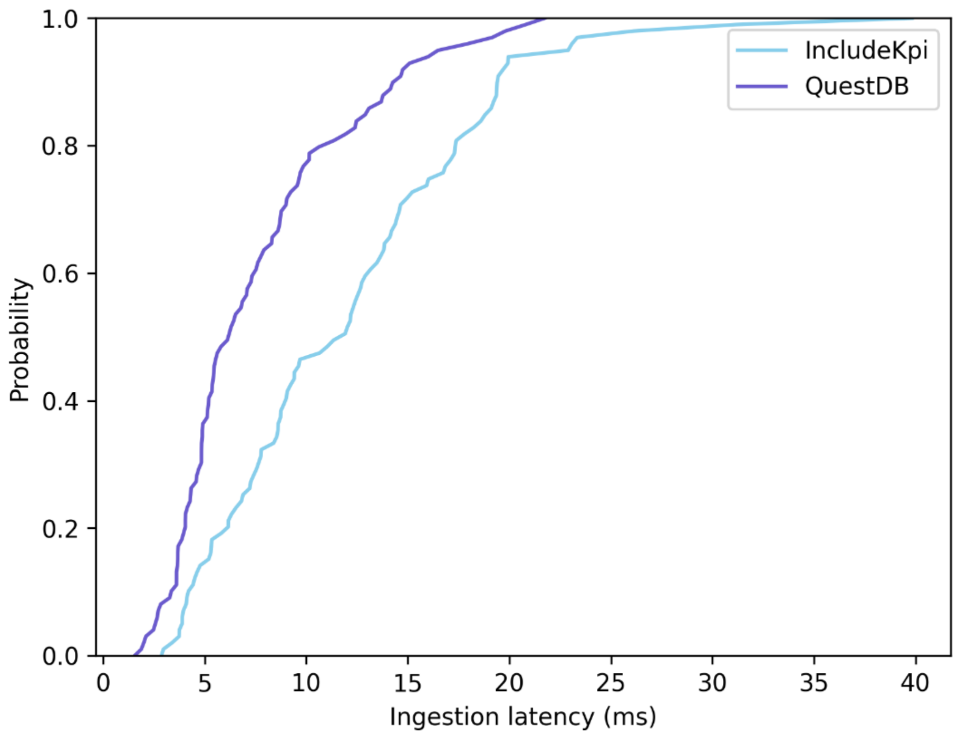


Figure 15: evaluation of the ingestion latency of the IncludeKpi gRPC method

Finally, to measure the overhead introduced by gRPC for data querying, we evaluated the performance of the QueryKpiData method. For this evaluation, we measured the time needed for querying the last 10^3 samples of a given KPI directly to the QuestDB and through the QueryKpiData gRPC method using the Monitoring client. This test has been repeated 100 times and the results are detailed in Figure 16. In this case the overhead latency introduced by gRPC is noticeably higher due to the larger amount of transmitted data than the previous experiment (1 sample vs 10^3 samples). The average latency obtained for querying 10^3 samples using the QueryKpiData gRPC was 105.4 ms compared to 13.7 ms when directly querying the data to the QuestDB. As a result, in this experiment, the average latency introduced by the monitoring component and the gRPC server was 91.7 ms. The value is still a very low number considering the amount of data that is being transmitted, and is still lower than the query latency results experienced when directly querying the InfluxDB (around 172 ms in average), as can be seen in the evaluation performed previously in.

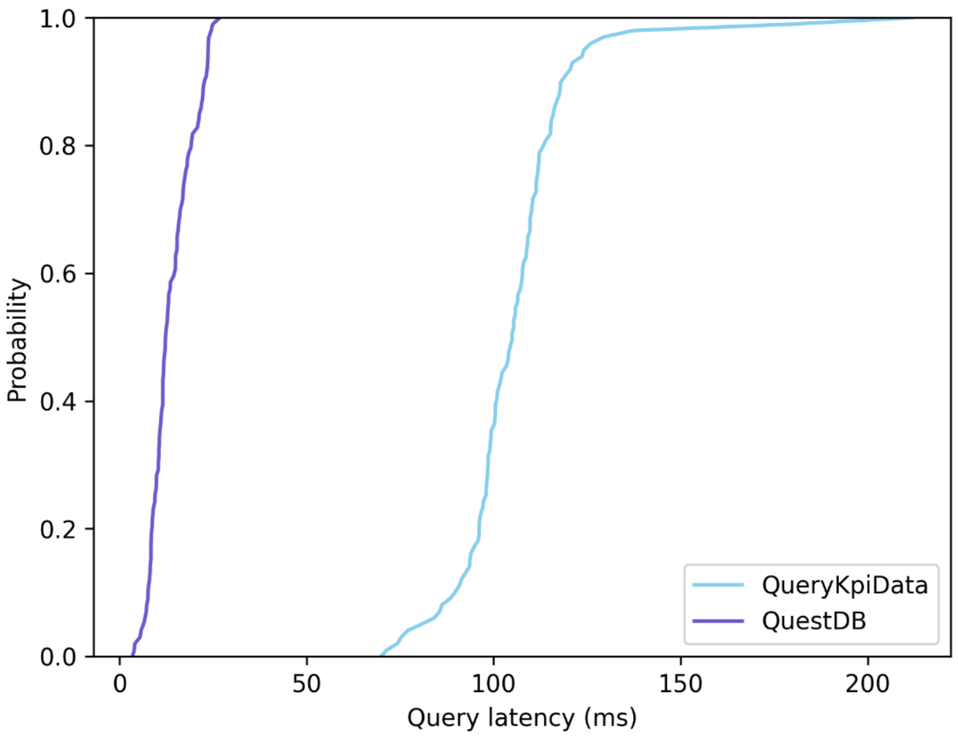


Figure 16: evaluation of the query latency of the QueryKpiData gRPC method

Ingesting and querying are the two main and basic operations of the Monitoring component, involved in the data monitoring, alarming and subscription functionalities. Therefore, we can affirm that these results are sufficient to evaluate the overall performance of the functionalities offered by the TeraFlowSDN Monitoring component.

3.3. Traffic Engineering Component

The Traffic Engineering Component (TE) provides Segment Routing (SR) path computation over the compatible infrastructure exposed by the Device component. It consolidates the devices in a Traffic Engineering Database (TED) from the information exposed by the Context component. In addition, it maintains and synchronizes the SR Label Switched Path (LSP) received from the devices and the ones it creates when the Service Components request it. The communication channel between the TE component and the other TeraFlow components (Service, Context and Device) is GRPC. Path Computation Element Protocol (PCEP) is the protocol communication channel with the devices (client) and TE component (server).

3.3.1. New Features/Extensions

The new release 2.0 component has been completely reengineered to provide complete integration and support for traffic engineering using path computation element protocol (PCEP). Moreover, the new features include:

- Integration with Service, Device, and Context components;
- TE workflow demonstration using emulated routers;
- Segment Routing paths are created using PCE-initiated LSPs.

3.3.2. Final Design

The TE component comprises two sub-applications: the TeraFlowSDN part which is in charge of all the TeraFlow-specific logic, and the PCE part and handles all the generic path computation element logic (as shown in Figure 17). The term “application” is derived from how Erlang Open Telecom Platform (OTP) software packages are named.

- **TeraFlowSDN Application**:
  - **GRPC Handler**:
    Responsible for talking gRPC to other services and handling gRPC requests.
  - **API Server**:
    Responsible for the high-level TeraFlowSDN API and logic. It contains both the handler for the TE service API and the proxy process to handle the other components API and events.
  - **Smart Resolver**:
    Responsible for resolving segment routing path given a set of constraints.
- **PCE Application**:
  - **TED**:
    The traffic engineering database built from the context topology. It will be used by the resolver to resolve a segment routing path.
  - **LSP DB**:
    The LSP database reflecting all the LSP created on the connected routers. It will be used by the resolver to resolve a segment routing path.
  - **PCE Server:**
    Expose the PCE API to the routers and implements the PCEP protocol.

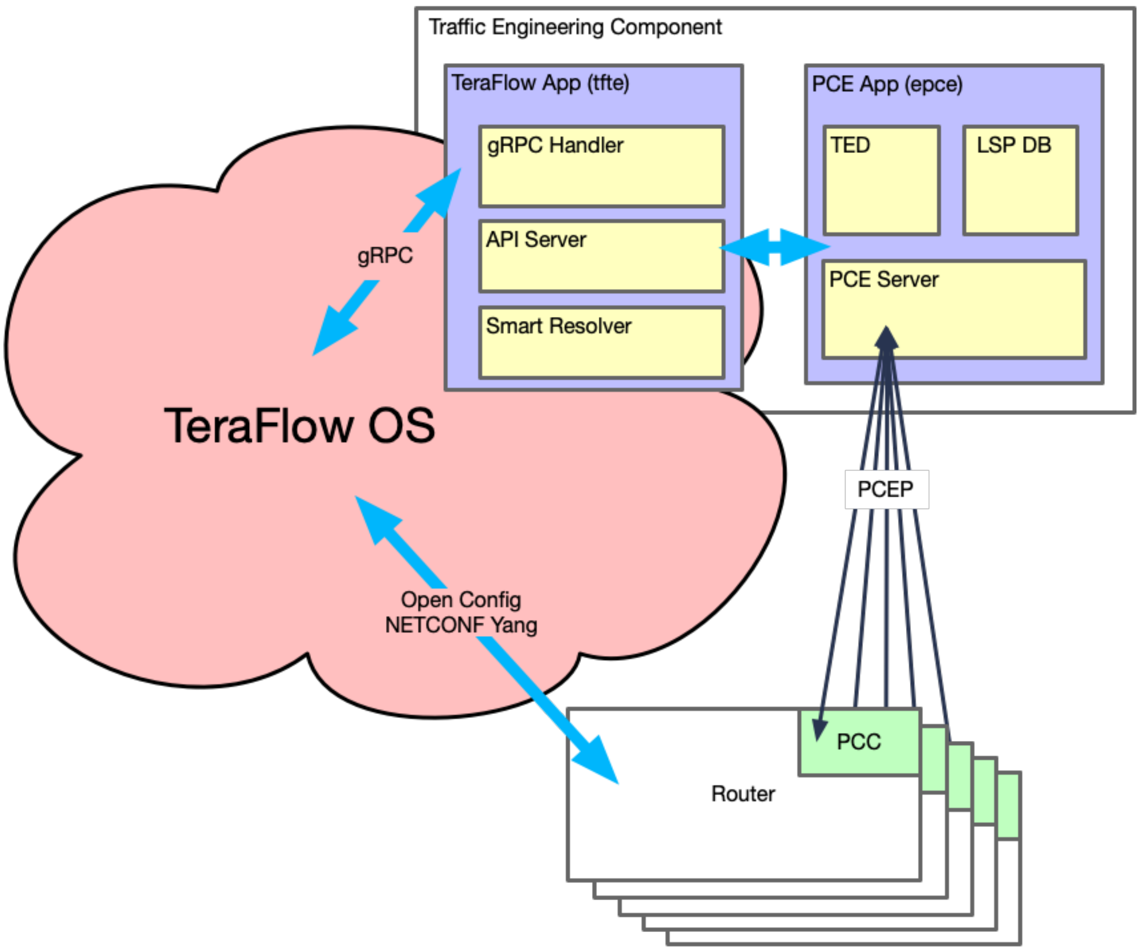


Figure 17: High-level design of the Traffic Engineering component

As an Erlang Open Telecom Platform service, the TE component is designed for high concurrency and high availability. Every process (worker) is supervised and part of a supervision tree. If case of an unrecoverable error, the process is crashed, and its supervisor restarts it from a known working state, as shown in Figure 18.

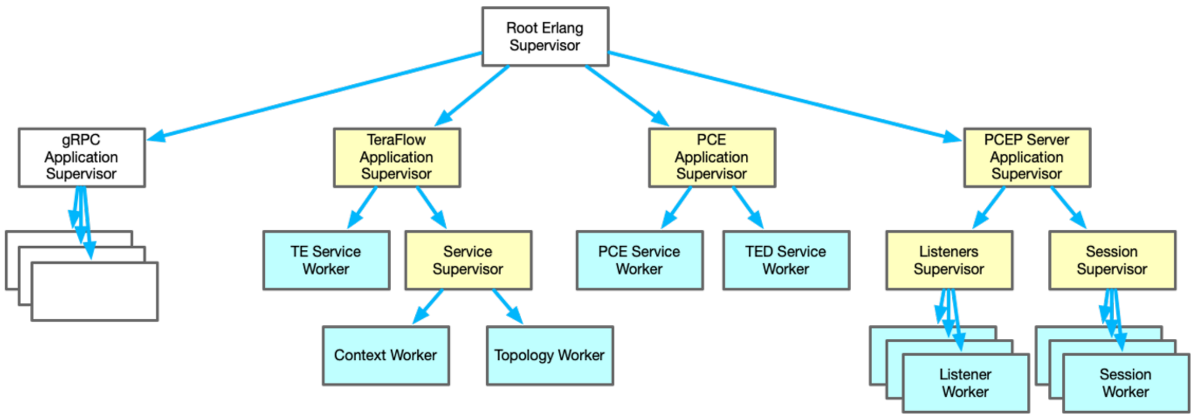


Figure 18: TE Component Supervision Tree

3.3.3. Final Interfaces

The final interface for the Traffic Engineering Component is described in Table 13. A Service Component calls the gRPS method *RequestLSP* to setup a segment routing path. All the information required for setting up the flow comes from the service configuration and the TED created from the context's topology. If a Service Component configuration changes, it can explicitly request the flow to be updated by calling UpdateLSP. Calling DeleteLSP will remove the flow from all involved routers when the service is terminated.

RPC Method Name	**Parameters**	**Results**
RequestLSP	Service	ServiceState
UpdateLSP	ServiceId	ServiceState
DeleteLSP	ServiceId	-

Table 13 TE component interface methods

3.3.4. Final Operational Workflows

This section exposes the workflow for the principal traffic engineering operations, as shown in Figure 19.

When starting the TE component, it will request the topology from the Context component, and then retrieve all the devices and links in the topology to build the traffic engineering database. It will then register for context events to update the TED when something changes.

To create a new segment-routing flow, the Service component call the *RequestLSP* method. The TE component resolves the path taking into account the service configuration, the traffic engineering database and the LSP database, it creates a new LSP and initiates it on the relevant router using PCEP.

When the Service component is terminated, it requests the removal of the segment routing LSP by calling the *DeleteLSP* method, and the TE component will update the LSP database and remove the LSP using PCEP.

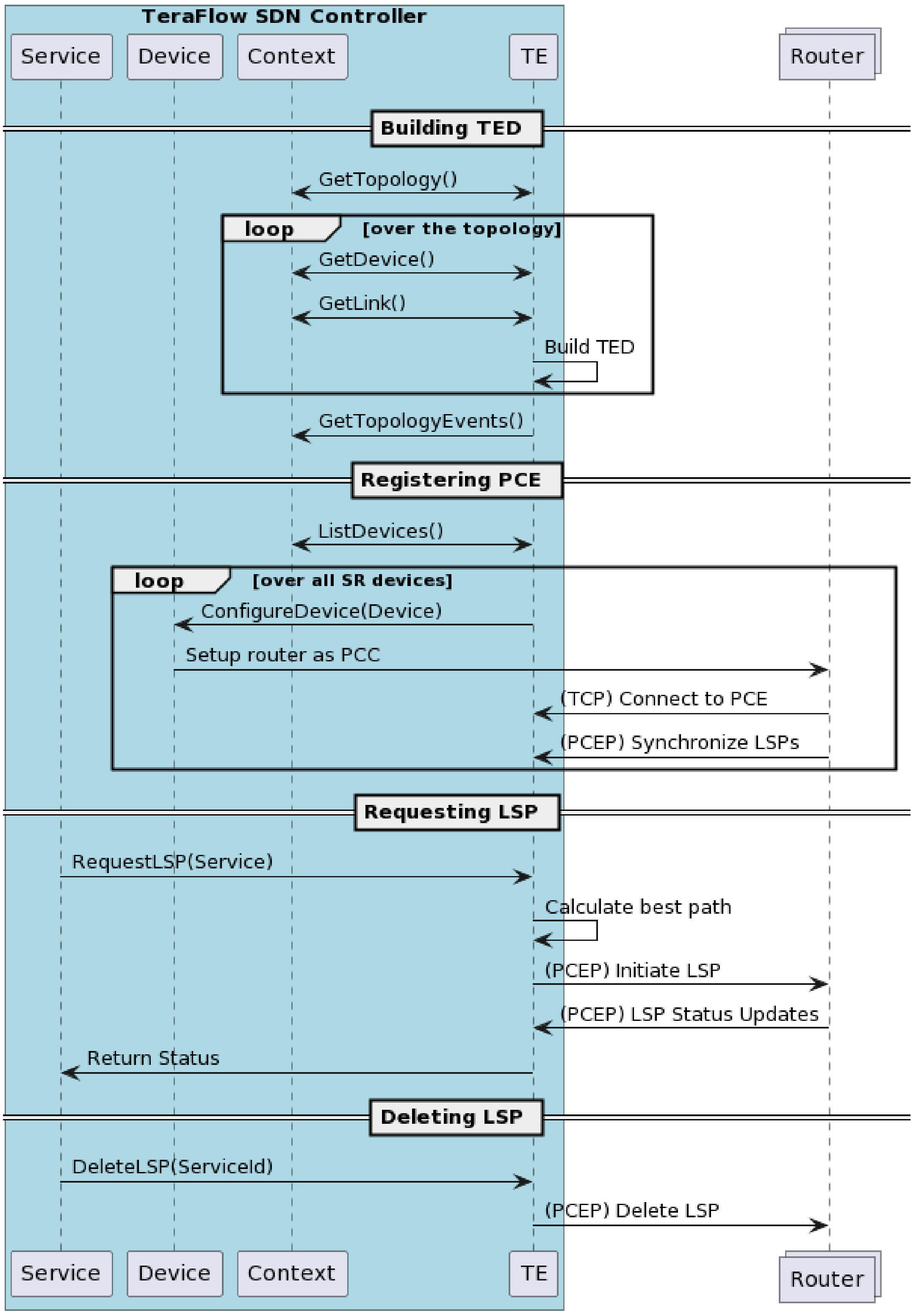


Figure 19: Traffic Engineering sequence diagram

3.3.5. Evaluation

The evaluation was made in a virtual machine and with a virtualized network setup with Linux namespaces and Free-Range Routing services for the IS-IS network and the PCEP clients.

Due to a discovered issue in the PCEP implementation of FRR, it has not been possible to benchmark the concurrent creation of multiple flows. However, we will investigate and notify our findings on the issue.

We measured the time to setup a unidirectional segment routing flow, and the time to update an already existing flow. The time is stable and mostly comprised of the FRR process to set up the segment routing path. Scalability in this context is highly dependent on the underlying network infrastructure.

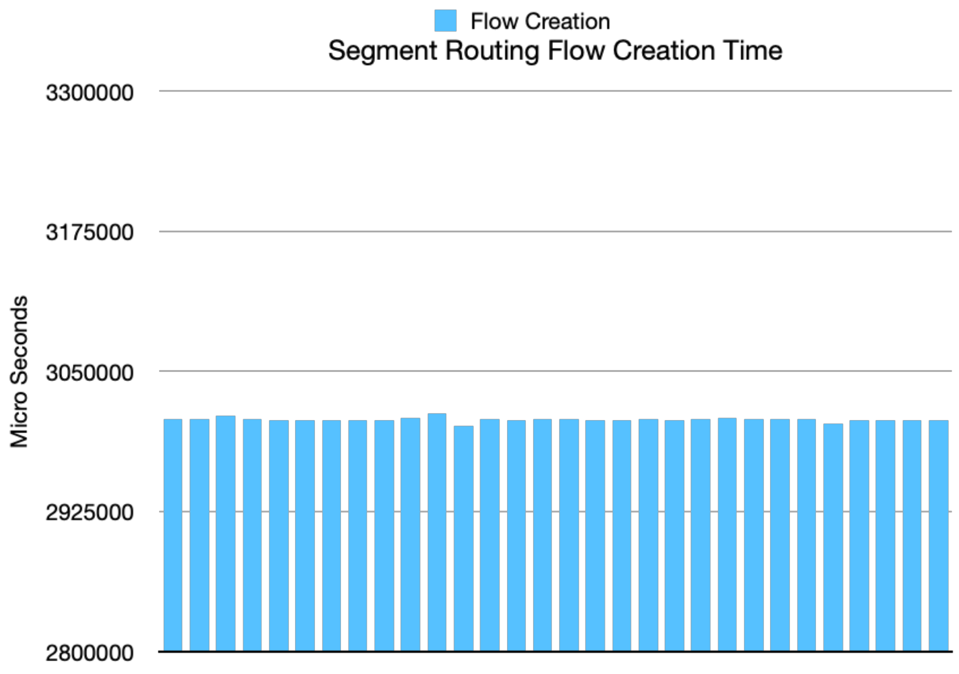


Figure 20: evaluation of the segment routing creation time

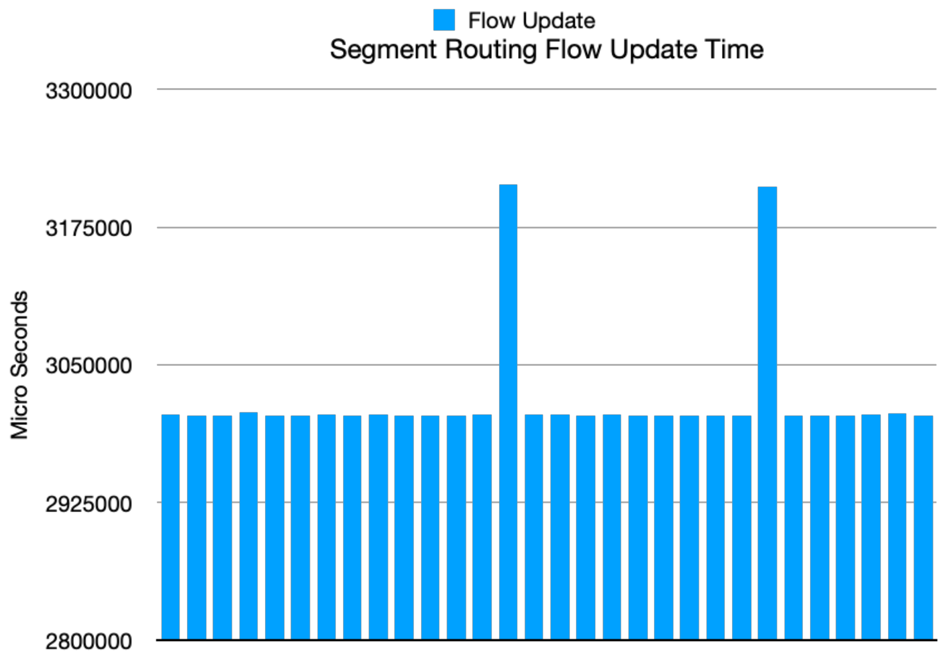


Figure 21: evaluation of the segment routing update time

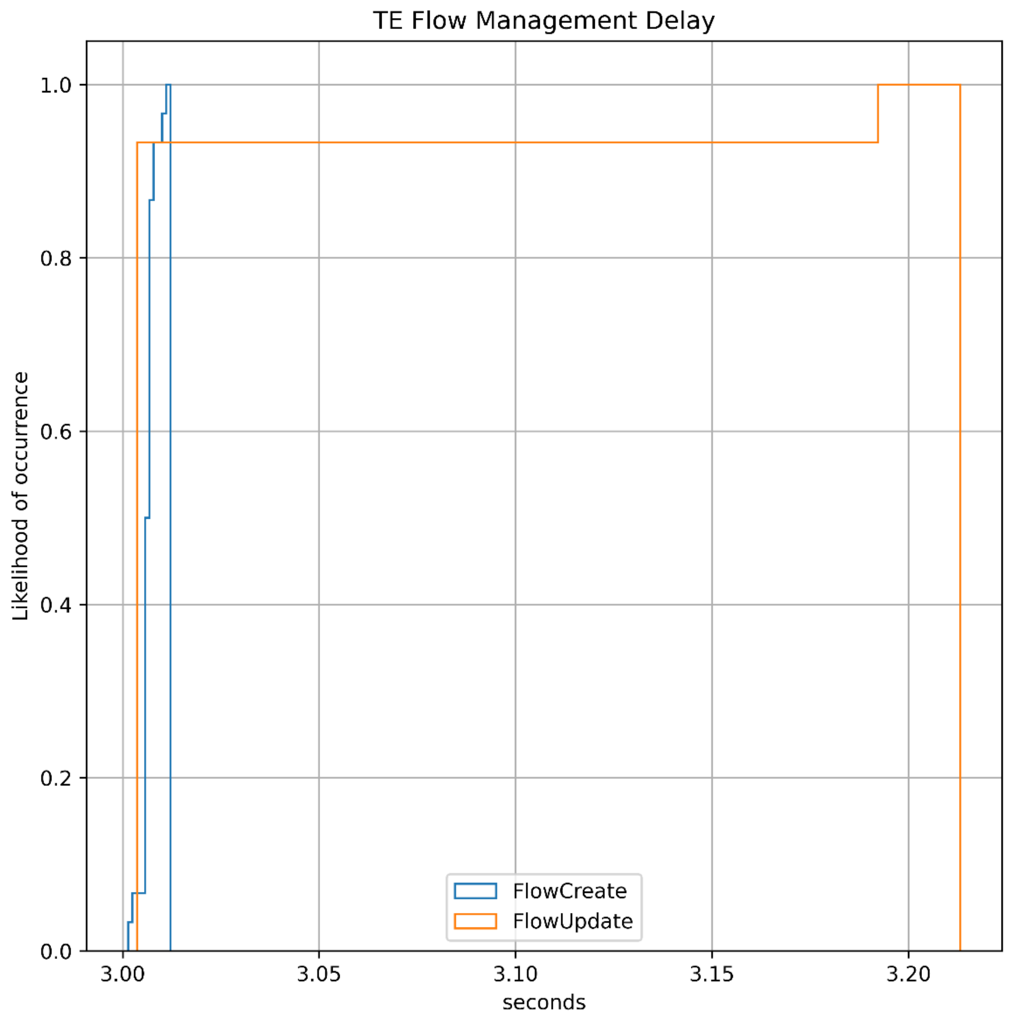


Figure 22: evaluation of the segment routing flow management delay

3.4. Path Computation Component

The Path Computation Component (PathComp) is designed and implemented to offer a dedicated and standalone entity for computing and to select the network resources when creating or modifying new/existing network connectivity services. Note that this component was not introduced in the former TeraFlow OS components reported in the previous D3.1 [5] [1].

The PathComp is fed by/relies on a set of inputs: i) the Context information entailing the topology/domain devices, links, attribute, existing services, etc.; ii) the targeted network service connectivity's characteristics specifying the network endpoints (devices and ports) and demanded requirements to be met in terms of guaranteed bandwidth, maximum latency, disjoint paths, etc.; and iii) a specific performance objective to be optimized by the PathComp, e.g., attaining an efficient use of the overall network resources, minimizing the total amount of consumed network power/energy, etc.

It is worth mentioning that the PathComp is designed to compute the path and resources for requests carrying either a single network connectivity service or multiple services. The latter is interesting when trying to restore diverse services affected by a network anomaly (e.g., link failure) or reoptimising the allocated resources (e.g., to reduce the overall network consumed power). For each service, the output of the PathComp for a successful computation describes: i) the set of devices and links forming the path; ii) specific switching/technology configuration parameters (e.g., the label in L2, optical frequency in L0, etc.). If the PathComp does not find a feasible path / resources to accommodate a service fulfilling the requirements, a response informing about that is sent by the PathComp.

3.4.1. New Features/Extensions

For the design and implementation of the PathComp the following characteristics have been considered:

- The aim is to concentrate in a specialized component any path computation and resource selection functions encompassing single or multiple domains, heterogeneous technologies forming the underlying infrastructure, etc.
- The PathComp operates as a server that can host a diverse pool of algorithms targeting several objective functions such as K-Shortest Path, Shortest Path, Energy-Aware Routing, etc.
- The PathComp exposes a defined interface and workflow to the rest of TeraFlow OS Components to compute paths and select the network resources. In this regard, the main component interacting with the PathComp is the Service but others may also use it when required. However, depending on the necessities, the Automation, Slice, or Inter-Domain Components could leverage the PathComp functionalities.

3.4.2. Final Design

The architecture of the PathComp is depicted in Figure 23, which is formed by two main elements: the Front-End and the Back-End Path Computations.

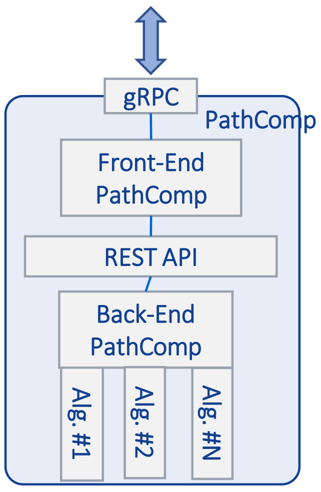


Figure 23: Architecture of the PathComp component.

- The Front-End PathComp handles the interactions with other TeraFlow OS components (e.g., Service, Automation, Slice, or Inter-Domain) to receive and process (new or update) connectivity service requests using the defined gRPC API (described below). Additionally, the Front-End PathComp can also trigger interactions with other components (e.g., Context or Monitoring) to retrieve information required or interesting to conduct the path computation and resource selection functions. The Front-End then performs the mapping between the gRPC messages to local commands based on a defined REST API (with JSON encoding). By doing so, the Front-End PathComp requests the actual computing execution to the Back-End module.
- The Back-End PathComp hosts the pool of devised Algorithms (as shown in Figure 23) to be executed. Thus, upon request (from the Front-End) a specific algorithm is triggered whose output is the returned.

The separation between the Front-End and Back-End modules is done to isolate both operations/functions: i) the interactions between the PathComp with TeraFlow SDN controller Components; and ii) the actual path computation and resource selection process which may result in an intensive computational process. By doing so, a 1: *N* Front-End / Back-Ends relationship could be built aiming at fostering the scalability and dynamism of the overall PathComp solution.

3.4.3. iFinal Interfaces

The PathComp component has two interfaces: i) an external one based on gRPC to interact with other TeraFlow OS components such as Service, Context, Automation, Slice, or Inter-Domain; ii) an internal one based on a REST API.

- **gRPC Interface**

The gRPC interface is offered to the rest of TeraFlow OS components for the sake of requesting and responding path computations and resource selections to the PathComp. Moreover, the PathComp leverages a method supported by the Context component to retrieve the information related to the topology, devices, links, etc. In Table 14, it is shown the specific rpc method supported by the PathComp where in brief, the set of services and a targeted algorithm are required to trigger the computations in the PathComp entity.

Table 14: gRPC PathComp interfaces

RPC Method Name	Parameters	Results
PathCompRequest	Services, Algorithm	Service
GetContext	ContextId	Context

- **REST Interface**

As mentioned above, the Front-End and Back-End modules of the PathComp component use a REST API to request/respond to the explicit path computations carried out in the Back-End module. The encoding is JSON-based. Three methods are supported over this interface as shown in Table 15. The GET method allows the Front-End querying/monitoring about the status of the Back-End entity, i.e., for “health” matters. More importantly, the POST and PUT methods enable the Front-End to request the computations for either a single or diverse network connectivity services. To this end, recall that it is passed the Context information along with the targeted algorithm identifier, and the individual service features (i.e., endpoints, constraints to be met, etc.). If the Back-End module succeeds in computing the paths for the requested services, the resulting devices, links, network resources are formatted in the response to be eventually configured/programmed by the TeraFlow OS. Note that the difference between using the POST or PUT messages is that in the first, the Back-End processes new incoming network connectivity services (i.e., not deployed), whilst in the PUT method, the Back end handles the re-computation of existing service, e.g., for re-optimization purposes. Thus, it is expected that in the PUT method additional information is provided about the currently occupied resources by the services to be re-computed by the Back-End module.

Table 15: REST API for the Front-End – Back-End modules of the PathComp

Method	**Endpoint URL**	**Results**
POST	/pathComp/api/v1/compRoute	Computed path and network resources
PUT	/pathComp/api/v1/compRoute	Computed path and network resources
GET	/pathComp/api/v1/health	OK with the status of the Back end PathComp element

3.4.4. Final Operational Workflows

Figure 24 depicts the workflow for requesting the path computation and resource selection for a new connectivity service. The request (PathCompRequest) is sent by the Service component specifying the list of services with their characteristics such as endpoints, bandwidth and latency needs, etc. In this request, it is also specified the targeted algorithm to be executed. Otherwise, the PathComp triggers a default algorithm. The PathComp then retrieves the context information which used as input information to seek a feasible path and resources. Within the PathComp, for each demanded service, the selected algorithm is executed iteratively. To this end, a local copy of the context information is always updated before triggering computation for the following service. Finally, the resulting path computation and selected resources are notified to the Service component via the PathCompReply message

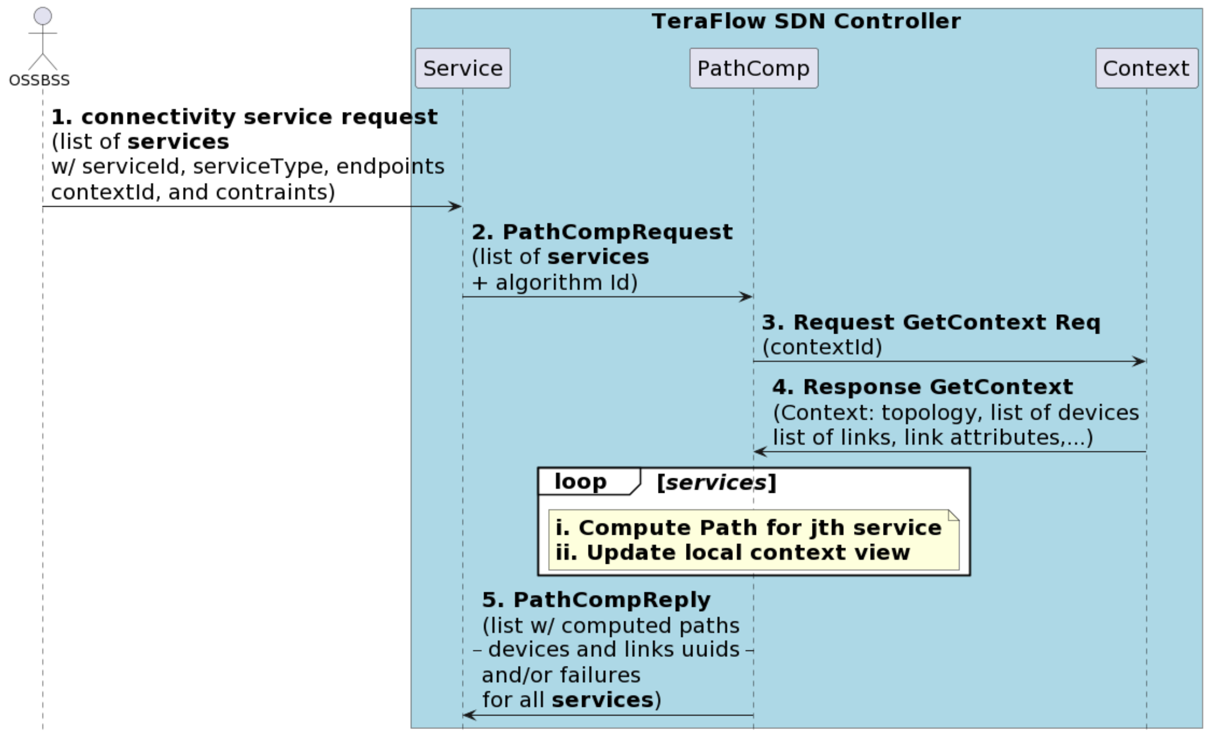

TeraFlow SDN Controller
OSSBSS
Service
PathComp
Context
1. connectivity service request (list of services w/ serviceId, serviceType, endpoints contextId, and contraints)
2. PathCompRequest (list of services + algorithm Id)
3. Request GetContext Req (contextId)
4. Response GetContext (Context: topology, list of devices list of links, link attributes,...)
loop [services]
i. Compute Path for jth service
ii. Update local context view
5. PathCompReply (list w/ computed paths – devices and links uuids – and/or failures for all services)

Figure 24: Workflow between Service and PathComp Components to request/respond to a new incoming network connectivity request

3.4.5. Evaluation

The evaluation of the PathComp component is tackled from both perspectives: functional addressing details of the control interface and protocol data, and from a more numerical view where it is shown the CDF of the PathComp delay when serving a set of dynamic network services and slices.

Functional Validation

In this part, it is described, and contents request message received from the Back-End element to trigger the route computation and resource selection for a specific set of network connectivity services. Figure 25, it is shown the contents of the REST API-based request (POST method) sent by the Front-End and processed by the Back end. The contents are divided into three blocks:

- Service request: specifies service attributes and computation needs such as the "pre-selected" algorithm (e.g., K Shortest Paths – KSP) to be triggered along with other arguments (e.g., maximum K value), the service identifier, the service endpoints (i.e., device and link identifiers), and the demanded requirements / constraints to be fulfilled (e.g., the bandwidth and the maximum end-to-end latency);
- Device List: contains the set of devices forming the transport infrastructure. For each device, it is described attributes such as the context and topology identifiers, the type (e.g., router), the endpoints (to connect to either other devices or terminating devices), and other characteristics like the consumed power upon *idle* state (needed for energy-aware routing);
- Link List: contains the set of links interconnecting the devices. For each link, it is described the endpoints being connected along with required attributes (e.g., delay, cost, available bandwidth, etc.).

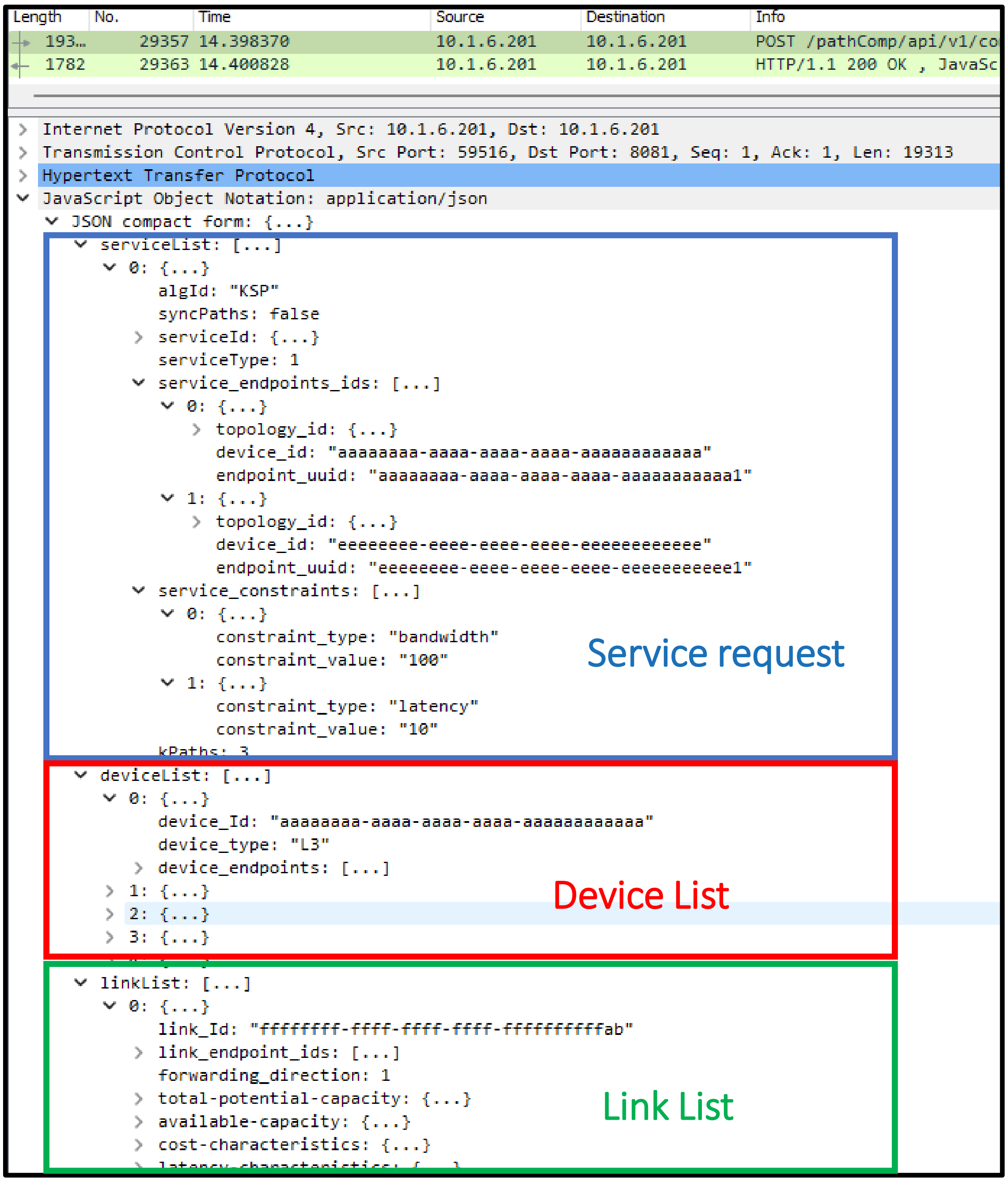
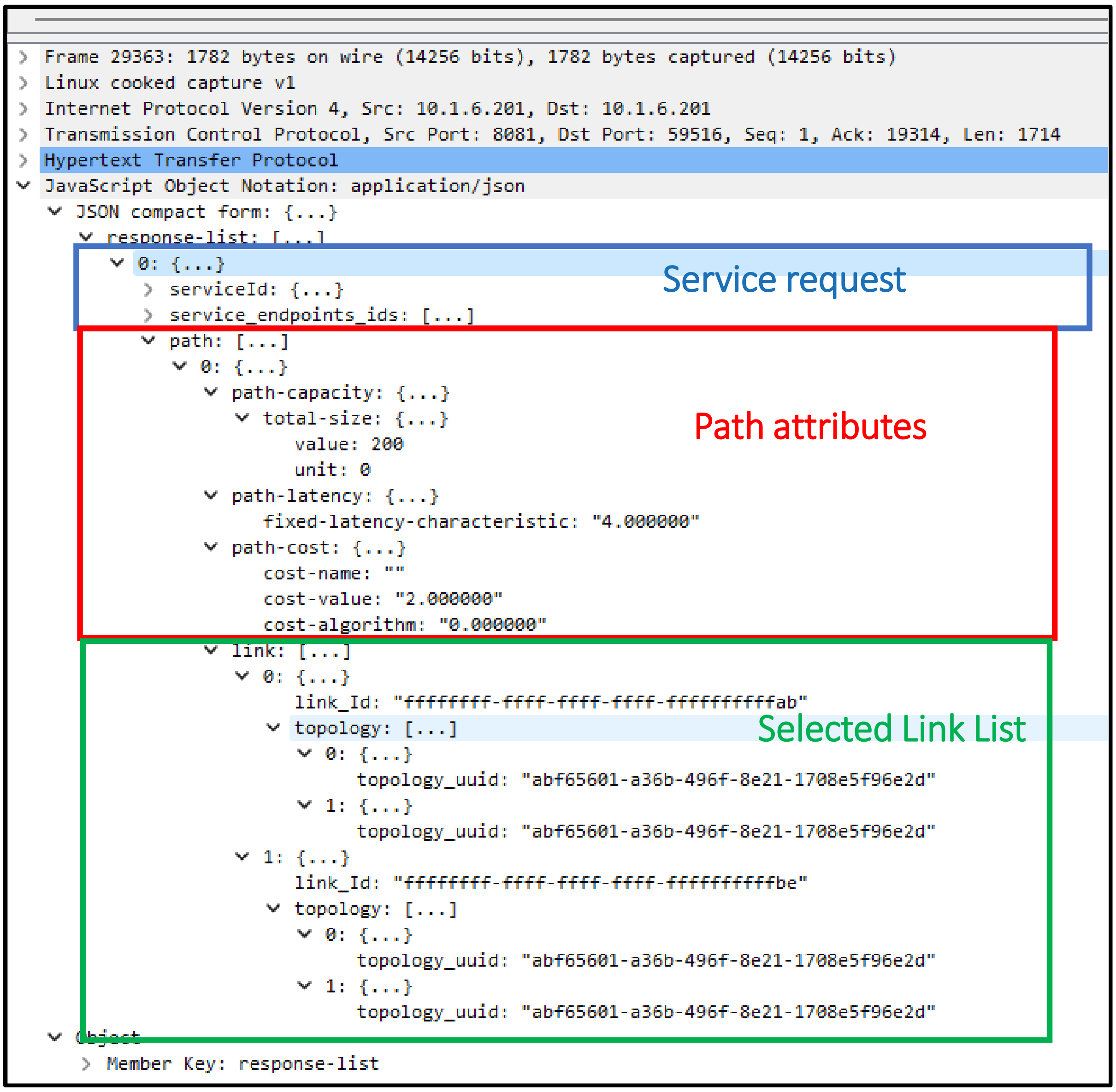

```
Length  No.    Time         Source       Destination  Info
193…    29357  14.398370    10.1.6.201   10.1.6.201   POST /pathComp/api/v1/co
1782    29363  14.400828    10.1.6.201   10.1.6.201   HTTP/1.1 200 OK , JavaSc

> Internet Protocol Version 4, Src: 10.1.6.201, Dst: 10.1.6.201
> Transmission Control Protocol, Src Port: 59516, Dst Port: 8081, Seq: 1, Ack: 1, Len: 19313
> Hypertext Transfer Protocol
v JavaScript Object Notation: application/json
  v JSON compact form: {...}
    v serviceList: [...]
      v 0: {...}
          algId: "KSP"
          syncPaths: false
        > serviceId: {...}
          serviceType: 1
        v service_endpoints_ids: [...]
          v 0: {...}
            > topology_id: {...}
              device_id: "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
              endpoint_uuid: "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaa1"
          v 1: {...}
            > topology_id: {...}
              device_id: "eeeeeeee-eeee-eeee-eeee-eeeeeeeeeeee"
              endpoint_uuid: "eeeeeeee-eeee-eeee-eeee-eeeeeeeeeee1"
        v service_constraints: [...]
          v 0: {...}
              constraint_type: "bandwidth"
              constraint_value: "100"
          v 1: {...}
              constraint_type: "latency"
              constraint_value: "10"
          kPaths: 3
    v deviceList: [...]
      v 0: {...}
          device_Id: "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
          device_type: "L3"
        > device_endpoints: [...]
      > 1: {...}
      > 2: {...}
      > 3: {...}
    v linkList: [...]
      v 0: {...}
          link_Id: "ffffffff-ffff-ffff-ffff-ffffffffffab"
        > link_endpoint_ids: [...]
          forwarding_direction: 1
        > total-potential-capacity: {...}
        > available-capacity: {...}
        > cost-characteristics: {...}
        > latency-characteristics: {...}
```

Figure 25: REST API Front-End / Back-End PathComp Request Content.

After triggering the selected algorithm at the Back end PathComp, if a path computation succeeds, this is sent back to the Front-End element to eventually proceed with the allocation through interacting with other TeraFlow SDN controllers (i.e., Service). The REST API-based response constructed by the Back-End element is depicted in Figure 26. The contents are:

- Service information: contains the service identifier and endpoints to allow the Front-End entity to resolve the path computation request and response exchange;
- Path attributes: carries specific output information such as the total end-to-end available capacity through the computed path, the accumulated end-to-end latency, and the aggregated total cost or consumed power once the service is eventually set up;

- Link List: it is listed the set the ordered set of the links forming the connectivity service between the two endpoints. In other words, it is contained the link information (context, topology and identifier) needed to allow the resource allocation when passed to the Service Component.



```
> Frame 29363: 1782 bytes on wire (14256 bits), 1782 bytes captured (14256 bits)
> Linux cooked capture v1
> Internet Protocol Version 4, Src: 10.1.6.201, Dst: 10.1.6.201
> Transmission Control Protocol, Src Port: 8081, Dst Port: 59516, Seq: 1, Ack: 19314, Len: 1714
> Hypertext Transfer Protocol
v JavaScript Object Notation: application/json
  v JSON compact form: {...}
    v response-list: [...]
      v 0: {...}
        > serviceId: {...}
        > service_endpoints_ids: [...]
        v path: [...]
          v 0: {...}
            v path-capacity: {...}
              v total-size: {...}
                  value: 200
                  unit: 0
            v path-latency: {...}
                fixed-latency-characteristic: "4.000000"
            v path-cost: {...}
                cost-name: ""
                cost-value: "2.000000"
                cost-algorithm: "0.000000"
            v link: [...]
              v 0: {...}
                  link_Id: "ffffffff-ffff-ffff-ffff-ffffffffffab"
                v topology: [...]
                  v 0: {...}
                      topology_uuid: "abf65601-a36b-496f-8e21-1708e5f96e2d"
                  v 1: {...}
                      topology_uuid: "abf65601-a36b-496f-8e21-1708e5f96e2d"
              v 1: {...}
                  link_Id: "ffffffff-ffff-ffff-ffff-ffffffffffbe"
                v topology: [...]
                  v 0: {...}
                      topology_uuid: "abf65601-a36b-496f-8e21-1708e5f96e2d"
                  v 1: {...}
                      topology_uuid: "abf65601-a36b-496f-8e21-1708e5f96e2d"
  v Object
    > Member Key: response-list
```

Figure 26: REST API Front-End / Back-End PathComp Response Content.

Numerical Results

The considered metric for assessing the PathComp component is the delay incurred by both the Front-End and Back-End elements when processing and serving computations requests. The elapsed time takes into consideration the above workflow: i) the Front-End retrieves the Context information, ii) the Front-End composes the REST API request message to the Back-End element; iii) the Front-End sends the request and wait till receive a response from the Back end PathComp; iv) the Front-End processes the response; v) the Front-End sends the response to the Service component.

To produce the CDF of the PathComp delay, 100 requests are generated where uniformly queries L2 and L3 network services and slices. The endpoints of every request are chosen randomly from the transport topology shown in Figure 27. The infrastructure is formed by 7 packet switches/routers which are controlled/programmed by a TeraFlow SDN controller instance. It is worth mentioning that the data plane is emulated and only the controller building blocks are considered to collect the numerical results

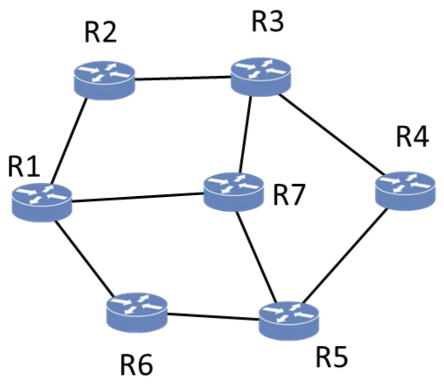


Figure 27: Transport Network Topology for PathComp Delay Performance Evaluation.

Each PathComp request is generated with a Poisson statistical model whose inter-arrival time is set to 200ms while the duration of the service/slice is modelled exponentially with a holding time of 10s. The considered algorithm to be executed regardless of the request type (connectivity service or slice) is the shortest path (i.e., Dijkstra algorithm). Finally, we consider that 5 replicas of the PathComp pod (e.g., Front-End and Back-End containers) are deployed behind a Kubernetes service for load balancing purposes.

In Figure 28, it is shown the CDF of the PathComp latency for the generated 100 requests. We observe that the delay ranges between 15 and 85 ms approximately.

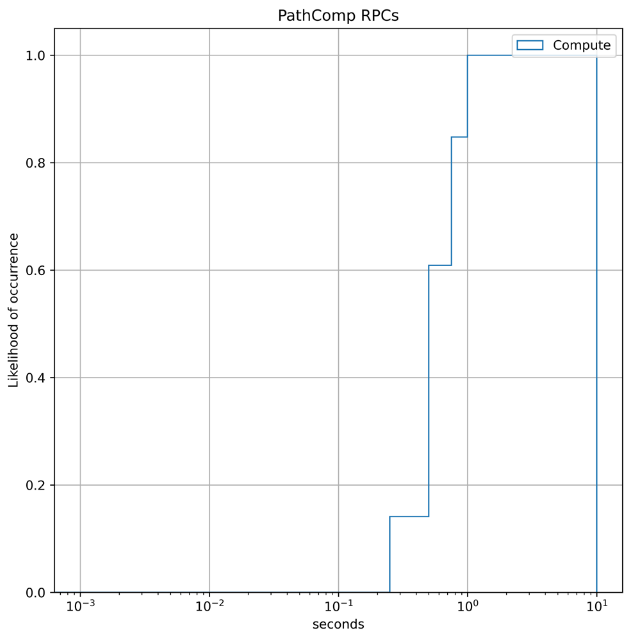


Figure 28: CDF for the PathComp Delay.

4. Hardware and L0/L3 Multi-layer Integration

This section provides the design overview, interfaces, operational workflows, and evaluation results of the core TeraFlowSDN components of T3.2, i.e., the SBI component (see Section 4.1), the Service component (see Section 4.2), and the Forecaster component (see Section 4.3).

4.1. SBI Component

This section describes the SBI component in charge of interacting with the underlying network equipment. Different protocols and data models might be needed to manage the network equipment; for this reason, the SBI component provides a Driver API that enables developers to implement new drivers and integrate them into TeraFlowSDN. We describe the SBI component's architectural design, the plugins' framework, the Driver interface, as well as the interface it exposes to the rest of the TeraFlowSDN components, while providing some performance evaluation results of the different southbound drivers.

4.1.1. New Features/Extensions

- Automated device discovery through interaction with the Automation component.
- Complete implementation of device monitoring functions.
- Improve OpenConfig driver templates.
- Validate and improve TAPI driver.
- Full support for P4 device configuration through a P4 driver implementation.
- Add new device driver managing for XR constellations via Infinera Pluggable Manager (IPM) controller.

4.1.2. Final Design

The available driver plugins are listed below, with a link to the corresponding subsection:

- An emulated driver plugin for testing purposes (see Section 4.1.3.1);
- An OLS ONF Transport API [TR547] driver plugin (see Section 4.1.3.2);
- An ONF TR-532 microwave driver plugin (see Section 4.1.3.3);
- A NETCONF [24] /OpenConfig [25] driver plugin for packet routers (see Section 4.1.3.4);
- A P4 [10] driver plugin for next-generation white box switches (see Section 4.1.3.5); and
- XR Constellation driver plugin (see Section 4.1.3.6).

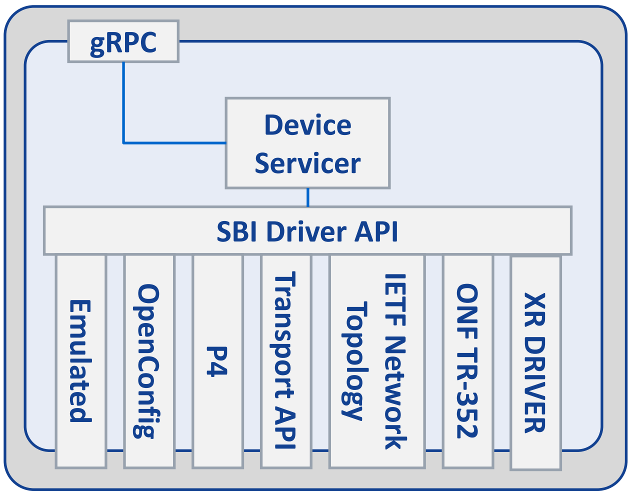


Figure 29: Architecture of the Device component

4.1.3. Device Plugins

Depending on the device to be managed, the TeraFlowSDN SBI component loads and uses the respective device plugin to translate abstract device operations into device-specific operations. The implemented drivers follow the Driver API described in Section 4.1.4. In the rest of this section, a variety of device plugins are presented.

4.1.3.1. Emulated Device Driver Plugin

Introduction

This section describes the Emulated Device Driver Plugin. This driver is used only for testing purposes to avoid strict dependencies on actual (physical or virtual) devices connected to the Device component for testing.

Supported Function

The function of this driver is very simple; it operates like a normal driver but, instead of interacting with physical/virtual devices, it uses internal memory for storing the configuration values provided. The driver implements all the methods described in the Device Driver API (see Section 4.1.4):

Device Driver RPC	Released on	Description
Constructor	v1	Constructor of an Emulated Device Driver. Provides a setting named as “endpoints” to define the endpoints that will be exposed by the emulated device.
Connect	v1	No real connection is established as the emulated driver does not need to connect to any external device.
Disconnect	v1	
GetInitialConfig	v1	Does nothing since the emulated device driver is constructed from scratch at each TeraFlow deployment.
GetConfig	v1	Retrieves specific configuration blocks according to the filtering parameters specified for the method.
SetConfig	v1	Updates/Deletes configuration blocks with the configuration rules specified in the method parameters.
DeleteConfig	v1	
SubscribeState	v1	Activates the monitoring of specific resources previously configured.

UnsubscribeState	v1	Deactivates the monitoring of specific resources previously configured.
GetState	v1	Periodically retrieves synthetic randomly generated values according to the configured sampling durations and intervals.

Table 16: List of RPCs supported by the Emulated TeraFlowSDN device driver

4.1.3.2. OLS ONF Transport API Driver Plugin

Introduction

This subsection describes the implementation of the Transport API (TAPI) Device Driver that serves as an SBI for the TeraFlow OS inside the Device component. As depicted in Figure 30, we consider the use of the TAPI Driver to interact with an Open Line System (OLS) Controllers in charge of managing underlying optical transport networks. In that way, the entire optical domain managed by the OLS controller is exposed to the TeraFlow OS as a single component with endpoints corresponding to the border endpoints in the optical network.

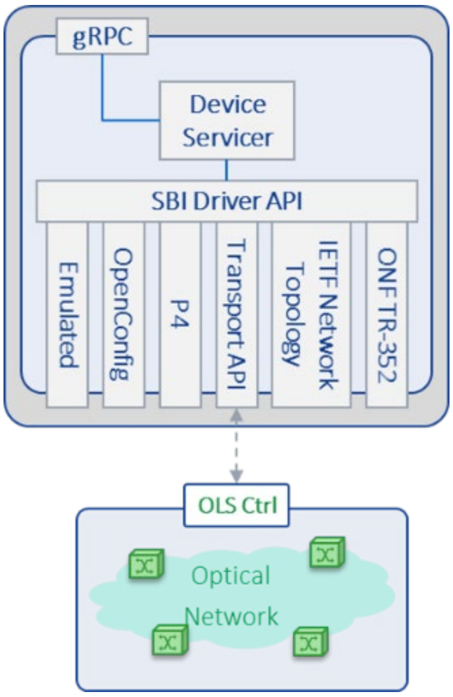


Figure 30: Architecture of the Device component's Transport API Driver

Supported Function

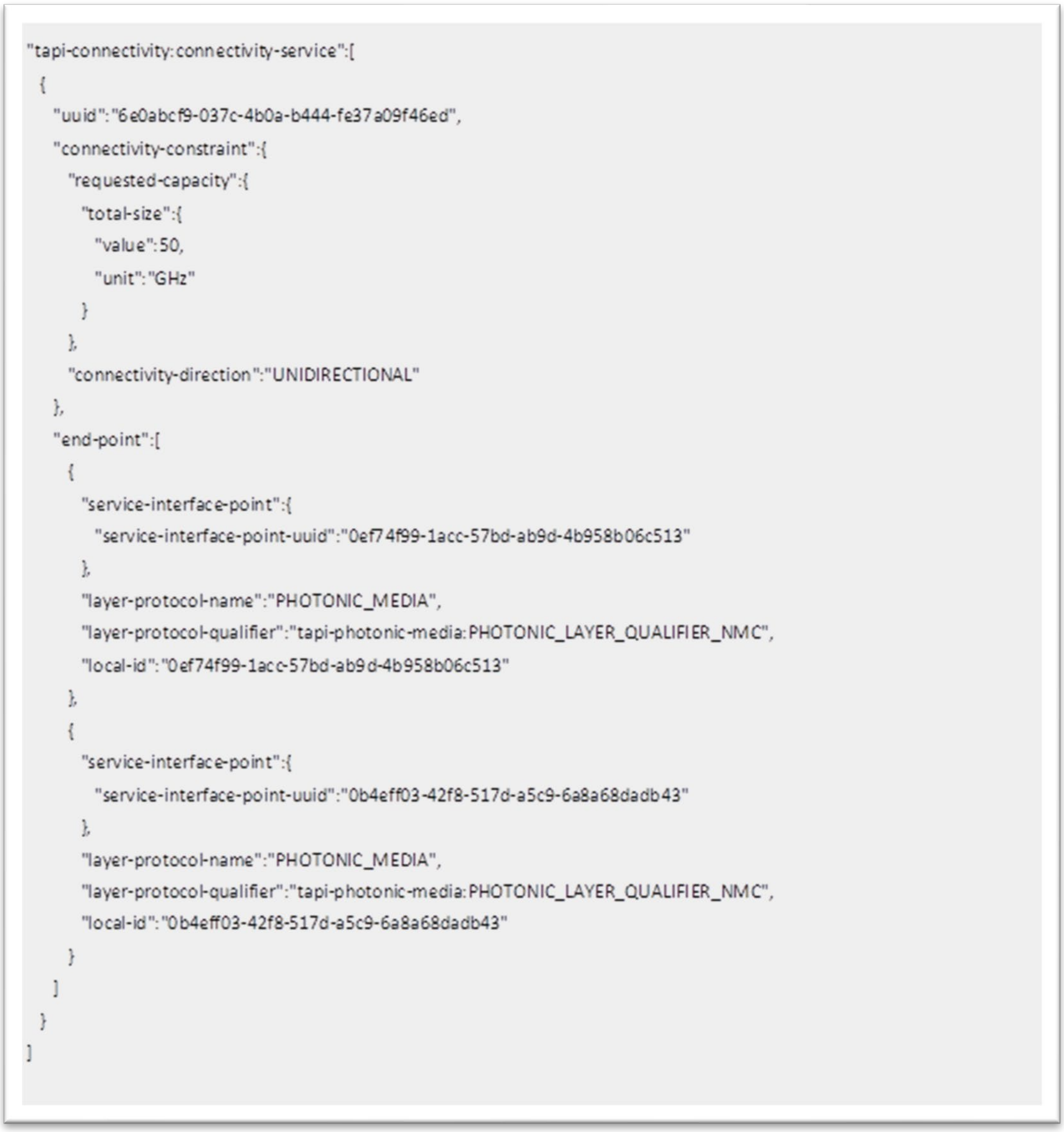
The goal of the TAPI Device Driver is to provide the function needed to establish basic communication with an OLS controller that will, in turn, manage the optical nodes. The interface to be implemented for TAPI-ready nodes is detailed in OpenAPI Specification (OAS) YAML files available in Table 17. In particular, the REST API URIs supported for the basic functionality of TAPI are described below:

- **GET /restconf/data/tapi-common:context**
  This URI provides the client with the TAPI context of the server, i.e., the topology, connectivity services, Service Interface Points (SIPs), as well as information about the name of the context and its Universal Unique Identifier (UUID). This URI is used to retrieve information about the TAPI server and to check effective connectivity with it.

- **GET /restconf/data/tapi-common:context/tapi-connectivity:connectivity-context/connectivity-service**
  This URI retrieves the connectivity services present in the TAPI server as well as the underlying connections that support them.
- **DELETE /restconf/data/tapi-common:context/tapi-connectivity:connectivity-context/connectivity-service={uuid}**
  This URI enables deletion of the connectivity service associated with a certain UUID.
- **POST /restconf/config/context/connectivity-service/{uuid}**
  This URI provides an endpoint for the creation of connectivity services. The data embedded in the body of the POST request is formatted as a JSON message. The JSON snippet in Figure 31 shows an example of creating a unidirectional 50 GHz connectivity service between two optical network endpoints.

Table 17: List of RPCs supported by the OLS ONF TAPI TeraFlowSDN device driver.

Device Driver RPC	Released on	Description
Constructor	v1	Constructor of a Transport API Device Driver.
Connect	v1	Validates connectivity with configured TAPI-enabled OLS controller.
Disconnect	v1	Does nothing since it uses basic non-persistent HTTP connections.
GetInitialConfig	v1	Does nothing since list of Service Interconnection Points (SIPs) is requested on-demand through GetConfig.
GetConfig	v1	Retrieves specific configuration blocks according to the filtering parameters specified for the method.
SetConfig	v1	Updates/Deletes connectivity services according to parameters specified in the method parameters.
DeleteConfig	v1	
SubscribeState	v1	Not Implemented.
UnsubscribeState	v1	
GetState	v1	



```
"tapi-connectivity:connectivity-service":[
 {
   "uuid":"6e0abcf9-037c-4b0a-b444-fe37a09f46ed",
   "connectivity-constraint":{
     "requested-capacity":{
       "total-size":{
         "value":50,
         "unit":"GHz"
       }
     },
     "connectivity-direction":"UNIDIRECTIONAL"
   },
   "end-point":[
     {
       "service-interface-point":{
         "service-interface-point-uuid":"0ef74f99-1acc-57bd-ab9d-4b958b06c513"
       },
       "layer-protocol-name":"PHOTONIC_MEDIA",
       "layer-protocol-qualifier":"tapi-photonic-media:PHOTONIC_LAYER_QUALIFIER_NMC",
       "local-id":"0ef74f99-1acc-57bd-ab9d-4b958b06c513"
     },
     {
       "service-interface-point":{
         "service-interface-point-uuid":"0b4eff03-42f8-517d-a5c9-6a8a68dadb43"
       },
       "layer-protocol-name":"PHOTONIC_MEDIA",
       "layer-protocol-qualifier":"tapi-photonic-media:PHOTONIC_LAYER_QUALIFIER_NMC",
       "local-id":"0b4eff03-42f8-517d-a5c9-6a8a68dadb43"
     }
   ]
 }
]
```

Figure 31: Example TAPI Create Connectivity Service.

4.1.3.3. ONF TR-532 / IETF Network Topology Microwave Driver Plugin

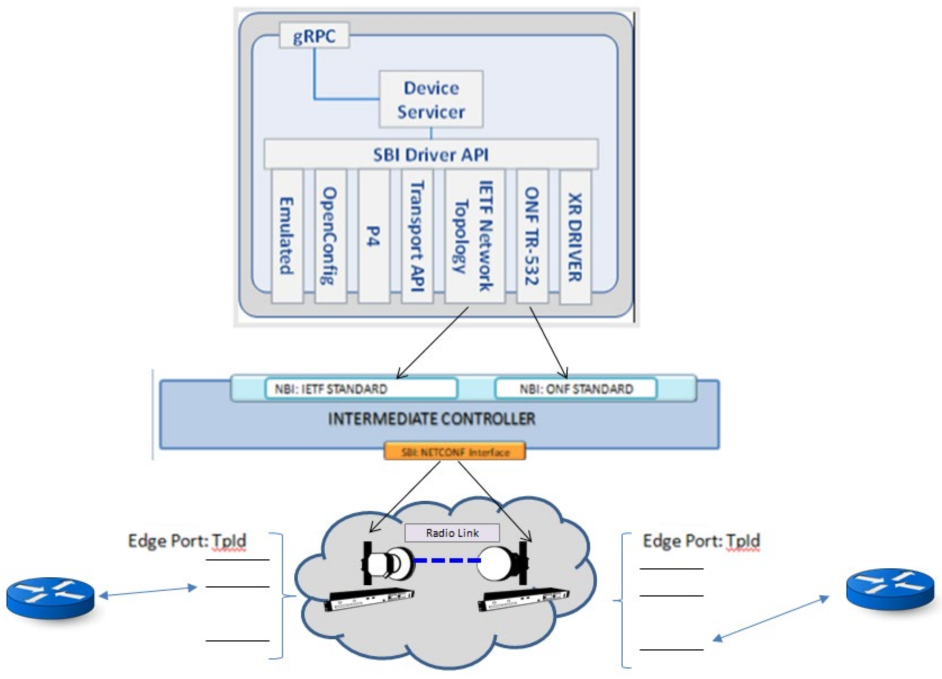


Figure 32: Microwave driver plugin design

IETF Network Topology and ONF TR-532 Device Driver are used to manage microwave devices involved in L3VPN services. The API interacts with an intermediate controller, the entity that interfaces with the specific MW network element.

The implementation follows the guidelines described in the Black Box abstraction in ETSI GS mWT 024. Each Microwave domain is abstracted and presented to Teraflow OS as a single virtual device exposing only edge ports.

Each MW domain is referred to with a single specific NodeId, while the exposed edge ports are referred as a concatenation of the device Id internal to the abstracted MW domain and the specific portId of that MW device

Table 18: List of the supported functions of the MW Device Drivers

Device Driver RPC	Description
Connect() GET: /nmswebs/restconf/data/ietf-network:networks	Check the Intermediate Controller connectivity status for the involved MW domain

GetConfig()
GET: /nmswebs/restconf/data/ietf-network:networks/network={:s}?fields={:s}
 	
GET: nmswebs/restconf/data/ietf-eth-tran-service:etht-svc
	Retrieve topological information about available edge ports
	Parameters used to invoke service are
	network= 'ETH-TOPOLOGY'
	fields= 'ietf-network-topology:link(link-id;destination(dest-node;dest-tp);source(source-node;source-tp));node(node-id;ietf-network-topology:termination-point(tp-id;ietf-te-topology:te/name)'

	The parameters returned by API are a list of {resource_key, resource_value} where
	resource_key = /endpoints/endpoint[node_id:tp_id]
	resource_value = {'uuid': tp_id, 'type': te['ietf-te-topology:te']['name']}
	
	retrieve list of services already configured in MW domain
	The parameters returned by API are a list of {resource_key, resource_value} where
	resource_key = /services/service[(service_name)]
	resource_value = {'uuid': service_name, 'type': service['etht-svc-type']}

SetConfig()
POST: /nmswebs/restconf/data/ietf-eth-tran-service:etht-svc
	create service identified by UUID involving
	Parameters used to invoke service are
	data = {
	'etht-svc-instances': [
	{
	'etht-svc-name': uuid,
	'etht-svc-type': 'ietf-eth-tran-types:p2p-svc',
	'etht-svc-end-points': [
	{
	'etht-svc-access-points': [
	{'access-node-id': node_id_src, 'access-ltp-id': tp_id_src, 'access-point-id': '1'}
	],
	'outer-tag': {'vlan-value': vlan_id, 'tag-type': 'ietf-eth-tran-types:classify-c-vlan'},
	'etht-svc-end-point-name': 'node_id_src:tp_id_src',
	'service-classification-type': 'ietf-eth-tran-types:vlan-classification'
	},
	{
	'etht-svc-access-points': [
	{'access-node-id': node_id_dst, 'access-ltp-id': tp_id_dst, 'access-point-id': '2'}
	],
	'outer-tag': {'vlan-value': vlan_id, 'tag-type': 'ietf-eth-tran-types:classify-c-vlan'},
	'etht-svc-end-point-name': 'node_id_dst:tp_id_dst',
	'service-classification-type': 'ietf-eth-tran-types:vlan-classification'
	}
	]
	}
	]
	}

DeleteConfig()
DELETE: /nmswebs/restconf/data/ietf-eth-tran-service:etht-svc/etht-svc-instances={:s}
	Delete service
	Parameters used to invoke service are
	data = {
	'etht-svc-instances': [{
	'etht-svc-name': uuid, }]
	}

4.1.3.4. OpenConfig Driver Plugin

This subsection describes the implementation of the OpenConfig Device Driver that serves as an SBI for the TeraFlow OS inside the Device component. As shown in Figure 33, we use the OpenConfig Driver to interact with L2 packet switches and L3 packet routers directly. Besides, two protocols are considered for the OpenConfig Driver, the NetConf protocol, used for configuring the devices, and the gNMI protocol, mainly used for telemetry streaming from these devices.

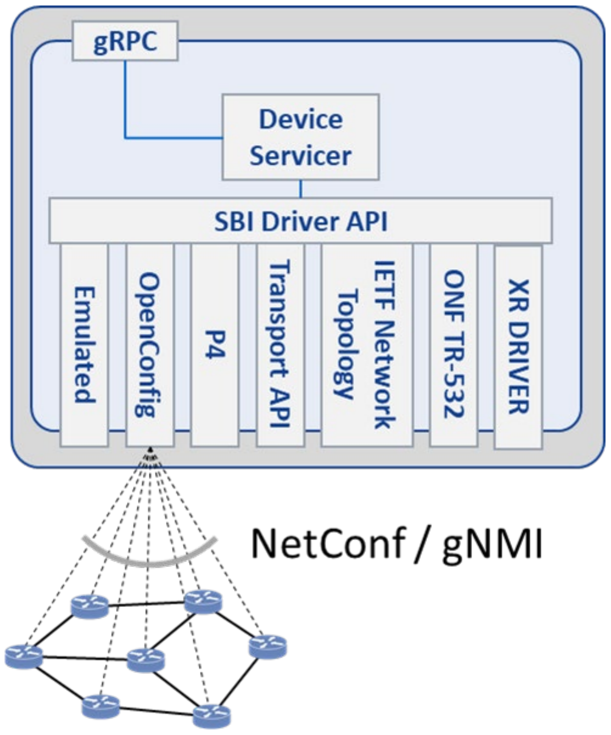


Figure 33: Architecture of the Device component's OpenConfig Driver.

4.1.3.5. NETCONF Protocol

The NETCONF protocol offers primitives to view and manipulate data, providing a suitable encoding as defined by the data model. Data is arranged into one or multiple configuration datastores (a set of configuration information required to get a device from its initial default state into a desired operational state). It enables remote access to a device and provides the rules by which multiple clients may access and modify a datastore within a NETCONF server (e.g., device). Note that NETCONF-enabled devices include a NETCONF server, while management applications include a NETCONF client.

The client and server communication data usually consists of XML/JSON-encoded RPC messages over a secure (commonly Secure Shell, SSH) connection. However, device Command Line Interfaces (CLIs) can be also wrapped around a NETCONF client.

NETCONF operations are organized in layers as shown in Figure 34. Upper **Content Layer** models the configuration and/or notification data that is exchanged between a client and a server. **Operations Layer** (e.g. <get-config>, <edit-config>) provides mechanisms for retrieving and manipulating that data. In the figure below, the **RPC Layer** enables to issue remote procedure calls for supporting the operations and issuing specialized commands not backed by data, e.g., device reboot. Finally, the bottom **Transport Protocol Layer** offers the means for exchanging the RPC messages in a secure manner.

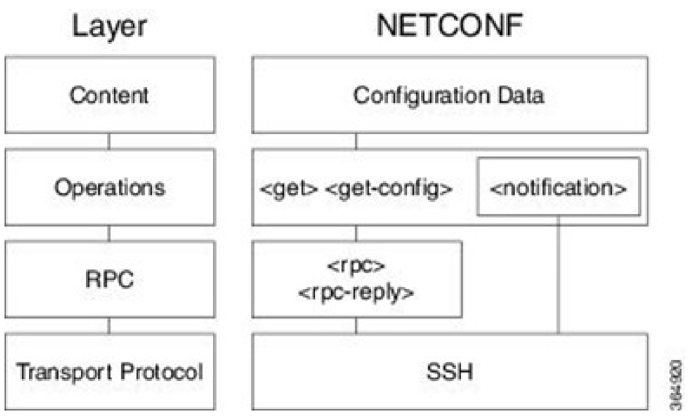


Figure 34: NETCONF Layering.

After establishing a session over a secure transport, both the client and the server send a *hello* message to announce their protocol capabilities, the supported data models, and the server's session identifier. After that announcement, the NETCONF session is up and ready for operation. The list of RPCs defined by the NETCONF protocol is summarized in Table 19. When accessing configuration or state data, with NETCONF operations, subtree filter expressions can select subtrees.

Table 19: List of RPCs supported by the NETCONF protocol.

RPC	**Description**
<get>	Retrieve running configuration and device state information.
<get-config>	Retrieve all or part of a specified configuration datastore.
<edit-config>	Edit a configuration datastore by creating, deleting, merging or replacing content.
<copy-config>	Copy an entire configuration datastore to another configuration datastore.
<delete-config>	Delete a configuration datastore.
<lock>	Lock an entire configuration datastore of a device.
<unlock>	Release a configuration datastore lock previously obtained with the <lock> operation.
<close-session>	Request graceful termination of a NETCONF session.

The Yet Another Next Generation (YANG) is a data modeling language, initially conceived to model configuration and state data for network devices. Models define the device configurations & notifications, capture semantic details, and make them more understandable. YANG is widely adopted as data modelling language across frameworks and Open-Source projects. In addition, there is a notable ongoing effort across the Standards Developing Organizations (SDOs) to model constructs (e.g., topologies, protocols), including packet and optical devices. There are, literally, hundreds of emerging standards across SDOs.

A YANG model includes a header, imports and include statements, type definitions, configurations, and operational data declarations as well as actions (RPC) and notifications. The language is expressive enough to:

- Structure data into data trees within the so-called datastores, by means of encapsulation of containers and lists, and to define constrained data types (e.g., following a given textual pattern).
- Condition the presence of specific data to the support of optional features.

- Allow the refinement of models by extending and constraining existing models (by inheritance/augmentation), resulting in a hierarchy of models.
- Define configuration and/or state data.

YANG has become the data modeling language of choice for multiple network control and management aspects due partly to its features, flexibility and the availability of tools. It is used to model the internal data and management operations of individual devices, entire networks, and services. Even pre-existing protocols (e.g., routing protocols such as BGP) can be configured in through YANG messages.

4.1.3.6. Data models and XML Templates

The XML templates of the following data templates can be found in Annex XML templates. The tables used in the following sections refer to the variables used in the templates, indicating the model path and the name used in the OpenConfig driver.

- **L2VPN**

The following scenario lists the required steps to configure an L2-VPN in ADVA Edgecore DRX-30 stacked node:

- Create L2-VPN network-instance
  - o <edit-config> - Create L2-VPN network-instance

Variable Name / Model Path	Description
ni_node_vpn_name **/network-instances/network-instance/name**	A unique name identifying the network instance.
ni_vpn_description **/network-instances/network-instance/config/description**	A free-form string to be used by the network operator to describe the function of this network instance.
ni_node_vpn_type **/network-instances/network-instance/config/type**	The type of network instance. The value of this leaf indicates the type of forwarding entries that should be supported by this network instance. Signalling protocols also use the network instance type to infer the type of service they advertise. Options: DEFAULT_INSTANCE (Global Table), L3VRF, L2VSI, L2P2P, L2L3.

- Configure interfaces/subinterfaces L2 parameters
  a. <edit-config> - Configure interfaces/subinterfaces L2 parameters (vlan-id)

Variable Name / Model Path	Description
ni_network_access_interface_id **/interfaces/interface/name**	Identifier for the physical interface. This means the name of the physical interface: i.e: GigabitEthernet0/0/0 or eth-0/0/0 depending on each vendor notation.
ni_network_access_termination _point **/interfaces/interface/name**	Specifies the name of the subinterface.
ni_network_access_mtu **/interfaces/interface/config/mtu**	Set the max transmission unit size in octets for the physical interface. If this is not set, the mtu is set to the operational default -- e.g., 1514 bytes on an Ethernet interface.
ni_network_access_description **/interfaces/interface/subinterfaces/su bint erface/config/description**	A textual description of the subinterface.

ni_network_access_vlan_id **/interfaces/interface/subinterfaces/subinterface/oc-vlan:vlan/ocvlan:match/oc-vlan:single-tagged/oc-vlan:config/oc-vlan:vlan-id**	VLAN identifier for single-tagged packets. Used for 802.1q ethernet frames.

- Add interfaces (endpoint) to L2-VPN network instance
    - <edit-config> - Add interfaces (endpoint) to L2-VPN network instance

Variable Name / Model Path	Description
ni_node_vpn_name **/network-instances/network-instance/name**	A unique name identifying the network instance.
ni_network_access_interface_id **/network-instances/network-instance/interfaces/interface/id**	Identifier for the physical interface. This means the name of the physical interface: i.e: GigabitEthernet0/0/0 or eth-0/0/0 depending on each vendor notation.
ni_network_access_termination _point **/network-instances/network-instance/interfaces/interface/id**	Specifies the name of the subinterface.

- Add virtual circuits (point-to-point, bi-directional pseudo-wire interconnection) to an L2-VPN network instance
- <edit-config> - Add virtual circuits (point-to-point, bi-directional pseudo-wire interconnection) to L2-VPN network instance

Variable Name / Model Path	Description
ni_node_vpn_name **/network-instances/network-instance/name**	A unique name identifying the network instance.
ni_node_connection_point **/network-instances/network-instance/connection-points/connection-point/connection-point-id**	A pointer to the configured identifier for the Endpoint.
ni_node_connection_point_id **/network-instances/network-instance/connection-points/connection-point/endpoints/endpoint/remote/config/virtual-circuit-identifier**	The virtual-circuit identifier that identifies the connection at the remote end-point.
ni_node_remote_system **/network-instances/network-instance/connection-points/connection-point/endpoints/endpoint/remote/config/remote-system**	The IP address of the device which hosts the remote end-point.

- **L3VPN**

The following scenario lists the required steps to configure an L3-VPN in ADVA Edgecore DRX-30 stacked node:

- Create L3-VPN network-instance
    - <edit-config> - Create L3-VPN network-instance

Variable Name / Model Path	Description
ni_node_vpn_name **/network-instances/network-instance/name**	A unique name identifying the network instance.
ni_node_vpn_type **/network-instances/network-instance/config/type**	The type of network instance. The value of this leaf indicates the type of forwarding entries that this network instance should support. Signalling protocols also use the network instance type to infer the type of service they advertise.

	Options: DEFAULT_INSTANCE (Global Table), L3VRF, L2VSI, L2P2P, L2L3.
ni_vpn_description **/network-instances/network-instance/config/description**	A free-form string to be used by the network operator to describe the function of this network instance.
ni_node_router_id **/network-instances/network-instance/config/router-id**	Router id of the router - an unsigned 32-bit integer expressed in dotted quad notation.
ni_node_route_distinguisher **/network-instances/network-instance/config/route-distinguisher**	The route distinguisher that should be used for the local VRF or VSI instance when it is signalled via BGP.

- Define routing protocols used within the sL3-VPN network instance
  - <edit-config> - Define BGP routing protocol for network-instance
  - <edit-config> - Define DIRECTLY CONNECTED routing protocol for network-instance

Variable Name / Model Path	**Description**
ni_node_vpn_name **/network-instances/network-instance/name**	A unique name identifying the network instance.
ni_node_policy_type **/network-instances/network-instance/protocols/protocol/identifier**	The on-the-wire encapsulation that should be used when sending traffic from this network instance.
ni_node_protocol_name **/network-instances/network-instance/protocols/protocol/name**	The label allocation mode to be used for L3 entries in the network instance Options: PER_PREFIX, PER_NEXTHOP, INSTANCE_LABEL.
ni_node_bgp_local_autonomous_system **/network-instances/network-instance/protocols/protocol/bgp/global/config/as**	The local autonomous system number that is to be used when establishing sessions with the remote peer or peer group, if this differs from the global BGP router autonomous system number.
ni_routing_protocol_bgp_router_id **/network-instances/network-instance/protocols/protocol/bgp/global/config/router-id**	Router id of the router - an unsigned 32-bit integer expressed in dotted quad notation.

- Configure interfaces/subinterfaces L3 parameters
  - <edit-config> - Configure interfaces/subinterfaces L3 parameters (IP address, address prefix, vlan id and MTU)

Variable Name / Model Path	**Description**
ni_network_access_interface_id **/interfaces/interface/name**	Identifier for the physical interface. This means the name of the physical interface: i.e: GigabitEthernet0/0/0 or eth-0/0/0 depending on each vendor notation.
ni_network_access_termination _point **/interfaces/interface/name**	Specifies the name of the subinterface.
ni_network_access_mtu **/interfaces/interface/config/mtu**	Set the max transmission unit size in octets for the physical interface. If this is not set, the mtu is set to the operational default -- e.g., 1514 bytes on an Ethernet interface.
ni_network_access_description **/interfaces/interface/subinterfaces/subinterface/config/description**	A textual description of the subinterface.
ni_network_access_vlan_id **/interfaces/interface/subinterfaces/subinterface/oc-vlan:vlan/oc-vlan:match/oc-**	VLAN identifier for single-tagged packets. Used for 802.1q ethernet frames.

vlan:single-tagged/oc-vlan:config/oc-vlan:vlan-id	
ni_network_access_address_ip **/interfaces/interface/subinterfaces/subinterface/oc-ip:ipv4/oc-ip:addresses/oc-ip:address/oc-ip:ip**	The IPv4 address on the subinterface.
ni_network_access_prefix **/interfaces/interface/subinterfaces/subinterface/oc-ip:ipv4/oc-ip:addresses/oc-ip:address/oc-ip:config/oc-ip:prefix-length**	The length of the subnet prefix.

- Add interfaces (endpoint) to L3-VPN network instance
    - <edit-config> - Add interfaces (endpoint) to an L3-VPN network instance

Variable Name / Model Path	Description
ni_node_vpn_name **/network-instances/network-instance/name**	A unique name identifying the network instance.
ni_network_access_interface_id **/network-instances/network-instance/interfaces/interface/id**	Identifier for the physical interface. This means the name of the physical interface: i.e: GigabitEthernet0/0/0 or eth-0/0/0 depending on each vendor notation.
ni_network_access_termination _point **/network-instances/network-instance/interfaces/interface/id**	Specifies the name of the subinterface.

- Create BGP Routing Policies Import/Export for an L3-VPN
    - <edit-config> create a routing-policy

Variable Name / Model Path	Description
rp_match_ext_community_set_name **/routing-policy/defined-sets/bgp-defined-sets/ext-community-sets/ext-community-set/ ext-community-set-name**	Label of the extended community set. This is used to reference the set in match conditions.
rp_match_ext_community_member **/routing-policy/defined-sets/bgp-defined-sets/ext-community-sets/ext-community-set/config/ ext-community-member**	Members of the extended community set.

- <editc-config> create BGP match conditions and action

Variable Name / Model Path	Description
rp_set_name **/routing-policy/policy-definitions/policy-definition/name**	Name of the top-level policy definition. This name is used in references to the current policy. It should be invoked by the ni_node_import_policy or ni_node_export_policy based on the policy type.
rp_set_statement_name **/routing-policy/policy-definitions/policy-definition/statements/statement/name**	Statement Name
rp_match_ext_community_set_name **routing-policy/policy-definitions/policy-definition/statements/statement/conditions/oc-bgp-pol:bgp-conditions/oc-bgp-pol:config/oc-bgp-pol:ext-community-set**	Label of the extended community set. This is used to reference the set in match conditions.
rp_action_policy_result	Select the final disposition for the route, either accept or reject.

/routing-policy/policy-definitions/policy-definition/statements/statement/actions/config/policy-result	Options: ACCEPT_ROUTE, REJECT_ROUTE

- Apply BGP Import/export Policy (Route Target) to an L3-VPN network instance
  - <edit-config> Apply BGP policy to network-instance

Variable Name / Model Path	Description
ni_node_vpn_name **/network-instances/network-instance/name**	A unique name identifying the network instance.
ni_node_import_policy **/network-instances/network-instance/inter-instance-policies/apply-policy/config/import-policy**	List of policy names in sequence to be applied on receiving a routing update in the current context.
ni_node_export_policy **/network-instances/network-instance/inter-instance-policies/apply-policy/config/export-policy**	List of policy names in sequence to be applied on receiving a routing update in the current context.

- Create protocol redistribution policies within an L3-VPN network instance
  - <edit-config> Creating protocol redistribution DIRECTLY_CONNECTED to BGP

Variable Name / Model Path	Description
ni_node_vpn_name **/network-instances/network-instance/name**	A unique name identifying the network instance.
ni_routing_protocol_src **/network-instances/network-instance/table-connections/table-connection/src-protocol**	The name of the protocol associated with the table which should be used as the source of forwarding or routing information.
ni_routing_protocol_dst **/network-instances/network-instance/table-connections/table-connection/dst-protocol**	The table to which routing entries should be exported.
ni_routing_protocol_af **/network-instances/network-instance/table-connections/table-connection/address-family**	The address family associated with the connection.

- **ACL**

The following scenario lists the required steps to configure an ACL in ADVA Edgecore DRX-30 stacked node:

- Create ACL-SET
  a. <edit-config> - Create ACL-SET (name, type, ACE ID, actions, parameters for the chosen type)

Variable Name / Model Path	Description
acl_name **/acl/acl-sets/acl-set/name**	Service Identifier
acl_type **/acl/acl-sets/acl-set/type**	The type of the service. The value of this leaf indicates the type of the rules that should be supported by this ACL Options: L2, IPV4, IPV6.
acl_id **/acl/acl-sets/acl-set/acl-entries/acl-entry/sequence-id**	ACL-ENTRY Identifier
acl_match_source_mac **/acl/acl-sets/acl-set/acl-entries/acl-entry/l2/**	For Type L2. Source address to match.

config/source-mac	
acl_match_destination_mac **/acl/acl-sets/acl-set/acl-entries/acl-entry/l2/config/destination-mac**	For Type L2. Destination address to match.
acl_match_source_ipv4 **/acl/acl-sets/acl-set/acl-entries/acl-entry/ipv4/config/source-address**	For Type IPV4. Source address to match.
acl_match_destination_ipv4 **/acl/acl-sets/acl-set/acl-entries/acl-entry/ipv4/config/destination-address**	For Type IPV4. Destination address to match.
acl_match_source_ipv6 **/acl/acl-sets/acl-set/acl-entries/acl-entry/ipv6/config/source-address**	For Type IPV6. Source address to match.
acl_match_destination_ipv6 **/acl/acl-sets/acl-set/acl-entries/acl-entry/ipv6/config/destination-address**	For Type IPV6. Destination address to match.
acl_match_protocol **/acl/acl-sets/acl-set/acl-entries/acl-entry/ipv{4 or 6}/config/protocol**	For Type IPV4 and IPV6. The protocol carried in the IP packet.
acl_match_dscp **/acl/acl-sets/acl-set/acl-entries/acl-entry/ipv{4 or 6}/config/dscp**	For Type IPV4 and IPV6. Value of diffserv codepoint.
acl_match_hop_limit **/acl/acl-sets/acl-set/acl-entries/acl-entry/ipv{4 or 6}/config/hop-limit**	For Type IPV4 and IPV6. The IP packet's hop limit.
acl_match_source_port **/acl/acl-sets/acl-set/acl-entries/acl-entry/transport/config/source-port**	For type IPV4. Source port or range.
acl_match_destination_port **/acl/acl-sets/acl-set/acl-entries/acl-entry/transport/config/destination-port**	For type IPV4. Destination port or range.
acl_match_tcp_flags **/acl/acl-sets/acl-set/acl-entries/acl-entry/transport/config/tcp-flags**	For type IPV4. List of TCP flags to match.
acl_match_forwarding_action **/acl/acl-sets/acl-set/acl-entries/acl-entry/actions/config/forwarding-action**	Specifies the forwarding action. One forwarding action must be specified for each ACL entry Options: ACCEPT, REJECT, DROP.
acl_match_log_action **/acl/acl-sets/acl-set/acl-entries/acl-entry/actions/config/log-action**	Specifies the log action and destination for matched packets. The default is not to log the packet. Options: NONE, SYSLOG.

- Add ACL-ENTRY to ACL-SET
    a. <edit-config> - Add ACL-ENTRY to ACL-SET (name, type, ACE ID, actions, parameters for the chosen type) (Same configuration as Create ACL-SET)
- Associate the ACL to an interface
- <edit-config> - Associate the ACL to an interface (interface ID, interface, sub interface, ACL name, ACL type, type of traffic)

Variable Name / Model Path	**Description**
acl_interface_id **/acl/interfaces/interface/id**	Interface identifier
acl_interface_name **/acl/interfaces/interface/interface-ref/config/interface**	Interface name
acl_interface_subinterface	Subinterface name

/acl/interfaces/interface/interface-ref/config/subinterface	
acl_interface_acl_set_name **/acl/interfaces/interface/ingress-acl-sets/{ingress or egress}-acl-set/set-name**	Name of the ACL to associate.
acl_interface_acl_set_type **/acl/interfaces/interface/ingress-acl-sets/{ingress or egress}-acl-set/type**	Type of the ACL to associate.

- **Inventory**

In this section, a "get" query is used to obtain information on all the device's components. This information is processed based on the component to display just the relevant information. The following table displays information that has been filtered based on the type of component.

Component type	**Data**
Chassis	• Component name • Component type • Vendor name • SW version
CPU	1. Component name 2. Component type
Fan	1. Component name 2. Component type
Power supply	1. Component name 2. Component type
Port	1. Port Name 2. Component type
Transceiver	A component of type=TRANSCEIVER is expected to be a subcomponent of a PORT component. A transceiver will always contain physical-channel(s), however when a line side optical-channel is present the physical-channel will reference its optical-channel. In this case, the optical-channels components must be subcomponents of the transceiver. The relationship between the physical-channel and the optical-channel allows for multiple optical-channels to be associated with a transceiver in addition to ensuring certain leaves are not duplicated in multiple components. 1. Operational state data for each component 2. Operational state data for port transceiver 3. Operational state data for channels

4.1.3.7. gNMI

Introduction

gNMI (gRPC Network Management Interface) is a network management protocol that uses gRPC (Google Remote Procedure Calls) as the underlying transport mechanism. It is a modern, open-source, and scalable protocol designed to be used for network configuration, monitoring, and management.

gNMI is based on the concept of a simple, high-level API that allows network devices to be managed in a uniform and consistent way. It uses a data model-based approach, where the data model defines the schema for the configuration and state data of the managed device. This allows gNMI to be used with various network devices, regardless of their specific vendor or model.

gNMI provides support for several important features, including:

- Transport security: gNMI uses TLS (Transport Layer Security) to communicate securely between the client and the server;
- Bi-directional streaming: gNMI allows the client and server to communicate in both directions simultaneously, allowing for real-time updates of network data;
- Versioning: gNMI supports versioning of the data model and the gNMI protocol itself, allowing for backwards compatibility and easy updates;
- Multi-tenancy: gNMI allows multiple clients to access the same network device, enabling support for multi-tenant environments.

Overall, gNMI is a powerful and flexible tool for managing modern networks and is gaining widespread adoption in the networking industry.

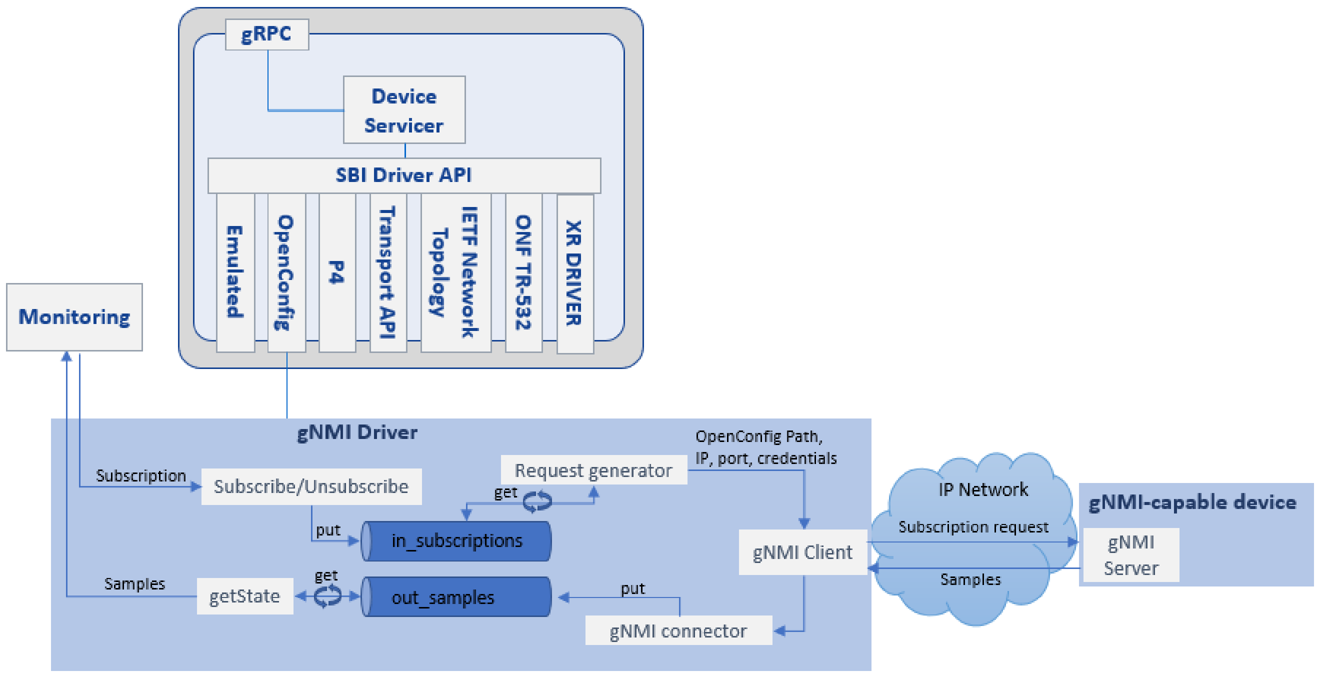


Figure 35: gNMI architecture driver plugin.

Supported Functions

This driver interrogates network devices to collect telemetry data and dumps it into the monitoring component. This is the list of RPCs supported by the gNMI Driver:

Device Driver RPC	Released on	Description
Connect	v1	Establishes a connection with a specific gNMI device by interrogating it and asking for the capabilities; it also starts the monitoring thread.
Disconnect	v1	Terminates the monitoring thread, disconnects the gRPC channel, sends the unsubscribe message to the device in question and stops the thread.
SubscribeState	v1	Checks the subscriptions queue and starts the telemetry streaming for a given device using a certain path. Then, it dumps the samples into the out_samples queue.
UnsubscribeState	v1	Puts the unsubscription request in the subscriptions queue.
GetConfig	v1	Provides the list of endpoints.

GetInitialConfig	v1	Not Implemented.
SetConfig	v1	
DeleteConfig	v1	

Table 20: gNMI supported GRPCs

4.1.3.8. P4 Whitebox Switches Driver Plugin

Introduction

Network devices are typically designed “bottom-up”, with fixed-function chips (i.e., with no run-time reconfigurability) being the heart of the system, thus determining how the device OS is realized and what functionality it can offer. Unfortunately, adding a new feature set to a fixed-function switch is a complex process that takes several months or even years as it requires hardware redesign.

The OpenFlow protocol [9] made a step forward by introducing an open interface to populate the forwarding tables (i.e., hash tables for Ethernet address lookup, longest-prefix match tables for IPv4/IPv6 and wildcard lookups for ACLs) in network switches, thus enabling software-based control planes to control switches from a variety of different vendors. However, OpenFlow still assumes the switches have a fixed behaviour (i.e., a fixed set of tables), typically described in the datasheet of a switch ASIC. This means that OpenFlow cannot change the switch behaviour, e.g., by adding new protocols.

P4 [10] was introduced in 2014 with the purpose of addressing the limitations of the OpenFlow SDN protocol as well as legacy networking paradigms. P4 is an open source, domain-specific programming language for next-generation network devices, also known as whiteboxes, which focuses on describing a “top-down” forwarding plane of programmable (non-fixed-function) switches. With programmable switches, there is no need for fixed protocols, such as OpenFlow. Instead, P4 treats programmable switches just like general purpose processors (e.g., CPUs or GPUs), allowing them to execute code written in a specific programming language (i.e., the P4 language). The code is first compiled by a P4 compiler and then loaded into the processor of the whitebox switch. This way, P4 lets network developers define what headers a switch should be able to parse (including custom or new headers), how to match on each header, and what actions the switch may perform on each header. In P4, OpenFlow is just another program, i.e., one of many possible ways to describe what a forwarding plane does.

Stratum OS

Since the introduction of P4 in 2014, a large community has been established, initially around the P4.org consortium, and since 2019 under the ONF umbrella [11]. In the same year (i.e., 2019), ONF announced the release of the Stratum [12] project as an open-source silicon-independent switch OS for SDN that runs on a variety of switching silicon and whitebox switch platforms. In addition, stratum exposes a set of next-generation SDN interfaces, including P4Runtime, OpenConfig, gRPC, gNMI, and gNOI, enabling greater programmability of forwarding behaviours in interchangeable forwarding devices, thus avoiding the vendor lock-in of today’s data planes through proprietary silicon interfaces and closed software APIs.

Native P4 support in TeraFlowSDN

The ONF has implemented several P4 device drivers on top of the Stratum OS (i.e., Barefoot, Mellanox, BMv2) with P4Runtime support within the ONOS SDN controller [13] ecosystem. The early

TeraFlowSDN P4 device driver prototype, introduced in D3.1 and the first TeraFlowSDN release (v1), assumed an intermediate SDN controller (i.e., ONOS) between the TeraFlowSDN controller and P4 devices, to exploit the existing P4Runtime implementation by ONOS.

However, during the second TeraFlowSDN release (v2), the TeraFlowSDN P4 device driver was re-designed with native P4 support. A new P4Runtime client has been natively incorporated in the TeraFlowSDN device driver and an overlay P4 Manager module abstracts low-level client interactions with the switch to offer key abstractions for the following P4 entities:

- Tables and table entries;
- Actions;
- Action profile members and groups;
- Meters and meter entries;
- Direct meters and direct meter entries;
- Counters and counter entries;
- Direct counters and direct counter entries;
- Controller packet metadata and packet replication entries

In the P4 data plane, TeraFlowSDN still embraces ONF initiatives by integrating P4 devices empowered by the Stratum OS. On the control plane side, TeraFlowSDN provides native support for P4 management operations through a built-in P4Runtime client; thus, no dependencies with external SDN controllers (i.e., ONOS) are required anymore. In the rest of this section, we describe how the TeraFlow SDN controller manages P4 whiteboxes. Figure 16 shows a high-level overview of the P4 TeraFlowSDN device driver plugin and how it interacts with P4 devices through P4Runtime and Stratum.

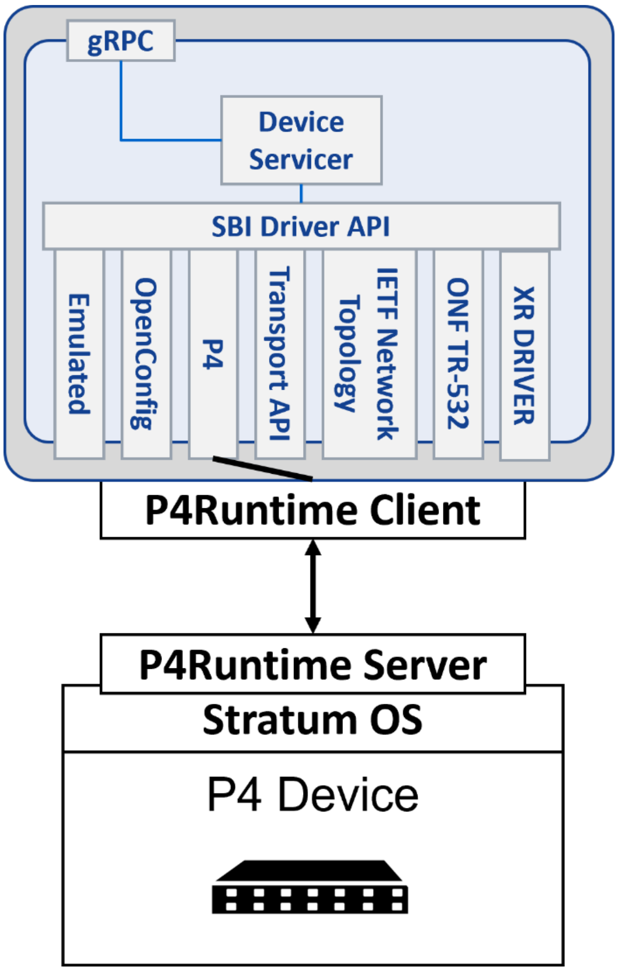


Figure 36: Architecture of the Device component's P4 driver plugin.

The TeraFlowSDN P4 pipeline

Before deploying a P4 program onto a P4 device, a pre-deployment workflow must be realized, as shown in Figure 36 above.

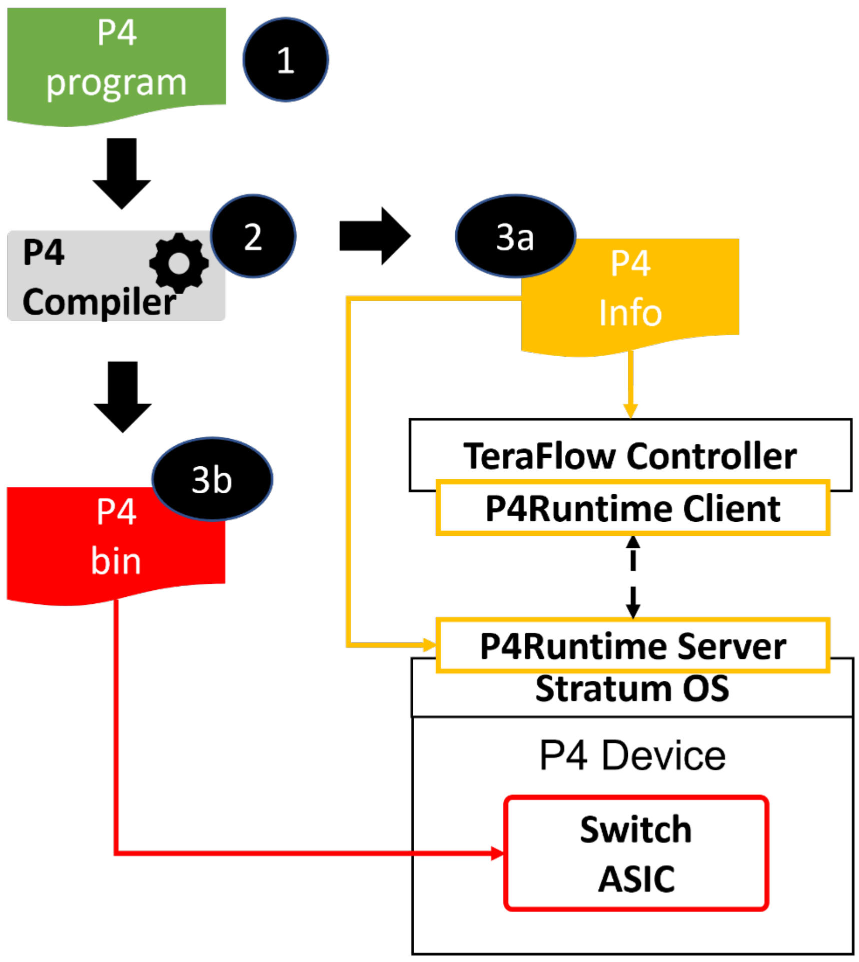


Figure 37: Required steps for a P4 SDN controller to install a P4 program on a P4 device.

Specifically, the desired P4 program needs to be written (step 1) by a network developer and compiled (step 2) by a P4 compiler. The P4 compiler generates two outputs:

- A "P4 Info" file (step 3a) which describes the "schema" of the P4 pipeline for runtime control. This schema captures P4 program attributes such as tables, actions, parameters, etc, in a target-independent format (I.e., same P4Info for a software switch, ASIC, etc.);
- A target-specific "P4 bin" binary (step 3b) used to realize a switch pipeline, such as a binary configuration for an application-specific integrated circuit (ASIC), a bitstream for a field-programmable gate array (FPGA), etc. At runtime the TeraFlowSDN controller uses a gRPC-based P4Runtime interface to manage the match-action pipelines specified in the P4 program.

P4 TeraFlowSDN Device Driver RPCs

The complete list of RPCs supported by the P4 TeraFlowSDN device driver is depicted in Table 21 below. Only two basic RPCs were released in the first TeraFlowSDN release (v1). Release v2 introduces support for configuring all the basic P4 entities through Set/Get/DeleteConfig RPCs and the GetInitialConfig RPC.

Table 21: List of RPCs supported by the P4 TeraFlowSDN device driver.

Device Driver RPC	Released on	Description
Connect	v1	Initiates a connection with a given P4 device on a given IP. The Connect RPC also offers custom settings which allow to pass

		the P4 binary file and the P4 info file to be installed on a given P4 device to realize a target forwarding logic.
Disconnect	v1	Tears a connection to a P4 device down.
GetConfig	v2	Retrieves either the entire device configuration or part of it, based on an input list of resource keys to be looked up.
GetInitialConfig	v2	Retrieves a basic initial configuration for a P4 device to allow automated onboarding of P4 devices through the Automation component.
SetConfig	v2	Installs a set of entries to the table(s) of the P4 pipeline, based on an input map of resource keys to resource values.
DeleteConfig	v2	Removes a set of entries from the table(s) of the P4 pipeline based on an input list of resource keys to be removed.

Future Work

The TeraFlowSDN device driver API also allows monitoring device resources (e.g., KPIs) through the SubscribeState, GetState, and UnsubscribeState RPCs. To achieve this functionality in P4, a dedicated working group by ONF, titled the P4.org Application group [15], suggests detailed P4 device resource monitoring via an in-band network telemetry (INT) specification [16]. We leave the implementation of such a feature in TeraFlowSDN as future work, as it falls outside of the scope of the TeraFlow EU project. However, UBI commits to this activity and hopes to engage the ETSI TFS community soon.

4.1.3.9. XR Constellation Driver Plugin

The functional driver presents XR constellation network point-to-multipoint connections as ports for configuring optical bandwidth and service endpoints. XR constellations are discovered from Infinera Intelligent Pluggable Manager (IPM) via north-bound REST-API. Optical bandwidth is delivered as an attribute to the XR Constellation driver and passed via REST-API to IPM.

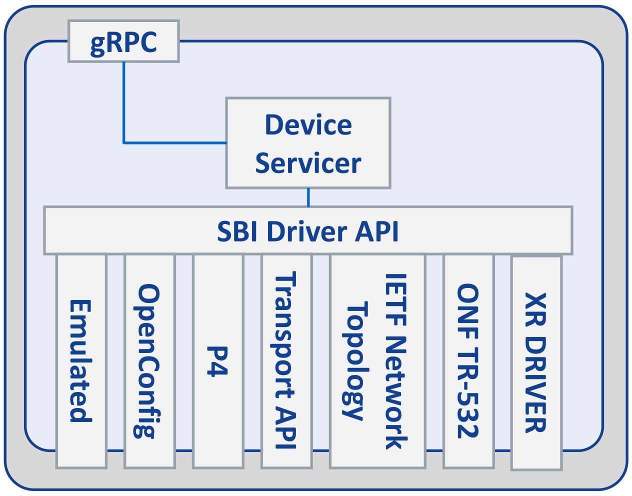


Figure 38. TeraFlowSDN device drivers

Infinera XR point-to-multipoint solution is the evolution of traditional point-to-point optical solution, reducing the required number of optical transceivers for connections and simplifying hub location with a simple optical combiner/demultiplexer. Point-to-multipoint hub and leaf XR modules create an

XR constellation, where the hub side XR module works controller for leaf XR modules enabling co-existence and coordinating optical behavior.

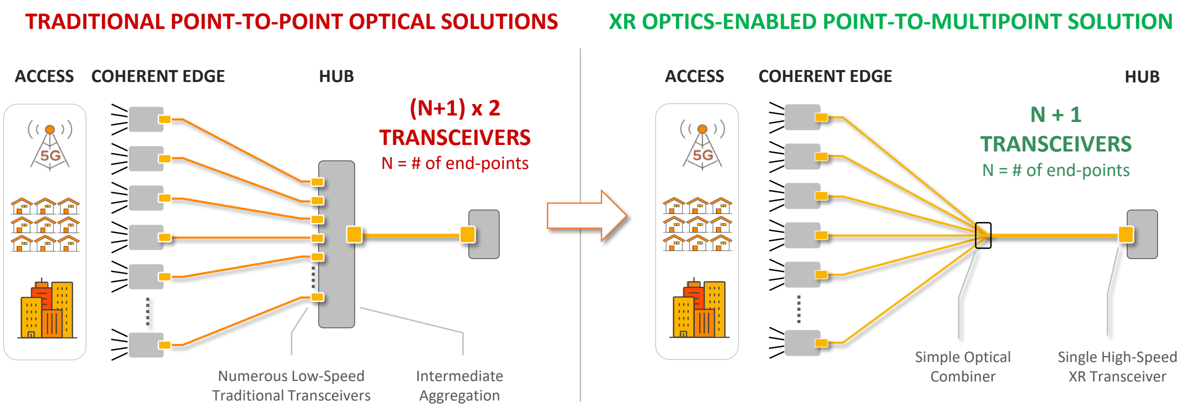


Figure 39. Infinera XR Optics point-to-multipoint concept

On point-to-multipoint configuration, XR pluggables are dual-managed, traditional pluggable basic configuration modes are done via standard CMIS/MSA interface via host NOS and service configuration (for example, L3/L3 services) via host NOS and Network Element Controller. XR modules point-to-multipoint optical network configuration is managed via IPM, where each individual XR modules connects via IP/CoAP protocol to IPM. To realize this IP communication link between IPM and XR module, network element may include XR Communication Agent SW component or alternative configure additional IP service for XR modules using dedicated VLAN connecting to local XR module. XR hub and leaf modules has an internal embedded bi-directional communication channel over optical path, allowing the leaf to share hub side IP connectivity towards IPM. Generally, IP access point can be on any or all hub and leaf modules. IPM has a north bound REST API available for external controllers, which the TeraFlowSDN XR Constellation driver connects to.

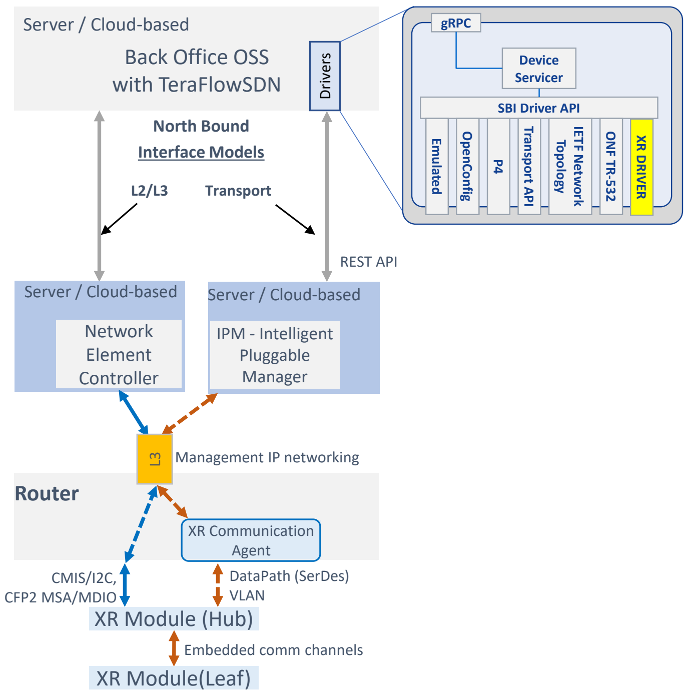


Figure 40. XR constellation management solution

Introduction

XR Constellation driver uses Infinera Intelligent Pluggable Manager (IPM) REST API to control XR pluggables. XR constellation driver behavior and functionality are developed, tested and demonstrated on two different environments: 1) emulated XR modules on XR Network emulator with IPM and SONiC device for 4x100G breakout constellation emulation and 2) with IPM and 400G SONiC device hosting XR pluggables on point-to-multipoint topology with traffic generator/analyzer in 400G VTI mode. XR transceiver 400G VTI mode allows Nx25 optical bandwidth allocation to VLAN for hub-leaf connection.

Introduction XR Network emulator environment for 4x100G breakout topology

Infinera developes XR network emulator environment, emulated XR pluggables software in hub-leaf roles with IP communication features/enhancements to support 1) both XR Communication Agent (XR-CA) use cases and DHCPv4/ND/DHCPv6/NTP (non XR-CA use case) and 2) XR modules hub-leaf internal network emulation and related functionality. XR network emulator setup is modelled to mimic 4x100G breakout network topology using 4x1G physical ports between SONiC L3 device and x86 server. When XR-CA is integrated to host router device, the XR modules IP address management is greatly simplified as XR modules uses host router IP addresses instead of dynamic IP address allocation via DHCPv4/ND/DHCPv6/NTP. With XR Network emulator, XR pluggable emulators and SONiC device usage, transition to real 400G SONiC and real XR pluggables is straightforward. XR pluggable emulators are used to develop IP functionalities and SW component/protocol stack extensions to support XR IP layer functionalities, as for example extensions to DHCP client modifications, ND extensions and other IP and internal communication channel selection and control logic. XR network emulation provides also virtualized leaf hosts, presenting connectivity through SONiC 1G ports to leaf devices.

On this environment, SONiC development focus on porting and integrating XR-CA SW component and verifying/hardening SONiC features to support both XR-CA and non-XRCA use case, and testing/verifying/hardening SONiC features.

IPM component is installed and maintained on x86 dedicated server, with additional services (DHCPv4/DHCPv6/NTP) supporting also non-XRCA use scenarios.

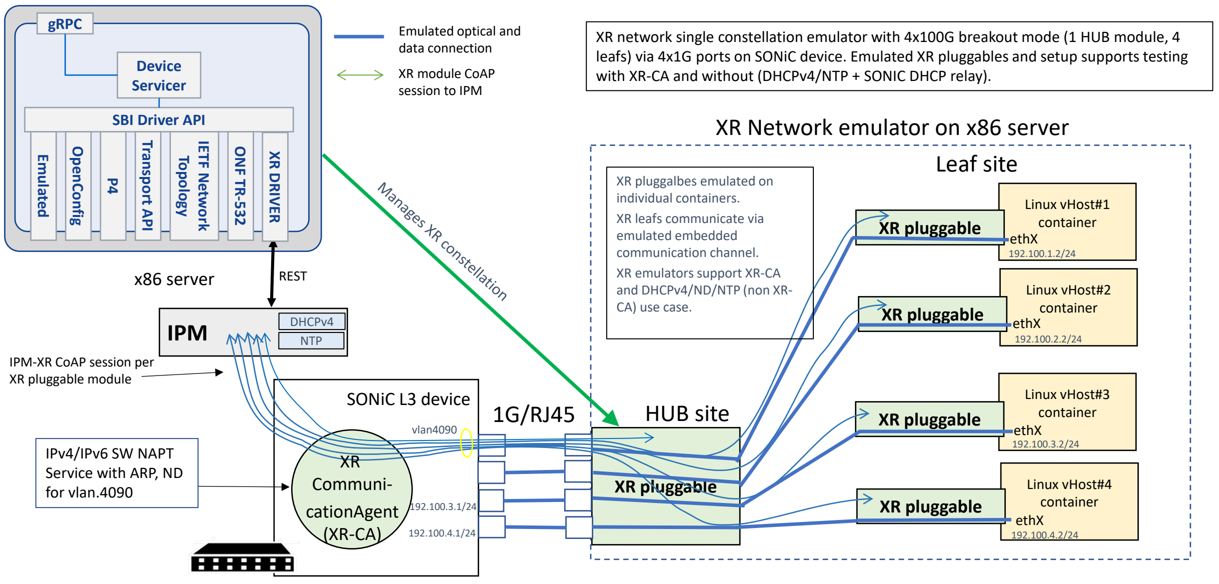

gRPC
Device Servicer
SBI Driver API
Emulated
OpenConfig
P4
Transport API
IETF Network Topology
ONF TR-532
XR DRIVER
x86 server
REST
IPM
DHCPv4
NTP
IPM-XR CoAP session per XR pluggable module
IPv4/IPv6 SW NAPT Service with ARP, ND for vlan.4090
SONiC L3 device
vlan4090
XR Communi-cationAgent (XR-CA)
192.100.3.1/24
192.100.4.1/24
1G/RJ45
Emulated optical and data connection
XR module CoAP session to IPM
Manages XR constellation
XR network single constellation emulator with 4x100G breakout mode (1 HUB module, 4 leafs) via 4x1G ports on SONiC device. Emulated XR pluggables and setup supports testing with XR-CA and without (DHCPv4/NTP + SONIC DHCP relay).
XR Network emulator on x86 server
Leaf site
XR pluggalbes emulated on individual containers.
XR leafs communicate via emulated embedded communication channel.
XR emulators support XR-CA and DHCPv4/ND/NTP (non XR-CA) use case.
HUB site
XR pluggable
XR pluggable
Linux vHost#1 container
ethX
192.100.1.2/24
XR pluggable
Linux vHost#2 container
ethX
192.100.2.2/24
XR pluggable
Linux vHost#3 container
ethX
192.100.3.2/24
XR pluggable
Linux vHost#4 container
ethX
192.100.4.2/24

Figure 41. TeraFowSDN and XR Network emulator environment

Introduction for XR transceivers 400G VTI-mode topology environment

Infinera develops the XR constellation driver to support XR VTI mode Nx25G bandwidth allocation and verifies its interoperability with IPM. XR VTI mode optical transport port is presented as {device}/{port}.{vlan}. Other components used in the environment include 400G SONiC device (Edgecore DSC240) hosting XR pluggables and XR-CA, 400G XR transceivers, optical splitter/decoupler and an external traffic generator/analyzer.

Infinera integrates and ports XR-CA SW component for SONiC/DCS240 with management interface support, enhances SONiC CMIS support and CLI commands for XR pluggable and verifies/hardens requires SONiC features with XR-CA and non XR-CA use case with DHCPv4/ND/NTP. Environment is used verify XR pluggables on different operation modes and optical parameters with SONiC. The environment is multipurposed, allowing for example XR module 4x100G breakout mode verification as well. Different XR pluggable firmware versions are verified on environment, providing feedback on XR transceivers interoperability on open SONiC Environment.

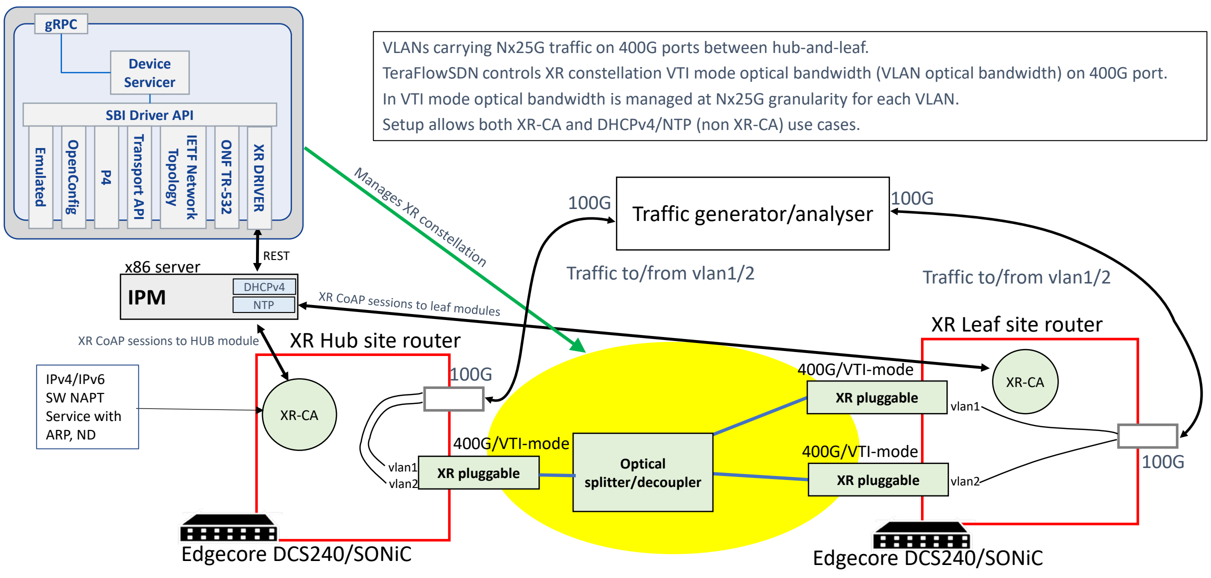

VLANs carrying Nx25G traffic on 400G ports between hub-and-leaf.
TeraFlowSDN controls XR constellation VTI mode optical bandwidth (VLAN optical bandwidth) on 400G port.
In VTI mode optical bandwidth is managed at Nx25G granularity for each VLAN.
Setup allows both XR-CA and DHCPv4/NTP (non XR-CA) use cases.

Figure 42 TeraFlowSDN and XR constellation 400G/VTI-mode environment

IPM REST interface

The attached embedded file contains the IPM REST interface description in the json file format (version v0.6.71). As new functionality is added to IPM, REST API and the json description will evolve accordingly. The current version is available at: https://labs.etsi.org/rep/tfs/controller/-/blob/develop/src/device/service/drivers/xr/ipm_rest_api_0_6_71.json

Table 22. XR Constellation driver RPCs

Device Driver RPC	Released on	Description
Connect	v2	Initiates a connection to IPM at the specified address and acquires access token using provided username and password parameters. Stores provide constellation hub module name parameter to scope driver instance constellation.
Disconnect	v2	Tears down connection to IPM.
GetConfig	v2	Optains topology and interfaces of the XR constellation.

GetInitialConfig	v2	Currently not needed, implemented as no-op.
SetConfig	v2	Creates or update bandwidth and connectivity of XR constellation to match provided parameters.
DeleteConfig	v2	Updated bandwith and connectivity of the XR constellation to reflected deleted services.

4.1.4. Final Interfaces

SBI is the TeraFlowSDN component in charge of interacting with the underlying network equipment or external/hierarchical controllers. Different protocols and data models might be needed to manage the network equipment or external controllers; for this reason, the Device component provides a Driver API that enables developers to implement new drivers and integrate them into the TeraFlowSDN.

Generic device driver Functions

- **Initialization** of the Driver provides a setting named as “endpoints” to define the endpoints that will be exposed by the device or external controller;
- **Connect** and **Disconnect** connects and disconnects to device or external controller;
- **GetInitialConfig** and **GetConfig** retrieve an initial configuration and the current configuration set for the device. **GetConfig** supports filtering of the resources according to the parameters specified for the method;
- **SetConfig** and **DeleteConfig** update and delete the configured resources for the driver;
- **SubscribeState** and **UnsubscribeState** activate and deactivate the monitoring of specific resources previously configured;
- **GetState** periodically retrieves state of device and provide information towards TeraFlowSDN.

4.1.5. Final Operational Workflows

The operational workflow for the Device component is depicted in Figure 43. The workflow has been generalized to cover all the Device Driver types. Besides, the workflow assumes a request from the Service component while not mandatory.

The workflow starts when some component, e.g., the Service component, triggers a device configuration, for instance, due to the processing of an UpdateService request. When the Device component receives the ConfigureDevice request, it first interrogates the Context component to gather the most up-to-date known state of the device stored on the Context component. Based on that information, it decides the appropriate Driver to be used. Next, it instantiates the selected Driver and, for each configuration rule to modified in the device, it first interrogates the device to get the most up-to-date state related to that configuration rule. Then, it correlates the state of the Context component with that of the device and decides the between updating or deleting the rules, if needed, or just keeping the rules as they are in the device. When all the requested configuration rules are processed, it updates the Context component with the most up-to-date device and configuration status according to the changes made. Finally, it returns the control to the caller service.

It is worth noting that, even when this workflow is general, it is complete enough to cover all the cases supported by the different Drivers. However, even when all Drivers implement the same API, the interactions between the Drivers and the devices are particular to each protocol, device and driver used. Besides, this workflow focuses on the details related to the Device component; for those details related to other components, such as the Service component, check section 4.2.4.

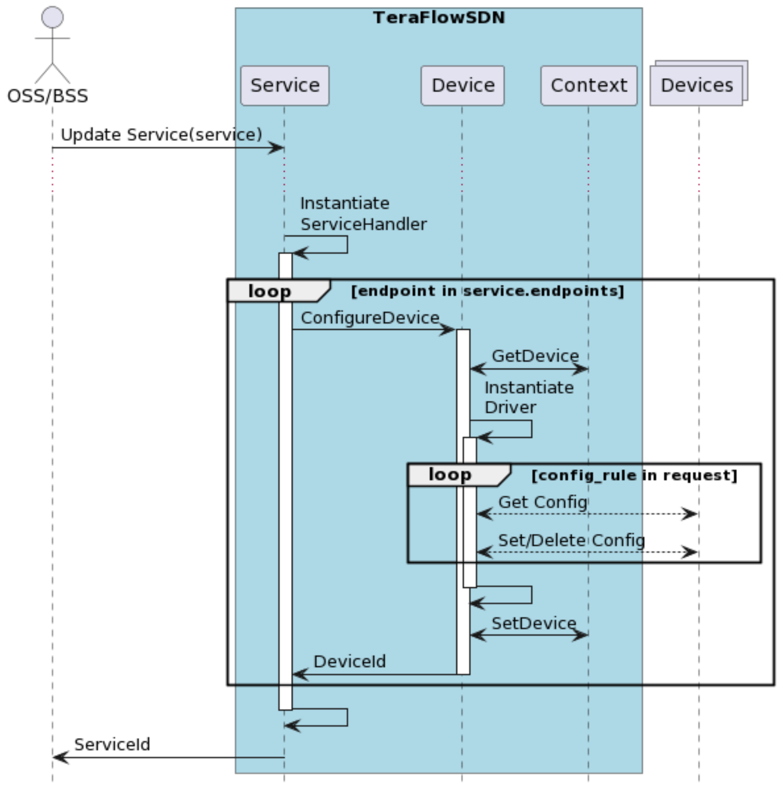


Figure 43: Exemplary workflow of a generic device configuration

4.1.6. Evaluation

In this section, we evaluate the functionality and performance of the different device drivers supported by the TeraFlow controller. Each of them is described in terms of the experimental setup used for the evaluation, and the evaluation results, which might include functional and performance results.

4.1.6.1. Emulated Device Driver Plugin

Experimental setup

The setup is deployed on a server equipped with a 24-core AMD Ryzen Threadripper 3960X processor clocked at 2.2 GHz, with 32 GB of DDR4 RAM, and 4TB HDD storage. On this server, TeraFlowSDN is deployed on top of MicroK8s v1.24.8 and no resource limits are set in order to allow stress testing.

The microservices deployed for the benchmark include: *i*) 1 replica of the Slice component; *ii*) 5 replicas of the Service component; *iii*) 5 replicas of the PathComp pod (e.g., Front-End and Back-End containers); *iv*) 1 replica of the Device component; and *v*) 1 replica of the Compute component. All pods are deployed behind a Kubernetes service for load balancing purposes (when applicable).

The transport network topology used for this assessment is that shown in Figure 27 and used for the PathComp performance evaluation. The topology is formed by 7 packet switches/routers which are controlled/programmed by a TeraFlow SDN controller instance. It is worth mentioning that the data plane is emulated, and only the controller building blocks are considered to collect the numerical results.

Numerical Results

The considered metric for assessing the Emulated Driver is the delay incurred by the driver in retrieving/storing/deleting the requested configuration rules in an internal memory. The elapsed time only takes into consideration the driver's internal processing time. To produce the CDF of the Emulated Driver delay, 100 requests are generated uniformly, selecting between L2 and L3 network services and slices. The endpoints of every request are chosen randomly from the transport topology. Each request is generated with a Poisson statistical model whose inter-arrival time is set to 200ms while the duration of the service/slice is modelled exponentially with a holding time of 10s.

In Figure 44, it is shown the CDF of the Emulated Device Driver latency for the generated 100 requests. It is worth noting that each request implies several rule configurations, retrievals and deletions. We observe that 80% of the requests changing the configuration take less than 1 ms. Besides, retrieval of the configuration takes longer due to having to explore the whole configuration rule memory kept by the emulated driver.

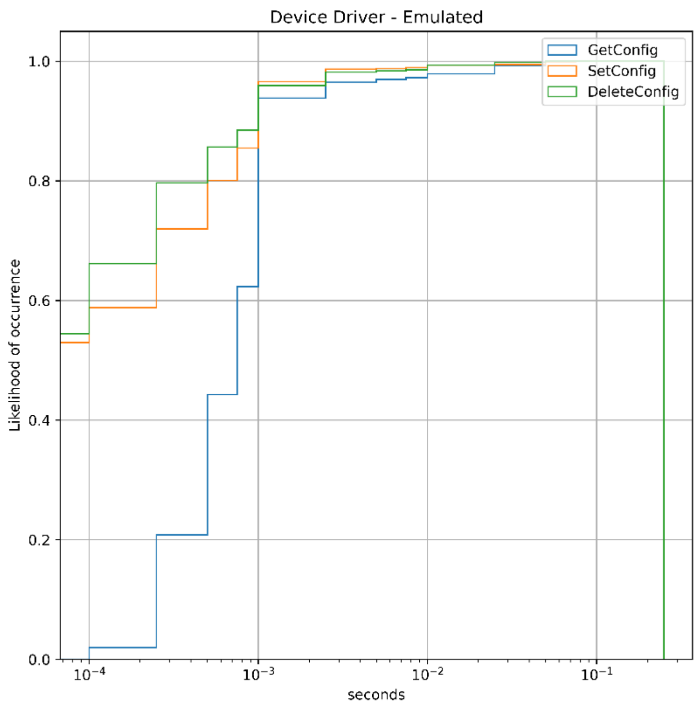


Figure 44: CDF for the Emulated Device Driver Delay.

We repeated the experiment two additional times only, including L2 services (Figure 45) and L3 services (Figure 46) in an isolated manner. We observe that L2 configuration rules take longer than the L3 counterpart. The reason is that L3 services generally include almost twice the number of configuration rules required for L2 due to the additional packet routing rules.

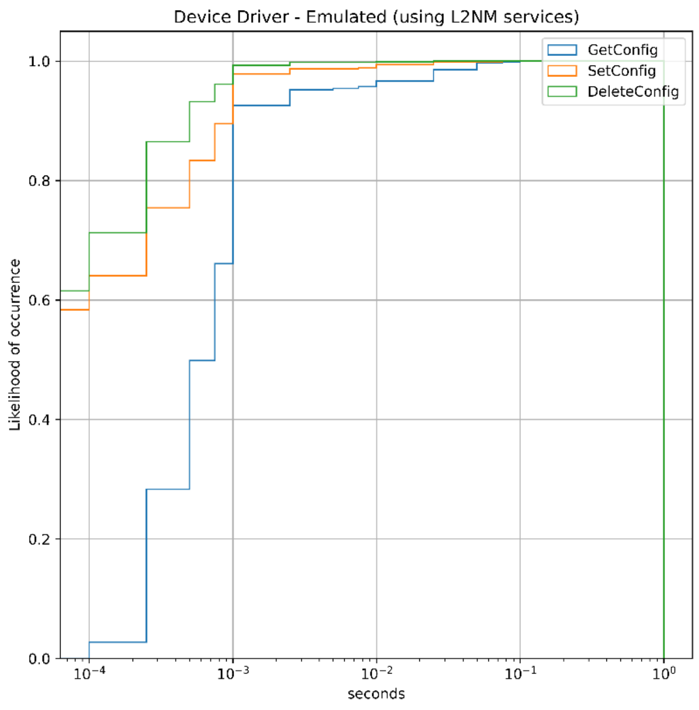


Figure 45: CDF for the Emulated Device Driver Delay (only L2 services).

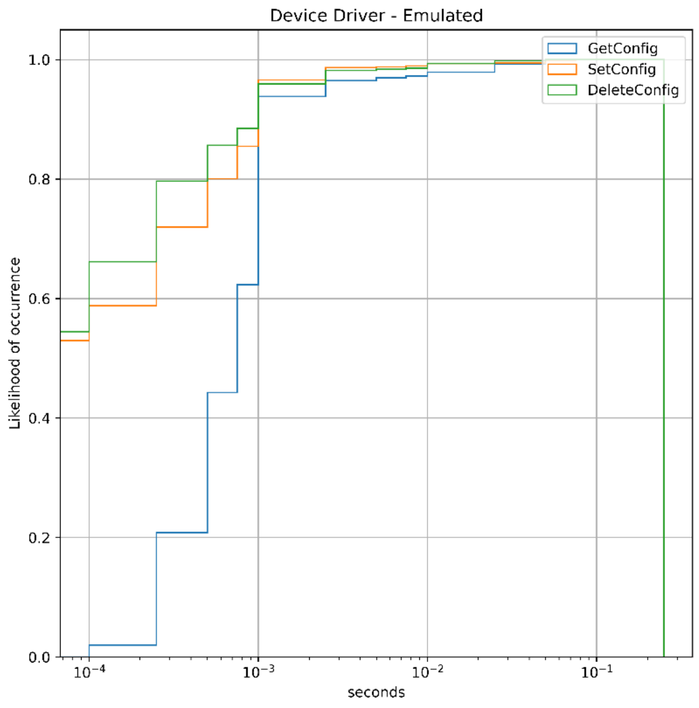


Figure 46: CDF for the Emulated Device Driver Delay (only L3 services).

4.1.6.2. OLS ONF Transport API Driver Plugin

Experimental setup

We used the same setup (server and micro-service deployment) described for the Emulated Device Driver (see section 4.1.6.1) extended with a proprietary Open Line System controller exposing a Transport API NBI and controlling an emulated underlying optical data plane formed by 4 nodes with 10 service interconnection points each of them.

Numerical Results

The considered metric for assessing the Transport API Driver is the delay incurred by the driver in retrieving/storing/deleting the configuration rules from/to the OLS controller, plus the dispatching time of these requests within the OLS controller. To produce the CDF of the Transport API Driver delay, 100 requests are generated. The endpoints of every request are chosen randomly from the abstract node extrapolated from the OLS controller where these endpoints are mapped to the SIPs of the TAPI context. Each request is generated with a Poisson statistical model whose inter-arrival time is set to 200ms while the duration of the requests is modelled exponentially with a holding time of 10s.

In Figure 47, it is shown the CDF of the Transport API Device Driver latency for the generated 100 requests. We observe that most configuration/deconfiguration requests take around 10 ms, while configuration retrieval requests take around 100 ms.

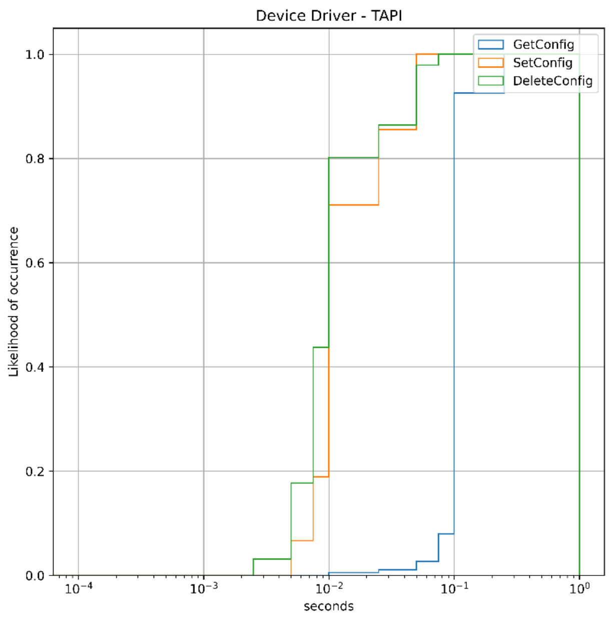


Figure 47: CDF for the Transport API Device Driver Delay.

4.1.6.3. ONF TR-532 Microwave Driver Plugin

Experimental setup

We used the same setup (server and micro-service deployment) described for the Emulated Device Driver (see section 4.1.6.1) extended with a proprietary SIAE MW intermediate controller exposing both IETF Network and ONF TR-532 models and controlling an underlying physical data plane formed by 2 nodes connected by a MW radio link traversed by an L2 services interconnecting 2 edge endpoints. The MW transport network topology used for this assessment is that shown in Figure 48.

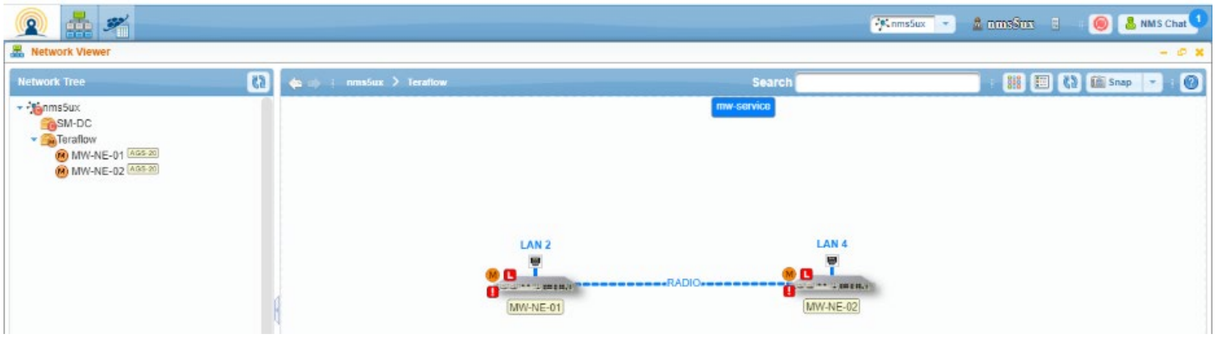


Figure 48: MW Transport Network Topology.

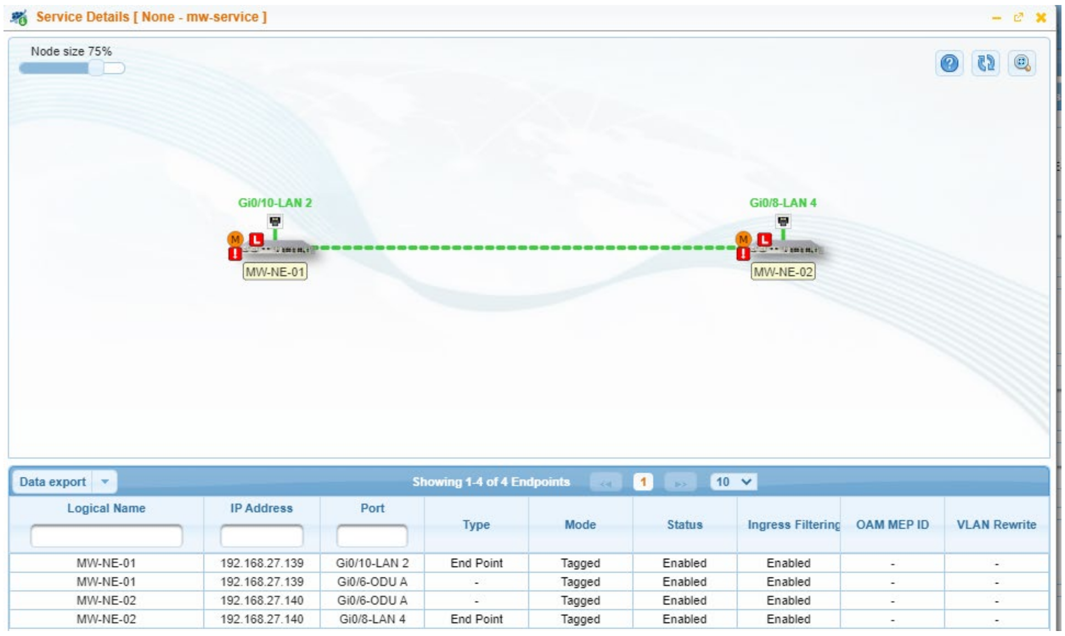
Functional Evaluation

We first evaluated the driver from a functional point of view. For that, we configured a single service named “mw-service” and checked that the configuration landed on the MicroWave controller and configured the devices. The specifications of the service are provided in Table 23.

Table 23. MicroWave Test Service Specifications

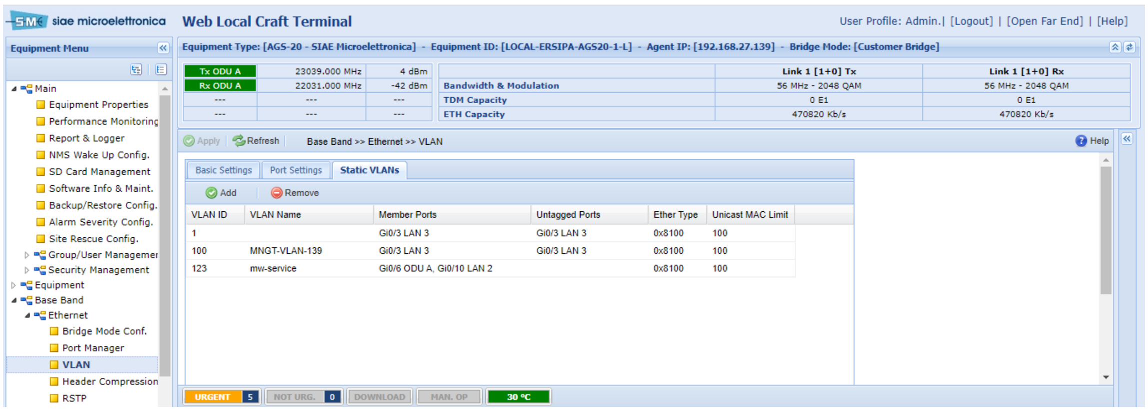
Service UUID	Service Type	Src EndPoint	Dst EndPoint	VLAN Id
mw-service	L2NM	192.168.27.139:10	192.168.27.140:8	123

Figure 49 depicts the resulting service in the MicroWave controller after receiving and configuring the request. Besides, Figure 50 and Figure 51 show the configuration offloaded by the MW controller into the microwave devices 192.168.27.139 and 192.168.27.140. In the figures, ports Gi0/6 correspond to the antennas, while ports Gi0/10 (LAN2) of 192.168.27.139 and Gi0/8 (LAN4) of 192.168.27.140 correspond to the LAN ports connected to packet devices.


Service Details [None - mw-service]
Node size 75%
Gi0/10-LAN 2
Gi0/8-LAN 4
MW-NE-01
MW-NE-02
Data export
Showing 1-4 of 4 Endpoints

Logical Name	IP Address	Port	Type	Mode	Status	Ingress Filtering	OAM MEP ID	VLAN Rewrite
MW-NE-01	192.168.27.139	Gi0/10-LAN 2	End Point	Tagged	Enabled	Enabled	-	-
MW-NE-01	192.168.27.139	Gi0/6-ODU A	-	Tagged	Enabled	Enabled	-	-
MW-NE-02	192.168.27.140	Gi0/6-ODU A	-	Tagged	Enabled	Enabled	-	-
MW-NE-02	192.168.27.140	Gi0/8-LAN 4	End Point	Tagged	Enabled	Enabled	-	-

Figure 49: Test “mw-service” on the MW controller configured by TeraFlow controller.

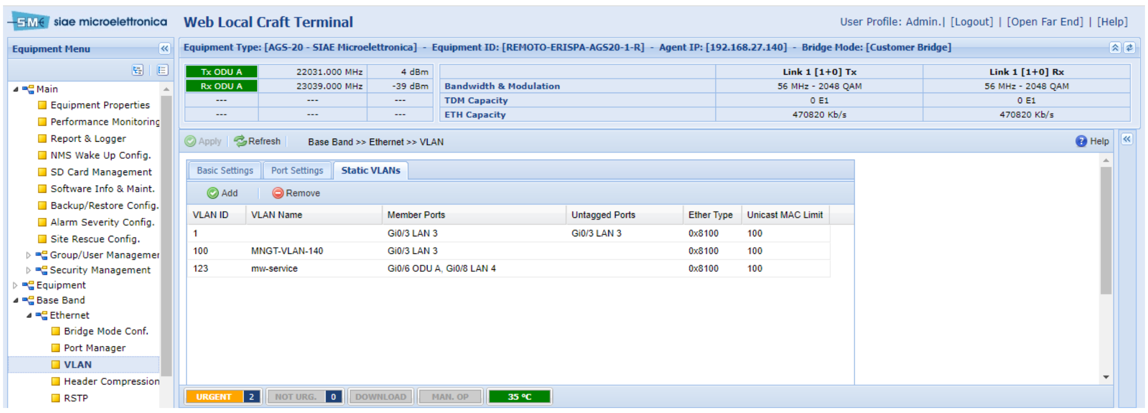


Equipment Type: [AGS-20 - SIAE Microelettronica] - Equipment ID: [LOCAL-ERSIPA-AGS20-1-L] - Agent IP: [192.168.27.139] - Bridge Mode: [Customer Bridge]

				Link 1 [1+0] Tx	Link 1 [1+0] Rx
Tx ODU A	23039.000 MHz	4 dBm			
Rx ODU A	22031.000 MHz	-42 dBm	Bandwidth & Modulation	56 MHz - 2048 QAM	56 MHz - 2048 QAM
---	---	---	TDM Capacity	0 E1	0 E1
---	---	---	ETH Capacity	470820 Kb/s	470820 Kb/s

VLAN ID	VLAN Name	Member Ports	Untagged Ports	Ether Type	Unicast MAC Limit
1		Gi0/3 LAN 3	Gi0/3 LAN 3	0x8100	100
100	MNGT-VLAN-139	Gi0/3 LAN 3	Gi0/3 LAN 3	0x8100	100
123	mw-service	Gi0/6 ODU A, Gi0/10 LAN 2		0x8100	100

Figure 50: Test service "mw-service" configured on device 192.168.27.139.



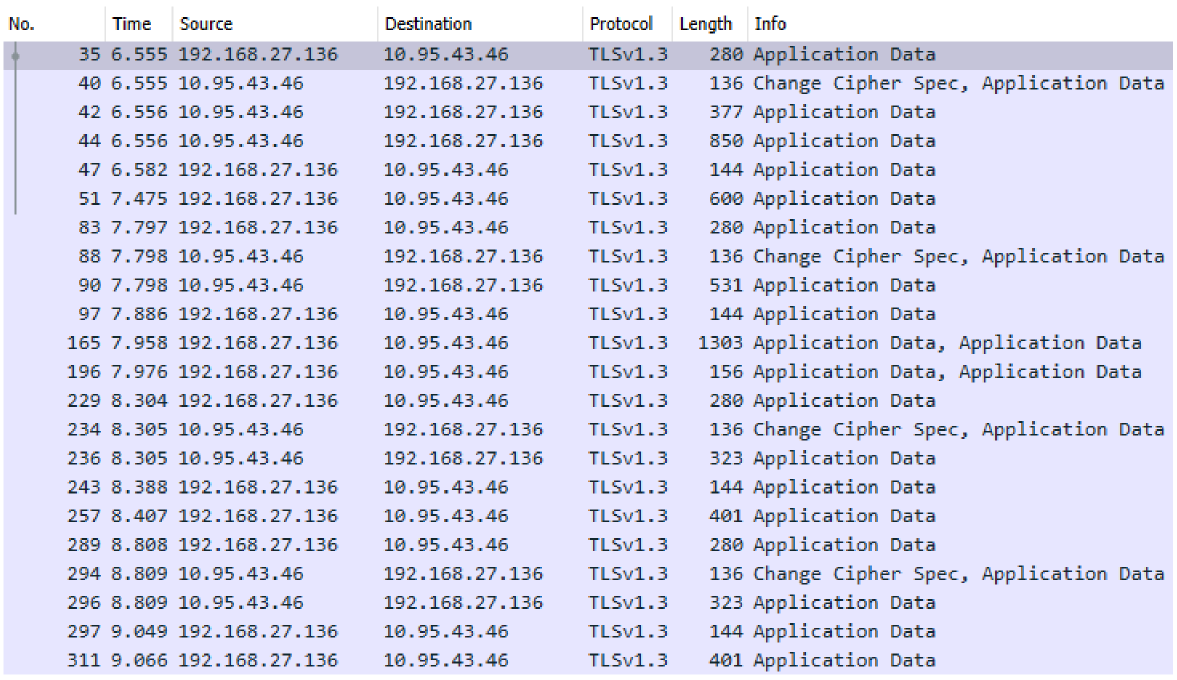
Equipment Type: [AGS-20 - SIAE Microelettronica] - Equipment ID: [REMOTO-ERISPA-AGS20-1-R] - Agent IP: [192.168.27.140] - Bridge Mode: [Customer Bridge]

				Link 1 [1+0] Tx	Link 1 [1+0] Rx
Tx ODU A	22031.000 MHz	4 dBm			
Rx ODU A	23039.000 MHz	-39 dBm	Bandwidth & Modulation	56 MHz - 2048 QAM	56 MHz - 2048 QAM
---	---	---	TDM Capacity	0 E1	0 E1
---	---	---	ETH Capacity	470820 Kb/s	470820 Kb/s

VLAN ID	VLAN Name	Member Ports	Untagged Ports	Ether Type	Unicast MAC Limit
1		Gi0/3 LAN 3	Gi0/3 LAN 3	0x8100	100
100	MNGT-VLAN-140	Gi0/3 LAN 3		0x8100	100
123	mw-service	Gi0/6 ODU A, Gi0/8 LAN 4		0x8100	100

Figure 51: Test service "mw-service" configured on device 192.168.27.140.

Finally, Figure 52 illustrates the Wireshark capture of the communication; note that communication is encrypted for security reasons, thus only encrypted messages are shown.



```
No.      Time   Source          Destination     Protocol  Length Info
     35 6.555 192.168.27.136  10.95.43.46     TLSv1.3    280 Application Data
     40 6.555 10.95.43.46     192.168.27.136  TLSv1.3    136 Change Cipher Spec, Application Data
     42 6.556 10.95.43.46     192.168.27.136  TLSv1.3    377 Application Data
     44 6.556 10.95.43.46     192.168.27.136  TLSv1.3    850 Application Data
     47 6.582 192.168.27.136  10.95.43.46     TLSv1.3    144 Application Data
     51 7.475 192.168.27.136  10.95.43.46     TLSv1.3    600 Application Data
     83 7.797 192.168.27.136  10.95.43.46     TLSv1.3    280 Application Data
     88 7.798 10.95.43.46     192.168.27.136  TLSv1.3    136 Change Cipher Spec, Application Data
     90 7.798 10.95.43.46     192.168.27.136  TLSv1.3    531 Application Data
     97 7.886 192.168.27.136  10.95.43.46     TLSv1.3    144 Application Data
    165 7.958 192.168.27.136  10.95.43.46     TLSv1.3   1303 Application Data, Application Data
    196 7.976 192.168.27.136  10.95.43.46     TLSv1.3    156 Application Data, Application Data
    229 8.304 192.168.27.136  10.95.43.46     TLSv1.3    280 Application Data
    234 8.305 10.95.43.46     192.168.27.136  TLSv1.3    136 Change Cipher Spec, Application Data
    236 8.305 10.95.43.46     192.168.27.136  TLSv1.3    323 Application Data
    243 8.388 192.168.27.136  10.95.43.46     TLSv1.3    144 Application Data
    257 8.407 192.168.27.136  10.95.43.46     TLSv1.3    401 Application Data
    289 8.808 192.168.27.136  10.95.43.46     TLSv1.3    280 Application Data
    294 8.809 10.95.43.46     192.168.27.136  TLSv1.3    136 Change Cipher Spec, Application Data
    296 8.809 10.95.43.46     192.168.27.136  TLSv1.3    323 Application Data
    297 9.049 192.168.27.136  10.95.43.46     TLSv1.3    144 Application Data
    311 9.066 192.168.27.136  10.95.43.46     TLSv1.3    401 Application Data
```

Figure 52: WireShark capture of communication between TeraFlow controller and MW controller.

Numerical Results

The considered metric for assessing the MW Device Driver is the delay incurred by the driver in retrieving/creating/deleting the L2 services between 2 edge endpoints of the MW domain, plus the time spent by the MW controller in configuring the underlying devices.

To produce the CDF of the MW Device Driver delay, 100 requests are generated. The endpoints of every request are chosen randomly from the LAN ports exposed by the abstract node extrapolated from the MW controller. Each request is generated with a Poisson statistical model whose inter-arrival time is set to 200ms while the duration of the requests is modelled exponentially with a holding time of 10s.

In Figure 53, it is shown the CDF of the MicroWave Device Driver latency for the generated 100 requests. We observe that most retrieve/configuration requests take around 1s, while deconfiguration requests take 5 seconds or more.

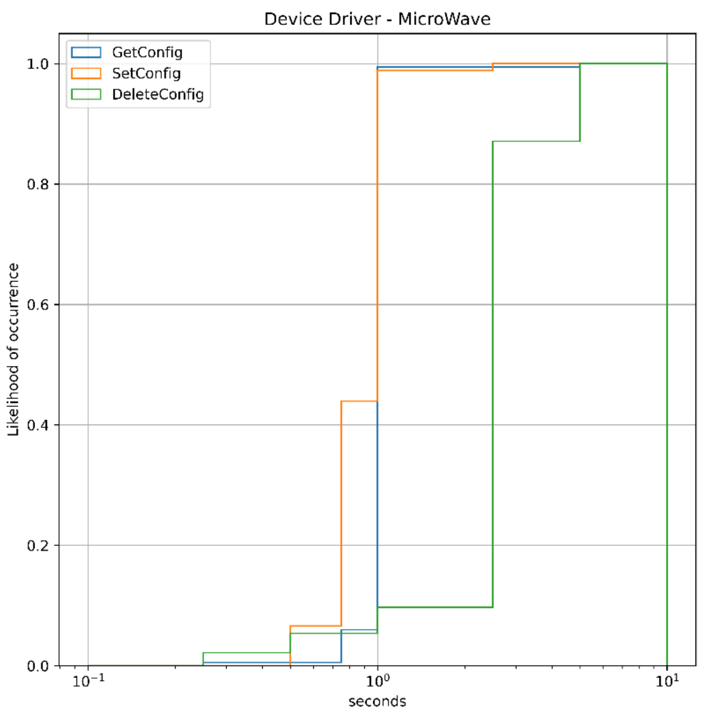


Figure 53: CDF for the MicroWave Device Driver Delay.

4.1.6.4. OpenConfig Driver Plugin

Experimental setup

To conduct the experiments of the OpenConfig device driver, we used a Virtual Machine running on top of OpenStack with an 8 virtual-core CPU (Intel Xeon Skylake IBRS Processor clocked at 2GHz) and 16 GB of DDR4 RAM. On this VM, TeraFlowSDN is deployed within MicroK8s v1.24.8.

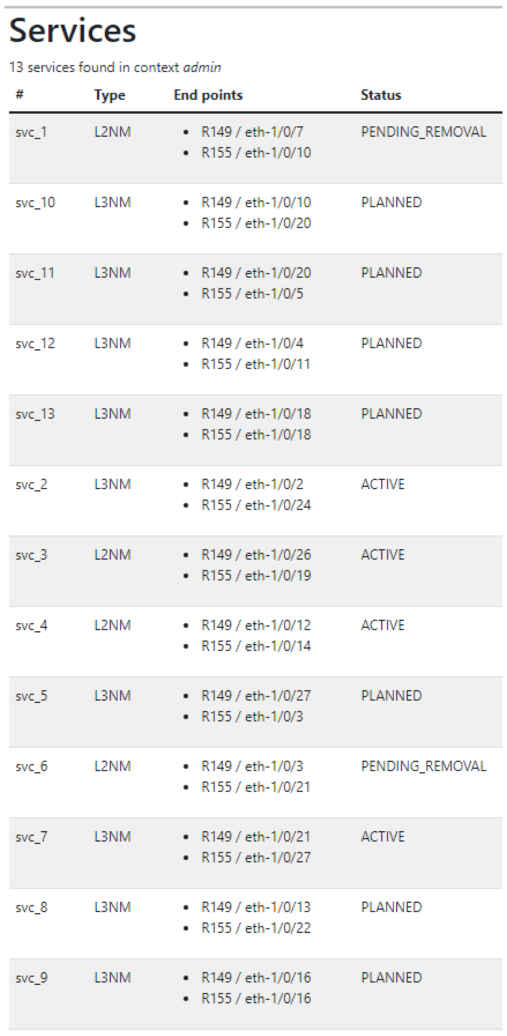
The microservices deployed for the benchmark include: *i*) 1 replica of the Slice component; *ii*) 1 replicas of the Service component; *iii*) 1 replicas of the PathComp pod (e.g., Front-End and Back-End containers); *iv*) 1 replica of the Device component; and *v*) 1 replica of the Compute component. All pods are deployed behind a Kubernetes service for load balancing purposes (when applicable).

The transport network topology used for this assessment is that shown in Figure 54. The topology is formed by 2 x Edgecore DRX-30 packet routers each running ADVA software NOS-OPX-B-21.1.1(8769) which are controlled/programmed by a TeraFlow SDN controller instance. Each router has 26 usable endpoints + 1 endpoint used for the link between the routers.

Figure 54: Topology used to assess the OpenConfig Device Driver.

Functional evaluation

To assess the capacity of TeraFlowSDN to configure the packet routers using OpenConfig, we triggered the creation of a number of services and retrieved the internal configuration of the packet routers where the services were configured. Figure 55 depicts the list of services in TeraFlowSDN WebUI while Table 24 contains the relevant segments of the resulting configuration deployed on the routers. Note that the interface connecting the routers among them is “eth-1/0/25.55”.



Services

13 services found in context *admin*

#	Type	End points	Status
svc_1	L2NM	• R149 / eth-1/0/7 • R155 / eth-1/0/10	PENDING_REMOVAL
svc_10	L3NM	• R149 / eth-1/0/10 • R155 / eth-1/0/20	PLANNED
svc_11	L3NM	• R149 / eth-1/0/20 • R155 / eth-1/0/5	PLANNED
svc_12	L3NM	• R149 / eth-1/0/4 • R155 / eth-1/0/11	PLANNED
svc_13	L3NM	• R149 / eth-1/0/18 • R155 / eth-1/0/18	PLANNED
svc_2	L3NM	• R149 / eth-1/0/2 • R155 / eth-1/0/24	ACTIVE
svc_3	L2NM	• R149 / eth-1/0/26 • R155 / eth-1/0/19	ACTIVE
svc_4	L2NM	• R149 / eth-1/0/12 • R155 / eth-1/0/14	ACTIVE
svc_5	L3NM	• R149 / eth-1/0/27 • R155 / eth-1/0/3	PLANNED
svc_6	L2NM	• R149 / eth-1/0/3 • R155 / eth-1/0/21	PENDING_REMOVAL
svc_7	L3NM	• R149 / eth-1/0/21 • R155 / eth-1/0/27	ACTIVE
svc_8	L3NM	• R149 / eth-1/0/13 • R155 / eth-1/0/22	PLANNED
svc_9	L3NM	• R149 / eth-1/0/16 • R155 / eth-1/0/16	PLANNED

Figure 55: List of Services configured by TeraFlowSDN using OpenConfig Driver.

Table 24. Configuration of Packet Routers done by TeraFlowSDN OpenConfig Driver

`hostname R149`
`router-id 5.5.5.5`
`!`
`port eth-1/0/2`
`   lldp transmit`
`   lldp receive`
`   enable`
`port eth-1/0/12`
`   lldp transmit`
`   lldp receive`
`   enable`
`port eth-1/0/21`
`   lldp transmit`
`   lldp receive`
`   enable`
`port eth-1/0/25`
`   lldp transmit`
`   lldp receive`
`   flow-monitoring`
`   enable`
`port eth-1/0/26`
`   lldp transmit`
`   lldp receive`
	`hostname R155`
	`router-id 5.5.5.1`
	`!`
	`port eth-1/0/14`
	`   lldp transmit`
	`   lldp receive`
	`   enable`
	`port eth-1/0/19`
	`   lldp transmit`
	`   lldp receive`
	`   enable`
	`port eth-1/0/24`
	`   lldp transmit`
	`   lldp receive`
	`   enable`
	`port eth-1/0/25`
	`   lldp transmit`
	`   lldp receive`
	`   flow-monitoring`
	`   enable`
	`port eth-1/0/27`
	`   lldp transmit`
	`   lldp receive`

```
   enable
!
interface inet eth-1/0/25.55
   ip address 99.5.55.2/24
   outer-tags 55
   ip ospf
   ip ospf network point-to-point
   ldp
   enable
interface ac eth-1/0/12.104
   outer-tags 104
   enable
interface ac eth-1/0/26.103
   outer-tags 103
   enable
!
router l3vpn
!
e-lan ELAN-AC:103 103
   vc VC-1 peer 5.5.5.1
   ac eth-1/0/26.103
   enable
e-lan ELAN-AC:104 104
   vc VC-1 peer 5.5.5.1
   ac eth-1/0/12.104
   enable
!
vrf svc_2-NetInst 2
   interface inet eth-1/0/2.102
      mtu 3000
      ip address 10.1.1.149/16
      outer-tags 102
      enable
   router bgp
      local-as 65002
      redistribute connected
   l3vpn route-distinguisher 65002:102
vrf svc_7-NetInst 7
   interface inet eth-1/0/21.107
      mtu 3000
      ip address 10.10.0.149/16
      outer-tags 107
      enable
   router bgp
      local-as 65007
      redistribute connected
   l3vpn route-distinguisher 65007:107
```

```
   enable
!
interface inet eth-1/0/25.55
   ip address 99.5.55.1/24
   outer-tags 55
   ip ospf
   ip ospf network point-to-point
   ldp
   enable
interface ac eth-1/0/14.104
   outer-tags 104
   enable
interface ac eth-1/0/19.103
   outer-tags 103
   enable
!
router l3vpn
!
e-lan ELAN-AC:103 103
   vc VC-1 peer 5.5.5.5
   ac eth-1/0/19.103
   enable
e-lan ELAN-AC:104 104
   vc VC-1 peer 5.5.5.5
   ac eth-1/0/14.104
   enable
!
vrf svc_2-NetInst 2
   interface inet eth-1/0/24.102
      mtu 3000
      ip address 10.1.1.155/16
      outer-tags 102
      enable
   router bgp
      local-as 65002
      redistribute connected
   l3vpn route-distinguisher 65002:102
vrf svc_7-NetInst 7
   interface inet eth-1/0/27.107
      mtu 3000
      ip address 10.10.0.155/16
      outer-tags 107
      enable
   router bgp
      local-as 65007
      redistribute connected
    l3vpn route-distinguisher 65007:107
```

Numerical Results

The considered metric for assessing the OpenConfig Driver is the delay incurred by the driver in retrieving/storing/deleting the configuration rules from/to the routers, including the time spent by the routers to process the changes. To produce the CDF of the OpenConfig Driver delay, 100 requests are generated. The endpoints of every request are chosen randomly from those ports available and not used in the routers. Each request is generated with a Poisson statistical model whose inter-arrival time is set to 1s while the duration of the requests is modelled exponentially with a holding time of 10s.

In Figure 56, it is shown the CDF of the OpenConfig Device Driver latency for the generated 100 requests. We observe that the majority of configuration/deconfiguration requests take 1 second or more, while configuration retrieval requests take around 10s. It is worth noting that this delay is almost 100% due to the processing time in the device. In particular, it can be seen that DeleteConfig, in general, takes less time than SetConfig, and GetConfig is the operation consuming more time since the entire data model needs to be traversed by the device NOS.

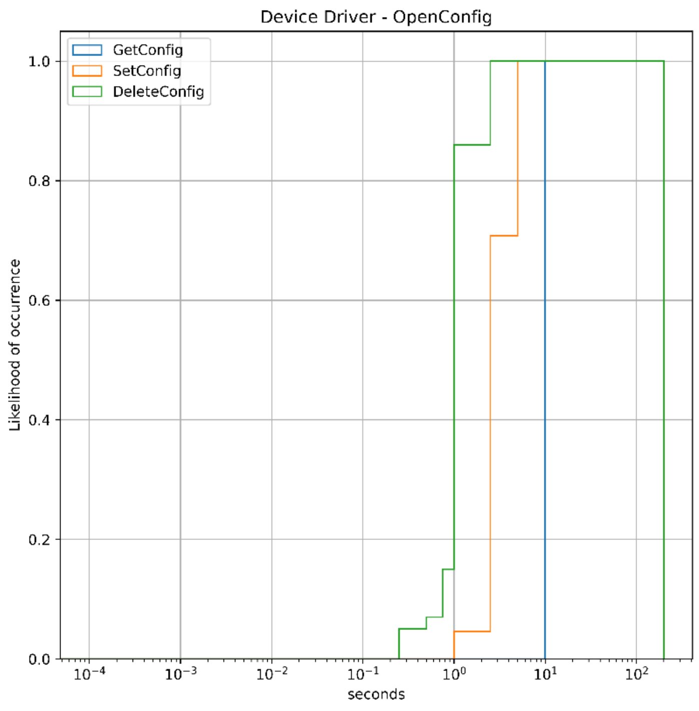


Figure 56: CDF for the OpenConfig Device Driver Delay.

4.1.6.5. P4 Whitebox Switches Driver Plugin

Experimental setup

This setup is deployed on a Dell PowerEdge R7515 chassis equipped with a 64-core AMD EPYC 7763 processor clocked at 2.45 GHz, with 256 GB of DDR4 RAM at 3200 MHz, and 2TB NVMe storage. The AMD processor is also equipped with 2MB L1 data cache, 2MB L1 instruction cache, 32MB L2 cache, and 256MB L3 cache. The L3 is shared among all 64 cores, while the rest are provisioned per-core.

TeraFlowSDN is deployed as a Kubernetes service on this server, and no resource limits are set to the SBI and Context components to allow stress testing.

Benchmarking procedure

To benchmark the P4 SBI driver we use an emulated (Mininet) topology of one or more software based P4 switches. We used the bmv2 P4 switch, which is already integrated into Mininet by ONF [14]. The objective of the benchmark is twofold:

- <u>Rule cardinality effects to quantify the latency:</u>
  - to install an exponentially increasing set of rules into a single P4 device.
  - to retrieve an exponentially increasing set of rules from a single P4 device.
  - to delete an exponentially increasing set of rules from a single P4 device.

- <u>Topology size effects to quantify the latency:</u>
    - to install a fixed configuration into a topology with increasing number of P4 devices.
    - to retrieve the configuration from a topology with increasing number of P4 devices.
    - to delete the configuration from a topology with increasing number of P4 devices.

P4 SBI driver performance vs. rule cardinality benchmark

The following benchmark measures the total execution time required by the P4 SBI driver to perform rule installation/retrieval/deletion operations on a single P4 device using an exponentially increasing number of rules in the ruleset. The length of the ruleset is 1, 10, 100, and 1000 rules respectively.

<u>Rule installation:</u> Figure 57 shows the total execution time (y axis) vs. time (x axis) during the execution of SetConfig RPCs, each with 1, 10, 100, and 1000 P4 device rules respectively. The graph is annotated with color-highlighted points, which indicate the time that each RPC was concluded:

i) The red point refers to the SetConfig RPC with 1 rule. This RPC is executed within 3.2 milliseconds.
ii) The green point refers to the SetConfig RPC with 10 rules. This RPC is executed within 16milliseconds.
iii) The blue point refers to the SetConfig RPC with 100 rules. This RPC is executed within 198 milliseconds.
iv) The white point refers to the SetConfig RPC with 1000 rules. This RPC is executed within 2.5 seconds.

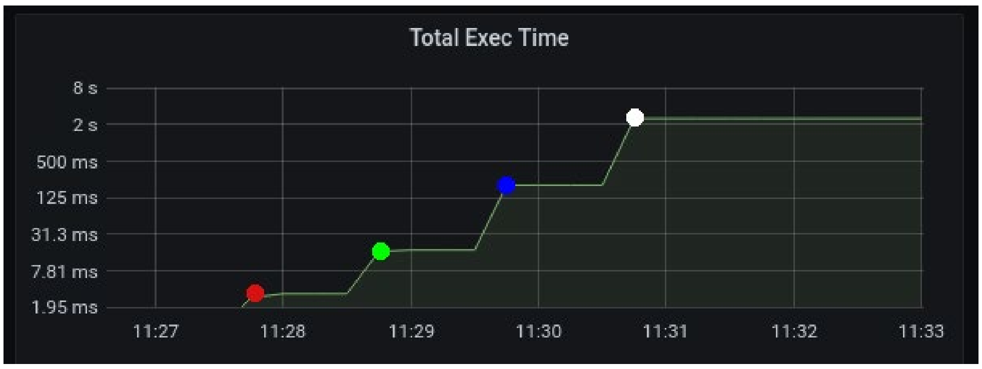


Figure 57: Latency to install P4 device configuration (SetConfig RPC) with exponentially increasing number of rules (1,10,100,1000).

<u>Rule retrieval:</u> Figure 58 shows the total execution time (y axis) vs. time (x axis) during the execution of GetConfig RPCs, each with 1, 10, 100, and 1000 P4 device rules respectively. The graph is annotated with color-highlighted points, which indicate the time that each RPC was concluded:

v) The red point refers to the GetConfig RPC with 1 rule. This RPC is executed within 12 milliseconds.
vi) The green point refers to the GetConfig RPC with 10 rules. This RPC is executed within 13 milliseconds.
vii) The blue point refers to the GetConfig RPC with 100 rules. This RPC is executed within 12 milliseconds.
viii) The white point refers to the GetConfig RPC with 1000 rules. This RPC is executed within 31.5 milliseconds.

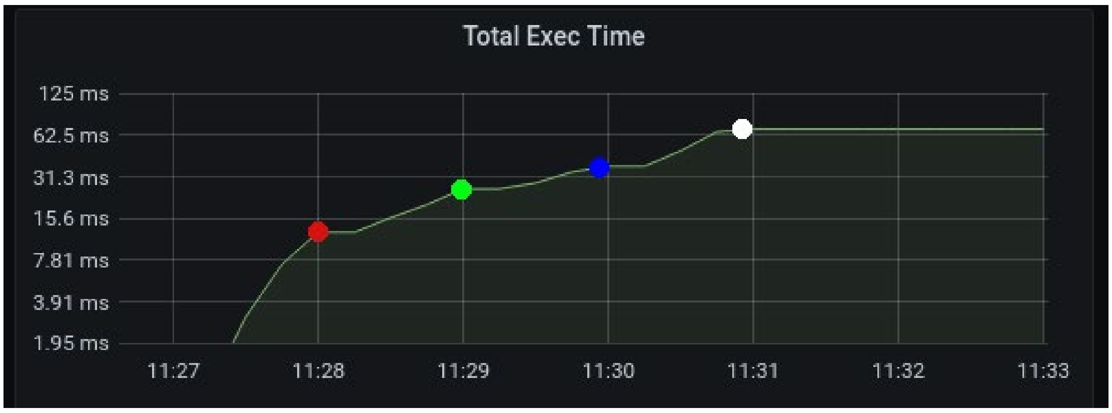


Figure 58: Latency to retrieve P4 device configuration (GetConfig RPC) with exponentially increasing number of rules (1,10,100,1000).

<u>Rule deletion:</u> Figure 59 shows the total execution time (y axis) vs. time (x axis) during the execution of DeleteConfig RPCs, each with 1, 10, 100, and 1000 P4 device rules respectively. The graph is annotated with color-highlighted points, which indicate the time that each RPC was concluded:

ix) The red point refers to the DeleteConfig RPC with 1 rule. This RPC is executed within 2.5 milliseconds.
x) The green point refers to the DeleteConfig RPC with 10 rules. This RPC is executed within 24.5 milliseconds.
xi) The blue point refers to the DeleteConfig RPC with 100 rules. This RPC is executed within 166 milliseconds.
xii) The white point refers to the DeleteConfig RPC with 1000 rules. This RPC is executed within 1.7 seconds.

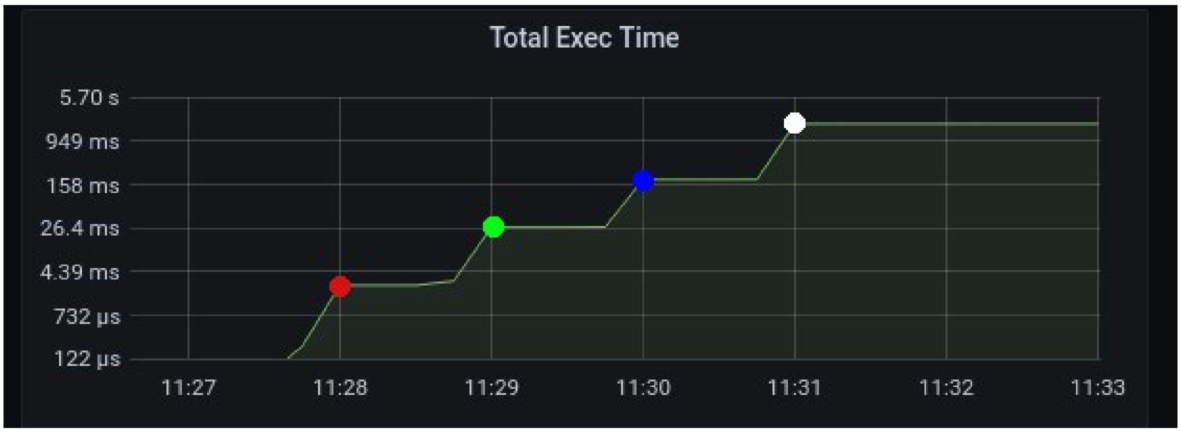


Figure 59: Latency to delete P4 device configuration with exponentially increasing number of rules (1,10,100,1000).

<u>Discussion:</u> To quantify how "heavy" each device driver RPC is, we fit functions on the values highlighted by the points on Figure 57, Figure 58 and Figure 59 above. SetConfig and DeleteConfig scale exponentially with an exponential increase in the number of rules as they both perform write operations to the Context Database. This DB is currently under re-design so as to support faster I/O and better scalability with increasing load; thus this performance is expected to improve throughout the final integration activities in WP5. On the other hand, GetConfig scales almost linearly with an exponential increase in the number of rules, which allows TeraFlow to retrieve device configuration rather quickly, even when this configuration is large.

P4 SBI driver performance vs. topology size benchmark

The following benchmark measures the total execution time the P4 SBI driver requires to perform rule installation/retrieval/deletion operations on topologies with (a linearly) increasing size. The size of the topology spans between 1-10 P4 devices. To quantify the impact of the topology size on the performance of the P4 SBI driver, we keep the number of rules installed/retrieved/deleted constant (1000 rules in total), thus the only variable in the benchmark is the topology size.

<u>Rule installation:</u> Figure 60 shows the total execution time (y axis) vs. time (x axis) during the execution of SetConfig RPCs on a topology with increasing size. In each execution, the topology size increases by one device. Specifically, the red point in Figure X indicates the total execution time to install 1000 rules on a topology with a single P4 device, while the green point in Figure X indicates the total execution time to install 1000 rules in total on a topology with 2 P4 devices (500 rules per device). Similarly, the black point in Figure X indicates the total execution time to install 1000 rules in total on a topology with 10 P4 devices (100 rules per device). The number of installed rules is kept constant (I.e., 1000 rules in total) to investigate how total execution time is affected by the topology size. The graph is annotated with color-highlighted points, which indicate the time that each RPC was concluded:

xiii) The red point refers to 1x SetConfig RPC on 1x device. This RPC is executed within 125 milliseconds.
xiv) The green point refers to 2x SetConfig RPCs on 2x devices. These RPCs are executed within 265 milliseconds.
xv) The blue point refers to 3x SetConfig RPCs on 3x devices. These RPCs are executed within 402 milliseconds.
xvi) The magenta point refers to 4x SetConfig RPCs on 4x devices. These RPCs are executed within 553 milliseconds.
xvii) The cyan point refers to 5x SetConfig RPCs on 5x devices. These RPCs are executed within 719 milliseconds.
xviii) The yellow point refers to 6x SetConfig RPCs on x devices. These RPCs are executed within 903 milliseconds.
xix) The white point refers to 7x SetConfig RPCs on 7x devices. These RPCs are executed within 1080 milliseconds.
xx) The purple point refers to 8x SetConfig RPCs on 8x devices. These RPCs are executed within 1250 milliseconds.
xxi) The orange point refers to 9x SetConfig RPCs on 9x devices. These RPCs are executed within 1420 milliseconds.
xxii) The black point refers to 10x SetConfig RPCs on 10x devices. These RPCs are executed within 1590 milliseconds.

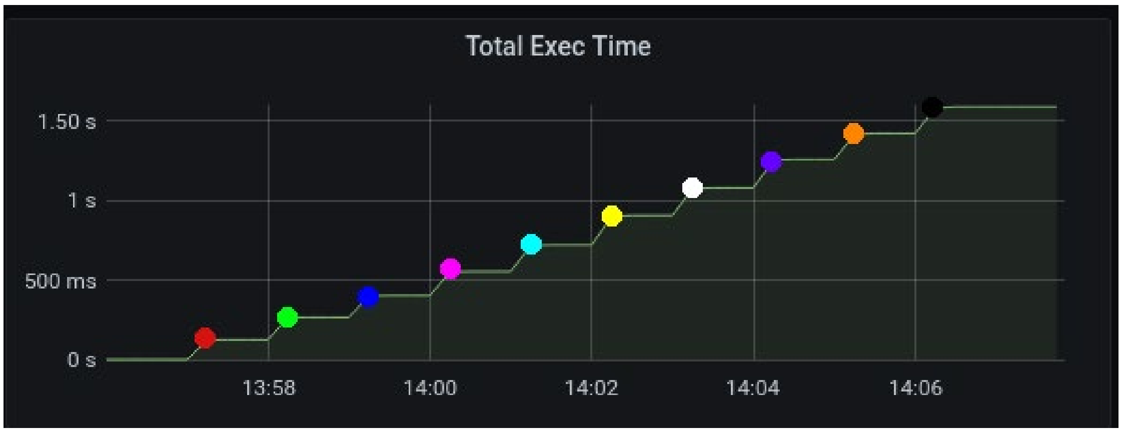


Figure 60: Latency to install 1000 rules in total atop P4 topologies of increasing size.

<u>Rule retrieval:</u> Figure 61 shows the total execution time (y axis) vs. time (x axis) during the execution of GetConfig RPCs on a topology with increasing size. In each execution, the topology size increases by one device as in the case of Figure 60 above. The number of installed rules is kept constant (I.e., 1000 rules in total) to investigate how total execution time is affected by the topology size. The graph is annotated with color-highlighted points, which indicate the time that each RPC was concluded:

xxiii) The red point refers to 1x GetConfig RPC on 1x device. This RPC is executed within 2 milliseconds.
xxiv) The green point refers to 2x GetConfig RPCs on 2x devices. These RPCs are executed within 4 milliseconds.
xxv)The blue point refers to 3x GetConfig RPCs on 3x devices. These RPCs are executed within 7 milliseconds.
xxvi) The magenta point refers to 4x GetConfig RPCs on 4x devices. These RPCs are executed within 10 milliseconds.
xxvii) The cyan point refers to 5x GetConfig RPCs on 5x devices. These RPCs are executed within 16 milliseconds.
xxviii) The yellow point refers to 6x GetConfig RPCs on x devices. These RPCs are executed within 24 milliseconds.
xxix) The white point refers to 7x GetConfig RPCs on 7x devices. These RPCs are executed within 32 milliseconds.
xxx)The purple point refers to 8x GetConfig RPCs on 8x devices. These RPCs are executed within 41 milliseconds.
xxxi) The orange point refers to 9x GetConfig RPCs on 9x devices. These RPCs are executed within 51 milliseconds.
xxxii) The black point refers to 10x GetConfig RPCs on 10x devices. These RPCs are executed within 63 milliseconds.

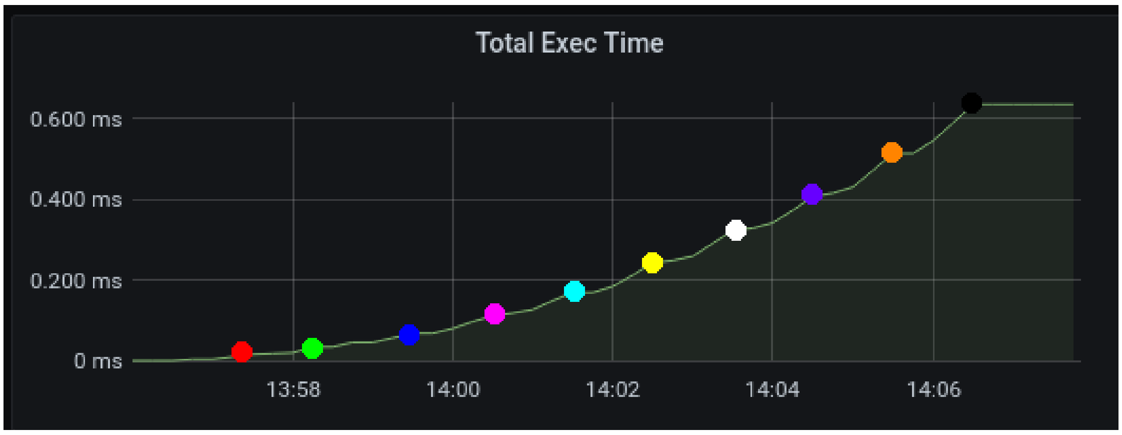


Figure 61: Latency to retrieve 1000 rules in total atop P4 topologies of increasing size.

<u>Rule deletion:</u> Figure 62 shows the total execution time (y axis) vs. time (x axis) during the execution of DeleteConfig RPCs on a topology with increasing size. In each execution, the topology size increases by one device as in the case of Figure 61 above. The number of installed rules is kept constant (i.e., 1000 rules in total) to investigate how total execution time is affected by the topology size. The graph is annotated with color-highlighted points, which indicate the time that each RPC was concluded:

xxxiii) The red point refers to 1x DeleteConfig RPC on 1x device. This RPC is executed within 190 milliseconds.

xxxiv) The green point refers to 2x DeleteConfig RPCs on 2x devices. These RPCs are executed within 304milliseconds.

xxxv) The blue point refers to 3x DeleteConfig RPCs on 3x devices. These RPCs are executed within 482 milliseconds.

xxxvi) The magenta point refers to 4x DeleteConfig RPCs on 4x devices. These RPCs are executed within 607 milliseconds.

xxxvii) The cyan point refers to 5x DeleteConfig RPCs on 5x devices. These RPCs are executed within 766 milliseconds.

xxxviii) The yellow point refers to 6x DeleteConfig RPCs on x devices. These RPCs are executed within 940 milliseconds.

xxxix) The white point refers to 7x DeleteConfig RPCs on 7x devices. These RPCs are executed within 1130 milliseconds.

xl) The purple point refers to 8x DeleteConfig RPCs on 8x devices. These RPCs are executed within 1280 milliseconds.

xli) The orange point refers to 9x DeleteConfig RPCs on 9x devices. These RPCs are executed within 1440 milliseconds.

xlii) The black point refers to 10x DeleteConfig RPCs on 10x devices. These RPCs are executed within 1600 milliseconds.

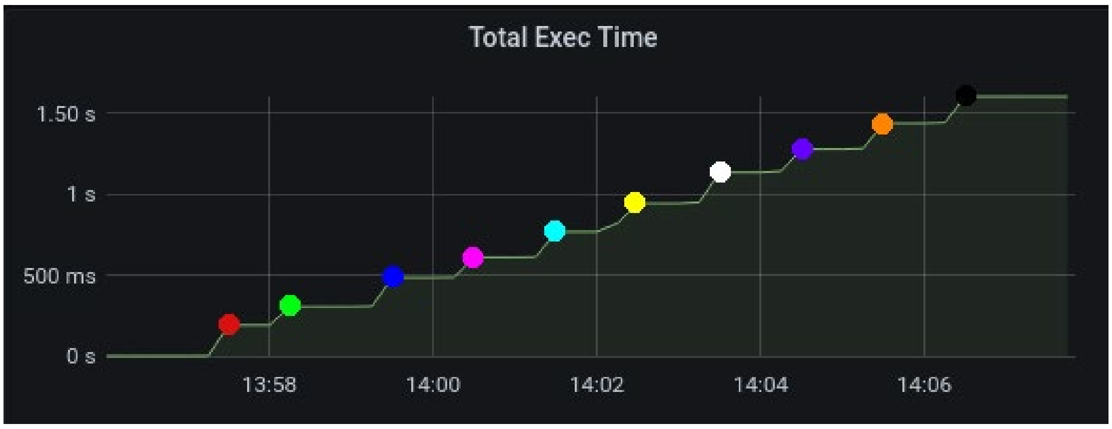


Figure 62: Latency to delete 1000 rules in total atop P4 topologies of increasing size.

Discussion

Fitting an equation on the execution times reported for the SetConfig and DeleteConfig RPCs, we see a linear behaviour. Specifically, the fitted function in each case is $F_{\mathrm{SC}} = 164x - 78.4$ for SetConfig and $F_{\mathrm{DC}} = 160x - 5.3$ for DeleteConfig, which both indicate an increase of the total execution time by 164 and 160 milliseconds per additional device, respectively. In the case of the GetConfig RPC, this cost is much lower as only 6.8 milliseconds per additional device are incurred to the total execution time.

4.1.6.6. XR Constellation Driver Plugin

Experimental setup

The setup uses two Dell PowerEdge R420 servers with 20 core Intel E5-2470@2.4GHz cpu, with 64GB@1333MHz memory and 1T SSD disks. One server runs TeraFlowSDN server and another Dell server runs IPM with emulated XR module constellation. Servers are physically located same place and sharing same subnet, having minimal latency between them.

Benchmarking procedure

IPM server has been configured HUB-LEAF reference configuration as show in Figure 63. Experimental setup adds/deletes service between HUB1 and LEAF1 to provide performance result, measuring TeraFlowSDN and IPM combined performance with GetConfig(), SetConfig(), DeleteConfig() methods.

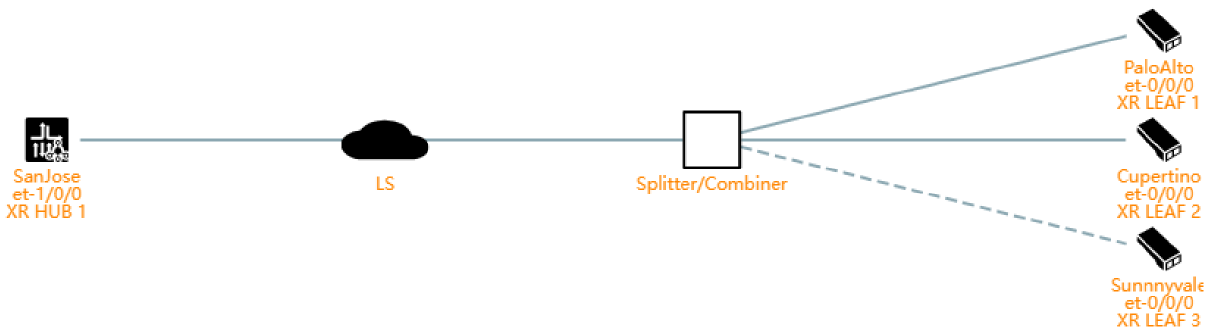


Figure 63. IPM XR Constellation reference setup for TeraFlow XR driver evaluation

On the TeraFlowSDN side, we onboard an emulated router setup as overlay to host the pluggables on TeraFlowSDN virtual routers. On Table 25 we detail an example request to create a service from TeraFlowSDN XR Constellation Driver log, where endpoints are identified as “input_sip” and "output_sip". The service is requested to have the given UUID "32af83e7-042e-47a4-939f-54e87fbc0906".

Table 25. Example of IPM connection creation request

```
[2022-12-22 10:54:21,972] INFO:device.service.drivers.xr.XrDriver:SetConfig[XR HUB 1@172.19.219.44]: resources=[('/service[32af83e7-042e-47a4-939f-54e87fbc0906]', '{"capacity_unit": "GHz", "capacity_value": 50.0, "direction": "UNIDIRECTIONAL", "input_sip": "XR HUB 1|XR-T4", "layer_protocol_name": "PHOTONIC_MEDIA", "layer_protocol_qualifier": "tapi-photonic-media:PHOTONIC_LAYER_QUALIFIER_NMC", "output_sip": "XR LEAF 1|XR-T1", "uuid": "32af83e7-042e-47a4-939f-54e87fbc0906"}')]
```

The response from IPM as seen from TeraFlowSDN XR Constellation Driver log is shown in Table 26 and contains the response accepting creating the connection request GET (8 ms), following with POST responses when actual connection has been created (29ms). TeraFlowSDN connection name uses "TF:<uuid>" syntax and IPM side connection identifier uses id: "/network-connection/<uuid>". The returned end-point identifiers are as given on SetConfig() request.

Table 26. Example of IPM connection creation response

```
[2022-12-22 10:54:21,993] INFO:device.service.drivers.xr.cm.cm_connection:process_http_response(): GET: https://172.19.219.44:443/api/v1/ncs/network-connections qparams=[('content', 'expanded'), ('q', '{"state.name": "TF:32af83e7-042e-47a4-939f-54e87fbc0906"}')] ==> 200

[2022-12-22 10:54:22,001] INFO:device.service.drivers.xr.cm.cm_connection:process_http_response(): POST: https://172.19.219.44:443/api/v1/ncs/network-connections qparams=None ==> 202

[2022-12-22 10:54:22,001] INFO:device.service.drivers.xr.cm.cm_connection:Created connection name: TF:32af83e7-042e-47a4-939f-54e87fbc0906, id: /network-connections/913fd8c1-f10d-453c-933d-fc58de19b97a, service-mode: XR-L1, end-points: [(XR HUB 1|XR-T4, 0), (XR LEAF 1|XR-T1, 0)]

[2022-12-22 10:54:22,001] INFO:device.service.drivers.xr.cm.tf:set_config_for_service: Created service 32af83e7-042e-47a4-939f-54e87fbc0906 as /network-connections/913fd8c1-f10d-453c-933d-fc58de19b97a (connection=name: TF:32af83e7-042e-47a4-939f-54e87fbc0906, id: /network-connections/913fd8c1-f10d-453c-933d-fc58de19b97a, service-mode: XR-L1, end-points: [(XR HUB 1|XR-T4, 0), (XR LEAF 1|XR-T1, 0)])
```

When HUB-LEAF service connection removal is requested, as shown in Table 27, with UUID, IPM GET/DELETE response pair confirms successful removal. On example, confirmation is received 54ms (GET response) and completion 67ms (DELETE response).

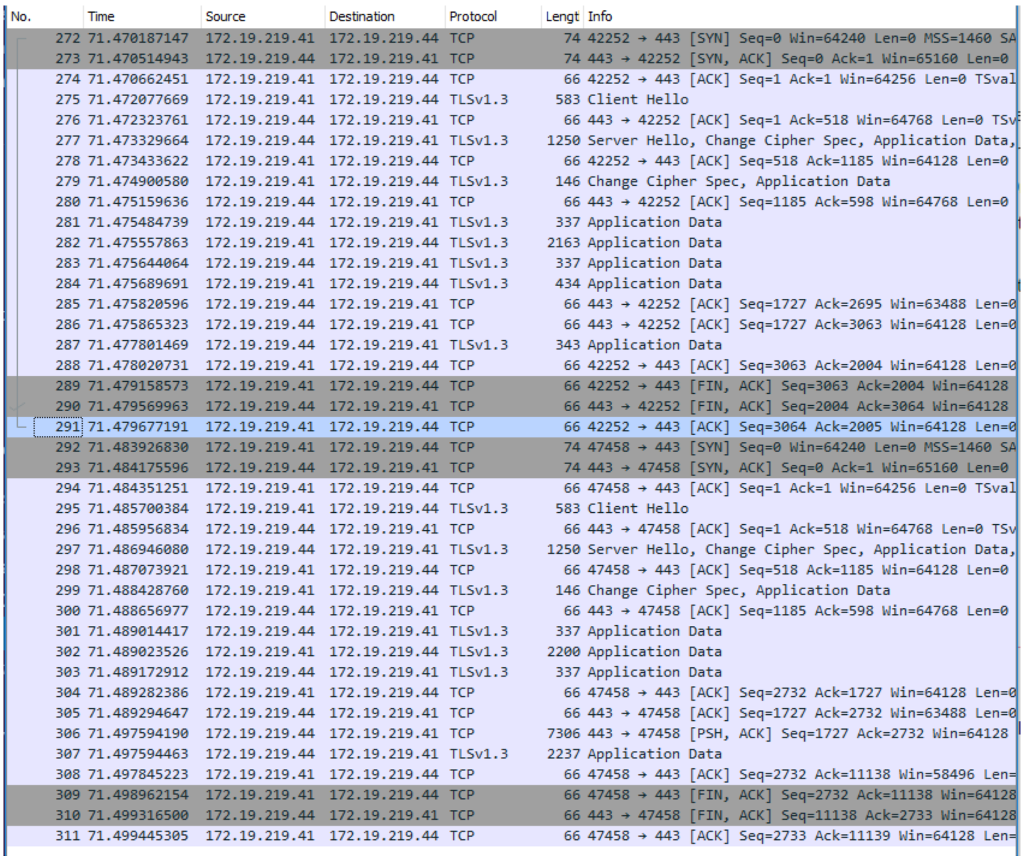
Table 27. Example IPM delete request-response

```
[2022-12-22 10:54:36,760] INFO:device.service.drivers.xr.XrDriver:DeleteConfig[XR HUB 1@172.19.219.44]: resources=[('/service[32af83e7-042e-47a4-939f-54e87fbc0906]', '{"uuid": "32af83e7-042e-47a4-939f-54e87fbc0906"}')]

[2022-12-22 10:54:36,814] INFO:device.service.drivers.xr.cm.cm_connection:process_http_response(): GET: https://172.19.219.44:443/api/v1/ncs/network-connections qparams=[('content', 'expanded'), ('q', '{"state.name": "TF:32af83e7-042e-47a4-939f-54e87fbc0906"}')] ==> 200

[2022-12-22 10:54:36,827] INFO:device.service.drivers.xr.cm.cm_connection:process_http_response(): DELETE: https://172.19.219.44:443/api/v1/ncs/network-connections/913fd8c1-f10d-453c-933d-fc58de19b97a qparams=None ==> 202

[2022-12-22 10:54:36,827] INFO:device.service.drivers.xr.cm.cm_connection:Deleted connection href='/network-connections/913fd8c1-f10d-453c-933d-fc58de19b97a'

[2022-12-22 10:54:36,827] INFO:device.service.drivers.xr.XrDriver:DeleteConfig: Connection 32af83e7-042e-47a4-939f-54e87fbc0906 deleted (was name: TF:32af83e7-042e-47a4-939f-54e87fbc0906, id: /network-connections/913fd8c1-f10d-453c-933d-fc58de19b97a, service-mode: XR-L1, end-points: [(XR HUB 1|XR-T4, 100), (XR LEAF 1|XR-T1, 100)])
```

Performance evaluation

On XR Constellation Driver performance test executes the same create/delete service sequence 39 times in row, having typically ~12.15 second delay between service requests.

The picture below shows Wireshark capture of SetConfig request on TeraFlowSDN server with two TCP flows (GET+POST, due TLS encryption these are not visible on Wireshark) between TeraFlowSDN (IP: 172.19.291.41) and IPM (IP: 172.19.219.44), with second timestamps 71.470 (packet 272) and 71.499 (packet 311), resulting same reference ~29ms time as seen on TeraFlowSDN XR Constellation driver logs timestamps with SetConfig() operation. From flow direction, one can make educated guess seeing GET operation (41 originates flow) and second part is POST (44 originates flow) as identified on TeraFlow SDN XR Constellation driver log.



```
No.   Time          Source         Destination    Protocol  Lengt Info
  272 71.470187147  172.19.219.41  172.19.219.44  TCP         74 42252 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SA
  273 71.470514943  172.19.219.44  172.19.219.41  TCP         74 443 → 42252 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0
  274 71.470662451  172.19.219.41  172.19.219.44  TCP         66 42252 → 443 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval
  275 71.472077669  172.19.219.41  172.19.219.44  TLSv1.3    583 Client Hello
  276 71.47232376l  172.19.219.44  172.19.219.41  TCP         66 443 → 42252 [ACK] Seq=1 Ack=518 Win=64768 Len=0 TSv
  277 71.473329664  172.19.219.44  172.19.219.41  TLSv1.3   1250 Server Hello, Change Cipher Spec, Application Data,
  278 71.473433622  172.19.219.41  172.19.219.44  TCP         66 42252 → 443 [ACK] Seq=518 Ack=1185 Win=64128 Len=0
  279 71.474900580  172.19.219.41  172.19.219.44  TLSv1.3    146 Change Cipher Spec, Application Data
  280 71.475159636  172.19.219.44  172.19.219.41  TCP         66 443 → 42252 [ACK] Seq=1185 Ack=598 Win=64768 Len=0
  281 71.475484739  172.19.219.44  172.19.219.41  TLSv1.3    337 Application Data
  282 71.475557863  172.19.219.41  172.19.219.44  TLSv1.3   2163 Application Data
  283 71.475644064  172.19.219.44  172.19.219.41  TLSv1.3    337 Application Data
  284 71.475689691  172.19.219.41  172.19.219.44  TLSv1.3    434 Application Data
  285 71.475820596  172.19.219.44  172.19.219.41  TCP         66 443 → 42252 [ACK] Seq=1727 Ack=2695 Win=63488 Len=0
  286 71.475865323  172.19.219.44  172.19.219.41  TCP         66 443 → 42252 [ACK] Seq=1727 Ack=3063 Win=64128 Len=0
  287 71.477801469  172.19.219.44  172.19.219.41  TLSv1.3    343 Application Data
  288 71.478020731  172.19.219.41  172.19.219.44  TCP         66 42252 → 443 [ACK] Seq=3063 Ack=2004 Win=64128 Len=0
  289 71.479158573  172.19.219.41  172.19.219.44  TCP         66 42252 → 443 [FIN, ACK] Seq=3063 Ack=2004 Win=64128
  290 71.479569963  172.19.219.44  172.19.219.41  TCP         66 443 → 42252 [FIN, ACK] Seq=2004 Ack=3064 Win=64128
  291 71.479677191  172.19.219.41  172.19.219.44  TCP         66 42252 → 443 [ACK] Seq=3064 Ack=2005 Win=64128 Len=0
  292 71.483926830  172.19.219.41  172.19.219.44  TCP         74 47458 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SA
  293 71.484175596  172.19.219.44  172.19.219.41  TCP         74 443 → 47458 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0
  294 71.484351251  172.19.219.41  172.19.219.44  TCP         66 47458 → 443 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval
  295 71.485700384  172.19.219.41  172.19.219.44  TLSv1.3    583 Client Hello
  296 71.485956834  172.19.219.44  172.19.219.41  TCP         66 443 → 47458 [ACK] Seq=1 Ack=518 Win=64768 Len=0 TSv
  297 71.486946080  172.19.219.44  172.19.219.41  TLSv1.3   1250 Server Hello, Change Cipher Spec, Application Data,
  298 71.487073921  172.19.219.41  172.19.219.44  TCP         66 47458 → 443 [ACK] Seq=518 Ack=1185 Win=64128 Len=0
  299 71.488428760  172.19.219.41  172.19.219.44  TLSv1.3    146 Change Cipher Spec, Application Data
  300 71.488656977  172.19.219.44  172.19.219.41  TCP         66 443 → 47458 [ACK] Seq=1185 Ack=598 Win=64768 Len=0
  301 71.489014417  172.19.219.44  172.19.219.41  TLSv1.3    337 Application Data
  302 71.489023526  172.19.219.41  172.19.219.44  TLSv1.3   2200 Application Data
  303 71.489172912  172.19.219.44  172.19.219.41  TLSv1.3    337 Application Data
  304 71.489282386  172.19.219.41  172.19.219.44  TCP         66 47458 → 443 [ACK] Seq=2732 Ack=1727 Win=64128 Len=0
  305 71.489294647  172.19.219.44  172.19.219.41  TCP         66 443 → 47458 [ACK] Seq=1727 Ack=2732 Win=63488 Len=0
  306 71.497594190  172.19.219.44  172.19.219.41  TCP       7306 443 → 47458 [PSH, ACK] Seq=1727 Ack=2732 Win=64128
  307 71.497594463  172.19.219.44  172.19.219.41  TLSv1.3   2237 Application Data
  308 71.497845223  172.19.219.41  172.19.219.44  TCP         66 47458 → 443 [ACK] Seq=2732 Ack=11138 Win=58496 Len=
  309 71.498962154  172.19.219.41  172.19.219.44  TCP         66 47458 → 443 [FIN, ACK] Seq=2732 Ack=11138 Win=64128
  310 71.499316500  172.19.219.44  172.19.219.41  TCP         66 443 → 47458 [FIN, ACK] Seq=11138 Ack=2733 Win=64128
  311 71.499445305  172.19.219.41  172.19.219.44  TCP         66 47458 → 443 [ACK] Seq=2733 Ack=11139 Win=64128 Len=
```

Figure 64. Wireshark capture from TeraFlowSDN and IPM TLS encrypted communication on SetConfig() operation.

The measured performance distribution result chart is presented on Figure 65.

- **GetConfig**() operations profile is mostly on 25ms-50ms bucket, and just few events 50ms-75ms and 75ms-100ms bucket;
- **SetConfig**() operations are profiled mostly to 50ms-75ms bucket, few remaining events in 75ms-100ms and 100ms-250ms buckets;
- **DeleteConfig**() operations are profiled mostly to 50ms-75ms bucket, and few remaining events in 25ms-50ms, 75ms-100ms and 100ms-250ms bucket.

As high-level finding shows, the TeraFlowSDN with XR driver and IPM provides a high probability solid response time for given operations on the used test scenario.

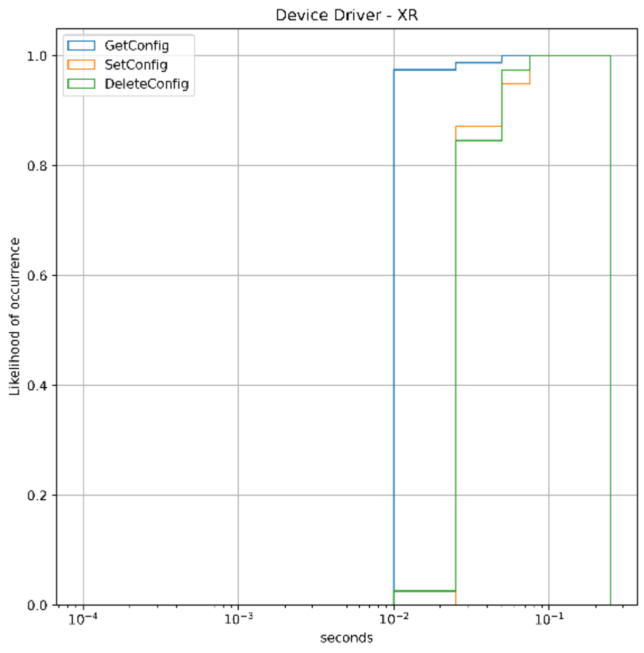


Figure 65. XR driver performance distribution chart.

4.2. Service Component

In this section, we describe the Service component in charge of managing the life-cycle of the connectivity services established in the network. For example, different service types could be requested and different protocols and data models might be used to configure the network equipment. For this reason, the Service component implements a Service Handler API that enables network operators to precisely define the service types they need to support and the behaviour for each of them. Finally, we describe the Service component's architectural design, the Service Handler API, and the interface it exposes to the rest of the TeraFlowSDN components. We also provide evaluation results of its operation.

4.2.1. New Features/Extensions

- Complete integration of Service with Device, Context, and Compute components;
- New Task Scheduler to manage dependency between services and connections, for instance, in multi-layer scenarios;
- Service handlers:
  - L2-VPN service for OpenConfig devices
  - L3-VPN service for Emulated devices
  - L3-VPN service for OpenConfig devices
    - Support for ACLs
  - Connectivity service for TAPI devices
  - L2 service handler for P4 devices
  - Microwave service handler
  - TAPI service handler extended to support XR constellation driver

4.2.2. 4Final Design

The available service handler plugins are listed below, with a link to the corresponding subsection:

- An L3NM emulated service handler plugin for testing purposes (see Section 4.1.3.1);
- An L2NM emulated service handler plugin for testing purposes (see Section 4.1.3.1);
- An L3NM OpenConfig service handler plugin for testing purposes (see Section 4.1.3.1);
- An L2NM OpenConfig service handler plugin for testing purposes (see Section 4.1.3.1);
- An OLS ONF Transport API [TR547] service handler plugin with support for the XR Constellation driver (see Section 4.1.3.2);
- An ONF TR-532 microwave service handler plugin (see Section 4.1.3.3); and
- A P4 service handler plugin for next-generation white box switches (see Section 4.1.3.5).

The internal architecture of the Service component is depicted in Figure 66. The gRPC block exposes the NBI to the rest of TeraFlow components while the Service Servicer dispatches the requests and issues path computation requests to the PathComp component. To support multi-layer service establishment, solutions returned by the PathComp component might consist of several (sub-)connections and (sub-)services. These elements are organized hierarchically and correspond to the connectivity services supporting the end-to-end service requested. If that service traverses, for instance, an L3 packet segment, and an L0 TAPI-managed optical segment, the resulting PathComp reply will contain an L3NM service that will depend on a TAPI service (both returned in the reply); in this example, the computed connections for these services will be returned.

The Service component incorporates a new Task Scheduler module responsible for organizing the returned (sub-)services and (sub-connections) and setup/teardown them in an ordered manner. Indeed, the Task Scheduler is responsible for choosing the appropriate service handler for each (sub-)service type.

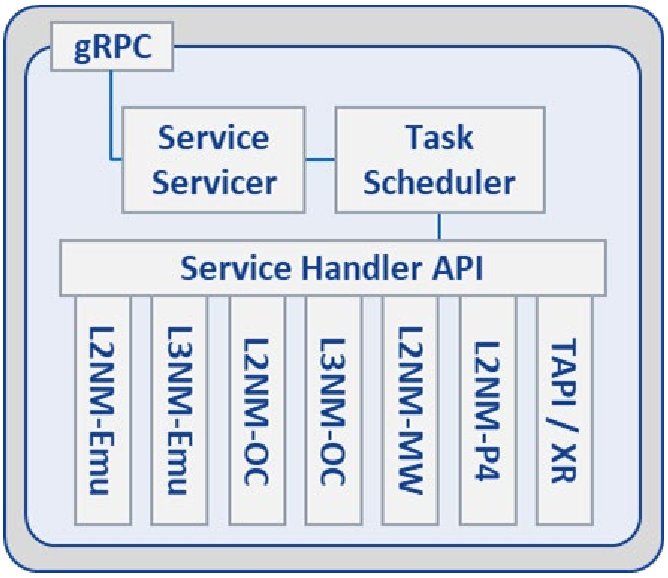


Figure 66: Architecture of the Service component.

4.2.2.1. L3NM -> Device (TID)

The L3NM is intended to provide a network-centric perspective of Layer 3 (L3) VPN services. This data model may improve communication between the service orchestrator and the network controller/orchestrator by allowing additional network-centric information to be included. It allows extra features such as resource management, or it can function as a multi-domain orchestration interface when logical resources must be managed.

The L3NM is not a model of customer service. The internal view of the service may be translated to a visible to consumers exterior view: the L3VPN Service Model (L3SM).

Customers' specified inputs can be provided into the L3NM. Such requests are often based on an L3SM template. For example, some sections of the L3SM module can be easily translated to the L3NM, while others are produced based on the requested service and local rules. Other components are exclusive to the service provider and do not map directly to the L3SM.

Once a global VPN service has been recorded by the L3NM, the actual activation and provisioning of the VPN service will entail a number of device modules to fine-tune the essential functionalities for service delivery. The VPN nodes support these functions, which may be handled via device YANG modules. A partial list of such device YANG modules is presented below:

- Routing management
- BGP
- PIM
- NAT management
- QoS management
- ACLs

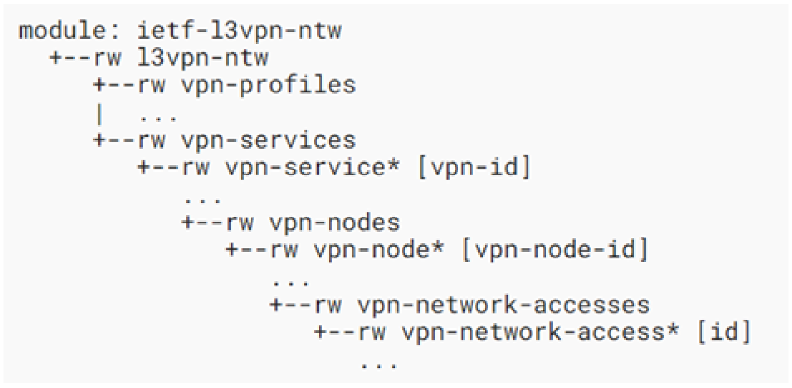
The L3NM ("ietf-l3vpn-ntw") is defined to manage L3VPNs in a service provider network. The "ietf-l3vpn-ntw" module, in particular, may be used to build, change, and retrieve a network's L3VPN services.

The "pyang" tool may create the module's entire tree diagram. However, because the entire tree is too lengthy, it is not included here. Subtrees are supplied instead for the reader's convenience.

- **Overall Structure of the Module**

The "ietf-l3vpn-ntw" module employs two primary containers: "vpn-profiles" and "vpn-services":

- The provider uses the 'vpn-profiles' container to store a collection of common VPN profiles that apply to one or more VPN services.
- The container 'vpn-services' manages the set of VPN services controlled within the service provider network. The data structure 'vpn-service' encapsulates a VPN service.

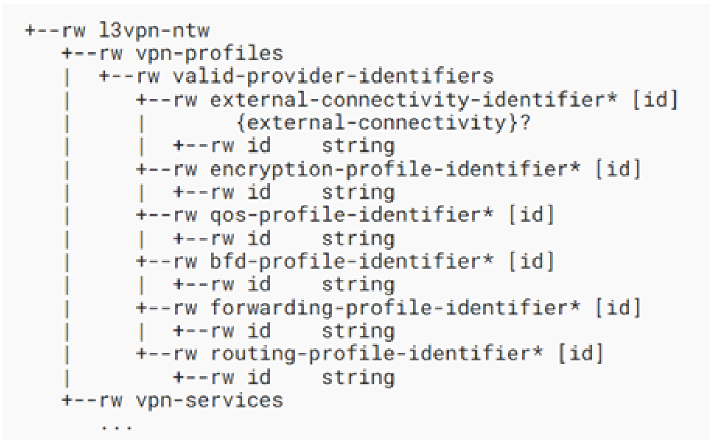


```
module: ietf-l3vpn-ntw
  +--rw l3vpn-ntw
     +--rw vpn-profiles
     |  ...
     +--rw vpn-services
        +--rw vpn-service* [vpn-id]
           ...
           +--rw vpn-nodes
              +--rw vpn-node* [vpn-node-id]
                 ...
                 +--rw vpn-network-accesses
                    +--rw vpn-network-access* [id]
                       ...
```

Figure 67: Overall L3NM Tree Structure.

o **VPN Profiles**

The container 'vpn-profiles' allows the VPN service provider to establish and manage a collection of VPN profiles that apply to one or more VPN services. When designing a VPN service, the model merely offers an identifier for these profiles to aid in recognizing and binding local regulations.

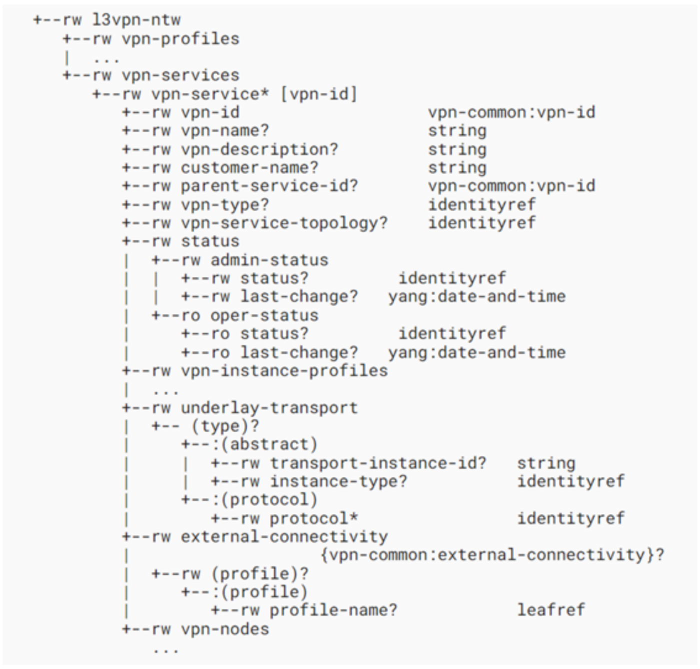


```
+--rw l3vpn-ntw
   +--rw vpn-profiles
   |  +--rw valid-provider-identifiers
   |     +--rw external-connectivity-identifier* [id]
   |     |        {external-connectivity}?
   |     |  +--rw id    string
   |     +--rw encryption-profile-identifier* [id]
   |     |  +--rw id    string
   |     +--rw qos-profile-identifier* [id]
   |     |  +--rw id    string
   |     +--rw bfd-profile-identifier* [id]
   |     |  +--rw id    string
   |     +--rw forwarding-profile-identifier* [id]
   |     |  +--rw id    string
   |     +--rw routing-profile-identifier* [id]
   |        +--rw id    string
   +--rw vpn-services
      ...
```

Figure 68: VPN Profiles Subtree Structure.

o **VPN Services**

The data structure 'vpn-service' abstracts a VPN service in the service provider network. Each 'vpn-service' is identifiable by a unique identifier: vpn-id. Such a 'vpn-id' is only relevant locally.

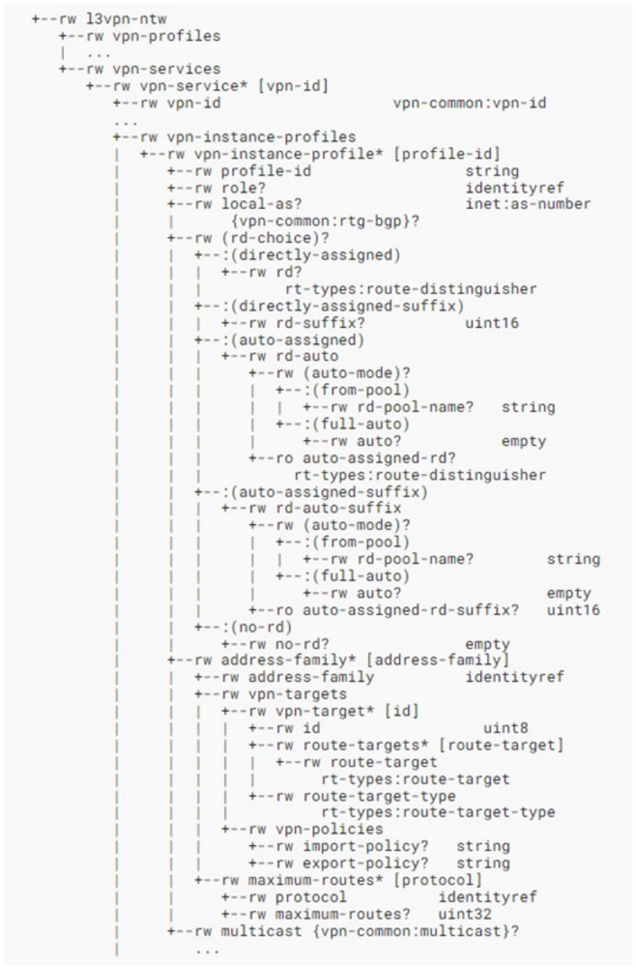


```
+--rw l3vpn-ntw
   +--rw vpn-profiles
   |  ...
   +--rw vpn-services
      +--rw vpn-service* [vpn-id]
         +--rw vpn-id                     vpn-common:vpn-id
         +--rw vpn-name?                  string
         +--rw vpn-description?           string
         +--rw customer-name?             string
         +--rw parent-service-id?         vpn-common:vpn-id
         +--rw vpn-type?                  identityref
         +--rw vpn-service-topology?      identityref
         +--rw status
         |  +--rw admin-status
         |  |  +--rw status?         identityref
         |  |  +--rw last-change?   yang:date-and-time
         |  +--ro oper-status
         |     +--ro status?         identityref
         |     +--ro last-change?   yang:date-and-time
         +--rw vpn-instance-profiles
         |  ...
         +--rw underlay-transport
         |  +-- (type)?
         |     +--:(abstract)
         |     |  +--rw transport-instance-id?   string
         |     |  +--rw instance-type?           identityref
         |     +--:(protocol)
         |        +--rw protocol*                identityref
         +--rw external-connectivity
         |                 {vpn-common:external-connectivity}?
         |  +--rw (profile)?
         |     +--:(profile)
         |        +--rw profile-name?            leafref
         +--rw vpn-nodes
            ...
```

Figure 69: VPN Services Subtree Structure.

- **VPN Instance Profiles**

VPN instance profiles are intended to factorize data nodes utilized at several levels of the model. The VPN service level defines generic VPN instance profiles, which are subsequently called at the VPN node and VPN network access levels. The 'profile-id' identifies each VPN instance profile. This identity is then used to refer to one or more VPN nodes, allowing the controller to identify generic resources to be set for a specific VRF instance.

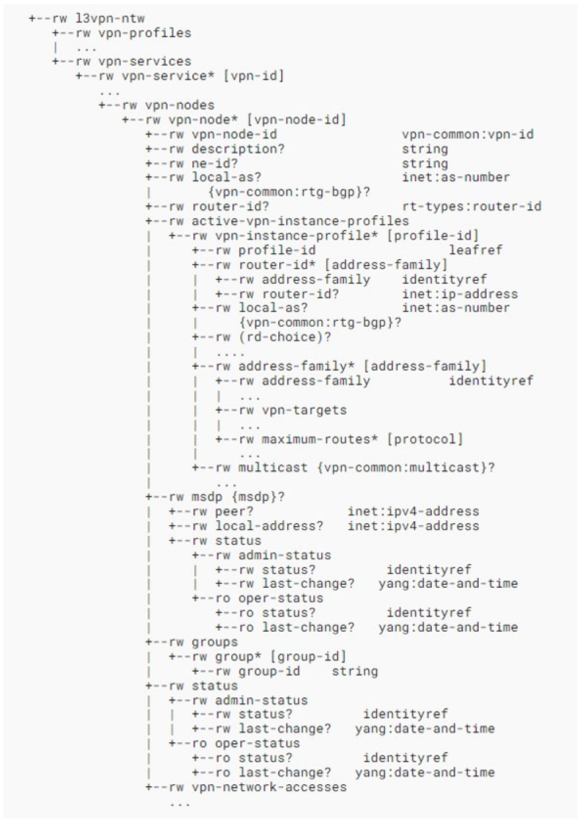


```
+--rw l3vpn-ntw
   +--rw vpn-profiles
   |  ...
   +--rw vpn-services
      +--rw vpn-service* [vpn-id]
         +--rw vpn-id                   vpn-common:vpn-id
         ...
         +--rw vpn-instance-profiles
         |  +--rw vpn-instance-profile* [profile-id]
         |     +--rw profile-id                 string
         |     +--rw role?                      identityref
         |     +--rw local-as?                  inet:as-number
         |     |       {vpn-common:rtg-bgp}?
         |     +--rw (rd-choice)?
         |     |  +--:(directly-assigned)
         |     |  |  +--rw rd?
         |     |  |          rt-types:route-distinguisher
         |     |  +--:(directly-assigned-suffix)
         |     |  |  +--rw rd-suffix?           uint16
         |     |  +--:(auto-assigned)
         |     |  |  +--rw rd-auto
         |     |  |     +--rw (auto-mode)?
         |     |  |     |  +--:(from-pool)
         |     |  |     |  |  +--rw rd-pool-name?   string
         |     |  |     |  +--:(full-auto)
         |     |  |     |     +--rw auto?           empty
         |     |  |     +--ro auto-assigned-rd?
         |     |  |           rt-types:route-distinguisher
         |     |  +--:(auto-assigned-suffix)
         |     |  |  +--rw rd-auto-suffix
         |     |  |     +--rw (auto-mode)?
         |     |  |     |  +--:(from-pool)
         |     |  |     |  |  +--rw rd-pool-name?        string
         |     |  |     |  +--:(full-auto)
         |     |  |     |     +--rw auto?                empty
         |     |  |     +--ro auto-assigned-rd-suffix?   uint16
         |     |  +--:(no-rd)
         |     |     +--rw no-rd?               empty
         |     +--rw address-family* [address-family]
         |     |  +--rw address-family          identityref
         |     |  +--rw vpn-targets
         |     |  |  +--rw vpn-target* [id]
         |     |  |  |  +--rw id                  uint8
         |     |  |  |  +--rw route-targets* [route-target]
         |     |  |  |  |  +--rw route-target
         |     |  |  |  |          rt-types:route-target
         |     |  |  |  +--rw route-target-type
         |     |  |  |          rt-types:route-target-type
         |     |  |  +--rw vpn-policies
         |     |  |     +--rw import-policy?   string
         |     |  |     +--rw export-policy?   string
         |     |  +--rw maximum-routes* [protocol]
         |     |     +--rw protocol           identityref
         |     |     +--rw maximum-routes?   uint32
         |     +--rw multicast {vpn-common:multicast}?
         |        ...
```

Figure 70: VPN Node Subtree Structure.

- **VPN Nodes**

The vpn-node is an abstraction that describes a collection of common policies that are implemented on a specific network node (usually a PE) and are part of a single L3VPN service. The 'vpn-node' command provides an argument that specifies the network node to which it is applied. If the 'ne-id' relates to a specific PE, the 'vpn-node' will very certainly be mapped to a VRF instance in the node. The paradigm, however, permits pointing to an abstract node. The network controller will select how to divide the 'vpn-node' into VRF instances in this situation.

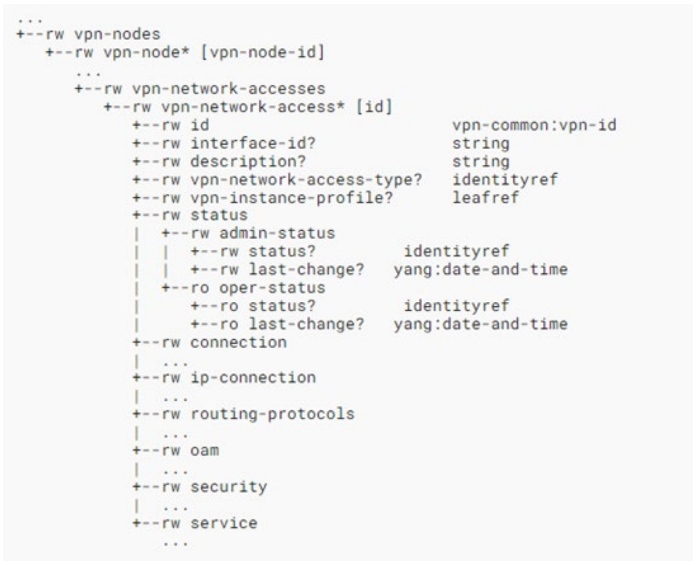


```
+--rw l3vpn-ntw
   +--rw vpn-profiles
   |  ...
   +--rw vpn-services
      +--rw vpn-service* [vpn-id]
         ...
         +--rw vpn-nodes
            +--rw vpn-node* [vpn-node-id]
               +--rw vpn-node-id                 vpn-common:vpn-id
               +--rw description?                string
               +--rw ne-id?                      string
               +--rw local-as?                   inet:as-number
               |       {vpn-common:rtg-bgp}?
               +--rw router-id?                  rt-types:router-id
               +--rw active-vpn-instance-profiles
               |  +--rw vpn-instance-profile* [profile-id]
               |     +--rw profile-id                  leafref
               |     +--rw router-id* [address-family]
               |     |  +--rw address-family     identityref
               |     |  +--rw router-id?         inet:ip-address
               |     +--rw local-as?             inet:as-number
               |     |     {vpn-common:rtg-bgp}?
               |     +--rw (rd-choice)?
               |     |  ....
               |     +--rw address-family* [address-family]
               |     |  +--rw address-family           identityref
               |     |  |  ...
               |     |  +--rw vpn-targets
               |     |  |  ...
               |     |  +--rw maximum-routes* [protocol]
               |     |     ...
               |     +--rw multicast {vpn-common:multicast}?
               |        ...
               +--rw msdp {msdp}?
               |  +--rw peer?            inet:ipv4-address
               |  +--rw local-address?   inet:ipv4-address
               |  +--rw status
               |     +--rw admin-status
               |     |  +--rw status?         identityref
               |     |  +--rw last-change?   yang:date-and-time
               |     +--ro oper-status
               |        +--ro status?         identityref
               |        +--ro last-change?   yang:date-and-time
               +--rw groups
               |  +--rw group* [group-id]
               |     +--rw group-id    string
               +--rw status
               |  +--rw admin-status
               |  |  +--rw status?         identityref
               |  |  +--rw last-change?   yang:date-and-time
               |  +--ro oper-status
               |     +--ro status?         identityref
               |     +--ro last-change?   yang:date-and-time
               +--rw vpn-network-accesses
                  ...
```

Figure 71: VPN Node Subtree Structure.

- VPN Network Accesses

The 'vpn-network-access' includes a set of data nodes that describe the access information for the traffic that belongs to a particular L3VPN.



```
...
+--rw vpn-nodes
   +--rw vpn-node* [vpn-node-id]
      ...
      +--rw vpn-network-accesses
         +--rw vpn-network-access* [id]
            +--rw id                         vpn-common:vpn-id
            +--rw interface-id?              string
            +--rw description?               string
            +--rw vpn-network-access-type?   identityref
            +--rw vpn-instance-profile?      leafref
            +--rw status
            |  +--rw admin-status
            |  |  +--rw status?         identityref
            |  |  +--rw last-change?   yang:date-and-time
            |  +--ro oper-status
            |     +--ro status?         identityref
            |     +--ro last-change?   yang:date-and-time
            +--rw connection
            |  ...
            +--rw ip-connection
            |  ...
            +--rw routing-protocols
            |  ...
            +--rw oam
            |  ...
            +--rw security
            |  ...
            +--rw service
               ...
```

Figure 72: VPN Network Access Subtree Structure.

4.2.2.2. L2NM -> Device (TID)

A network controller, for example, can expose the L2NM to a service controller within the service provider's network. The paradigm may be employed in the communication interface between the entity that interacts directly with the customer and the entity in charge of network orchestration and control by allowing additional network-centric information.

The L2NM enables features such as exposing operational parameters, selecting transport protocols, and precedence. It also functions as a multi-domain orchestration interface.

The L2NM is designed to support a wide range of Layer 2 Virtual Private Networks, including:

- Virtual Private LAN Service (VPLS);
- Virtual Private Wire Service (VPWS);
- Various flavors of EVPNs.

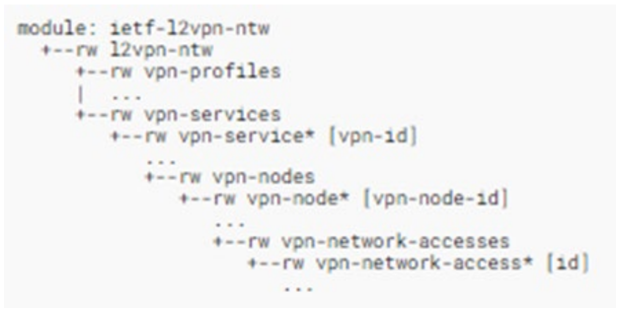
The L2NM is intended to quickly accommodate future Layer 2 VPN flavors and processes.

The L2NM is used to manage L2VPNs within a service provider's network. The 'ietf-l2vpn-ntw' module may be used to create, edit, remove, and retrieve L2VPN services in a network controller. The module is intended to reduce the volume of customer-related data. The "pyang" tool may create the module's entire tree diagram. Because it is too lengthy, that tree is not included here. Subtrees are supplied instead for the reader's convenience.

- Overall Structure of the Module

The 'ietf-l2vpn-ntw' module employs two major containers:

- The provider uses the 'vpn-profiles' container to establish and manage a collection of common VPN profiles that apply to VPN services.
- The container 'vpn-services' manages the set of L2VPN services controlled in the service provider network. By adding a new instance of 'vpn-service,' the module allows you to create a new L2VPN service. The data structure 'vpn-service' encapsulates the VPN service.



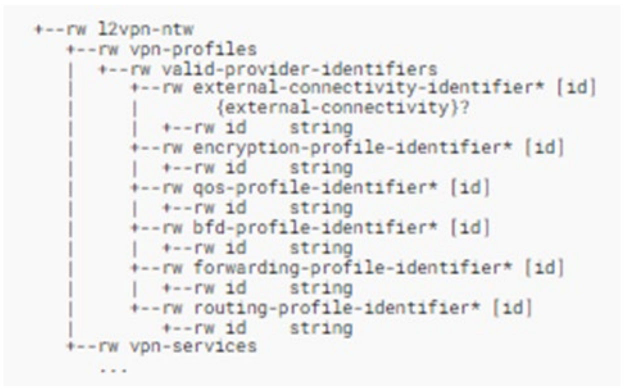
```
module: ietf-l2vpn-ntw
  +--rw l2vpn-ntw
     +--rw vpn-profiles
     |  ...
     +--rw vpn-services
        +--rw vpn-service* [vpn-id]
           ...
           +--rw vpn-nodes
              +--rw vpn-node* [vpn-node-id]
                 ...
                 +--rw vpn-network-accesses
                    +--rw vpn-network-access* [id]
                       ...
```

Figure 73: Overall L2NM Tree Structure.

o **VPN Profiles**

A VPN service provider uses the 'vpn-profiles' container to establish and manage a collection of VPN profiles that apply to one or more VPN services.

Each VPN service provider has its own definition of these profiles. When designing a VPN service, the model simply contains an identifier for these profiles to make detecting and binding local regulations easier.

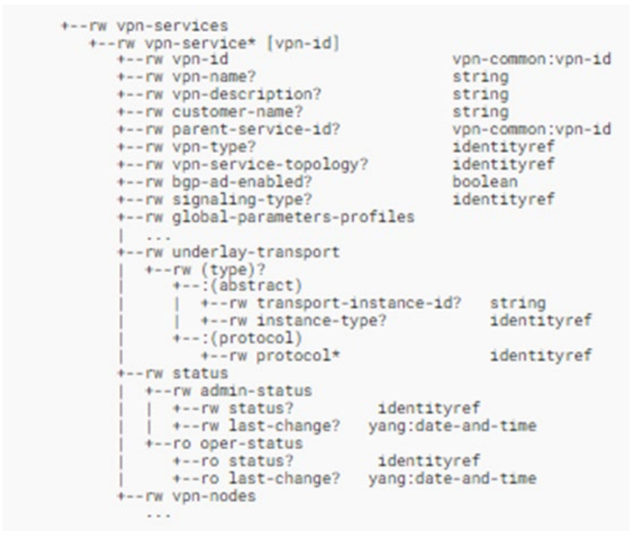


```
+--rw l2vpn-ntw
   +--rw vpn-profiles
   |  +--rw valid-provider-identifiers
   |     +--rw external-connectivity-identifier* [id]
   |     |       {external-connectivity}?
   |     |  +--rw id     string
   |     +--rw encryption-profile-identifier* [id]
   |     |  +--rw id     string
   |     +--rw qos-profile-identifier* [id]
   |     |  +--rw id     string
   |     +--rw bfd-profile-identifier* [id]
   |     |  +--rw id     string
   |     +--rw forwarding-profile-identifier* [id]
   |     |  +--rw id     string
   |     +--rw routing-profile-identifier* [id]
   |        +--rw id     string
   +--rw vpn-services
      ...
```

Figure 74: VPN VPN Profiles Subtree

- **VPN Services**

The data structure vpn-service abstracts an L2VPN service in the service provider network. Each 'vpn-service' is identifiable by a unique identifier: vpn-id. Such a 'vpn-id' has no relevance outside of the network controller.

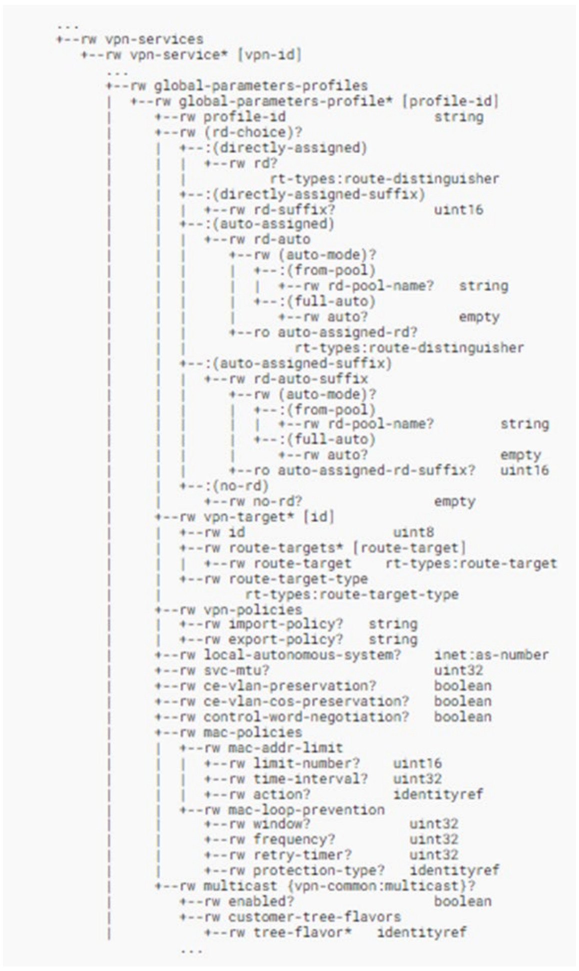


```
+--rw vpn-services
   +--rw vpn-service* [vpn-id]
      +--rw vpn-id                          vpn-common:vpn-id
      +--rw vpn-name?                       string
      +--rw vpn-description?                string
      +--rw customer-name?                  string
      +--rw parent-service-id?              vpn-common:vpn-id
      +--rw vpn-type?                       identityref
      +--rw vpn-service-topology?           identityref
      +--rw bgp-ad-enabled?                 boolean
      +--rw signaling-type?                 identityref
      +--rw global-parameters-profiles
      |  ...
      +--rw underlay-transport
      |  +--rw (type)?
      |     +--:(abstract)
      |     |  +--rw transport-instance-id?   string
      |     |  +--rw instance-type?           identityref
      |     +--:(protocol)
      |        +--rw protocol*                identityref
      +--rw status
      |  +--rw admin-status
      |  |  +--rw status?         identityref
      |  |  +--rw last-change?   yang:date-and-time
      |  +--ro oper-status
      |     +--ro status?         identityref
      |     +--ro last-change?   yang:date-and-time
      +--rw vpn-nodes
         ...
```

Figure 75: VPN VPN Service Subtree

- **Global Parameters Profiles**

The 'global-parameters-profile' specifies parameters that can be reused for the same L2VPN service instance ('vpn-service'). Global parameter profiles are established at the VPN service level, enabled at the VPN node level, and then utilized at the VPN network access level if they are activated. The 'profile-id' identifies each VPN instance profile. Some data nodes can be configured at the VPN node and VPN network access levels. These altered values supersede the global values.

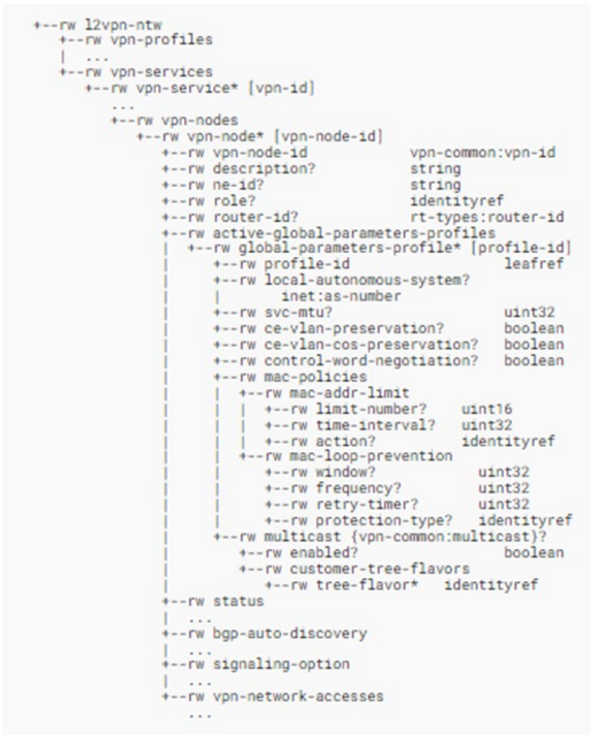


```
...
+--rw vpn-services
   +--rw vpn-service* [vpn-id]
      ...
      +--rw global-parameters-profiles
      |  +--rw global-parameters-profile* [profile-id]
      |     +--rw profile-id                         string
      |     +--rw (rd-choice)?
      |     |  +--:(directly-assigned)
      |     |  |  +--rw rd?
      |     |  |          rt-types:route-distinguisher
      |     |  +--:(directly-assigned-suffix)
      |     |  |  +--rw rd-suffix?                   uint16
      |     |  +--:(auto-assigned)
      |     |  |  +--rw rd-auto
      |     |  |     +--rw (auto-mode)?
      |     |  |     |  +--:(from-pool)
      |     |  |     |  |  +--rw rd-pool-name?   string
      |     |  |     |  +--:(full-auto)
      |     |  |     |     +--rw auto?           empty
      |     |  |     +--ro auto-assigned-rd?
      |     |  |             rt-types:route-distinguisher
      |     |  +--:(auto-assigned-suffix)
      |     |  |  +--rw rd-auto-suffix
      |     |  |     +--rw (auto-mode)?
      |     |  |     |  +--:(from-pool)
      |     |  |     |  |  +--rw rd-pool-name?        string
      |     |  |     |  +--:(full-auto)
      |     |  |     |     +--rw auto?                empty
      |     |  |     +--ro auto-assigned-rd-suffix?   uint16
      |     |  +--:(no-rd)
      |     |     +--rw no-rd?                   empty
      |     +--rw vpn-target* [id]
      |     |  +--rw id                     uint8
      |     |  +--rw route-targets* [route-target]
      |     |  |  +--rw route-target    rt-types:route-target
      |     |  +--rw route-target-type
      |     |          rt-types:route-target-type
      |     +--rw vpn-policies
      |     |  +--rw import-policy?   string
      |     |  +--rw export-policy?   string
      |     +--rw local-autonomous-system?    inet:as-number
      |     +--rw svc-mtu?                    uint32
      |     +--rw ce-vlan-preservation?       boolean
      |     +--rw ce-vlan-cos-preservation?   boolean
      |     +--rw control-word-negotiation?   boolean
      |     +--rw mac-policies
      |     |  +--rw mac-addr-limit
      |     |  |  +--rw limit-number?    uint16
      |     |  |  +--rw time-interval?   uint32
      |     |  |  +--rw action?          identityref
      |     |  +--rw mac-loop-prevention
      |     |     +--rw window?            uint32
      |     |     +--rw frequency?         uint32
      |     |     +--rw retry-timer?       uint32
      |     |     +--rw protection-type?   identityref
      |     +--rw multicast {vpn-common:multicast}?
      |        +--rw enabled?                    boolean
      |        +--rw customer-tree-flavors
      |           +--rw tree-flavor*   identityref
               ...
```

Figure 76: VPN Global Parameters Profiles Subtree.

- **VPN Nodes**

The vpn-node is an abstraction representing a collection of policies applied to a network node that is part of a single vpn-service. A 'vpn-node' has 'vpn-network-accesses,' which are the interfaces used in the VPN's establishment. The 'vpn-network-accesses' are linked to the client sites.



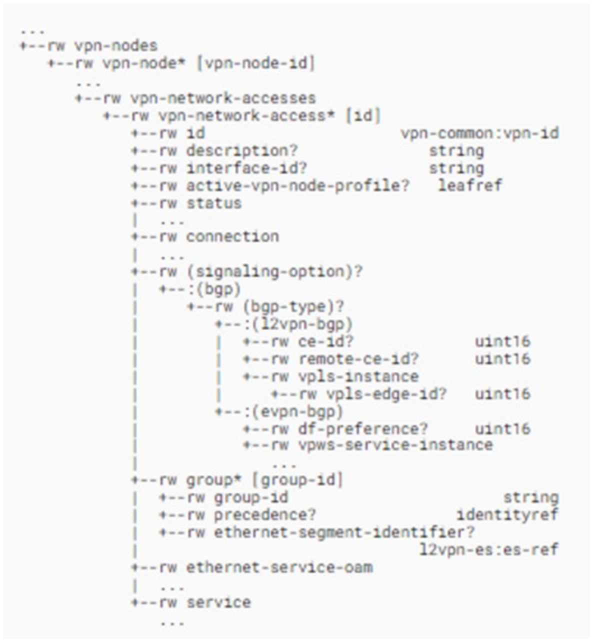
```
+--rw l2vpn-ntw
   +--rw vpn-profiles
   |  ...
   +--rw vpn-services
      +--rw vpn-service* [vpn-id]
         ...
         +--rw vpn-nodes
            +--rw vpn-node* [vpn-node-id]
               +--rw vpn-node-id              vpn-common:vpn-id
               +--rw description?             string
               +--rw ne-id?                   string
               +--rw role?                    identityref
               +--rw router-id?               rt-types:router-id
               +--rw active-global-parameters-profiles
               |  +--rw global-parameters-profile* [profile-id]
               |     +--rw profile-id                   leafref
               |     +--rw local-autonomous-system?
               |     |       inet:as-number
               |     +--rw svc-mtu?                     uint32
               |     +--rw ce-vlan-preservation?        boolean
               |     +--rw ce-vlan-cos-preservation?    boolean
               |     +--rw control-word-negotiation?    boolean
               |     +--rw mac-policies
               |     |  +--rw mac-addr-limit
               |     |  |  +--rw limit-number?     uint16
               |     |  |  +--rw time-interval?    uint32
               |     |  |  +--rw action?           identityref
               |     |  +--rw mac-loop-prevention
               |     |     +--rw window?             uint32
               |     |     +--rw frequency?          uint32
               |     |     +--rw retry-timer?        uint32
               |     |     +--rw protection-type?    identityref
               |     +--rw multicast {vpn-common:multicast}?
               |        +--rw enabled?                  boolean
               |        +--rw customer-tree-flavors
               |           +--rw tree-flavor*    identityref
               +--rw status
               |  ...
               +--rw bgp-auto-discovery
               |  ...
               +--rw signaling-option
               |  ...
               +--rw vpn-network-accesses
                  ...
```

Figure 77: VPN Nodes Subtree.

- **VPN Network Accesses**

A 'vpn-network-access' is a point of access to a VPN service. In other words, this container contains the parameters that characterize the access information for the traffic associated with a certain L2VPN.

A 'vpn-network-access' contains information such as the connection used to define the access, the precise Layer 2 service needs, and so on.



```
...
+--rw vpn-nodes
   +--rw vpn-node* [vpn-node-id]
      ...
      +--rw vpn-network-accesses
         +--rw vpn-network-access* [id]
            +--rw id                         vpn-common:vpn-id
            +--rw description?                  string
            +--rw interface-id?                 string
            +--rw active-vpn-node-profile?       leafref
            +--rw status
            |  ...
            +--rw connection
            |  ...
            +--rw (signaling-option)?
            |  +--:(bgp)
            |     +--rw (bgp-type)?
            |        +--:(l2vpn-bgp)
            |        |  +--rw ce-id?              uint16
            |        |  +--rw remote-ce-id?       uint16
            |        |  +--rw vpls-instance
            |        |     +--rw vpls-edge-id?    uint16
            |        +--:(evpn-bgp)
            |           +--rw df-preference?      uint16
            |           +--rw vpws-service-instance
            |              ...
            +--rw group* [group-id]
            |  +--rw group-id                            string
            |  +--rw precedence?                    identityref
            |  +--rw ethernet-segment-identifier?
            |                                  l2vpn-es:es-ref
            +--rw ethernet-service-oam
            |  ...
            +--rw service
               ...
```

Figure 78: VPN Network Access Subtree.

4.2.2.3. P4-based L2 service handler (UBI)

The P4 network programming language leverages a table-based model to describe the forwarding pipeline of a network element. This pipeline is only realized when P4 tables are populated with forwarding rules, which match traffic and perform one or more actions on the patched packets/flows. This makes P4 different from e.g., YANG-based programming models, which have no notion of runtime rules. To this end, dedicated service handlers are required for P4-based topologies, as the service layer needs to oversee the underlying topology and provision forwarding rules to all devices across the service path.

In TeraFlowSDN release v2, an L2 service handler is developed for P4-based topologies. This service handler implements the same service L2NM type described previously in this section but uses the P4 device driver plugin to translate connectivity requirements into forwarding rules for the underlying P4 devices. Figure X depicts an integrated TeraFlowSDN environment, where the P4 L2NM service handler operates atop the P4 SBI driver, also in tandem with Context, Monitoring, Service, Path Computation, Policy, and the WebUI components. This deployment manages an example P4 topology used to demonstrate the functionality of the P4 service handler, although the concept is abstract and can work with any P4 topology.

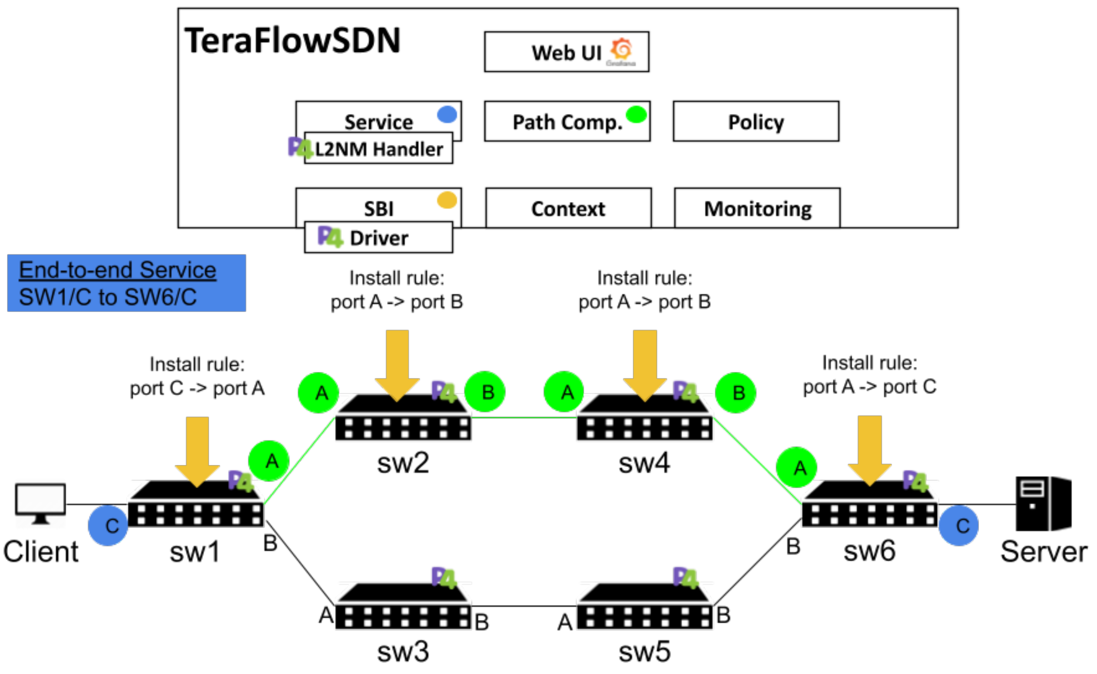
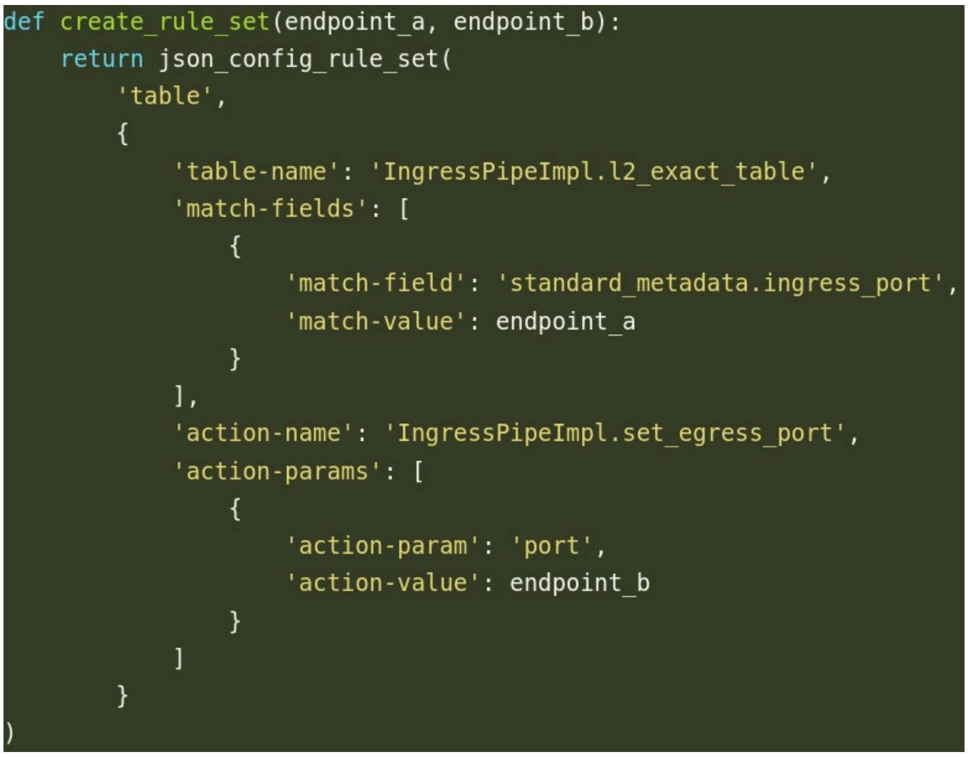


Figure 79: P4 L2NM service handler on an example P4-based topology.

As shown in Figure 79, a six-node topology of software-based P4 switches interconnects a client with a remote server. TeraFlowSDN gets a request to establish connectivity between two endpoints, i.e., a client connected to port C of sw1 and a server connected to port C of sw6. To do so, the Service component uses a Path Computation Engine (PCE) in TFS to compute an end-to-end path. As shown in Figure 80, the Path Computation component replies with a list of endpoints that comprise the path sw1-->sw2-->sw4-->sw6. Then, the P4 service handler parses the returned endpoints and computes forwarding rules for the switches that the path traverses. In this example, 4 forwarding rules are sent by the P4 device driver to sw1, sw2, sw4, and sw6 as shown in Figure 79. After all switches install the requested rules, the end-to-end path is established. To facilitate rule installation, the L2NM P4 service handler provides a rule template (see Figure Y) where the endpoints' information is configurable.

Specifically, the template rule matches on a configurable ingress port of a P4 device and output the packet to a configurable egress port using a corresponding P4 action. This way, rules for multiple devices and endpoints can be replicated and configured very quickly, while introducing no complexity to the service handler's code.



```
def create_rule_set(endpoint_a, endpoint_b):
    return json_config_rule_set(
        'table',
        {
            'table-name': 'IngressPipeImpl.l2_exact_table',
            'match-fields': [
                {
                    'match-field': 'standard_metadata.ingress_port',
                    'match-value': endpoint_a
                }
            ],
            'action-name': 'IngressPipeImpl.set_egress_port',
            'action-params': [
                {
                    'action-param': 'port',
                    'action-value': endpoint_b
                }
            ]
        }
)
```

Figure 80: Code snippet with P4 rule template using configurable endpoints.

This service handler will be used in WP5 to realize a service restoration workflow atop P4 topologies as already described in D5.2 [6].

4.2.3. Final Interfaces

Service is the TeraFlowSDN component in charge of managing the creation, update and removal of connectivity services through the SBI component. Different connectivity service types, device types and device protocols might be needed to support the connectivity service management; for this reason, the Service component provides a Service Handler API that enables developers to implement new service handlers and integrate them into the TeraFlowSDN.

4.2.3.1. L2VPN Network Model (TID)

This use case allows the provisioning, modification and deletion of a Layer 2 VPN service spanning one or more IP/MPLS routers via the SDN controller using a subset of the L2NM data model (RFC 9291). The L2VPN creates a virtual routing network instance (known as ELAN) in each of the routers involved in service deployment. The routing instance (ELAN) created at every router allows the routing information propagation between the sites involved in the service.

The pre-requisites for this use case are the followings:

1. Reachability between PE devices where the L2 VPN is going to be deployed.

The main set of functionalities covered in this use case are the followings:

2. To provide a name and a description for the new L2VPN service;
3. The type and data plane encapsulation in this use case is limited to L2VPNs (type=L2VSI) over MPLS (encapsulation-type=MPLS);
4. To configure the operational state and configuration parameters relating to the forwarding database of the network instance;
5. Enable/disable the configured network instance on the network element (Note: Some vendor's implementations could not allow the disable function, so the network instance will be enabled immediately after creation and maintain enable until it is deleted);
6. Attach subinterface(s) to be bound to the L2VPN;
7. Attach connection points within a forwarding instance.

Functional Requirements

The following steps are used to create the L2VPN Service. Each step has an associated set of functionalities:

Create VPN-Nodes:

- Create ELAN Name
- Change Operational status / Administrative status

Create each VPN Network access to the corresponding VPN Node. The VPN network access refers to the L2 termination-points to the VPN-Node:

- Create Interfaces $(1 \dots n)$;
- For each interface Add Interface-type;
- Add VLAN information to an interface;
- Add MTU configuration to interface;
- Change Administrative Status on the interface.

Attach interface to Network Instance:

- Add interface to VPN Node.

Attach virtual circuits to Network Instance:

- Add remote end-point.

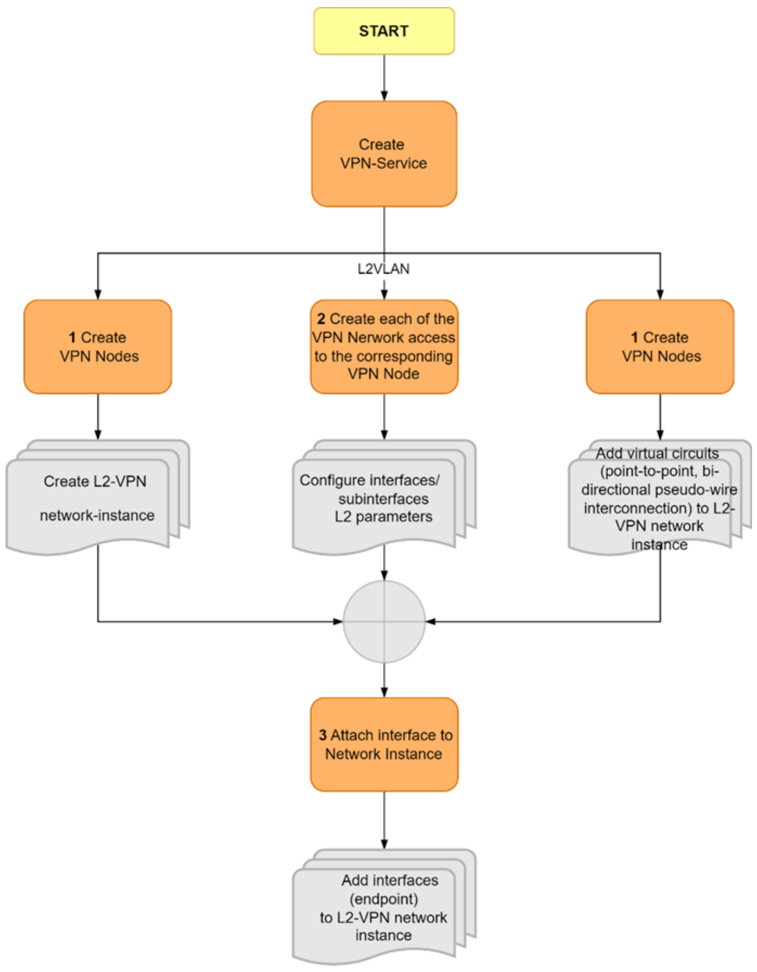

START
Create VPN-Service
L2VLAN
1 Create VPN Nodes
2 Create each of the VPN Nerwork access to the corresponding VPN Node
1 Create VPN Nodes
Create L2-VPN network-instance
Configure interfaces/ subinterfaces L2 parameters
Add virtual circuits (point-to-point, bi-directional pseudo-wire interconnection) to L2-VPN network instance
3 Attach interface to Network Instance
Add interfaces (endpoint) to L2-VPN network instance

Figure 81: L2VPN functional requirements.

4.2.3.2. 1L3VPN Network Model (TID)

L3VPN services are widely deployed in IP/MPLS networks. This use case allows the provisioning, modification and deletion of a Layer 3 VPN service spanning one or more IP/MPLS routers via the SDN controller using a subset of the L3NM data model (RFC 9182). The L3VPN creates a virtual routing network instance (usually known as VRF) in each of the routers involved in service deployment. The routing instance (VRF) created at every router allows the routing information propagation between the sites involved in the service.

The pre-requisites for this use case are the followings:

- Reachability between PE devices where the L3 VPN is going to be deployed;
- BGP reachability with VPN address family enabled.

The main set of functionalities covered in this use case are the followings:

- To provide a name and a description for the new L3VPN service;
- The type and data plane encapsulation in this use case is limited to BGP-based L3VPNs (type=L3VRF) over MPLS (encapsulation-type=MPLS);
- Set the route distinguisher (RD) used for each VRF when it is signaled via BGP. In this use case, the RD must be explicitly provided;

- It is possible to configure a router-id to identify the routing device and used by BGP and OSPF to function in a routing instance;
- Enable/disable the configured network instance on the network element (Note: Some vendors implementation could not allow disable it so that the network instance will be enabled immediately after creation and keep enable until it is deleted);
- The label allocation (per prefix, per next-hop or a single label per VRF) is selected by each router vendor. For ADVA, which allows its selection, it is fixed to single label per VRF;
- To enable the Address Families (AF) supported within the L3VPN. Note that some vendors enable all families by default (Juniper);
- To configure Route Target for export and import. The controller will take care of creating the policies in the device automatically;
- Create vpn_node (VRF) profiles to reuse when there are multiple VRFs;
- VPN topologies can be indicated (hub & spoke, full mesh, custom) as informative. The topology is achieved via RT assigned for import and export;
- Attach subinterface(s) to be bound to the L3VPN at L2 (single or double tagging, Lag members if apply) and configure L3 information (IP address, Loopback type interface);
- To attach existing routing policies to VPN for import and export;
- To control the VPN lifecycle using status variables such as: pre-deployment, testing or up;
- To add a static route to a VPN node (VRF) indicating prefix, next hop and (optional) metric.

Functional Requirements

The following steps are used to create the L3VPN Service, each step has associated a set of functionalities:

Create VPN-Nodes:

- Create VRF Name;
- Create Router ID;
- Create Route Distinguisher;
- Attach Import / Export Policies;
- Change Operational status / Administrative status;
- Define Service Topology (e.g., hub-spoke, multi-point-to-multi-point);
- Define node Topology-role;
- Create Maximum Route Limits into a VRF.

Create each of the VPN Network access to the corresponding VPN Node. The VPN network access refers to the L3 termination-points to the VPN-Node:

- Crate Interfaces $(1 \dots n)$;
- For each interface Add Interface-type;
- For each interface Add Interface Encapsulation-type;
- Add VLAN information to an interface;
- Add Addressing parameters to an interface;
- Add MTU configuration to interface;
- Change Administrative Status on the interface.

Attach interface to Network Instance:

- Add interface to VPN Node.

Create Import and Export conditions:

- Add a routing policy to the VPN Node;
- Create community members into a community set;
- Add community values to routes imported from a particular protocol;
- Filter based on Prefix List.

Attach import/export policy to Network Instance:

- Add import policy to VPN Node;
- Add export policy to VPN Node.

Add Routing protocol to the CE-PE connectivity:

- Add CE-PE static routing connectivity;
- Add static routing redistribution into the VRF.

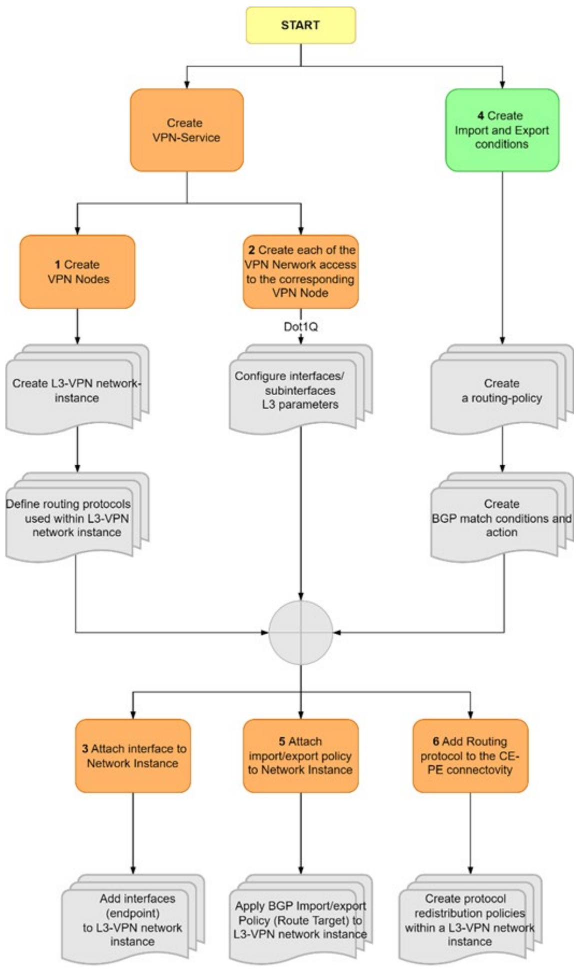

START
Create VPN-Service
4 Create Import and Export conditions
1 Create VPN Nodes
2 Create each of the VPN Nerwork access to the corresponding VPN Node
Dot1Q
Create L3-VPN network-instance
Configure interfaces/ subinterfaces L3 parameters
Create a routing-policy
Define routing protocols used within L3-VPN network instance
Create BGP match conditions and action
3 Attach interface to Network Instance
5 Attach import/export policy to Network Instance
6 Add Routing protocol to the CE-PE connectovity
Add interfaces (endpoint) to L3-VPN network instance
Apply BGP Import/export Policy (Route Target) to L3-VPN network instance
Create protocol redistribution policies within a L3-VPN network instance

Figure 82: L3VPN functional requirements.

4.2.3.3. P4-based L2 service handler

The L2NM P4 service handler implements two key RPCs that the Service component exposes to other TeraFlowSDN components or third parties. These RPCs are shown in Table 28. These RCPs allow to create a set of endpoints that a service traverses, update these endpoints in case of a change, and delete these endpoints when the service is no longer active. With these two RPCs, the service handler can establish L2 connectivity on arbitrary P4 topologies.

Table 28: RPCs implemented by the L2NM P4 service handler

RPC Method Name	Parameters	Results
SetEndpoint	List of endpoints	List of boolean status/endpoint operation
DeleteEndpoint	List of endpoints	List of boolean status/endpoint operation

4.2.3.4. Microwave service handler

The MW Service Handler implements two key RPCs that the Service component exposes to be used by other TeraFlow SDN components. This RPC is described in Table 29 below.

With these two RPCs, the service handler can manage the configuration of an L3VPN service traversing the MW domain.

These two RCPs allow both to create an L2 path identified by Service UUID between two MW edge endpoints that are traversed by an L3VPN service and then to delete the path when it is no more required.

Table 29: RPCs implemented by the MW service handler

RPC Method Name	Parameters	Results
SetEndpoint	L2 Service UUID List of edge endpoints	Boolean status reporting operation result
DeleteEndpoint	L2 Service UUID	Boolean status reporting operation result

4.2.3.5. REST servicer handler

The TAPI service handler implements two key RPCs that the Service component exposes to other TeraFlowSDN components or third parties. These RPCs are shown in Table 30. These RCPs allow to create a set of endpoints that a service traverses, update these endpoints in case of a change, and delete these endpoints when the service is no longer active. With these two RPCs, the service handler can establish TAPI connectivity on arbitrary TAPI-enabled OLS controllers.

Table 30: RPCs implemented by the TAPI service handler

RPC Method Name	Parameters	Results
SetEndpoint	List of endpoints	List of boolean status/endpoint operation
DeleteEndpoint	List of endpoints	List of boolean status/endpoint operation

TAPI service handler is extended to support XR Constellation driver due to service layer synergies. In addition, the TAPI service is extended with a new attribute describing planned optical bandwidth on XR announced port, which XR Constellation driver updates via IPM REST API towards XR constellation.

4.2.4. Final Operational Workflows

The operational workflows for the Service component are described in this section. The workflows have been generalized to cover all combinations of connectivity service types, service handlers, device types and device drivers. Besides, the workflow assumes a request from an external OSS/BSS while it is not mandatory and might come from other TeraFlow components.

Three workflows directly initiated by external components/systems are described, i.e., *CreateService*, *UpdateService*, and *DeleteService*. Besides, given its complexity, an additional workflow corresponding to the *Execute* method of the internal Task Scheduler used by the Service component. This *TaskScheduler:Executor* method is described separately since it is used by both the *UpdateService* and the *DeleteService*.

The *CreateService* workflow (shown in Figure 83) starts when some entity, e.g., the Slice component or an OSS/BSS, triggers a connectivity service configuration. Given that some systems might require to reserve a connectivity service identifier before adding endpoints, constraints, etc. to that connectivity service, the creation of a connectivity service is done in two steps. The first step consists in creating an empty connectivity service where only the connectivity service type is specified and returning the identifier. The second step (described below) consists in updating the connectivity service by populating the required fields.

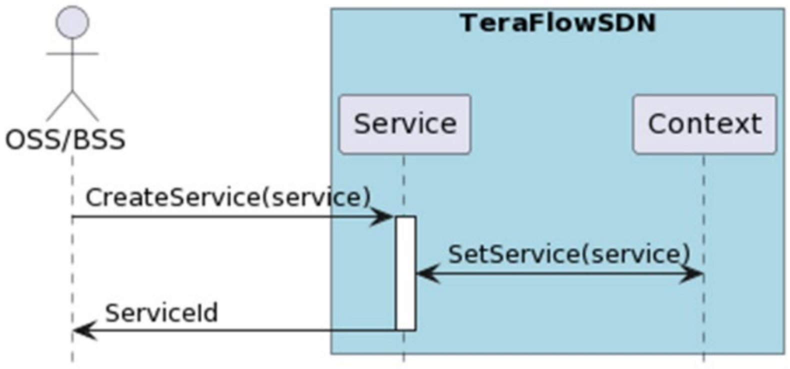


Figure 83: Generic CreateService workflow

Now focusing on the *UpdateService* (shown in Figure 84), the Service component first interrogates the Context component to retrieve the most up-to-date version of the connectivity service. Next, it sets the state of the connectivity service to PLANNED and requests a path computation to the PathComp component to decide whether new paths are required for possibly new endpoints added or constraints changed.

The response of the PathComp component includes, at least, the connectivity service requested. If a path has been found for that service, it also carries the connection path computed for it. Besides, in case of traversing multiple layers, for instance, a packet-layer connection requiring new underlying optical connection(s), the reply will also include one or more sub-services with the corresponding sub-connections supporting them.

Then, a Task Scheduler is instantiated, and the (sub-)services and (sub-)connections computed by the PathComp are correlated and scheduled. For instance, when an optical connection supports a packet connection, the Service component should first setup the optical-layer connection and only when it becomes operational, setup the packet-layer connection. Finally, when correlated and scheduled, the

Execute method of the TaskScheduler (described below) is executed to perform all the required configuration operations. Upon completion, the service is updated in the Context component, and the service identifier is returned to the caller entity.

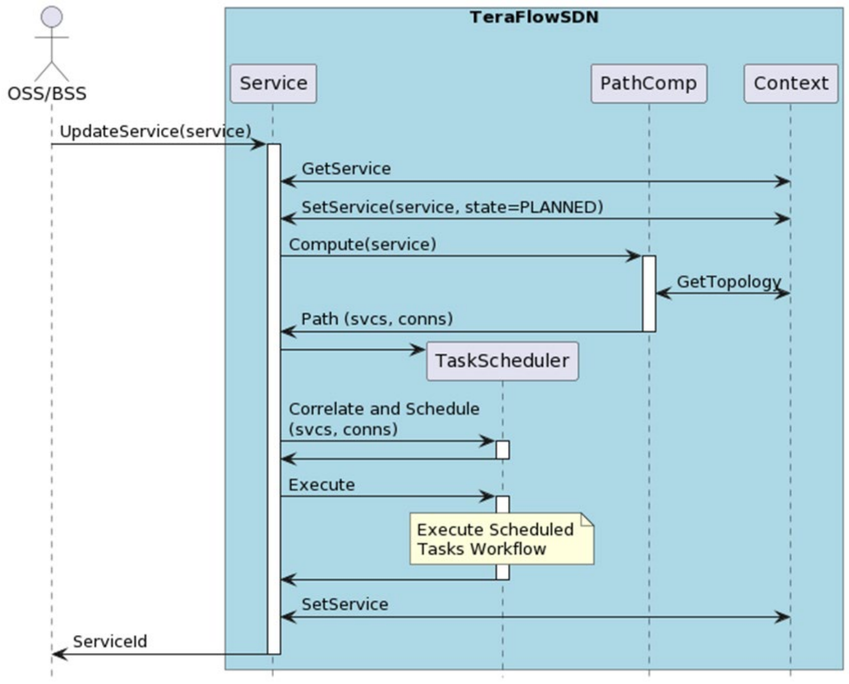


Figure 84: Generic UpdateService workflow.

The DeleteService workflow (depicted in Figure 85) begins by retrieving from the context component the Service to be deleted and marking it as PENDING_REMOVAL. Then, a TaskScheduler is instantiated and populated with the services and connections properly retrieved from the Context component and scheduled according to their dependencies. Again, a connection cannot be deleted if it supports an ACTIVE service. A service supported by sub-services needs to be deactivated and removed before the sub-services and sub-connections are torn down. When populated, the *Execute* method of the Task Scheduler is executed and, upon completing the execution of the tasks, the control is returned to the caller entity.

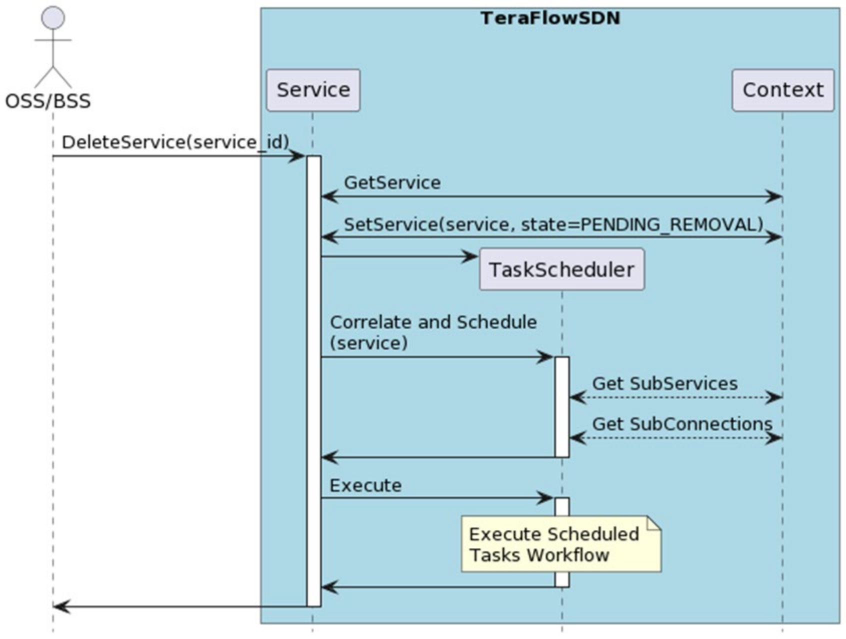

TeraFlowSDN
OSS/BSS
Service
Context
DeleteService(service_id)
GetService
SetService(service, state=PENDING_REMOVAL)
TaskScheduler
Correlate and Schedule (service)
Get SubServices
Get SubConnections
Execute
Execute Scheduled Tasks Workflow

Figure 85: Generic DeleteService workflow.

The Task Scheduler is responsible for executing the tasks to set-up and tear-down services and connections in the appropriate order. For that, the correlation and scheduling methods consist in creating a Directed Acyclic Graph of tasks and assigning as predecessors the tasks that need to be completed before the actual task can be initiated.

As an illustrative example, a connection between routers R1 and R3 in the Multi-layer network depicted in Figure 86 needs to be established. For that, we need to pass through R2, but there is no direct connection between R1 and R2, and between R2 and R3. Thus, we must establish virtual links (dotted lines) passing through the underlying optical network (Reconfigurable Optical Add Drop Multiplexers depicted in green).

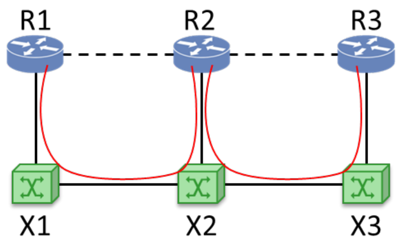


Figure 86: Example Multi-Layer Network.

To resolve this, the PathComp component would retrieve 3 services and 3 connections as follows:

- **<u>Main Service (Svc:R1-R3)</u>** on the packet layer (uses L2NM or L3NM depending on connection settings) between routers R1 and R3 that depends on connection R1-R3.
    - **<u>Connection (Conn:R1-R3)</u>** that depends on sub-services R1-R2 and R2-R3.
        - **<u>Sub-Service (Svc:R1-R2)</u>** on the optical layer (TAPI service handler) between transponders in routers R1 and R2 that depends on connection R1-R2.
            - **<u>Connection (Conn:R1-R2)</u>** on the optical layer.
        - **<u>Sub-Service (Svc:R2-R3)</u>** on the optical layer (TAPI service handler) between transponders in routers R2 and R3 that depends on connection R2-R3.

- **Connection (Conn:R2-R3)** on the optical layer.

The set of supported schedulable tasks are:

- For setup operations:
    - **ConfigureConnection(connection)**: uses the selected service handler to compose the set of configuration rules for the traversed devices and interacts with the Device component to apply these configurations.
- For teardown operations:
    - **DeconfigureConnection(connection)**: uses the selected service handler to compose the set of deconfiguration rules for the traversed devices and interacts with the Device component to apply these deconfigurations.
    - **DeleteService(service)**: Removes a service when it has been deconfigured on the traversed devices.
- For both operations:
    - **SetServiceState(service, new state)**: Changes the state of a (sub-)service during the setup process (e.g., mark a service as ACTIVE, PENDING_REMOVAL, etc).

The set of tasks that would result from the Task Scheduler would be as follows (note that multiple options are feasible):

- *ServiceSetState(Svc:R1-R3, state=PLANNING)*
- *ServiceSetState(Svc:R1-R2, state=PLANNING)*
- *ConnectionConfigure(Conn:R1-R2)*
- *ServiceSetState(Svc:R1-R2, state=ACTIVE)*
- *ServiceSetState(Svc:R2-R3, state=PLANNING)*
- *ConnectionConfigure(Conn:R2-R3)*
- *ServiceSetState(Svc:R2-R3, state=ACTIVE)*
- *ConnectionConfigure(Conn:R1-R3)*
- *ServiceSetState(Svc:R1-R3, state=ACTIVE)*

The scheduling for a delete operation would be extrapolated by reversing the sequence of tasks, replacing tasks *ServiceSetState*(*PLANNED*) by *ServiceDelete*(), replacing *ConnectionConfigure*() by *ConnectionDeconfigure*(), and adding required *ServiceSetState*(*PENDING_REMOVAL*) tasks at the beginning of the sequence.

Now that the Task Executor has been described conceptually, the workflow for the *Execute* method of the Task Scheduler illustrated in Figure 87 can be understood.

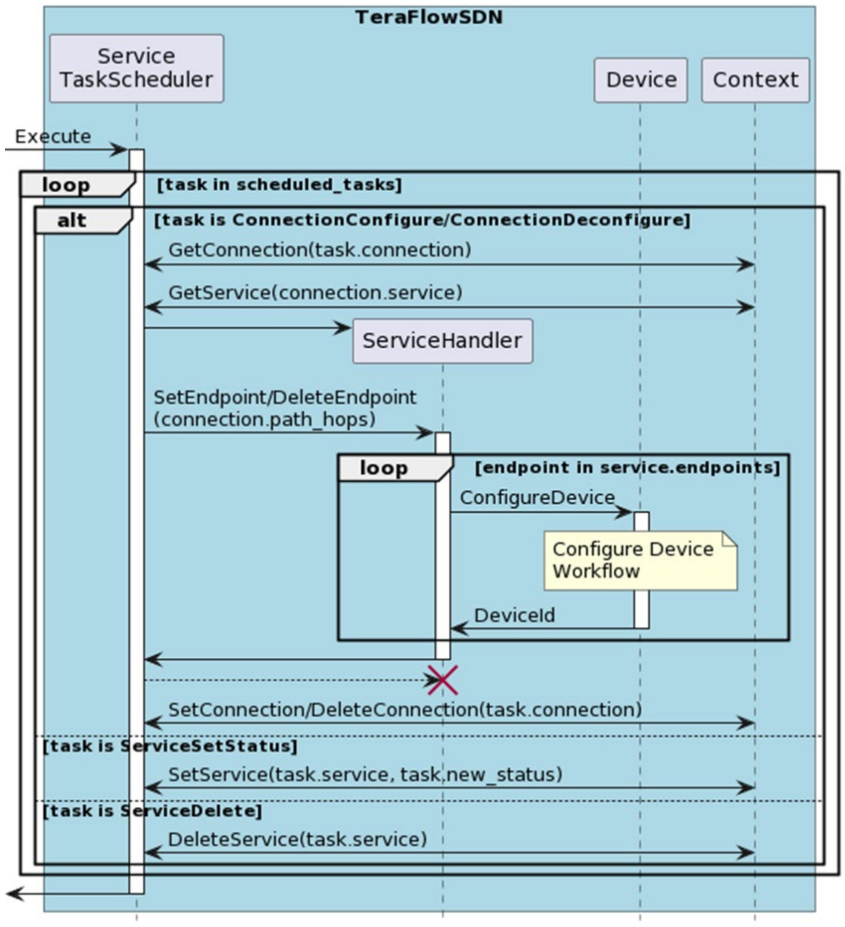

TeraFlowSDN
Service TaskScheduler
Device
Context
Execute
loop
[task in scheduled_tasks]
alt
[task is ConnectionConfigure/ConnectionDeconfigure]
GetConnection(task.connection)
GetService(connection.service)
ServiceHandler
SetEndpoint/DeleteEndpoint (connection.path_hops)
loop
[endpoint in service.endpoints]
ConfigureDevice
Configure Device Workflow
DeviceId
SetConnection/DeleteConnection(task.connection)
[task is ServiceSetStatus]
SetService(task.service, task.new_status)
[task is ServiceDelete]
DeleteService(task.service)

Figure 87: Generic TaskScheduler::Execute workflow.

Even though this workflow is general, it is complete enough to cover all the cases supported by the different Service Handlers. Note that the *ConnectionConfigure* and *ConnectionDeconfigure* are the responsible for instantiating the appropriate service handlers taking as input the service types and the specifications of the traversed devices (e.g., device type, supported drivers, etc). Besides, the *ConnectionConfigure* and *ConnectionDeconfigure* tasks use the *ConfigureDevice* RPC provided by the Device component to interact with the traversed devices. The details on the ConfigureDevice workflow are presented in section 4.1.5.

4.2.5. Evaluation

The Service component's evaluation consists of evaluating each Service Handler using reasonable scenarios. In particular, execution time spent by the different methods of the Service Handler API. Note that methods evaluated are those in charge of setting and deleting endpoints, i.e., SetEndpoint and DeleteEndpoint. The other four methods SetConfigRule, DeleteConfigRule, SetConstraint and DeleteConstraint are included in the Service Handler API, but are still not implemented. They are left as placeholders for future extensions to manage dynamic changes in constraints and configuration rules not addressed in the current implementation of the TeraFlow controller.

PERFORMANCE NOTE: Current activities involve integrating a new implementation of the Context component based on CockroachDB, a distributed, scalable and performant relational database. The current implementation is based on a database that does not supporting concurrency and with limited performance, which results in bad performance results. Given that Service Handlers highly rely on Context component and Device (which in turn makes use of Context to correlate configuration rules to be configured), the performance reported in this section should be considered preliminary and for functional evaluation only. We expect to achieve better performance after finishing this integration. The final results will be released in deliverable D5.3 [8].

4.2.5.1. L2VPN Network Model

Experimental setup

We used the same setup (server and micro-service deployment) described for the OpenConfig Device Driver (see section 4.1.3.4).

Functional Evaluation

The functional evaluation of the L2VPN Network Model (L2NM) Service Handler using OpenConfig (L2NM OpenConfig) was done together with the OpenConfig Device Driver (see Section 4.1.3.4). Service Handler performance evaluation results were filtered from the overall L2NM+L3NM performance evaluation.

Numerical Results

The considered metric for assessing the L2NM OpenConfig Service Handler is the delay incurred by the service handler, including the overhead of the device component, the corresponding OpenConfig Device Driver, the Context component, and the setup of the underlying routers. The specifications of the requests are those described in Section 4.1.3.4 for the OpenConfig Device Driver.

The results obtained from the OpenConfig device driver in section 4.1.6.4, directly affect the performance of the L2NM OpenConfig Service Handler, as both the SetEndpoint and DeleteEndpoint RPCs invoke low-level RPCs to the TeraFlowSDN SBI component, which re-directs these RPCs to the SetConfig/GetConfig/DeleteConifig RPCs of the OpenConfig device driver.

In Figure 88, it is shown the CDF of the L2NM OpenConfig Service Handler latency for the generated 100 requests. We observe that the majority of SetEndpoint and DeleteEndpoint requests take more than 100 seconds. The origin of this delay comes from: (i) the configuration of the rules in the packet routers, and (ii) from the database receiving many query and update requests in parallel and having to process them sequentially. The first delay comes by construction of the equipment and is out of the scope of the TeraFlow project activities. We expect to resolve the second delay after completing the integration of the new version of the Context component using the CockroackDB.

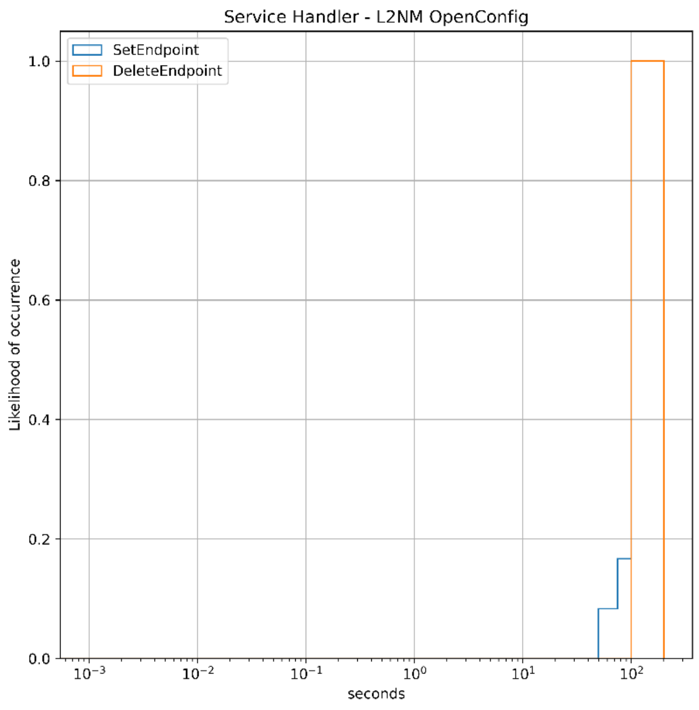


Figure 88: CDF for the L2VPN Network Model Service Handler with OpenConfig Delay.

4.2.5.2. L3VPN Network Model

Experimental setup

We used the same setup (server and micro-service deployment) described for the OpenConfig Device Driver (see Section 4.1.3.4).

Functional Evaluation

The functional evaluation of the L3VPN Network Model (L3NM) Service Handler using OpenConfig (L3NM OpenConfig) was done together with the OpenConfig Device Driver (see Section 4.1.3.4). Service Handler performance evaluation results were filtered from the overall L2NM+L3NM performance evaluation.

Numerical Results

The considered metric for assessing the L3NM OpenConfig Service Handler is the delay incurred by the service handler, including the overhead of the device component, the corresponding OpenConfig Device Driver, the Context component, and the setup of the underlying routers. The specifications of the requests are those described in Section 4.1.3.4 for the OpenConfig Device Driver.

The results obtained from the OpenConfig device driver in section 4.1.6.4, directly affect the performance of the L3NM OpenConfig Service Handler, as both the SetEndpoint and DeleteEndpoint RPCs invoke low-level RPCs to the TeraFlowSDN SBI component, which re-directs these RPCs to the SetConfig/GetConfig/DeleteConifig RPCs of the OpenConfig device driver.

In Figure 89, it is shown the CDF of the L3NM OpenConfig Service Handler latency for the generated 100 requests. We observe that majority of SetEndpoint and DeleteEndpoint requests take more than 100 seconds. The origin of this delay comes from: (i) the configuration of the rules in the packet routers, and (ii) from the database receiving many query and update requests in parallel and having to process them sequentially. The first delay comes by construction of the equipment and is out of the scope of the TeraFlow project activities. We expect to resolve the second delay after completing the integration of the new version of the Context component using the CockroackDB.

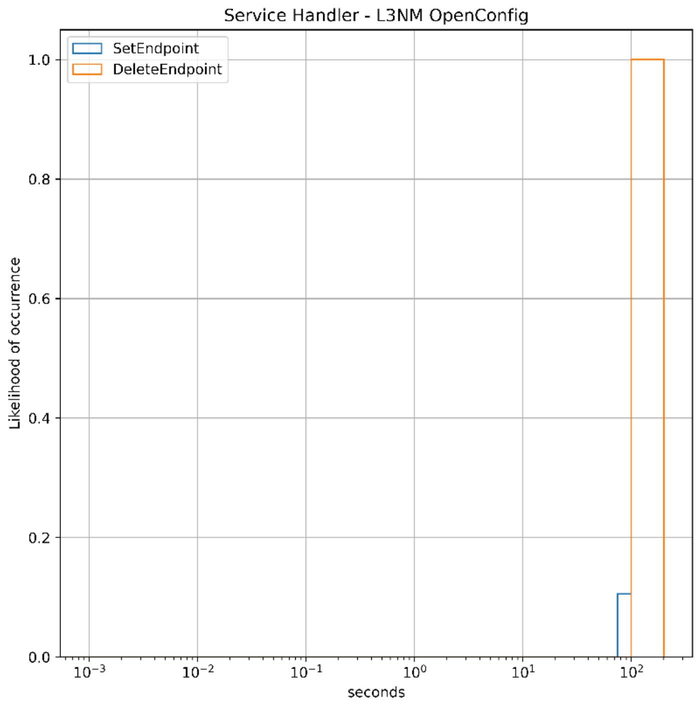


Figure 89: CDF for the L3VPN Network Model Service Handler with OpenConfig Delay.

4.2.5.3. P4-based L2 service handler

Experimental setup

To conduct the experiments of the L2NM P4 service handler, we used the same setup as in the case of the P4 SBI driver (see Section 4.1.6). TeraFlowSDN is deployed as a Kubernetes service on this testbed, and no resource limits are se to the Service, SBI, and Context components to allow stress testing.

Benchmarking procedure

To benchmark the P4 service handler we use an emulated (Mininet) topology of one or more software-based P4 switches. In each experiment, an increasing number of L2NM services are created atop pairs of P4 devices. The number of services increases linearly between 1-5 services.

The following benchmark measures the total time (in seconds) required to perform service creation/deletion for 1-5 services. The CreateService RPC is invoked by an external client, which measures the time it takes to complete the service request before issuing the next one.

Service creation: Figure 90 depicts the total execution time in seconds (y-axis) required to provision 1-5 L2 P4 services vs. the time (x-axis). As highlighted by the points, this time is:

- 144ms for 1 service (indicated by the red point);
- 523ms for 2 services (indicated by the green point);
- 884ms for 3 services (indicated by the blue point);
- 1610ms for 4 services (indicated by the magenta point);
- 2600ms for 5 services (indicated by the white point).

We fit a linear function on these points to quantify the additional time per service, which is 600ms per service.

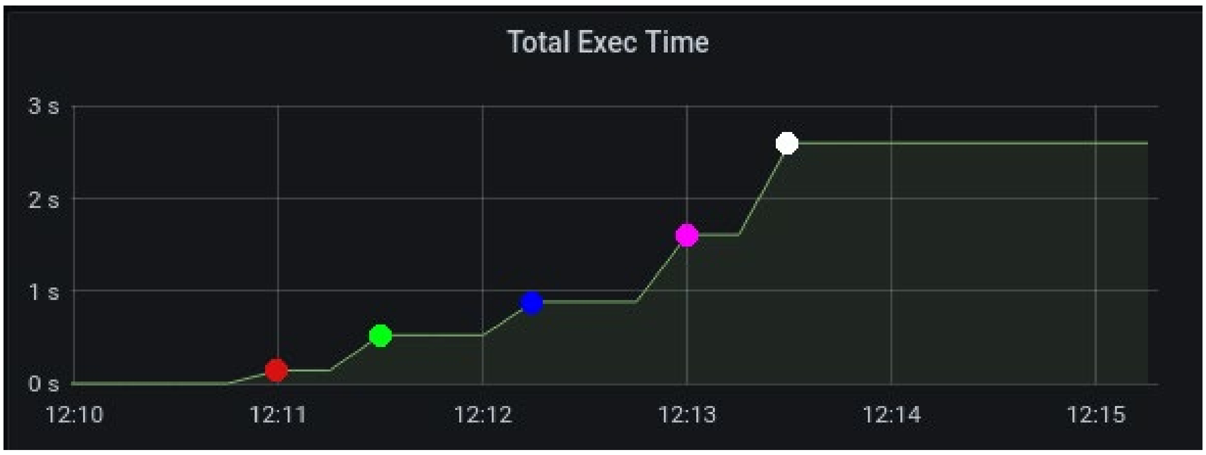


Figure 90: Latency to establish 1-5 L2NM services atop pairs of P4 devices.

Service deletion: Figure 91 depicts the total execution time in seconds (y-axis) required to decommission 1-5 L2 P4 services vs. the time (x-axis) using the SetEndpoint RPC of the P4-based L2 service handler. As highlighted by the points, this time is:

- 180ms for 1 service (indicated by the red point)
- 440ms for 2 services (indicated by the green point)
- 1230ms for 3 services (indicated by the blue point)
- 1890ms for 4 services (indicated by the magenta point)
- 2790ms for 5 services (indicated by the white point)

We fit a linear function on these points to quantify the additional time per service, which is 667ms per service using the DeleteEndpoint RPC of the P4-based L2 service handler. This time is slightly higher than the SetEndpoint RPC, but still comparable.

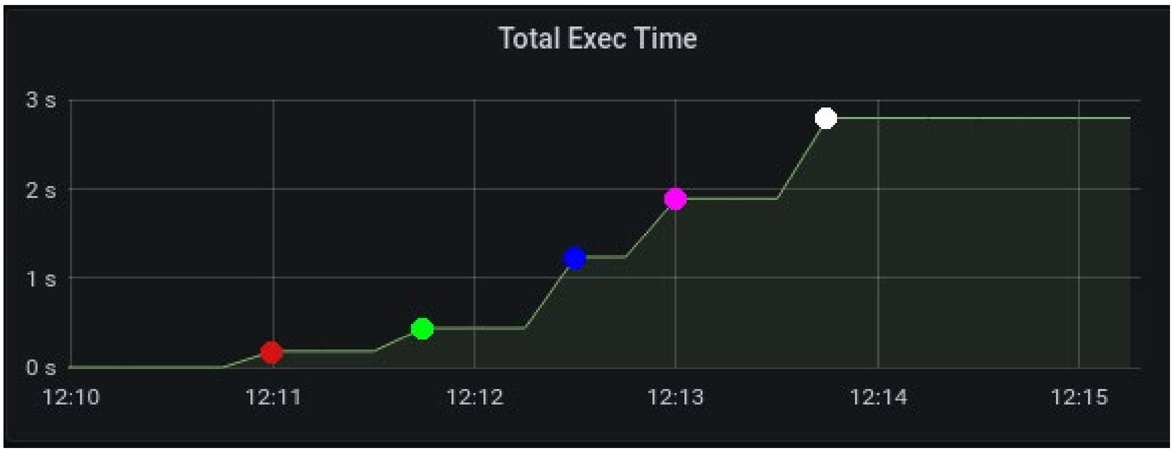


Figure 91: Latency to tear 1-5 L2NM services down atop pairs of P4 devices.

The results obtained from the P4 device driver in Section Y above directly affect the performance of the P4 service handler, as both the SetEndpoint and DeleteEndpoint RPCs invoke low-level RPCs to the TeraFlowSDN SBI component, which re-directs these RPCs to the SetConfig/GetConfig/DeleteConifig RPCs of the P4 device driver.

4.2.5.4. Microwave service handler

Experimental setup

We used the same setup (server and micro-service deployment) described for the MicroWave Device Driver (see section 4.1.6.3).

Functional Evaluation

The functional evaluation of the MicroWave Service Handler was done together with the MicroWave Device Driver (see section 4.1.6.3).

Numerical Results

The considered metric for assessing the MW Service Handler is the delay incurred by the service handler, including the overhead of the device component, the corresponding MicroWave Device Driver, the Context component, the MicroWave controller, and the setup of the underlying Network Equipment. The specifications of the requests are those described in section 4.1.6.3 for the MicroWave Device Driver.

In Figure 92, it is shown the CDF of the MicroWave Service Handler latency for the generated 100 requests. We observe that the majority of SetEndpoint and DeleteEndpoint requests take around 50 seconds. The origin of this delay comes from: (i) the database receiving many query and update requests in parallel and having to process them sequentially. We expect to resolve this issue after completing the integration of the new version of the Context component using the CockroackDB; and (ii) from the MicroWave Network Equipment having to process sequentially the configuration of the rules in the devices. This second limitation comes by construction of the equipment and is out of the scope of the TeraFlow project activities.

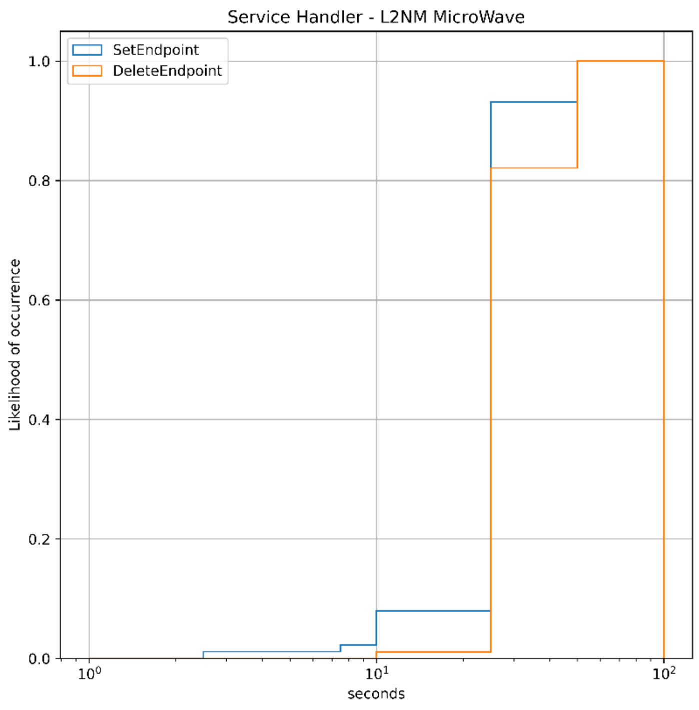


Figure 92: CDF for the MicroWave Service Handler Delay.

4.2.5.5. REST servicer handler (TAPI and XR Constellation)

Experimental setup

We used the same setup (server and micro-service deployment) described for the Transport API Device Driver Evaluation (see section 4.1.6.2) including the proprietary Open Line System controller exposing a Transport API NBI and controlling an emulated underlying optical data plane formed by 4 nodes with 10 service interconnection points each of them.

Numerical Results

The metric for assessing the Transport API Service Handler is the delay incurred by the service handler to (de)configure the optical connections on the OLS controller through the Device component and the Transport API device driver. This includes the dispatching time of these requests within the OLS controller and the overheads introduced by the Device and Context components. The request generation is as for the Transport API Device Driver Evaluation (see section 4.1.6.2).

In Figure 93, it is shown the CDF of the Transport API Service Handler latency for the generated 100 requests. We observe that the majority of configuration/deconfiguration requests take around 10s. We are currently on the integration phase of CockroachDB, a distributed and scalable database, in Context component. We expect to achieve much better performance after finishing this integration.

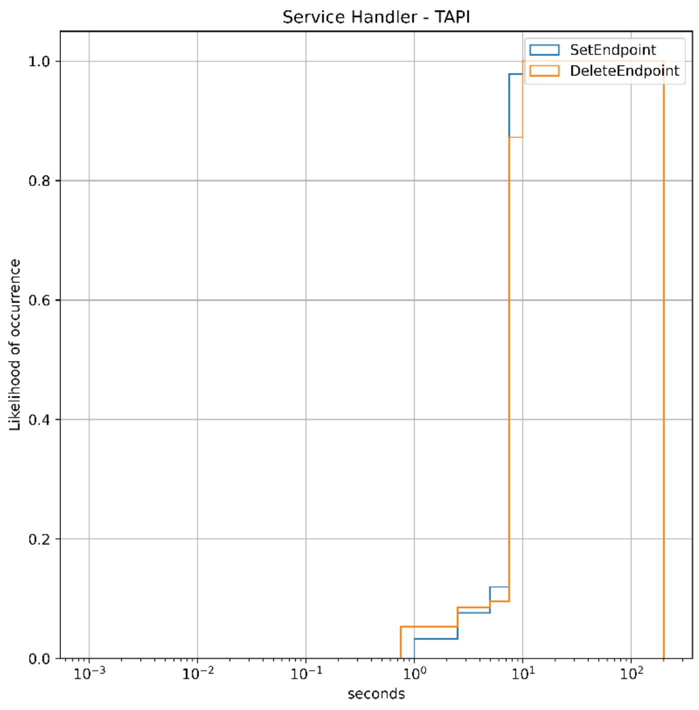


Figure 93: CDF for the Transport API Service Handler Delay.

4.3. Forecaster Component

The final report on the Forecaster TeraFlowSDN component, focuses on:

- The new features introduced to this component;
- The design overview of the Forecaster component;
- The set of interfaces of the Forecaster component;
- The operational workflows of the Forecaster component;
- A performance evaluation study.

4.3.1. New Features/Extensions

This section summarizes the complete features of the Forecaster component, as it is a new component part of TeraFlowSDN release v2.

- Forecaster can obtain the historic of requested and serviced connectivity services with duration and capacity constraints from Context component;
- Forecaster includes ML algorithms (currently using Prophet) for predicting traffic forecasts;
- Traffic forecasts are analyzed before determining whether to accept a new service request.

4.3.2. Final Design

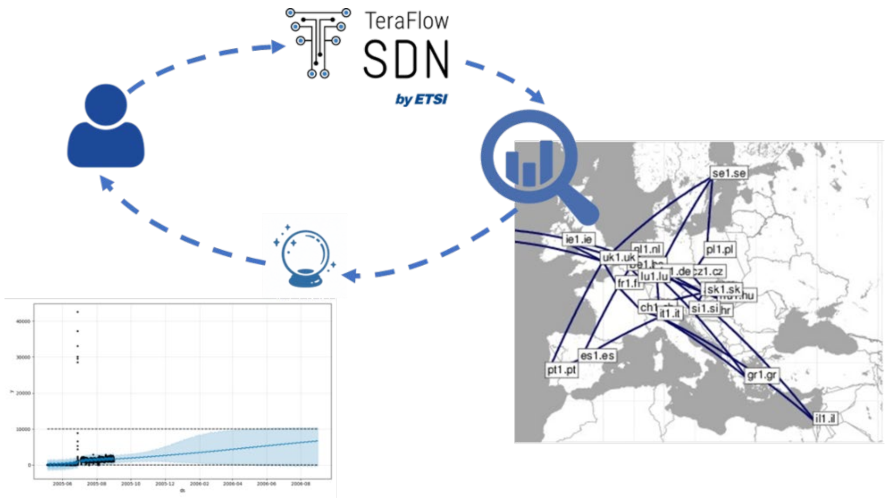


Figure 94 Traffic forecasting with SDN controller. Source of sample network: [SDNlib10]

Figure 94 shows the proposed component behaviour. The TeraFlowSDN controller can monitor current network status and obtain timestamped traffic matrices on top of as an example pan-European core network (we are using Geant as an example [19]). Then, a traffic forecasting application (forecaster) can introduce ML-based algorithms to predict the network status in the future. This information can be provided to the OSS/BSS in order to trigger necessary link updates, as well as provided back to the SDN controller in order to limit resources allocated to specific network links.

Forecaster is a novel TFS component that can perform proactive SDN traffic prediction (i.e., forecasts) using ML algorithms. For example, it is able to collect real-time Key Performance Indicators (KPI), such as link occupancy, and use ML algorithms to forecast where and when a problem (e.g., link resource unavailable) is likely to occur, so as to reroute traffic before it happens.

Multiple traffic forecasting libraries can be introduced to provide the component with multiple engines. One of the possibilities mentioned above is the Prophet [20] library, which already includes a seasonal model for data forecasts. Another possibility could be the introduction of AutoML [21]. AutoML does not require training; thus, no previous data modelling is required.

4.3.3. Final Interfaces

The interfaces provided by the Forecaster component (see for reference Table 31) provide a complete network forecast (GetForecastOfTopology) by introducing a topology identifier or obtain a specific forecast of a link (GetForecastOfLink), by introducing a link identifier. A Forecast is a data structure that contains an array of timestamped predictions for a specified time period for a specific link. Another provided interface allows to check if a new requested connectivity service will have resources available in the future. To this end a link resource availability threshold is configured and the Forecast component provides a ForecastPrediction with the decision.

RPC Method Name	**Parameters**	**Results**
GetForecastOfTopology	context.TopologyId	Forecast
GetForecastOfLink	context.LinkId	Forecast
CheckService	context.ServiceId	ForecastPrediction

Table 31 Forecaster interface definition

4.3.4. Final Operational Workflows

Figure 95 describes the proposed sequence diagram to provide traffic forecasts within TeraFlowSDN controller. Two different usages are depicted. The first refers to link/topology availability forecast, while the second focuses on the analysis of link availability for an upcoming connectivity service request.

A user, through Business Support System (BSS) or Operations Support System (OSS) can request a link/topology traffic usage forecast (Step 1) that is received by the Forecaster Component. At the receipt of a traffic forecast request, topology (Steps 2-3) and Service (Steps 4-5) are requested to Context, which acts as the TFS database. A forecast is computed using the received information and provided to the user (Step 6).

Another supported feature is the analysis of a service requested by the OSS/BSS (Step 7). Service component requests a forecast prediction to Forecast component based on the received Service request (Step 8). The prediction mechanism detailed in the previous paragraph is detailed (Step 9) and a decision is taken using a certain bandwidth utilization threshold and notified to Service component (Step 10), which can decide not to allocate resources based on the prediction or follow standard service allocation procedures

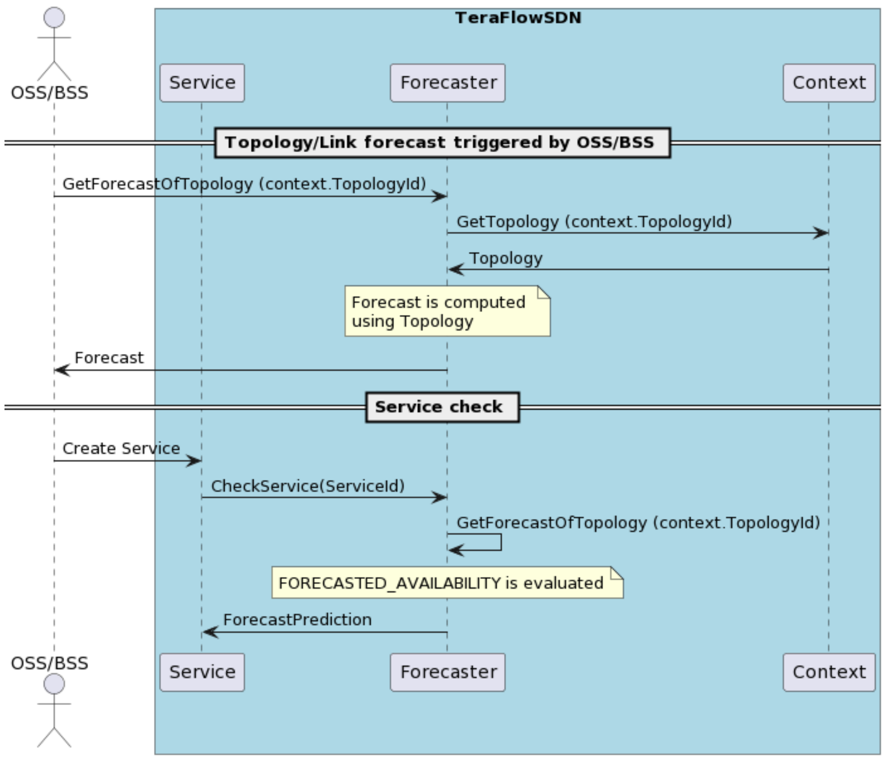

TeraFlowSDN
OSS/BSS
Service
Forecaster
Context
Topology/Link forecast triggered by OSS/BSS
GetForecastOfTopology (context.TopologyId)
GetTopology (context.TopologyId)
Topology
Forecast is computed using Topology
Forecast
Service check
Create Service
CheckService(ServiceId)
GetForecastOfTopology (context.TopologyId)
FORECASTED_AVAILABILITY is evaluated
ForecastPrediction

Figure 95 Forecaster Sequence Diagram

4.3.5. Evaluation

The Forecaster component has been programmed using python. It offers a gRPC interface based on a protocol buffer, and it can interact with the other necessary TFS micro-service (i.e., Context and

Service). It includes Prophet library [PRO22] and it is expected to be extended with other forecasting libraries, such as AutoML.

In order to provide meaningful forecasts, we have introduced in TFS an emulated GEANT network with link traffic matrices provided in [19].

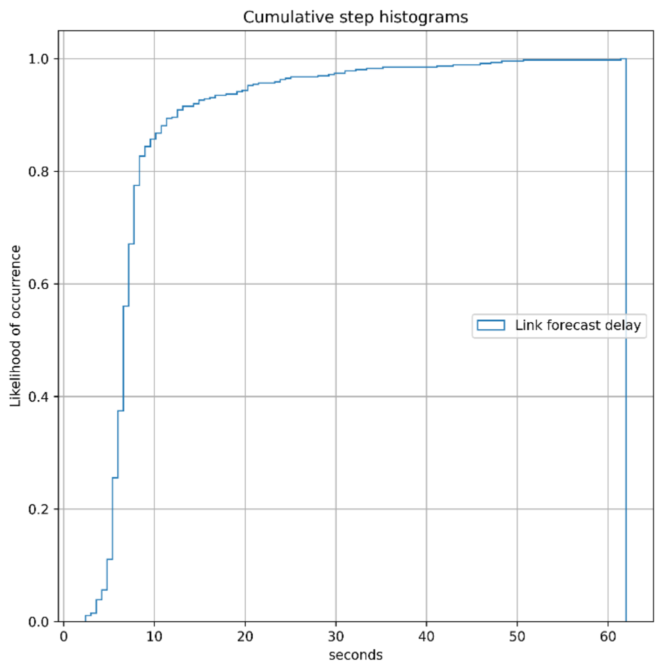


Figure 96 CDF of traffic forecasting delay

Figure 96 shows the Cumulative Distribution Function for the 464 link traffic forecast delays computed for the specified GEANT network. The mean link forecast delay is 8.72 seconds and its standard deviation is 6.72 seconds.

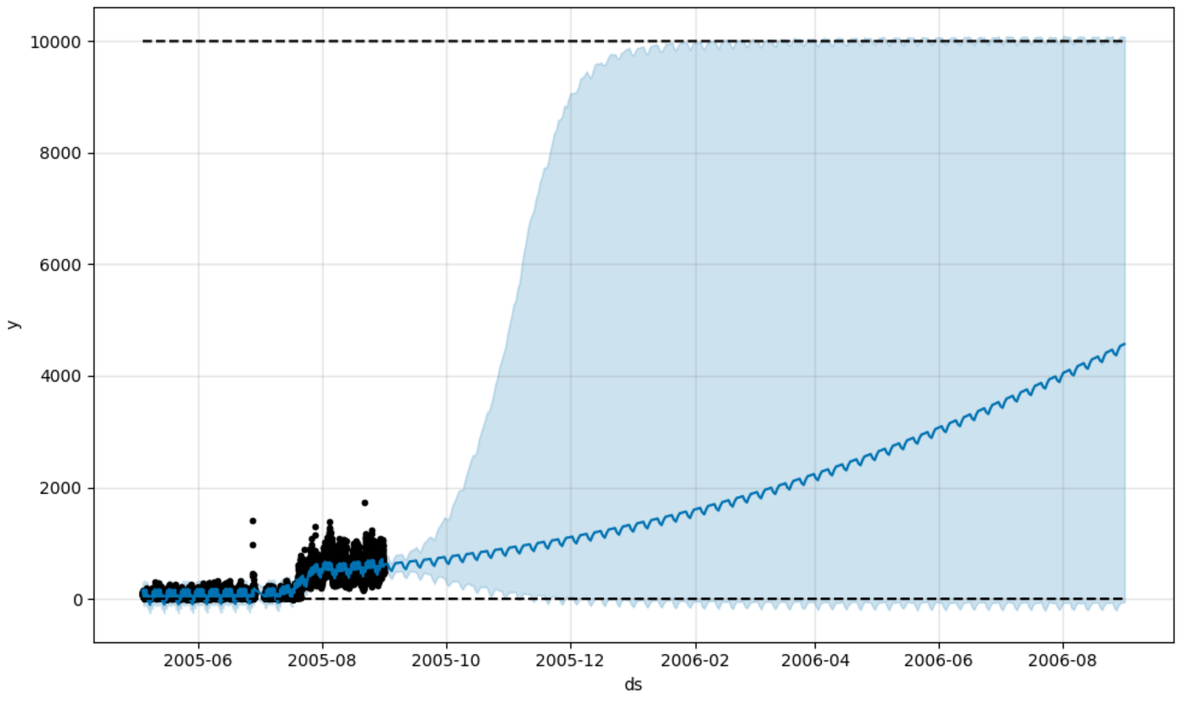


Figure 97 Example of traffic forecast for de-at link

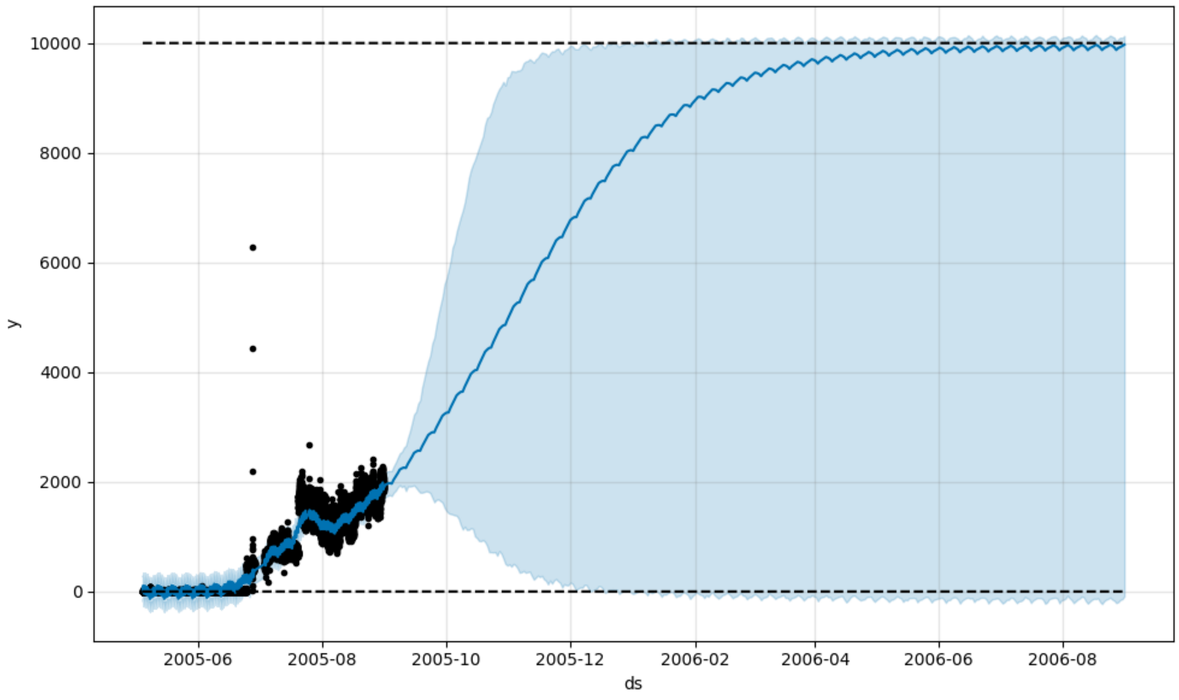


Figure 98 Example of traffic forecast for hr-at link

Figure 97 and Figure 98 provide two examples of link forecasts. It can be observed that a yearly forecast has been generated per link. Other necessary values considered have been the maximum limit for bandwidth usage in order not to forecast negative values. Figure 97 has a smoother predicted growth rate, while Figure 98 growth rate is almost exponential.

5. SDN Automation

This section provides the final design overview, interfaces, operational workflows, and performance evaluation results of the TeraFlowSDN components of T3.3, i.e., the Automation component (see Section 5.1) and the Policy Management component (see Section 5.2).

5.1. Automation (ZTP) Component

The final report on the Automation TeraFlowSDN component, also known as zero-touch provisioning (ZTP) component, focuses on:

- The new features introduced to this component since the first TeraFlowSDN release (see Section 5.1.1);
- The final design overview of the Automation component (see Section 5.1.2);
- The final set of interfaces of the Automation component (see Section 5.1.3);
- The final operational workflows of the Automation component (see Section 5.1.4);
- A performance evaluation study of key Automation RPCs (see Section 5.1.5).

5.1.1. New Features/Extensions

This section summarizes the new features added to the Automation component during the second year of the TeraFlow EU project, as part of the TeraFlow release v2. These features are added atop the TeraFlow release v1, which was announced in March 2022.

- Automated device update using the newly developed ztpUpdate RPC;
- Automated device deletion using the newly developed ztpDelete RPC;
- Additional integration tests to better support the TeraFlow uses cases.

5.1.2. Final Design

The Automation component aims to provide zero-touch device onboarding, reconfiguration, and deletion functions to the TeraFlowSDN controller and similar SDN controllers or overlay network management tools. To meet the automation design objectives, the Automation component is designed according to Figure 99.

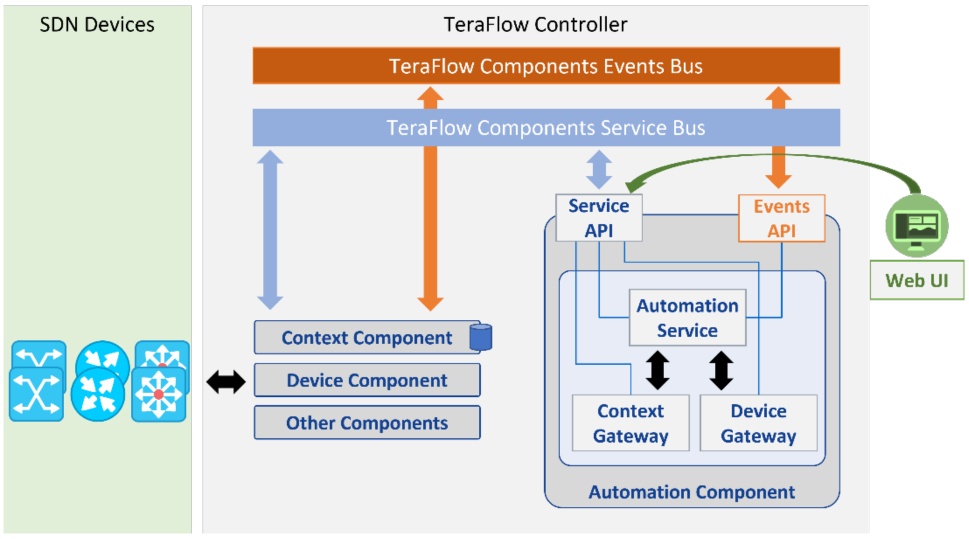


Figure 99: Overview of the final design of the TeraFlowSDN Automation component.

As shown in Figure 99, the Automation component mainly interacts with the Context and Device components. This is done via internal classes of the Automation component that implement (i) gRPC clients towards Device and Context components, acting as service consumers of external components' services and (ii) an overlay gateway layer that interacts with both services through the gRPC clients, specifically:

- The ContextGateway interface communicates with a Context Service gRPC client to invoke key RPC functions described in context.proto file;
- The ContextService interface implements the getDevice() and getDeviceEvents() methods by communicating with a Context gRPC client through the use of ContextGateway interface;
- The DeviceGateway interface communicates with a Device Service gRPC client to invoke key RPC functions described in device.proto file;
- The DeviceService interface implements the getInitialConfiguration(), configureDevice(), and deleteDevice() methods by communicating with a Device gRPC client through the use of DeviceGateway interface.

The Automation component implements an internal Automation Service (see Figure 99) of similar architecture, which consumes the Device and Context services above to offer Automation services as follows:

- The AutomationGateway interface implements all the RPCs that are described in automation.proto file. Section 5.1.3 provides details about these RPCs;
- The AutomationService interface implements the addDevice(), updateDevice(), and deleteDevice() methods by communicating with a Context gRPC client and a Device gRPC client through the use of ContextService interface and DeviceService interface respectively.

5.1.3. Final Interfaces

The Automation component offers two interfaces. The first interface, titled "Service API" in Figure 99, exposes basic automation functions to the rest of the TeraFlowSDN components. The second interface, titled "Events API" in Figure 99, allows the Automation component to register, to receive, thus react upon relevant events from key TeraFlowSDN components. Both interfaces are described in the rest of this section.

Automation Service API

Table 32 displays an overview of the RPC methods exposed by the Automation component. Specifically, the main RPC methods provide a way to automatically (i) onboard a new device (i.e., ztpAdd), (ii) reconfigure an already onboarded device (i.e., ztpUpdate), and (iii) remove an onboarded device (i.e., ztpDelete) or all the onboarded devices (i.e., ztpDeleteAll). In addition to those key functions, the Automation component also exposes two read-only RPCs that allow other TeraFlowSDN components to access the current state of device roles associated to the various devices. Specifically, the Automation component allows querying a device role using a device role ID as input (i.e., ztpGetDeviceRole) or querying a list of device roles associated with a specific device ID (i.e., ztpGetDeviceRolesByDeviceId).

Table 32: Service interface definition for the Automation component.

RPC Method Name	Parameters	Results
ztpAdd	DeviceRole	DeviceRoleState
ztpUpdate	DeviceRoleConfig	DeviceRoleState
ztpDelete	DeviceRole	DeviceRoleState
ztpDeleteAll	context.Empty	DeviceDeletionResult
ztpGetDeviceRole	DeviceRoleId	DeviceRole
ZtpGetDeviceRolesByDeviceId	context.DeviceId	DeviceRoleList

Automation Events API

Apart from the main Automation services, the Automation component exploits a publish-subscribe TeraFlowSDN mechanism to dynamically associate components with relevant events that require immediate actions. This is the role of the “Events’ API”. The Automation component relates its services with three basic events, as shown in Table 33.

Table 33: Events’ publish-subscribe interface for the Automation component.

Event Name	Triggered by	Triggers	Results

DEVICE_ADD	Context component	Automation component ztpAdd RPC	New Device object with an associated:
			DeviceStatus = ENABLED
			DeviceRoleState = ZTP_DEV_STATE_CREATED

DEVICE_UPDATE	Web UI component or external entity	Automation component ztpUpdate RPC	Updated Device object with an associated: • DeviceRoleState = ZTP_DEV_STATE_UPDATED
DEVICE_DELETE	Web UI component or external entity	Automation component ztpDelete RPC	Deleted Device object with an associated: • DeviceStatus = DISABLED • DeviceRoleState = ZTP_DEV_STATE_DELETED

5.1.4. Final Operational Workflows

This section provides detailed sequence diagrams for the most important RPCs of the Automation component, highlighting the interaction of the Automation component with other TeraFlowSDN components or external entities.

Automated device onboarding workflow

Upon issuing a request to add a new device, the operator calls the Device component's AddDevice method. This method initiates a connection with the requested device, and if successfully connected, the device driver obtains the device configuration. This configuration is turned into a DeviceConfig object and pushed to the Context database. A "DEVICE_ADD" event is generated upon success, notifying that a new device is associated with a TeraFlowSDN device driver plugin. As a result, the Context component generates a notification through the "Events API". The Automation component receives this event, thus the ztpAdd RPC is automatically triggered as shown in Figure 100. First, the Automation component requests the new Device object from the Context database, resulting in a "getDevice" call to the Device component. Then, if this device is not already configured, the Automation component requests this device's initial configuration parameters by issuing a "getInitialConfig" RPC to the Device component through the Context component. Upon receiving an updated DeviceConfig object, the Automation component loops through the configuration entries of its local object and updates the relevant entries according to the newly fetched DeviceConfig object. Next, the updated device object is pushed to the Device component and stored to the Context database via a "configureDevice" RPC. Upon success, the Automation component flips the DeviceStatus bit of the Device object to "ENABLED" and the respective ZTPDeviceState to "CREATED" while generating relevant events. Finally, these events can be consumed by other TeraFlowSDN components, e.g., to begin management and monitoring routines for this device. Note that if the DeviceStatus of the newly arrived device is already ENABLED, the device is already provisioned, thus no action is taken by the ztpAdd RPC, while a relevant warning is issued.

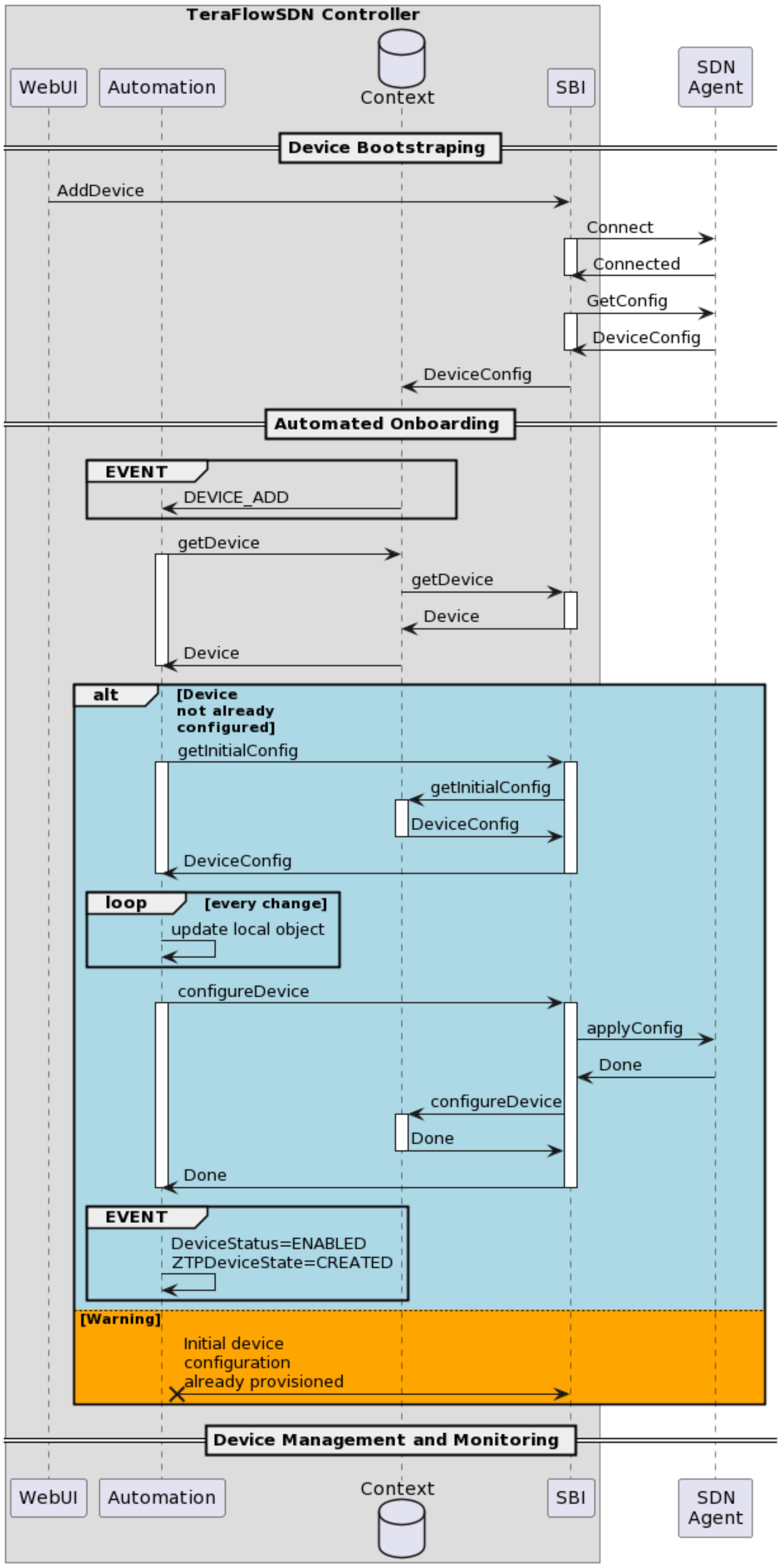


Figure 100: Zero-Touch Provisioning of a new device into TeraFlowSDN.

Automated device update workflow

Device update requests can be issued by an operator using the ztpUpdate RPC, This RPC receives a new candidate DeviceConfig object for a certain device as input, as shown in Figure 101. First, the Automation component requests the respective Device object from the Context database, resulting in a “getDevice” call to the Device component. Then, if this device is enabled, the Automation component loops through the configuration entries of the newly retrieved object and updates the relevant entries according to the newly fetched DeviceConfig object. Next, the updated Device object is pushed to the Device component and stored to the Context database via a “configureDevice” RPC. Upon success, the Automation component flips the ZTPDeviceState to “UPDATED” while generating a relevant event. Note that, if the device to be updated is not enabled, the ztpUpdate RPC outputs a relevant warning indicating an inability to update a disabled device.

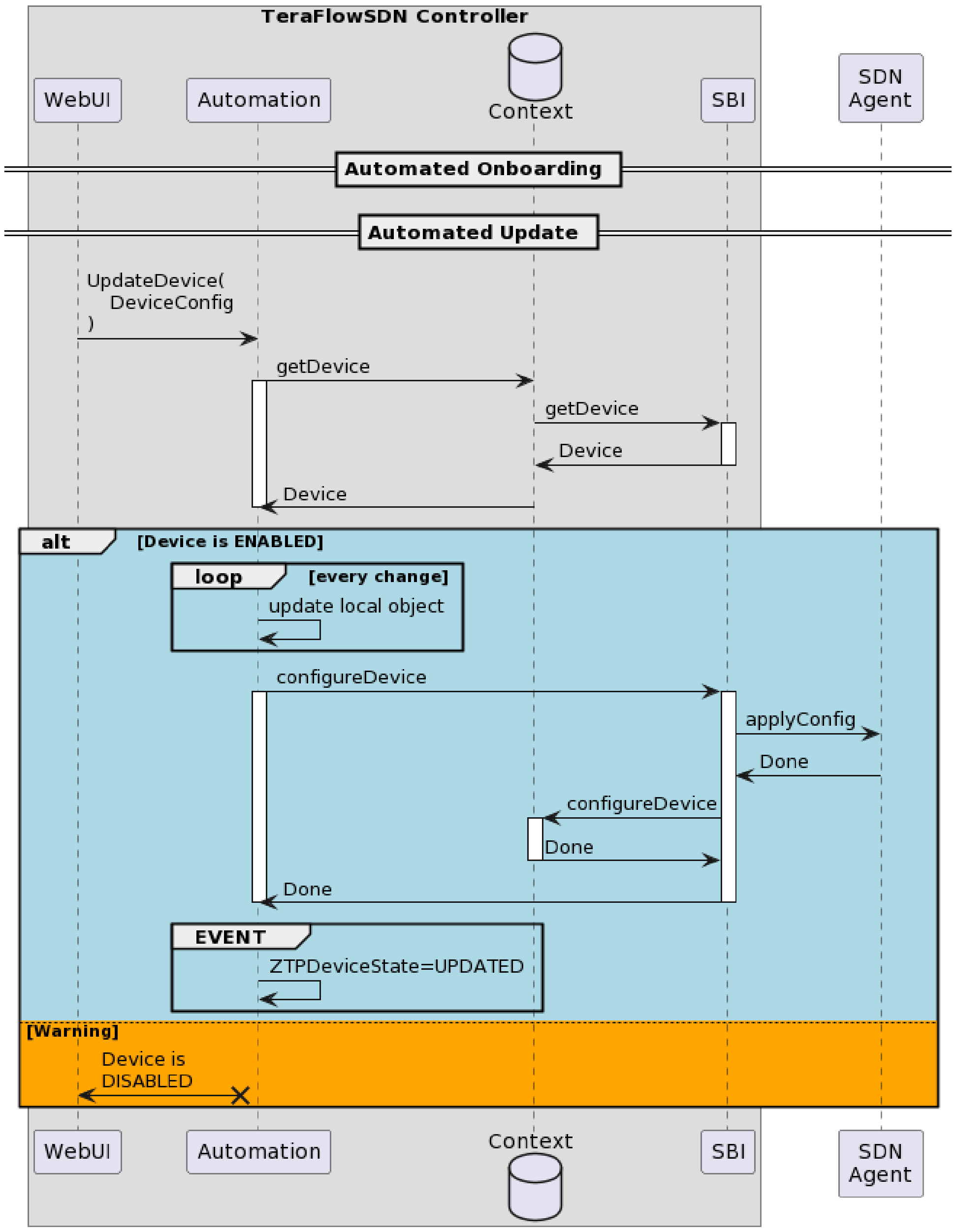


Figure 101: Zero-Touch Update of a device into TeraFlowSDN.

Automated device deletion workflow

Device delete requests can be issued by an operator using the ztpDelete RPC as shown in Figure 102. Upon such a request, the Automation component requests the respective Device object from the Context database, resulting in a “getDevice” call to the Device component. Then, if this device is enabled, the Device object is deleted from the Context database via a “deleteDevice” RPC issued to the Device and Context components in turn. Upon success, the Automation component flips the DeviceStatus bit of the Device object to “DISABLED” and the respective ZTPDeviceState to “DELETED” while generating the respective events. Note that, if the device to be deleted is not enabled, the ztpDelete RPC outputs a warning indicating an inability to delete an already disabled device.

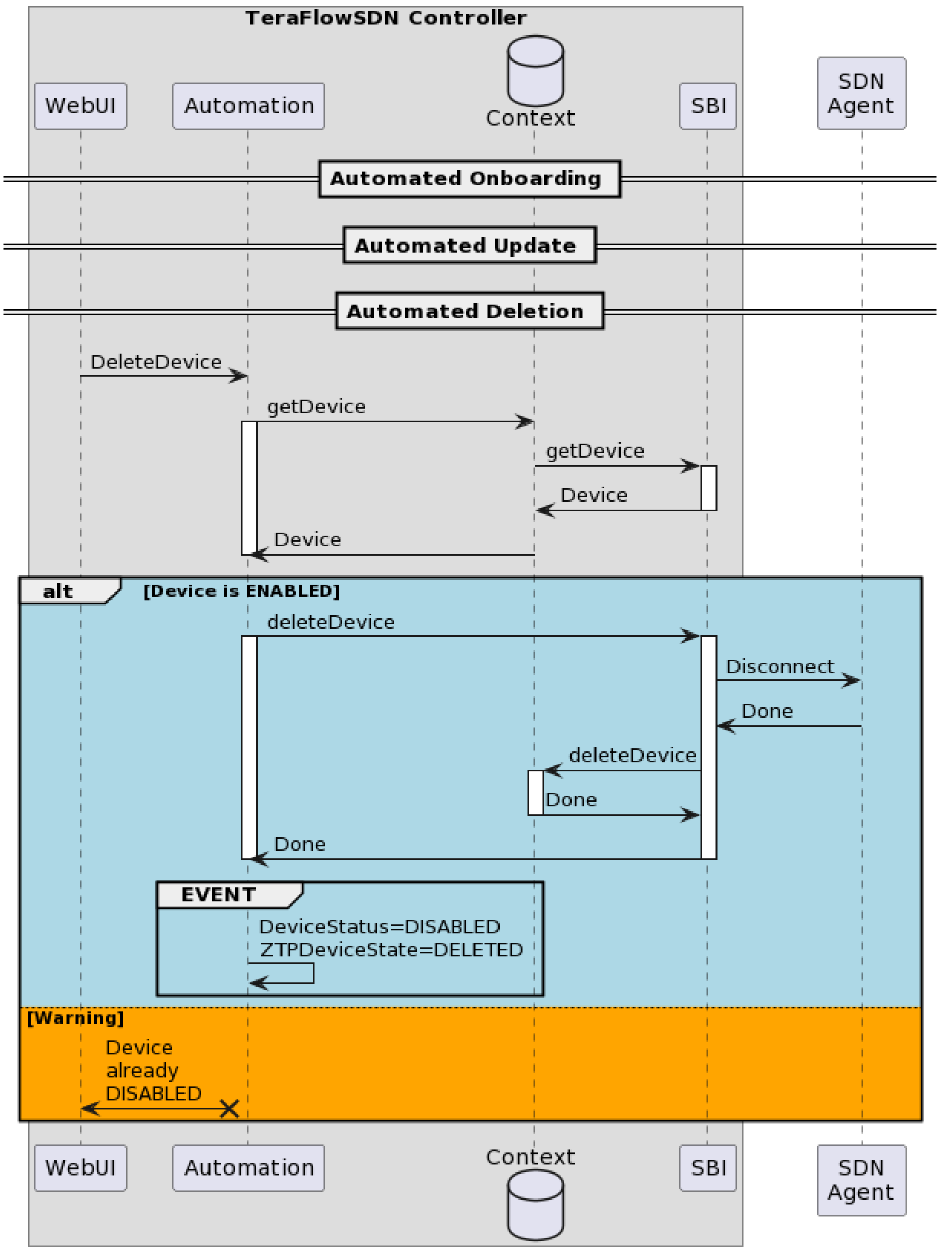


Figure 102: Zero-Touch Deletion of a device from TeraFlowSDN.

5.1.5. Evaluation

This section evaluates the three basic RPCs of the Automation component, i.e., ztpAdd, ztpUpdate, and ztpDelete, using an actual TeraFlowSDN deployment with the Automation, SBI, and Context components in action.

Experimental setup

This setup is deployed on the same testbed as in the case of the P4 service handler (see Section 4.2.5). TeraFlowSDN is deployed as a Kubernetes service on a server, and no resource limits are set to the Automation, SBI and Context components to allow stress testing.

Benchmarking procedure

To benchmark the three RPCs of the Automation component, the Grafana K6 [17] testing suite is employed. This tool allows to emulate multiple gRPC clients that in the case of the Automation component trigger the desired RPCs. K6 allows to pin each emulated client on a different thread, thus invoke concurrent calls to the Automation component for stressing its behaviour under high load. To stress test the Automation component a large number of devices is required. As this is hard to provision using real hardware, software-based devices are employed, using the emulated device driver plugin of the SBI component.

The following benchmarks measure the total time (in seconds) required to perform zero-touch add/update/delete operations using an exponentially increasing number (i.e., 1, 10, 100, 500, and 800) of emulated devices. Each ztp RPC is invoked by a client deployed on a dedicated thread to ensure concurrency. Errorbars are used to report the time in the y-axis. The central point corresponds to the median latency, while the whiskers correspond to minimum and maximum latencies respectively.

Zero-touch device addition benchmarking

In the first experiment, the Automation, SBI, and Context components are deployed in clean state and an exponentially increasing number of emulated devices is provisioned. This is done by calling the CreateDevice RPC of the SBI component, which results in a Connect RPC per device by the emulated device driver. This in turn triggers device bootstrapping, which produces an event that is captured by the Automation component. Specifically, this event automatically invokes the ztpAdd RPC, which undertakes the provisioning of an initial configuration to the underlying device as described by the ztpAdd workflow in Figure 103.

Figure 103 shows the time required for the Automation component to provision an exponentially increasing number of emulated devices. The number of devices is shown in the x-axis, while the y-axis displays the total time to realize zero-touch provisioning for this increasing number of devices. The clock ticks when the first RPC is issued by K6 and stops when the last RPC is concluded, which implies that all devices are in ZTPDeviceState=CREATED. As shown in Figure 103, provisioning configuration for a single device takes around 50ms. The median latency increases up to 12s for 800 devices. To identify the trend, we fit a function to the median latencies shown in Figure 103, which shows a polynomial increase of the latency as follows:

$$\text{Latency} = 1104x^2 - 3630x + 2699,$$

where x is the number of devices.

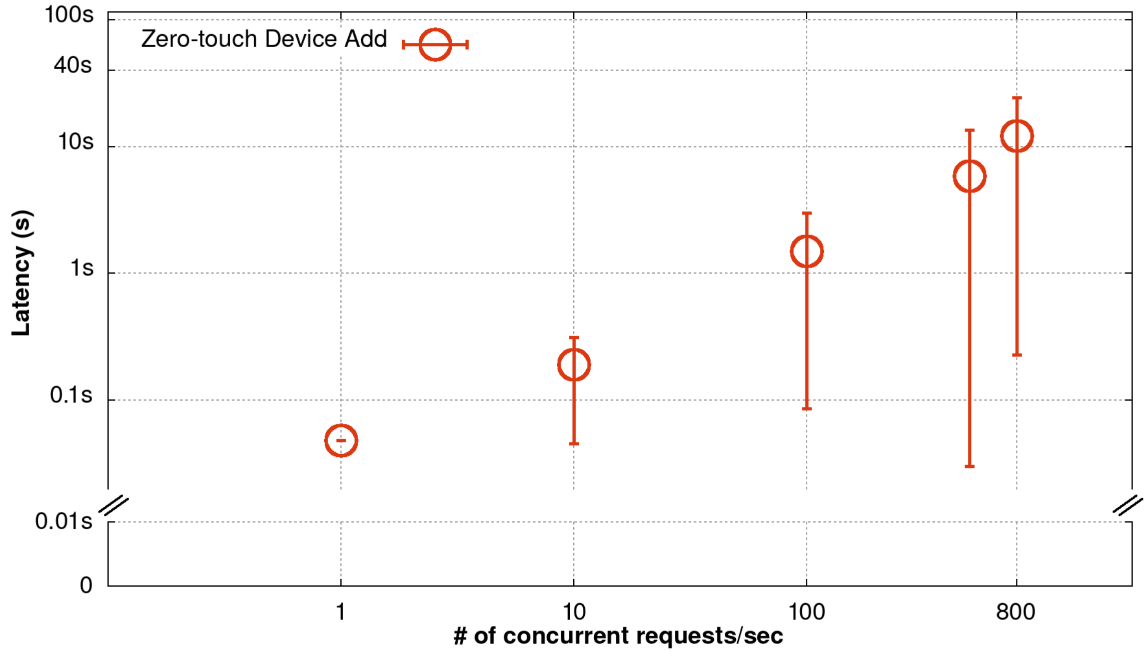


Figure 103: Zero-Touch add benchmark using an exponentially increasing number of emulated devices.

Zero-touch device update benchmarking

In this experiment, the same number of devices is considered. This device must be already provisioned to update a device through the ztpUpdate RPC of the Automation component. For this reason, this experiment considers the devices pre-deployed and pre-configured. The objective is to update the existing configuration of each device using the automated workflow of the ztpUpdate as described in Figure 104. The clock

Figure 104 shows the time required for the Automation component to update an exponentially increasing number of emulated devices. The number of devices is shown in the x-axis, while the y-axis displays the total time to realize zero-touch update for this increasing number of devices. The clock ticks when the first RPC is issued by K6 and stops when the last RPC is concluded, which implies that all devices are in ZTPDeviceState=UPDATED. As shown in Figure 104, updating the configuration for a single device takes around 37ms. The median latency increases up to 14.5s for 800 devices, which makes this RPC slightly heavier than the ztpAdd. To identify the trend, we fit a function to the median latencies shown in Figure 104, which shows a polynomial increase of the latency as follows:

$$\text{Latency} = 1206x^2 - 3415x + 2102,$$

where x is the number of devices.

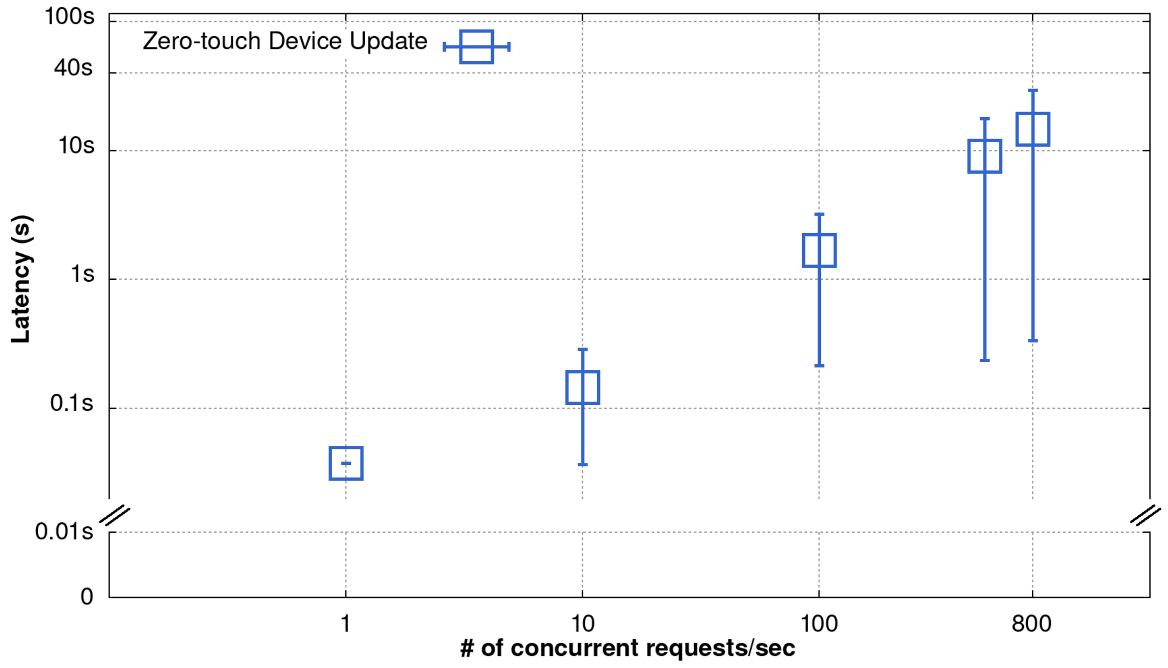


Figure 104: Zero-Touch update benchmark using an exponentially increasing number of emulated devices.

Zero-touch device deletion benchmarking

This final experiment follows a similar set up with the update benchmark. An increasing number of devices is pre-deployed and pre-configured, hence the goal of this experiment is to decommission these devices using the ztpDelete RPC automatically.

Figure 105 shows the time required for the Automation component to delete an exponentially increasing number of emulated devices. The number of devices is shown in the x-axis, while the y-axis displays the total time to realize zero-touch deletion for this increasing number of devices. As shown in Figure 105, decommissioning a single device takes around 46ms. The median latency increases up to 15s for 800 devices, which is the highest compared to ztpAdd and ztpUpdate. This is because deleting a device also TBD. To identify the trend, we fit a function to the median latencies shown in Figure 105, which shows a polynomial increase of the latency as follows:

$$\text{Latency} = 1217x^2 - 3223x + 1830,$$

where x is the number of devices.

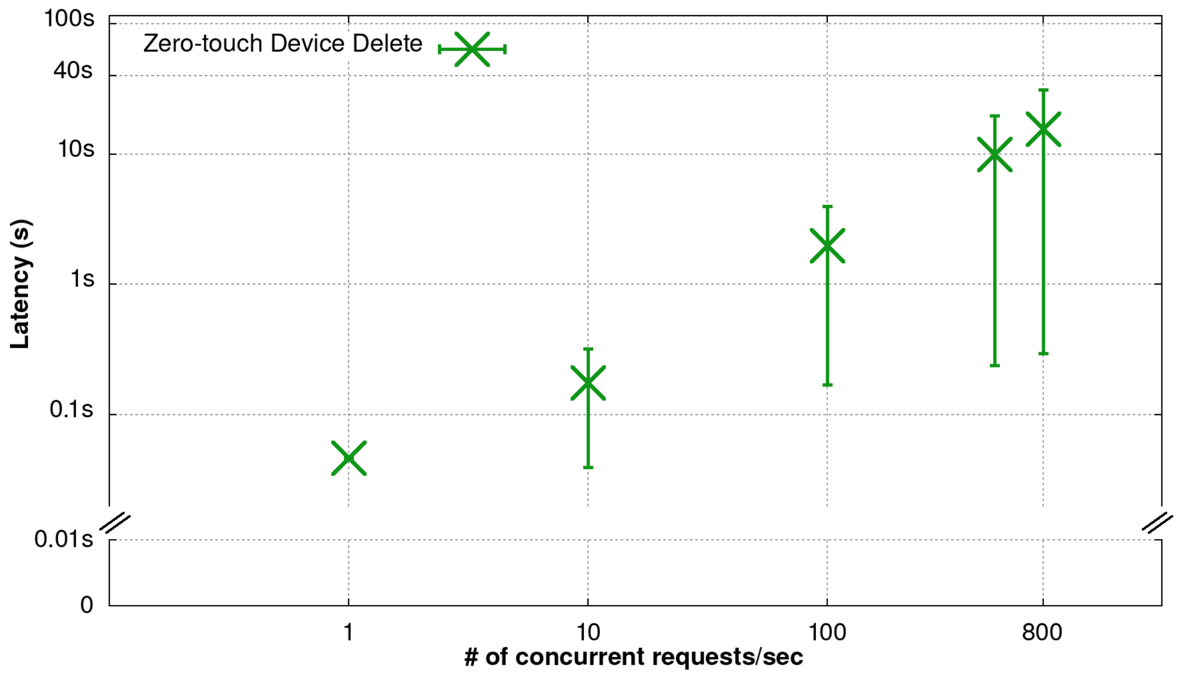


Figure 105: Zero-Touch deletion benchmark using an exponentially increasing number of emulated devices.

Summary

Despite the exponential increase of the added/updated/deleted devices, the service time of the Automation RPCs is still polynomial. Note that to perform these RPCs, the Automation component interacts with two other TeraFlowSDN components (SBI and Context), thus this is an integration-level benchmark. This demonstrates that TeraFlowSDN components are carefully designed to accommodate high loads.

5.2. Policy Management Component

Network policy may be defined as a collection of rules that dictate the behaviours of network resources, which may include devices (physical or virtual) and functional components. Within the TeraFlow project, we use policy rules for both:

- high-level objectives, which are often referred to as "intent" statements due to the declarative nature of the request;
- low-level objectives, applied to specific devices or when making network resource assignment decisions. With often use imperative statements when processing "network policy".

The TeraFlow Policy Management component uses "event-driven management" [16]; this approach provides a valuable method to monitor state change of managed objects and resources and enable automatic triggering of responses to events based on an established set of rules. The TeraFlow event-driven policy provides rapid autonomic responses to specific conditions, enabling self-management behaviours, such as self-configuration, self-healing, self-optimization, and self-protection. The TeraFlow Policy Management Component utilises an emerging technical technique called "Event Condition Action" (ECA) to provide event-driven benefits [16]. ECA Policy enables actions to be automatically triggered based on when certain events in the network occur while certain conditions hold. Thus, ECA facilitates limited logic to be delegated to network devices and functional components for automating specific required behaviour.

5.2.1. New Features/Extensions

This section summarizes the new features added to the Policy Management component during the second year of the TeraFlow EU project, as part of the TeraFlow release v2. These features are added atop the TeraFlow release v1, which was announced in March 2022.

- Drastically revised ECA-based policy model (as compared to TeraFlow release v1) with a basic policy rule object that specializes to (i) service-oriented policy rules (network-wide policies spanning across a number of devices that a service might traverse) and (ii) device-specific policy rules (device-level policies);
    - These changes introduced additional service-level and device-level policy RPCs. Section 5.2.3 provides further details about all Policy Management RPCs.
- Implementation of all eight (8) Policy Management RPCs. In the TeraFlow release v1, only a draft Policy Management protobuf was implemented, with a “skeleton” Policy Management component. In this second TeraFlow release, a complete Policy Management implementation is provided;
- Unit and integration tests to support the TeraFlow uses cases.

5.2.2. Final Design

The goal of the Policy Management component, also abbreviated as Policy component, is to translate a network operator’s high-level network policy statements into a correct set of low-level instructions that realize this policy across the various network elements. To meet this objective, the Policy component is designed according to Figure 106.

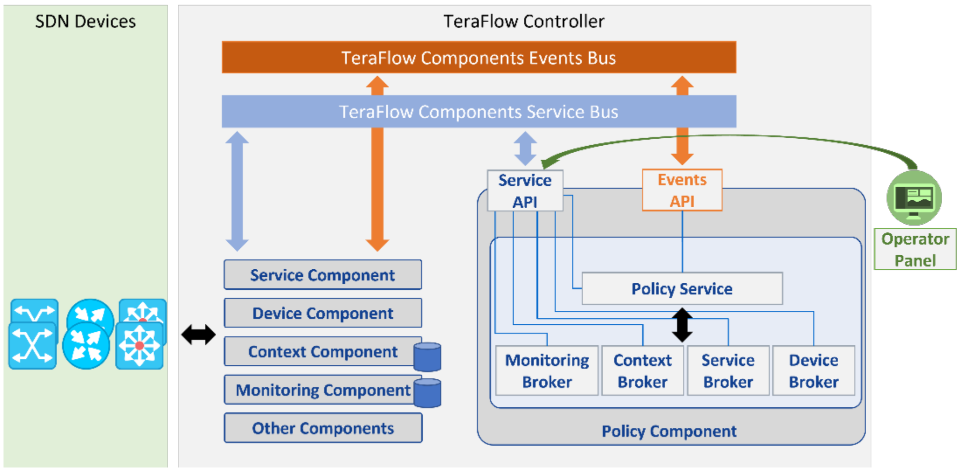


Figure 106: Overview of the final design of the TeraFlowSDN Policy component.

As shown in Figure 106, the Policy Management component mainly interacts with the Context, Monitoring, Device, and Service components. This is done via internal classes of the Policy Management component that implement (i) gRPC clients towards the Context, Monitoring, Device, and Service components, acting as service consumers of external components’ services and (ii) an overlay gateway layer that interacts with all these services through the gRPC clients. Specifically:

- The ContextGateway interface communicates with a Context Service gRPC client to invoke key RPC functions described in context.proto file;

- The ContextService interface implements the GetService(), GetDevice(), GetPolicyRule(), SetPolicyRule(), and DeletePolicyRule() methods by communicating with a Context gRPC client through the use of ContextGateway interface;
- The MonitoringGateway interface that communicates with a Monitoring service gRPC client to invoke key RPC functions described in monitoring.proto file;
- The MonitoringService interface that implements the MonitorKpi(), SetKpiAlarm(), and GetAlarmResponseStream() methods by communicating with a Monitoring gRPC client through the use of the MonitoringGateway interface;
- The DeviceGateway interface communicates with a Device Service gRPC client to invoke key RPC functions described in device.proto file;
- The DeviceService interface implements the configureDevice() method by communicating with a Device gRPC client through the use of DeviceGateway interface;
- The ServiceGateway interface that communicates with a Service service gRPC client to invoke key RPC functions described in service.proto file;
- The ServiceService interface implements the UpdateService() method by communicating with a Service gRPC client through the use of the ServiceGateway interface.

The Policy Management component implements an internal Policy Service (see Figure 106) of similar architecture, which consumes the Context, Monitoring, Device, and Service services above to offer Policy Management services as follows:

- The PolicyGateway interface implements all the RPC functions that are described in policy.proto file. Section 5.2.3 provides details about these RPCs;
- The PolicyService interface that implements the Policy RPC methods by communicating with a Monitoring gRPC client, a Context gRPC client, a Service gRPC client, and a Device gRPC client through the MonitoringService, ContextService, ServiceService, and DeviceService interfaces, respectively.

<u>Basic policy rule format</u>

A basic policy rule contains the following set of information, as introduced in the policy.proto file and shown in Table 34.

Table 34: Key elements of a basic policy rule object.

Basic policy rule element	Description
Policy rule ID	A unique policy rule ID in string format.
Policy rule state	The state of this policy rule in the policy component's internal state machine: • POLICY_UNDEFINED • POLICY_FAILED • POLICY_INSERTED • POLICY_VALIDATED • POLICY_PROVISIONED • POLICY_ACTIVE • POLICY_ENFORCED • POLICY_INEFFECTIVE • POLICY_EFFECTIVE • POLICY_UPDATED • POLICY_REMOVED

Policy rule priority	The priority of a policy rule encoded as a non-negative integer. The lower the number the higher the priority of the rule.
List of policy rule conditions	A list of policy rule conditions encoded in the policy_condition.proto. Each condition is a triplete of: • Monitoring.KPI ID • Numerical Operator (=, ≠, <, ≤, >, ≥) • Monitoring. KPI value
Boolean operator for policy rule conditions	Boolean operator between multiple policy rule conditions. Supported operators are AND or OR.
List of policy rule actions	A list of policy rule actions encoded in the policy_action.proto. Each action is a tuple of (i) a predefined set of action operations followed by (ii) a corresponding list of action parameters. For example, service-related actions act upon a service's configuration rules or constraints, while device-related actions act upon device parameters, such as device status, port(s) status, etc.

Service and device-level policy rules

This basic policy rule is inherited by two other types of policy rules as follows:

- PolicyRuleService inherits all attributes of a basic policy rule associated with a particular service ID and optionally, a list of devices through which the service traverses. If no devices are provided, all devices being traversed by the service are assumed;
- PolicyRuleDevice inherits all attributes of a basic policy rule associated with a list of devices on which the policy rule focuses.

Core logic: The Policy service implements the internal finite state machine depicted in Figure 107. Operators can add a new policy to the TeraFlowSDN ecosystem by calling the policyAdd RPC (either for service-based policies or for device-level policies) as shown on the top left part in Figure 107. At this stage, the state of a newly arriving policy is marked as INSERTED and the input policy is stored in the Context database through a "SetPolicyRule" RPC. Upon successfully parsing and validating the content of the requested policy, the input policy transitions to the VALIDATED state. In the case that the validation process fails, e.g., due to an unsupported policy condition (i.e., KPI) or an invalid service/device ID being provided, the state machine transitions to the FAILED state, which results in a corresponding failure of the policyAdd RPC.

Next, an already validated policy rule needs to be provisioned through an interaction with the Monitoring component. Specifically, an input policy rule contains a list of policy rule conditions in the form of AND-separated or OR-separated "KPI_ID NUMERICAL_OPERATOR KPI_Value" patterns. For instance, a policy rule condition for a certain service could be "PacketLoss > 0.2". Such a condition instructs the Policy component to (i) Monitor each KPI in a list of KPIs, (ii) set a new KPI alarm from a list of KPIs, and (iii) get a stream of KPI alarm responses, when a certain registered KPI alarm is triggered. With this set of RPCs issued to the Monitoring component, a policy rule can be successfully subscribed for conditional alarms (or events), thus transition the state machine to the PROVISIONED state (see Figure 107).

Once a set of KPIs are registered to the Monitoring component, the Policy component waits for potential alarms to be triggered. Once such an alarm is thrown, a corresponding policy rule transitions to the ACTIVE state. This is the time for the Policy component to apply the (list of) action(s) associated with the policy, for which the alarm is thrown. Upon the enforcement of the action(s), which can be done either at the level of a service or a (set of) device(s), the policy rule transitions to the ENFORCED state as shown in Figure 107. At this state, it is critical for the network operator to know whether the applied action(s) has been effective or not. To do so, the Policy component offers an internal policy assessment loop that monitors each policy rule's KPIs for a configurable period after the enforcement of the policy actions. If the monitored KPIs keep on reporting alarms past the enforcement of the policy actions, the policy rule transitions to the INEFFECTIVE state. Other relevant TeraFlowSDN components, such as the WebUI, could register for policy events with such a state, to notify the network operators for policy rules that require attention. In this case, potential remedy actions could be triggered by network operators, by updating ineffective policies (e.g., with more effective actions) or deleting ineffective policy rules as shown in Figure 107. In the case of no further alarms triggered after the enforcement of policy actions, the policy state is marked as EFFECTIVE.

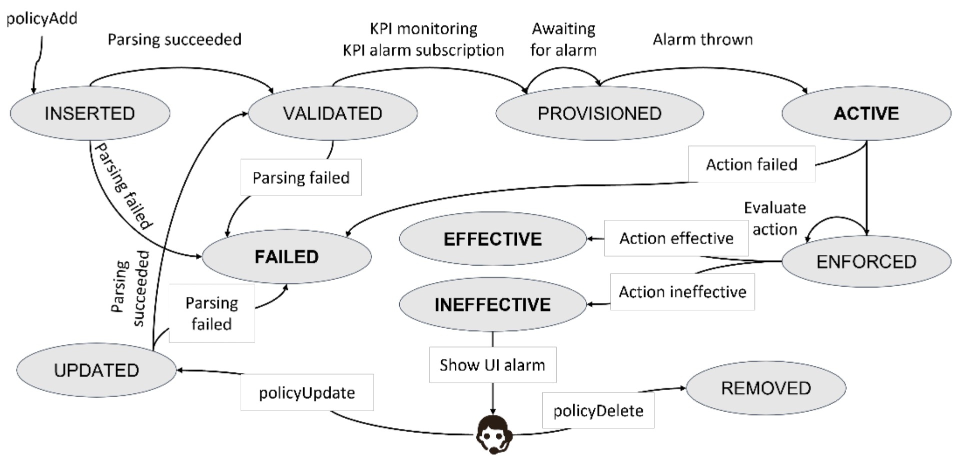


Figure 107: Internal state machine of the TeraFlowSDN Policy component.

5.2.3. Final Interfaces

The Policy Management component offers two interfaces. The first interface, titled "Service API" in Figure 106, exposes basic policy management functions to the rest of the TeraFlowSDN components. The second interface, titled "Events API" in Figure 106 allows the Policy Management component to register, to receive, thus react upon relevant events from key TeraFlowSDN components. Both interfaces are described in the rest of this section.

Policy Management Service API

Table 35 displays an overview of the RPC methods exposed by the Policy Management component. Specifically, the main RPC methods provide a way to (i) add a new policy, either at the service level (i.e., policyAddService) or at the device level (i.e., policyAddDevice), (ii) update an already provisioned policy, either at the service level (i.e., policyUpdateService) or at the device level (i.e., policyUpdateDevice), and (iii) remove a provisioned (i.e., policyDelete). In addition to those key functions, the Policy Management component also exposes three read-only RPCs that allow other TeraFlowSDN components to access the current state of policies. Specifically, the Policy Management component allows querying for service-level (GetPolicyService) and device-level (GetPolicyDevice) policies by their ID or querying the list of policies associated with a specific service ID (GetPolicyByServiceId).

Table 35: Service interface definition for the Policy Management component.

RPC Method Name	Parameters	Results
PolicyAddService	PolicyRuleService	PolicyRuleState
PoicyAddDevice	PolicyRuleDevice	PolicyRuleState
PolicyUpdateService	PolicyRuleService	PolicyRuleState
PoicyUpdateDevice	PolicyRuleDevice	PolicyRuleState
PoicyDelete	PolicyRuleId	PolicyRuleState
GetPolicyService GetPolicyDevice GetPolicyByServiceId	PolicyRuleId PolicyRuleId context.ServiceId	PolicyRuleService PolicyRuleDevice PolicyRuleServiceList

Policy Management Events API

Apart from the main Policy Management services, the Policy component exploits a publish-subscribe TeraFlowSDN mechanism to dynamically associate policy conditions with relevant events that require immediate actions. Integrating the Policy component with the Monitoring component has made it possible to associate policy conditions (i.e., statements on the value of KPIs monitored by the Monitoring component) with alarms using the alarm subsystem of the Monitoring component. When a registered KPI value exceeds some predefined thresholds, the Monitoring component uses the Events API channel of the TeraFlowSDN controller to notify subscribed components (e.g., the Policy Management component) about the exceeded KPI threshold(s). This way, the Policy Management component uses an asynchronous mechanism to trigger policy actions upon the received alarms. In a similar way, when a policy rule transitions to a new state (e.g., ACTIVE, EFFECTIVE/INEFFECTIVE, etc.) other TeraFlowSDN component could subscribe to the Context component, waiting for such events to be raised, thus inform the network operator through the WebUI in a complete dynamic manner.

5.2.4. Final Operational Workflows

In this section, a detailed sequence diagram is provided for the most important RPCs of the Policy Managment component. These diagrams highlight both the interaction of the Policy Management component with other TeraFlowSDN components or external entities, as well as the prominent effect of the ECA model in the design of the Policy Management component.

Service-level policy creation workflow

To apply a new service-level policy to the network, a network operator needs to trigger the policyAddService RPC of the Policy Management component as shown in Figure 108. This can be done via the Slice component, when a new service is created, and the network operator wishes to associate this service with a new policy. The policyAddService call will provide the Policy Management component with a Policy rule object which contains several internal objects denoting the service associated with the policy rule, a set of conditions for this rule to apply, and a set of actions to be enforced once the condition(s) is(are) met. First, the Policy Management component parses the received policy and validates that the policy rule object refers to a valid service ID and a valid set of KPIs and actions. Then, the policy rule is stored to the Context database and the policy rule is marked as VALIDATED. At this point, Policy returns a successful response to the Slice component (which cascades back to the WebUI and OSS/BSS), while internally the Policy component begins the provisioning of the policy rule as follows.

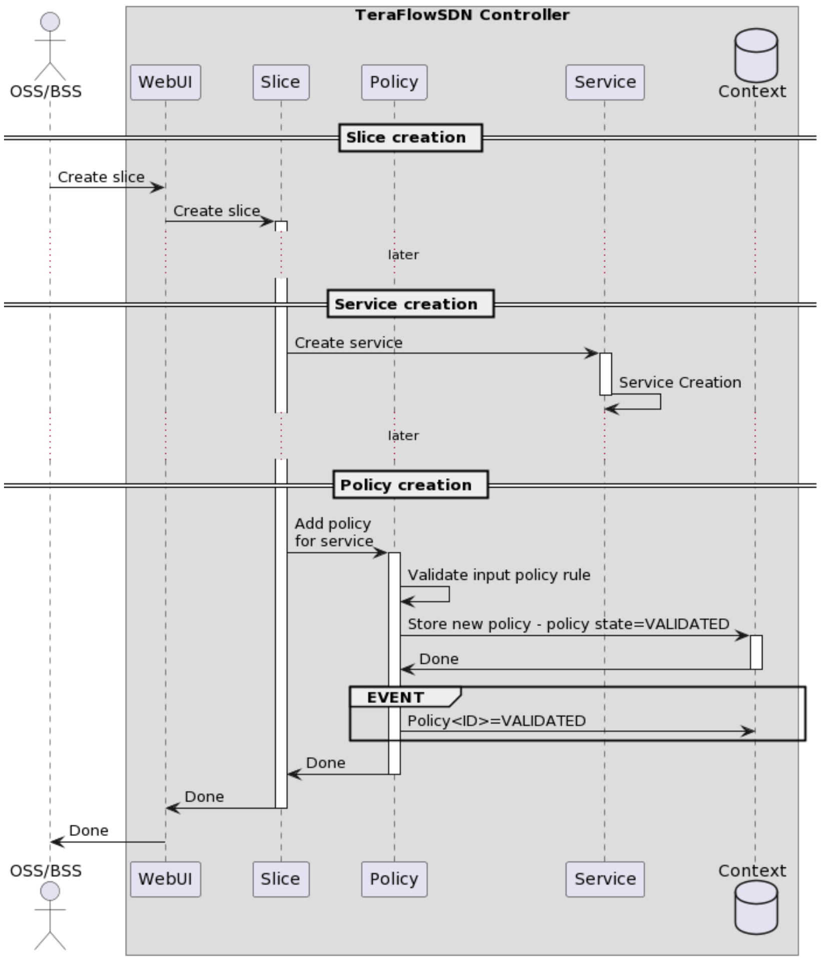


Figure 108: Service-level policy creation through the TeraFlowSDN policyAddService RPC.

Service-level policy subscription workflow

Next, the input policy rule conditions are parsed to identify which KPIs need to be requested from the Monitoring component as shown in Figure 109. This entails the (i) SetKPI RPC to create a new KPI in case it does not already exist, (ii) MonitorKPI RPC to instruct the Monitoring component to begin retrieving data for this KPI, (iii) SetKPIAlarm RPC to register to events when the KPI exceeds some range of values or specific threshold, and (iv) GetAlarmResponse Stream RPC to receive the generated alarms when the KPI condition will be met. Once all these RPCs succeed, the policy rule transitions to the PROVISIONED state.

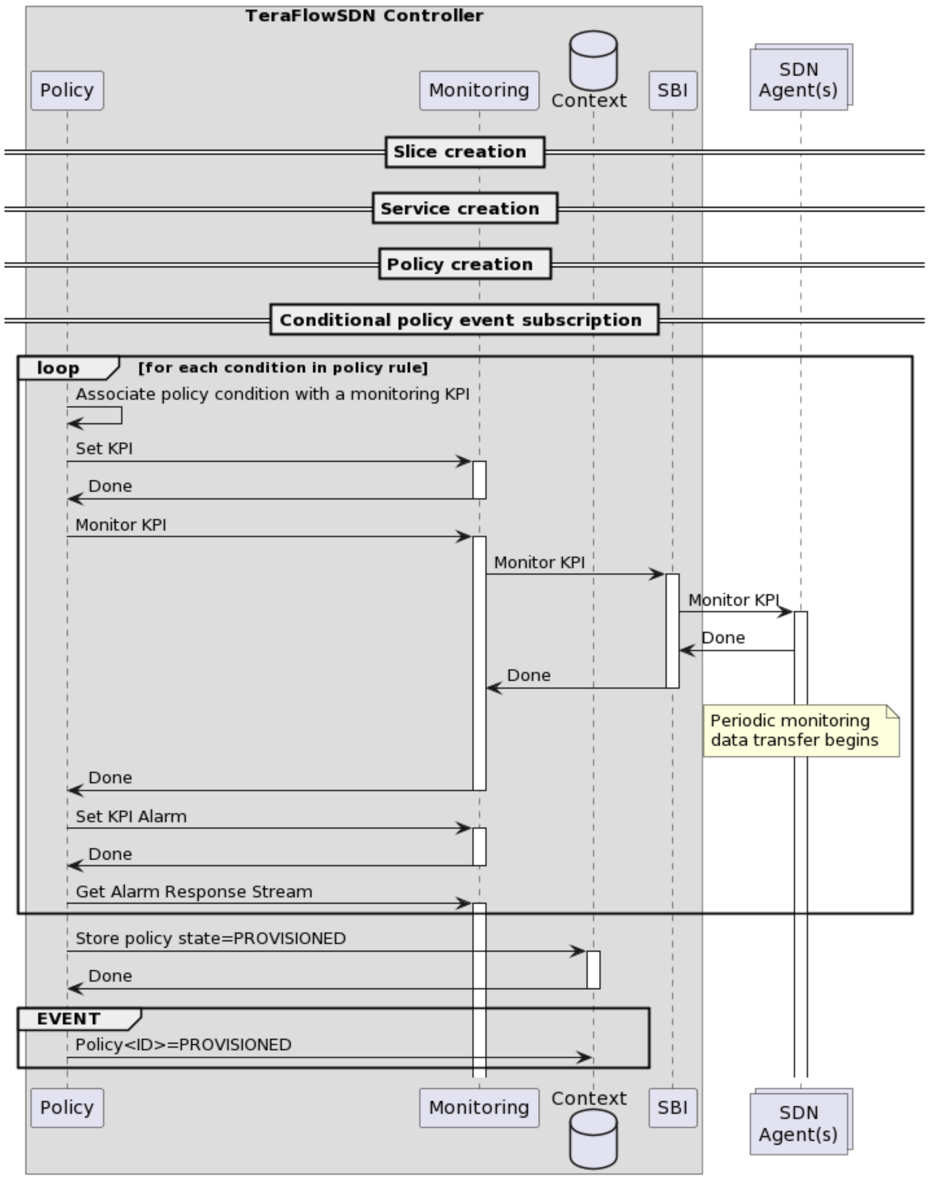

TeraFlowSDN Controller
Policy
Monitoring
Context
SBI
SDN Agent(s)
Slice creation
Service creation
Policy creation
Conditional policy event subscription
loop [for each condition in policy rule]
Associate policy condition with a monitoring KPI
Set KPI
Done
Monitor KPI
Monitor KPI
Monitor KPI
Done
Done
Periodic monitoring data transfer begins
Done
Set KPI Alarm
Done
Get Alarm Response Stream
Store policy state=PROVISIONED
Done
EVENT
Policy<ID>=PROVISIONED

Figure 109: Service-level policy subscription.

Service-level policy triggering and enforcement workflow

At a later stage, an asynchronous event will be generated when the KPI meets the requested condition(s) as shown in Figure 110. This event further transitions the policy rule to the ACTIVE state, as the Policy Management component is now ready to apply the corresponding policy actions. To do so, the affected service is first retrieved from the Context component and its local configuration is updated by applying the list of policy actions. To enforce the updates, the UpdateService RPC is called, resulting in Service component interactions with the various underlying devices through the Device component. At this point, the policy rule transitions to the ENFORCED state.

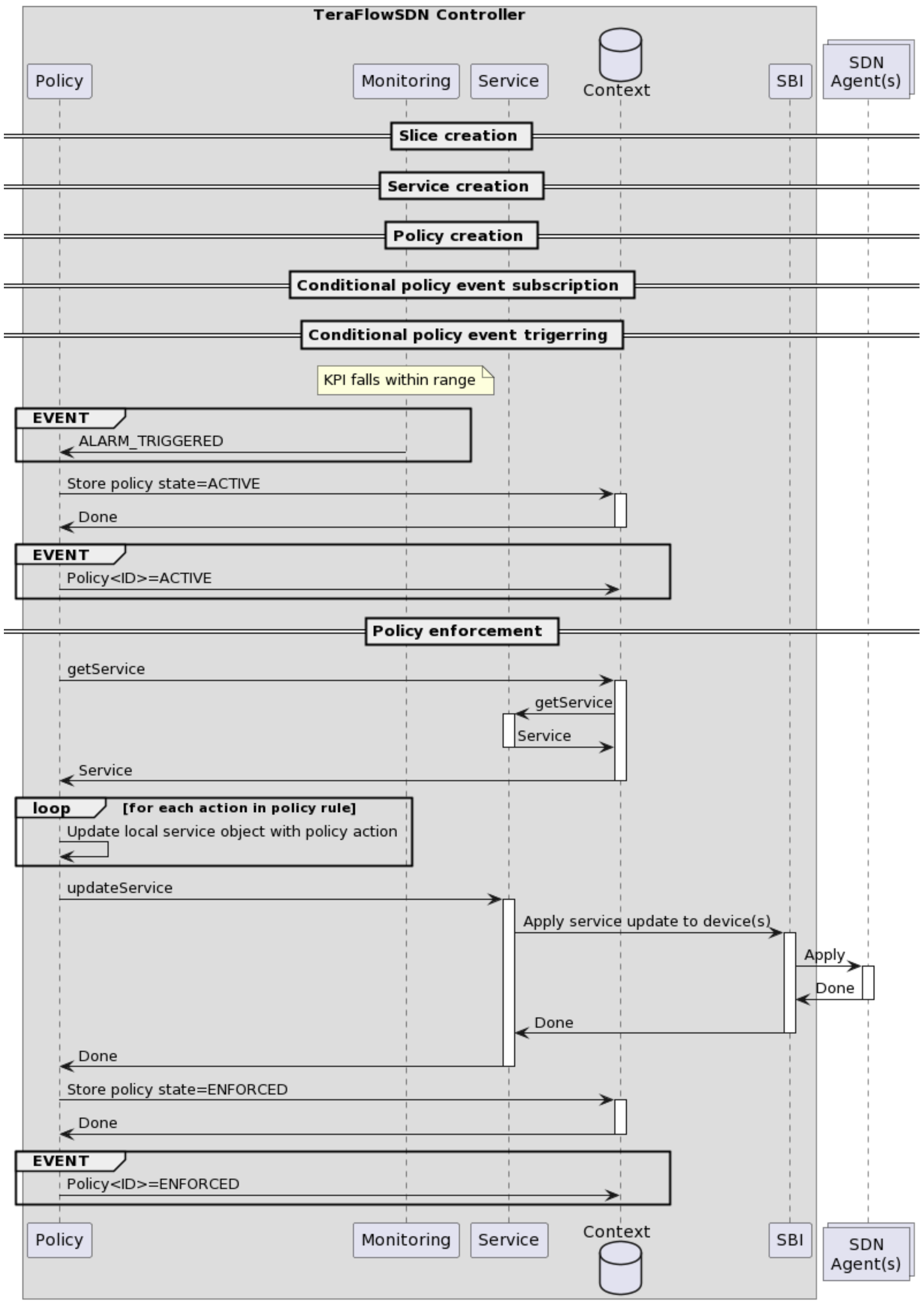


Figure 110: Service-level policy triggering and enforcement.

Device-level policy creation workflow

To apply a new device-level policy to the network, a network operator needs to trigger the policyAddDevice RPC of the Policy Management component as shown in Figure 111. The key difference between device-level and service-level policy creation is the caller. The device-level policy creation is triggered directly by the operator (or a UI), while the Slice component calls the service-level policy creation after the creation of a service for the target policy. Otherwise, Figure 110 and Figure 111 present an identical workflow for creating and validating service-level and device-level policies.

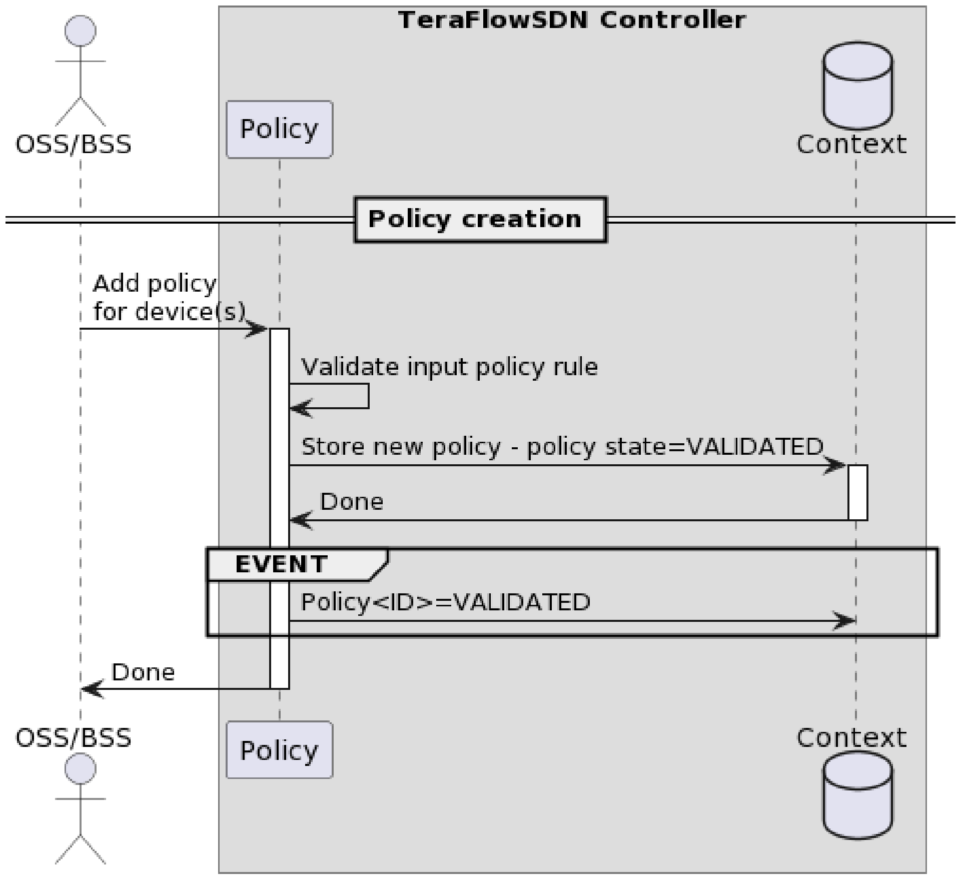


Figure 111: Device-level policy creation.

Device-level policy subscription workflow

To subscribe a device-level policy to the Monitoring component, a similar workflow is followed as in the case of the service-level policy subscription shown in Figure 112.

Device-level policy triggering and enforcement workflow

Device-level policies are enforced differently than service-level policies as shown in Figure 112. Specifically, the device-level policy enforcement method loops across a list of devices, fetches each device from the Context component, and applies a (set of) action(s) to the local device object. Once this is done, the updated device configuration is communicated to the Device component through a configureDevice RPC and the policy is considered as ENFORCED.

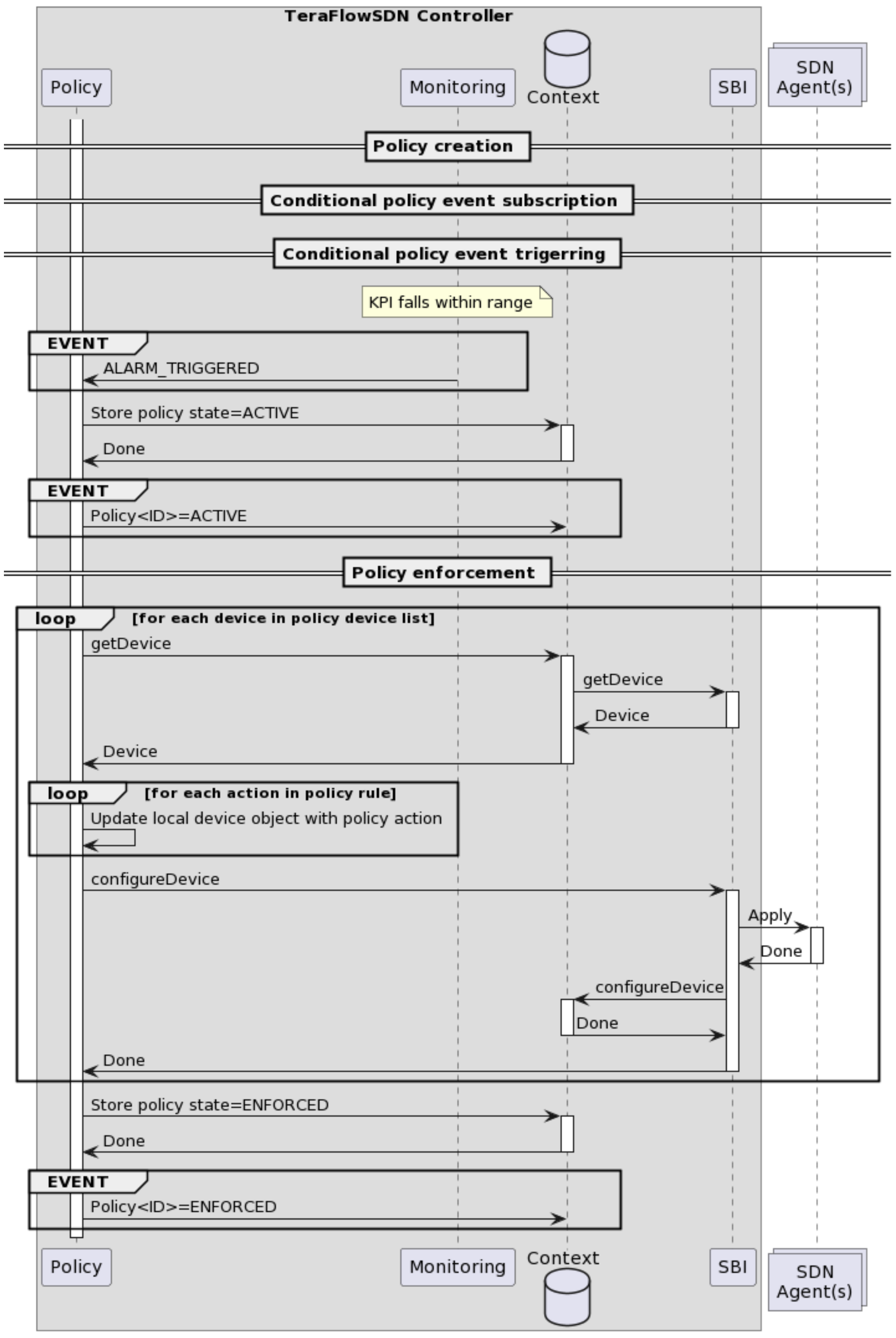


Figure 112: Device-level policy triggering and enforcement.

Policy assessment workflow

Once a service-level or a device-level policy is enforced as shown in Figure X and Figure Y, respectively, an internal policy mechanism is invoked to assess whether the enforced action was effective or not (see Figure 113). In the case that no alarms are raised after the enforcement of the policy action, the policy rule is marked as EFFECTIVE. Otherwise, the policy rule is marked as INEFFECTIVE and its state is stored in the Context database so that other components could register for alarms when such a policy state is stored in the database. This gives TeraFlowSDN an efficient way of asynchronously informing the network operator immediately after a policy rule state change is flipped. Ineffective policies will cause WebUI alerts that an operator can immediately observe and act upon, either through a policy update or policy delete operation as follows.

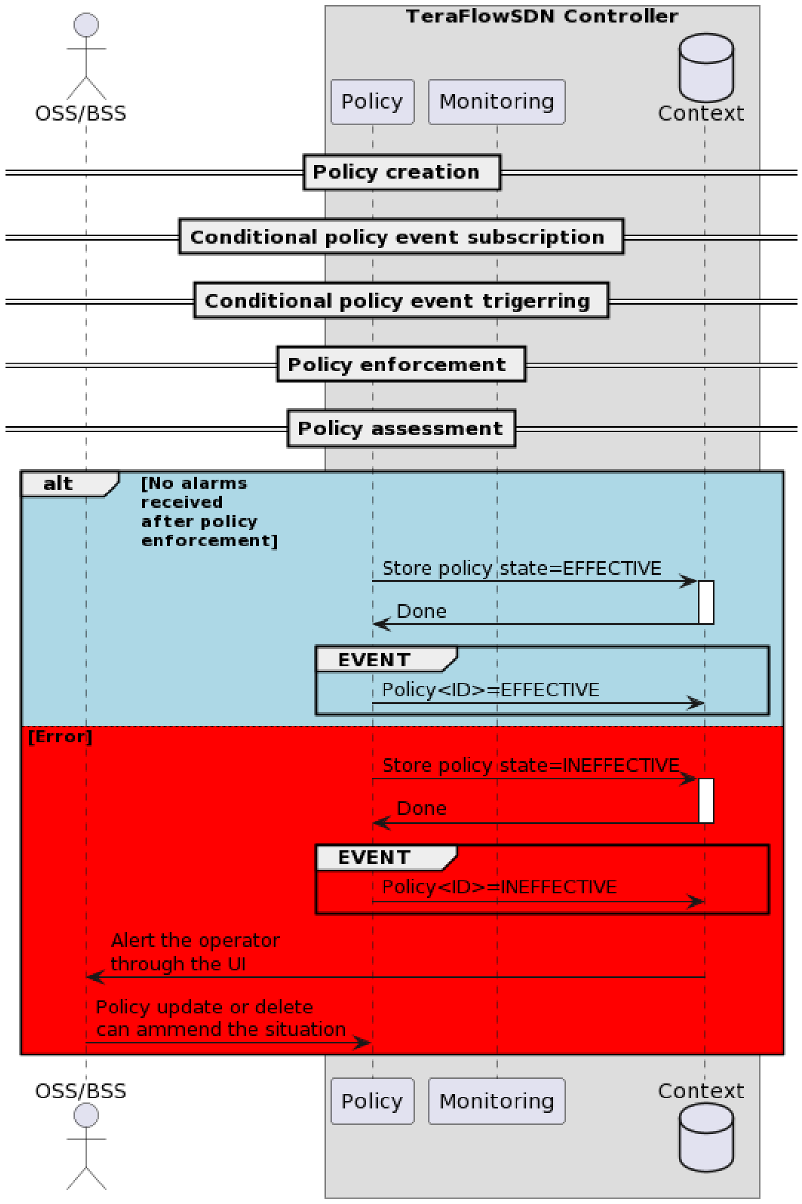

TeraFlowSDN Controller
OSS/BSS
Policy
Monitoring
Context
Policy creation
Conditional policy event subscription
Conditional policy event trigerring
Policy enforcement
Policy assessment
alt
[No alarms received after policy enforcement]
Store policy state=EFFECTIVE
Done
EVENT
Policy<ID>=EFFECTIVE
[Error]
Store policy state=INEFFECTIVE
Done
EVENT
Policy<ID>=INEFFECTIVE
Alert the operator through the UI
Policy update or delete can ammend the situation

Figure 113: Policy assessment.

Generic policy update workflow (Both for service-level and device-level policies)

To update a policy, the policyUpdateService or policyUpdateDevice RPCs are similarly called by the network operator. As in the case of the policyAdd RPCs, first the validation of the updated policy rule is required to return a status to the caller. Then, the update concerns either the conditional part of the policy or the action part of the policy, or even both. For each condition in the updated policy rule, if the corresponding KPI is not already registered by the previous policyAdd RPC, the same set of RPCs are issued to the Monitoring component (I.e., SetKPI, MonitorKPI, SetKPIAlarm, and GetAlarmResponseStream). Otherwise, the values of already registered KPIs are checked and if a KPI has updated values, a SetKPIAlarm is re-invoked to update the KPI thresholds. The same process is repeated for the policy actions to update the list of actions according to the content of the update policy request. At this point the policy transitions to the UPDATED state and an alarm is expected soon after the (updated) policy KPIs trigger an alarm. In this case, the exact same workflow is followed as in the policyAddService RPC.

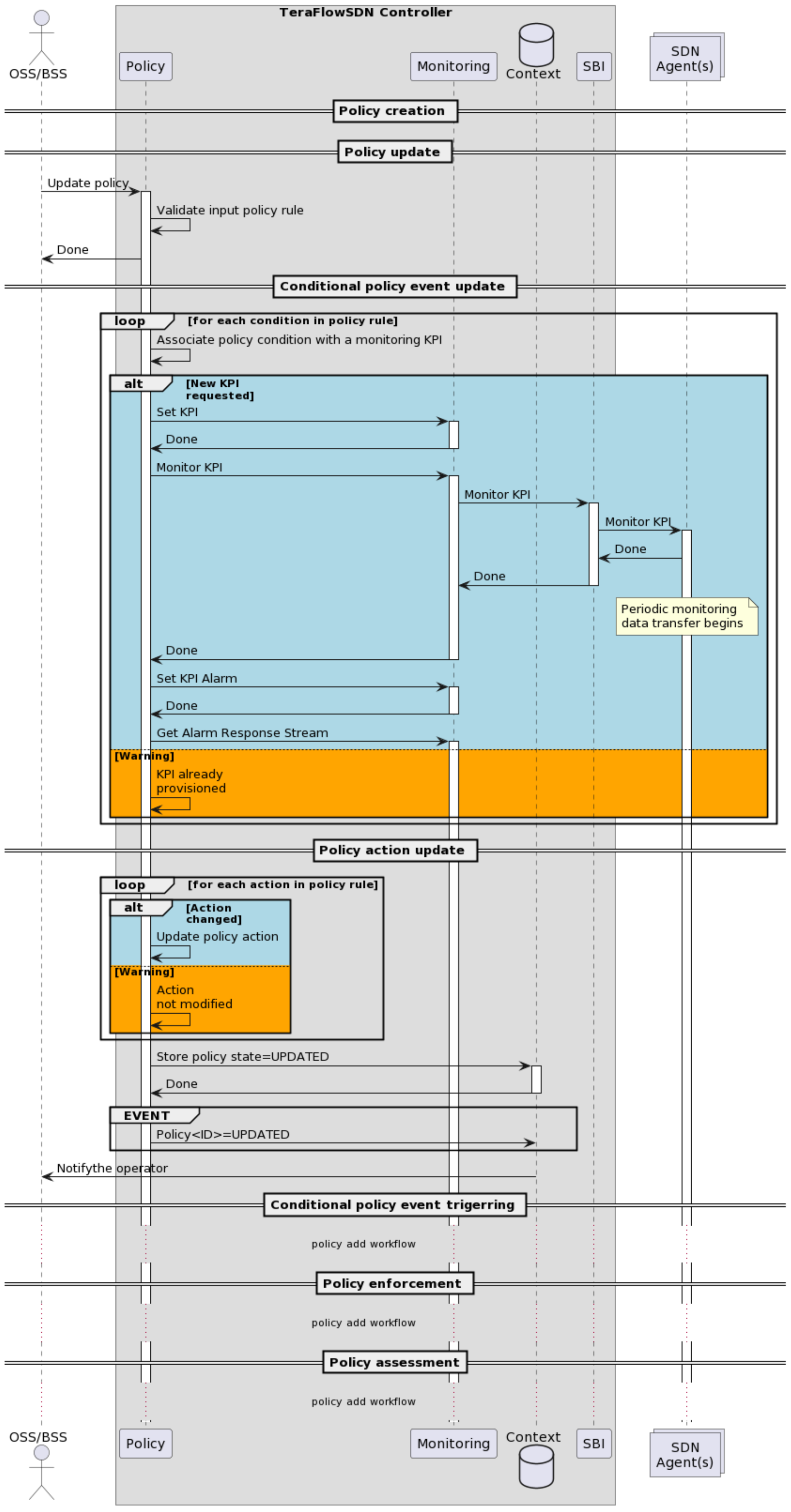


Figure 114: Policy update for service-level and device-level policies.

Generic policy deletion workflow (Both for service-level and device-level policies)

To delete a policy rule, e.g., as part of the deletion of a slice, network operators can issue a policyDelete RPC as shown in Figure Y. First the Policy Management component validates that the deletion request refers to a valid policy ID. Then, a communication with the Monitoring component is initiated to delete the KPI alarms associated to the policy rule. This ensures that no future alarms will be received from the Monitoring component about this policy. Next, the policy rule is deleted from the Context database and a relevant event is generated to notify other components about the successful deletion of the policy rule. This event also could be captured by the WebUI component and visualized to the network operator through a pop-up banner or colour highlights.

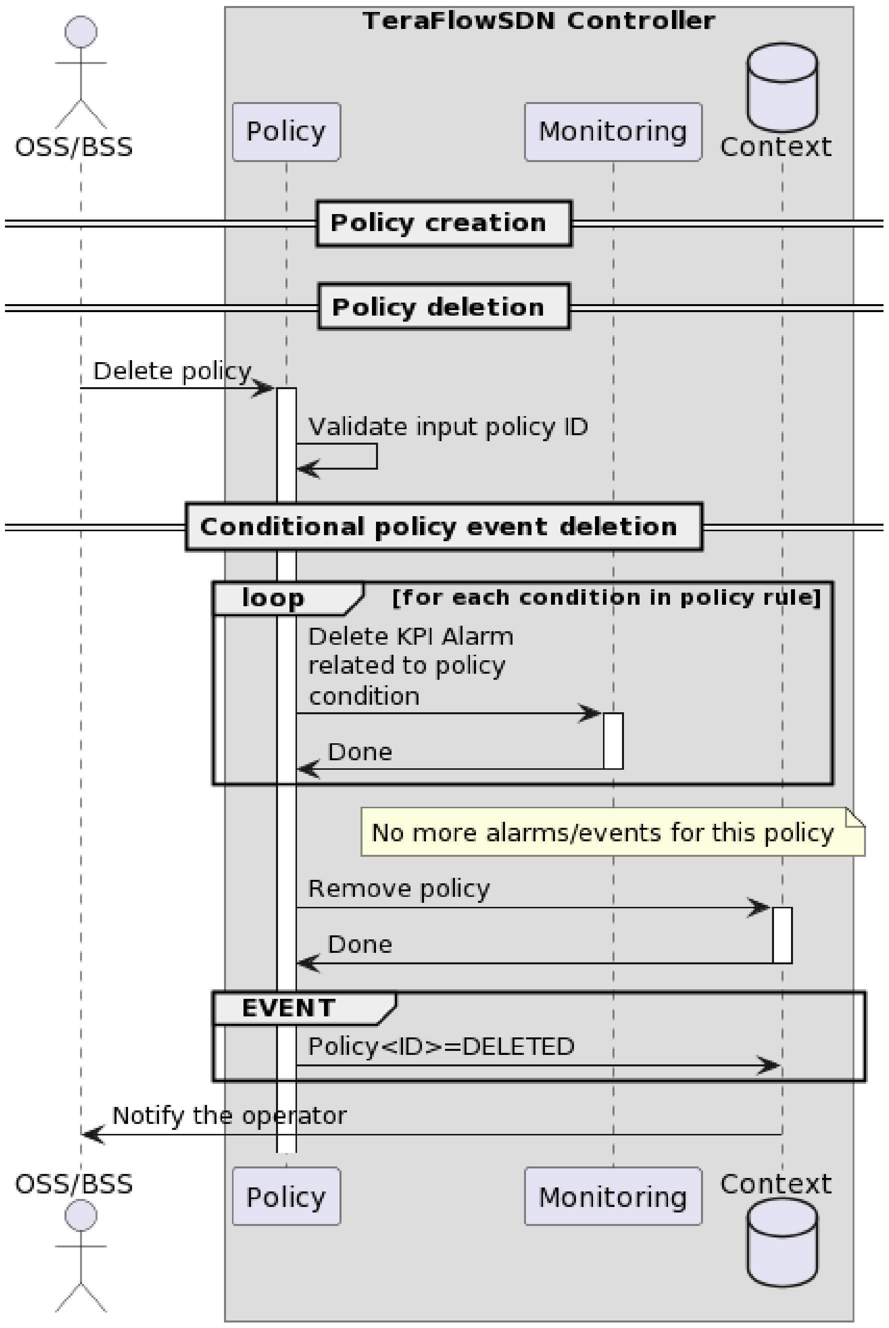


Figure 115: Policy deletion for service-level and device-level policies.

5.2.5. Evaluation

This section evaluates three basic RPCs of the Policy Management component, i.e., policyAddService, policyUpdateService, and policyDelete, using an actual TeraFlowSDN deployment with the Policy, Service, and Context components in action.

Experimental setup

The experimental setup for this evaluation is identical to the one used for the Automation component in Section 5.1.5. TeraFlowSDN is deployed as a Kubernetes service on the test server, and no resource limits are set to the Policy and Context components to allow stress testing.

Benchmarking procedure

To benchmark the three RPCs of the Policy component, the Grafana K6 [17] testing suite is employed as in the case of the Automation component in Section 5.1.5. This tool allows emulating multiple gRPC clients that in the case of the Policy component trigger the target RPCs. K6 allows pinning of each emulated client on a different thread, thus invoke concurrent calls to the Policy component for stressing its behaviour under high load. To stress-test policy add/update/remove operations only a basic integration with the Context component is required. This is because, the Policy component is designed with an asynchronous processing model in mind. Specifically, when an operator requires to add/update/delete a policy, the Policy component validates the inputs and immediately returns a status depending on whether the requested policy object (for add and update operations) or policy ID (for the delete operation) was valid or not. This allows non-blocking requests to the Policy component, while internal threads can be allocated to process the incoming requested as needed. This is also reflected by the workflow diagrams introduced in Section 5.2.4.

The following benchmarks measure the total time (in seconds) required to perform policy add/update/delete operations using an exponentially increasing number (i.e., 1, 10, 100, 500, and 800) of incoming requests. Each RPC is invoked by a client deployed on a dedicated thread to ensure concurrency. Errorbars are used to report the time in the y-axis. The central point corresponds to the median time, while the whiskers correspond to minimum and maximum time respectively. To test the target policy RPCs, an example service needs to be set up. We used an emulated OLS connectivity service atop emulated packet routers as an example. The benchmark creates an increasing number of policy rules using the policy rule template shown in Table 36 as a basis. This rule implies that a TestKPI needs to be created and associated with the target service. When the TestKPI is False, this policy rule will trigger no action. Note that this benchmark does not aim to test the runtime operations of a policy (i.e., how a policy action is enforced), but rather quantify the overhead of requesting an increasing number of policies from TeraFlowSDN. For this reason, the action part of the policy rule has no effect on this benchmark.

Table 36: Policy rule used for the benchmarking of the service-based policy add/update/delete operations.

Policy attribute	**Value**
Service ID	6942d780-cfa9-4dea-a946-a8a0b3f7eab2
Initial policy rule state	UNDEFINED
Policy rule priority	0
Policy rule condition list	TestKPI = False (created for the needs of this benchmark)
Boolean policy rule operator	UNDEFINED (not used as the condition list has a single item)
Policy rule action	NO_ACTION

Service-based policy addition benchmark

The Policy, Service, and Context components are deployed as a clean state in the first experiment. An example L2NM service is created, to which the policies under test refer. Then, the benchmark employs an exponentially increasing number of incoming policy add requests. This is done by calling the policyAddService RPC of the Policy component with the example policy rule shown n Figure X as an input. To test multiple rules for the same service, the benchmark automatically updates the policy rule ID to create an increasing number of rules for the same service. Figure 116 shows the total amount of time (y-axis in seconds) to accommodate an increasing number of rules (i.e., 1 to 800 in x-axis). The clock ticks when the first RPC is issued by K6 and stops when the last RPC is concluded, which implies that all policies are provisioned. As shown in Figure X, inserting a single policy takes around 48ms. The median latency increases up to 10s for 800 policies. To identify the trend, we fit a function to the median latencies shown in Figure 116, which shows a polynomial increase of the latency as follows:

$$\text{Latency} = 798x^2 - 2166x + 1303$$

where x is the number of added policies.

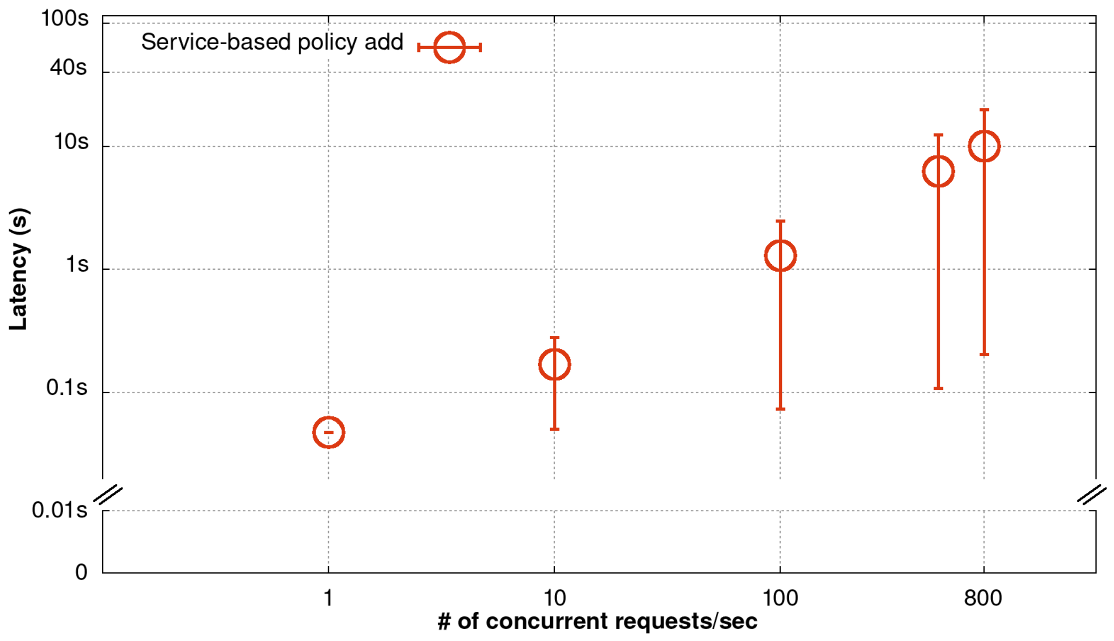


Figure 116: Service-based policy add benchmark using an exponentially increasing number of incoming policy add requests.

Service-based policy update benchmark

In the second experiment, an increasing number of policies is assumed to be pre-provisioned. The objective of this benchmark is to quantify how much time a network operator requires to update this increasing number of available policies using the policyUpdateService RPC. Figure 117 shows the total amount of time (y-axis in seconds) to accommodate an increasing number of policy rule updates (i.e., 1 to 800 in x-axis). Updating a single policy takes around 22ms, which is half the time to insert it. The median latency increases up to 561ms for 800 policies, which renders policy updates at least an order of magnitude faster than policy insertions at scale. To identify the trend, we fit a function to the median latencies shown in Figure 117, which shows a polynomial increase of the latency as follows:

$$\text{Latency} = 39x^2 - 95x + 71,$$

where x is the number of updated policies.

The polynomial coefficients justify our reasoning as the slope of the polynomial function in the case of the policy update RPC is less steep than the slope of the policy add RPC.

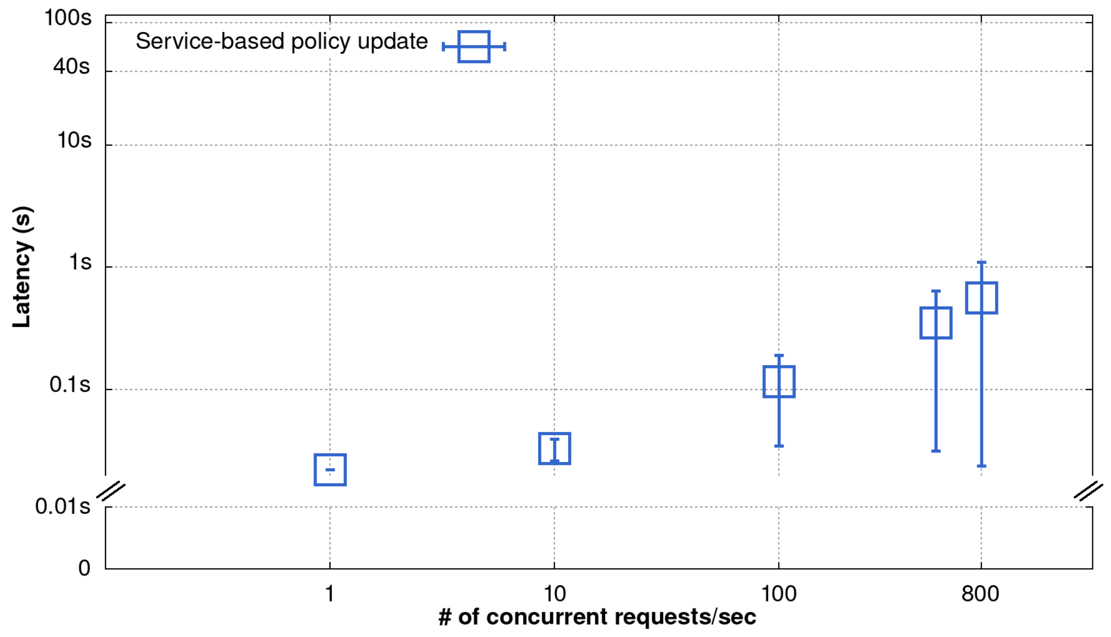


Figure 117: Service-based policy update benchmark using an exponentially increasing number of incoming policy update requests.

Policy deletion benchmark

Finally, an operator should be able to delete available policies. As in the case of the update benchmark, this benchmark assumes a number of pre-provisioned policies which the operator deletes using the policyDelete RPC. This RPC can delete both service and device-level policies as the only input to this RPC is the policy ID to be deleted. Figure 118 shows the total amount of time (y-axis in seconds) to remove an increasing number of policy rules (i.e., 1 to 800 in x-axis). Deleting a single policy takes around 23ms, which is similar to the policy update case. The median latency increases up to 480ms for 800 policies, which makes policy deletion slightly faster than policy updates. To identify the trend, we fit a function to the median latencies shown in Figure 118, which shows a polynomial increase of the latency as follows:

$$\text{Latency} = 24x^2 - 14x - 7$$

where x is the number of deleted policies.

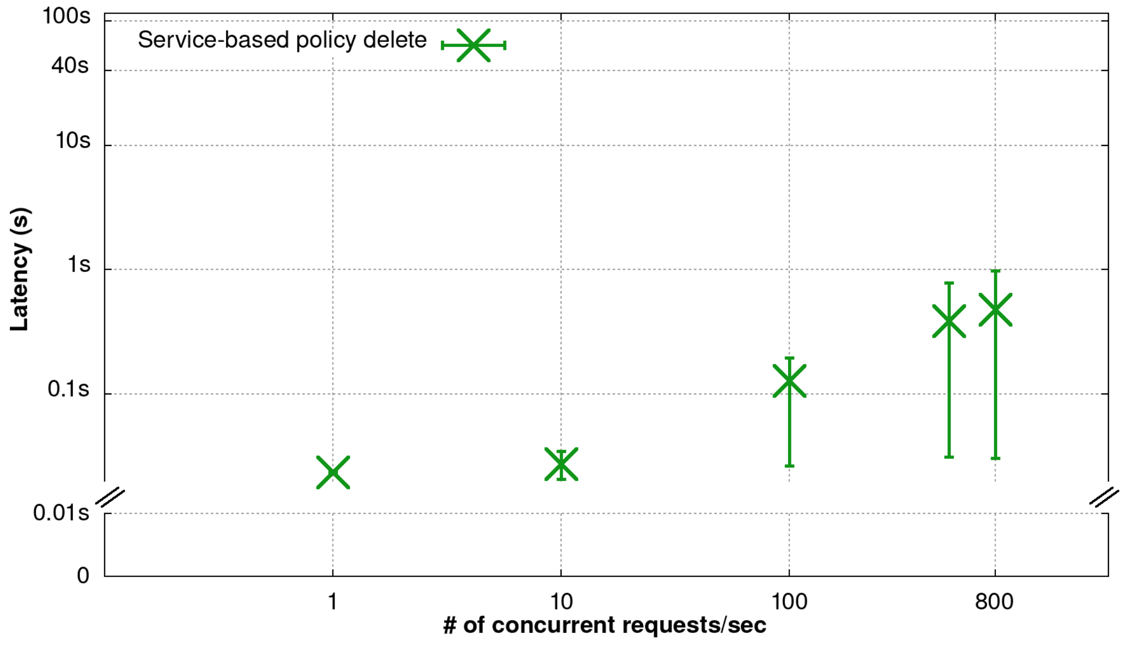


Figure 118: Generic policy benchmark using an exponentially increasing number of incoming policy update requests.

Summary

Policy add operations introduce some reasonable overhead that a network operator takes once off, when creating those policies. On-the-fly policy updates and fast, which suggests that network operators should rather update a policy rather than delete it and create a new one from scratch. Nevertheless, all policy RPCs are control plane operations that are not expected to be heavily utilized by network operators. However, it is useful to quantify the overhead of issuing policy operations. In a large network with lots of dynamic traffic pattern updates, a network operator might need to introduce frequent policy changes, which TeraFlowSDN appears to provision rather quickly.

6. Transport Network Slicing and Multi-tenancy

This section provides the design overview, interfaces, operational workflows, and evaluation results of the core TeraFlowSDN components of T3.4, i.e., the Slice Management component (see Section 6.1).

6.1. Slice Management Component

Network Slices provide the necessary connectivity with a set of specific commitments of network resources between a number of endpoints over a shared underlay network [22]. In this context, transport network slices are provided to support connectivity with a dedicated Service Level Agreement (SLA), which shall be mapped as a technology abstract intent, regardless of the underlying implementation (e.g., L2VPN or L3VPN). Thus, transport network slices once deployed shall be monitored and enforced, in terms of the established SLA constraints/requirements. The current IETF Network Slice Service YANG Model allows the request for the necessary connectivity constraints [23].

We define a slice group as the entity consisting of one or multiple slices with a unique group identifier. One slice belongs to one and only one slice group. Slice grouping requires a mechanism to map a slice into its slice group, also known as slice template or slice blueprint. From our transport network perspective, slice grouping can be based on the mapping of slice SLA requirements to the existing set of slice groups. Thus, slice grouping introduces the need for a clustering algorithm to find service optimization while preserving the slice SLA.

6.1.1. New Features/Extensions

- Initial version of Slice Management component;
- Alignment with NBI component;
- Slice Grouping.

6.1.2. Final Design

The architectural design of the slice component, shown in Figure is quite simple. It just includes the gRPC service interface and a servicer module implementing the logic of the different methods described in section 6.1.3.

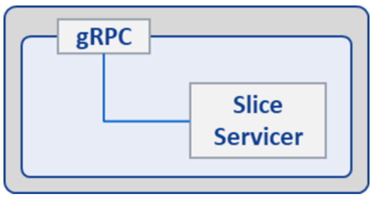


Figure 119 Architecture of the Slice component

Figure 120 provides an example of a slice request based on [23]. The requested slice includes a service-id along with a requested Service Level Objective (SLO) and Service Level Expectation (SLE) policy. By doing so, several metrics can be included, for example SLO ``one way minimum guaranteed bandwidth'' and SLO ``guaranteed availability''. Figure 120 shows an example of slice request based on [23] data model.

```
{
    "service-id": "exp-slice",
    "service-slo-sle-policy":
    {
      "policy-description":"video-service-policy",
      "metric-bounds":
      {
        "metric-bound":
          [
          {
            "metric-type": "service-slo-one-way-bandwidth",
            "metric-unit": "mbps"
            "bound": "5000"
          },
          {
            "metric-type": "service-slo-availability",
            "bound": "99.99%"
          }
          ]
      }
    }
}
```

Figure 120 Slice JSON request based on [Wu22]

6.1.3. Final Interfaces

Table 37 displays an overview of the RPC methods exposed by the Slice component to manage Transport Network Slices ("slices" for short). Specifically, the RPC methods exposed are:

- **CreateSlice**: creates a new slice. As it happens with the Service component, it just instantiates the slice in the Context database and retrieves the identifier;
- **UpdateSlice**: enables to manipulate the slice by managing the endpoints, constraints, and configuration rules, and implementing the appropriate changes through the Service and Device components;
- **DeleteSlice**: removes an existing slice de-configuring the appropriate supporting services and device configuration rules;
- **OrderSliceWithSLA**: similar to UpdateSlice, but explicitly demands to take into consideration the provided SLAs definitions. If slice with SLA already exists, returns the slice; otherwise, creates the slice;
- **RunSliceGrouping**: triggers a procedure to optimize the underlying services and re-maps them to the slices matching the requirements.

Table 37: Service interface definition for the Slice component

RPC Method Name	**Parameters**	**Results**
CreateSlice	context.Slice	context.SliceId
UpdateSlice	context.Slice	context.SliceId
DeleteSlice	context.SliceId	---
OrderSliceWithSLA	context.Slice	context.SliceId
RunSliceGrouping	---	---

6.1.4. Final Operational Workflows

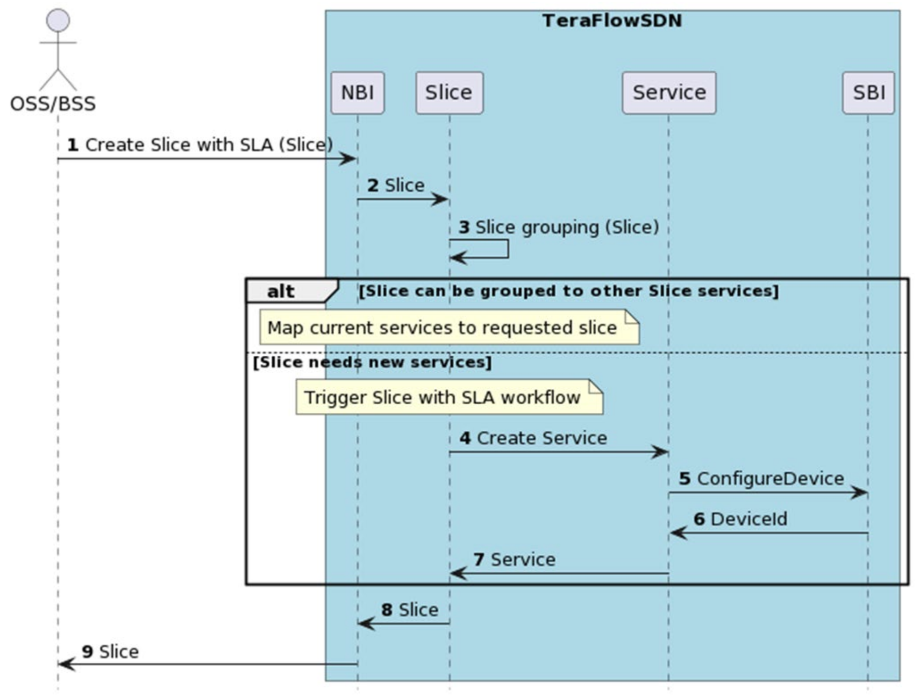

TeraFlowSDN
OSS/BSS
NBI
Slice
Service
SBI
1 Create Slice with SLA (Slice)
2 Slice
3 Slice grouping (Slice)
alt
[Slice can be grouped to other Slice services]
Map current services to requested slice
[Slice needs new services]
Trigger Slice with SLA workflow
4 Create Service
5 ConfigureDevice
6 DeviceId
7 Service
8 Slice
9 Slice

Figure 121 Workflow to provide slice grouping

The workflow depicted in Figure 121 provides slice grouping based on slice requests that demand randomly distributed service availability and allocated bandwidth. We define a slice group as an entity consisting of one or multiple slices with a unique group identifier. One slice belongs to one and only one slice group. Slice grouping requires a mechanism to map a slice into its slice group, also known as a slice template or slice blueprint. From our transport network perspective, slice grouping can be based on mapping slice SLA requirements to the existing set of slice groups. Thus, slice grouping introduces the need for a clustering algorithm to find service optimization while preserving the slice SLA.

Step 1 shows the request for the transport network slice, received from the NorthBound Interface (NBI) via a RESTconf interface. Such a request is then translated/mapped into the used TeraFlowSDN protocol buffer and sent to the Slice component for its processing (Step 2). Finally, in step 3, the slice grouping algorithm is triggered, detailed below.

The outcome of the slice grouping algorithm can result in two options:

- the slice request is mapped to an existing slice group, or;
- a new slice group might be required. In the first case, the slice resources are related to a current/existing slice group and, using Steps 8 and 9, it is notifed the allocated resources for the request to the Operation Support System (OSS) and Business Support System (BSS).

In case new resources need to be allocated, the Slice component requests the necessary connectivity services to the Service component. The resources are then allocated following the necessary SDN orchestration mechanisms (steps 4-7). The underlying resource orchestration workflow applied by TeraFlowSDN is detailed in D2.2.

6.1.5. Evaluation

Experimental setup

We used the same setup (server and micro-service deployment) described for the Emulated Device Driver Evaluation (see section 4.1.6.1).

PERFORMANCE NOTE: Current activities involve integrating a new implementation of the Context component based on CockroachDB, a distributed, scalable, and high-performance relational database. The current implementation is based on a database not supporting concurrency and with limited performance, which results in poor performance results. Given that Slice relies on Service and Device that, in turn highly rely on Context component, the performance reported in this section should be considered preliminary and for functional evaluation only. We expect to achieve better performance after finishing this integration. The final results will be released in deliverable D5.3 [8].

The considered metric for assessing the Slice component is the delay incurred by the component to setup/teardown the transport network slices through the Service and Device components using the Emulated device driver and the L2NM/L3NM Emulated service handlers. This includes the dispatching time of the requests as well as the overheads introduced by the Service, Device and Context components. The request generation is similar to that for the Emulated device driver evaluation (see section 4.1.6.1); however, to produce the CDF of the Slice component delay, the requests are generated uniformly selecting between L2 and L3 network slices. The endpoints of every request are chosen randomly from the transport topology in the previous section. Each request is generated with a Poisson statistical model whose inter-arrival time is set to 200ms while the duration of the service/slice is modelled exponentially with a holding time of 10s.

In Figure 122, it is shown the CDF of the Slice component latency for the generated requests. It is worth noting that each request implies several rule retrievals, configurations and/or deletions. We observe that the requests changing the configuration takes between 1 and 10 seconds. As stated beforehand, we are currently on the integration phase of CockroachDB, a distributed and scalable database, in the Context component. We expect to achieve much better performance after finishing this integration. The final results will be reported in D5.3 [8].

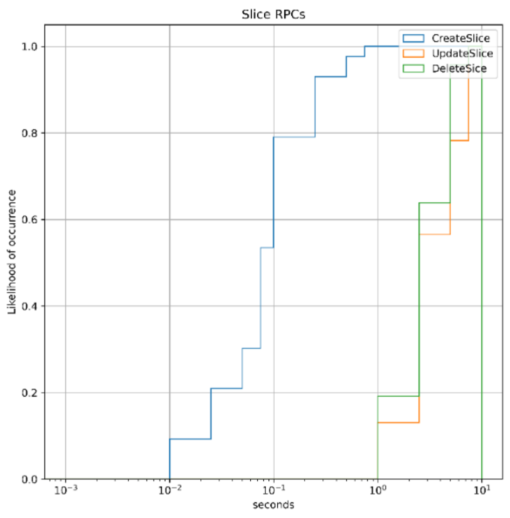


Figure 122: CDF for the Slice component Delay.

Slice Grouping Evaluation

To support the slice grouping based on the requested SLO/SLE, we use the K-Means clustering algorithm. This is an unsupervised machine learning algorithm, which groups data into a determined (i.e., K) number of clusters. These number of clusters is defined by the user and, K-means groups the data into that specific number of clusters. This is the reason why a technique is needed to determine the optimal number of clusters for every specific case.

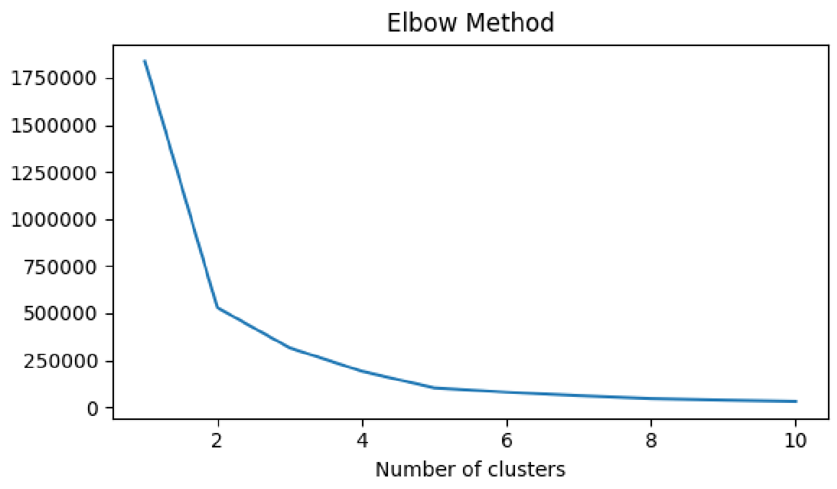


Figure 123 shows the application of the Elbow method to select the number of clusters on the received requests. We have run K-means algorithm for a K value ranging from 1 to 10. For each result, we have computed the sum of the squared distances from each point to its assigned center. These plotted values allow us determining the best value of K (i.e., 2 clusters in the proposed demonstration). Finally, Figure 124 plots the received transport slice requests and the clusters to which they are related.

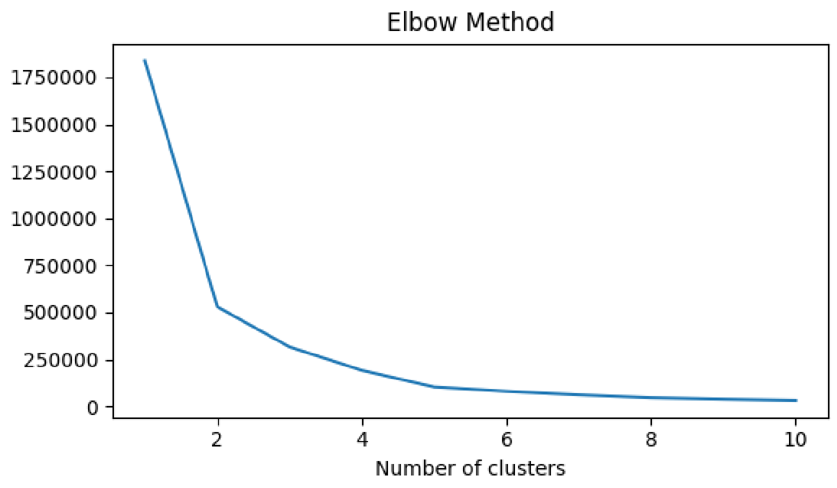


Figure 123 Number of clusters convergence after K-means application

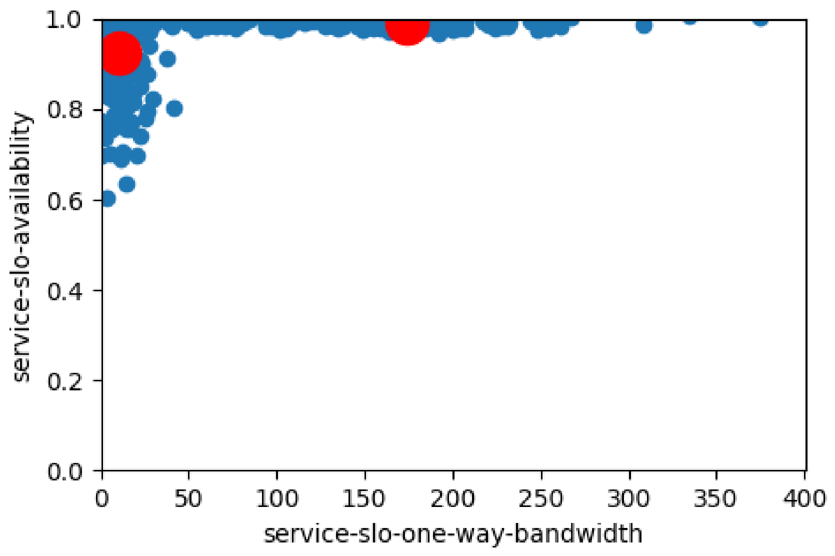


Figure 124 Example of slices grouped in two clusters

7. Conclusions

This deliverable serves as a reference document for the design, interface specification, workflows, and evaluation of core TeraFlowSDN components, touching upon important areas of modern network operating systems, including (i) scalable high-performance SDN control plane, (ii) heterogeneous SDN hardware integration, (iii) service and OS lifecycle automation, and (iv) network slice management.

The documented components are engineered as parts of a fully disaggregated cloud-native network operating system to address the above objectives. The source code, mechanics, documentation, and installation guidelines of the core TeraFlowSDN components are provided in MS3.3 [3]. The intention of this document was to:

- provide additional context on the formal description and architecture of each component through the "Final Design" section;
- document the exposed services per component (i.e., "Final Interfaces" section);
- highlight the interactions between different TeraFlowSDN components or with external entities ("Final Workflows" section);
- evaluate basic component features in terms of performance and/or scalability ("Evaluation" section).

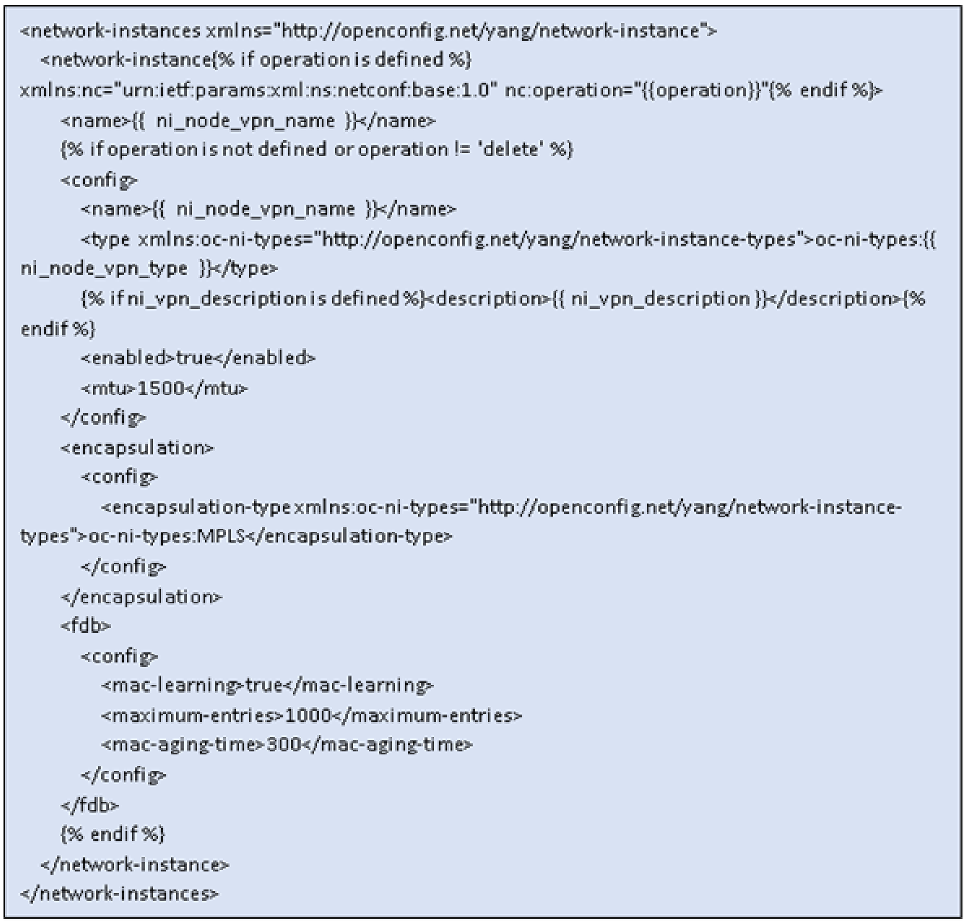
The developments for the reported components are part of the second TeraFlowSDN release, which will be the basis for the experimentation and validation activities in the context of WP5.

8. ANNEX: XML templates

In this Annex are the XML templates used in Section 4.1.3.4.1.2 Data Templates and XML templates.

8.1. L2VPN

1. Create L2-VPN network-instance



```
<network-instances xmlns="http://openconfig.net/yang/network-instance">
  <network-instance{% if operation is defined %}
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation="{{operation}}"{% endif %}>
    <name>{{ ni_node_vpn_name }}</name>
    {% if operation is not defined or operation != 'delete' %}
    <config>
      <name>{{ ni_node_vpn_name }}</name>
      <type xmlns:oc-ni-types="http://openconfig.net/yang/network-instance-types">oc-ni-types:{{
ni_node_vpn_type }}</type>
      {% if ni_vpn_description is defined %}<description>{{ ni_vpn_description }}</description>{%
endif %}
      <enabled>true</enabled>
      <mtu>1500</mtu>
    </config>
    <encapsulation>
      <config>
        <encapsulation-type xmlns:oc-ni-types="http://openconfig.net/yang/network-instance-
types">oc-ni-types:MPLS</encapsulation-type>
      </config>
    </encapsulation>
    <fdb>
      <config>
        <mac-learning>true</mac-learning>
        <maximum-entries>1000</maximum-entries>
        <mac-aging-time>300</mac-aging-time>
      </config>
    </fdb>
    {% endif %}
  </network-instance>
</network-instances>
```

2. Configure interfaces/subinterfaces L2 parameters

```
<interfaces xmlns="http://openconfig.net/yang/interfaces"
        xmlns:oc-ip="http://openconfig.net/yang/interfaces/ip" >
 <interface>
  <name>{{ni_network_access_interface_id}}.{{ni_network_access_termination _point}}</name>
  <config>
   <name>{{ni_network_access_interface_id}}.{{ni_network_access_termination _point }}</name>
   <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">ianaift: l2vlan</type>
   {% if mtu is defined %}<mtu>{{ni_network_access_mtu}}</mtu>{% endif%}
   <enabled>true</enabled>
  </config>
  <subinterfaces>
   <subinterface>
    <index>0</index>
    <config>
     <index>0</index>
     <description>{{description}}</description>
    {% if vendor=="ADVA" and vlan_id is not defined %}
     <untagged-allowed xmlns="http://www.advaoptical.com/cim/adva-dnos-oc-
interfaces">true</untagged-allowed>
     {% endif%}
    </config>
    {% if ni_network_access_vlan_id is defined %}
    <vlan xmlns="http://openconfig.net/yang/vlan">
     <match>
      <single-tagged>
       <config>
        <vlan-id>{{ni_network_access_vlan_id}}</vlan-id>
       </config>
      </single-tagged>
     </match>
    </vlan>
    {% endif%}
   </subinterface>
  </subinterfaces>
 </interface>
</interfaces>
```

3. Add interfaces (endpoint) to L2-VPN network instance

```
<network-instances xmlns="http://openconfig.net/yang/network-instance">
  <network-instance>
    <name>{{ ni_node_vpn_name }}</name>
    <interfaces>
      <interface>
        <id>{{ni_network_access_interface_id}}.{{ni_network_access_termination _point}}</id>
        <config>
          <id>{{ni_network_access_interface_id}}.{{ni_network_access_termination _point}}</id>
          <interface>{{ni_network_access_interface_id}}.{{ni_network_access_termination
_point}}</interface>
          <subinterface>0</subinterface>
        </config>
      </interface>
    </interfaces>
  </network-instance>
```

4. Add virtual circuits (point-to-point, bi-directional pseudo-wire interconnection) to L2-VPN network instance

```
<network-instances xmlns="http://openconfig.net/yang/network-instance">
  <network-instance>
    <name>{{ni_node_vpn_name}}</name>
    <connection-points>
      <connection-point>
        <connection-point-id>{{ni_node_connection_point}}</connection-point-id>
        <config>
          <connection-point-id>{{ni_node_connection_point}}</connection-point-id>
        </config>
        <endpoints>
          <endpoint>
            <endpoint-id>{{ni_node_connection_point}}</endpoint-id>
            <config>
              <endpoint-id>{{ni_node_connection_point}}</endpoint-id>
              <precedence>1</precedence>
              <type xmlns:oc-ni-types="http://openconfig.net/yang/network-instance-types">oc-
ni-types:REMOTE</type>
            </config>
            <remote>
              <config>
                <virtual-circuit-identifier>{{ni_node_connection_point_id}}</virtual-circuit-
identifier>
                <remote-system>{{ni_node_remote_system}}</remote-system>
              </config>
            </remote>
          </endpoint>
        </endpoints>
      </connection-point>
    </connection-points>
  </network-instance>
```

8.2. L3VPN

- `Create L3-VPN network-instance`

```
<network-instances xmlns="http://openconfig.net/yang/network-instance">
   <network-instance{% if operation is defined %}
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation="{{operation}}"{% endif %}>
      <name>{{ni_node_vpn_name}}</name>
      {% if operation is not defined or operation != 'delete' %}
      <config>
         <name>{{ ni_node_vpn_name }}</name>
         <type xmlns:oc-ni-types="http://openconfig.net/yang/network-instance-types">oc-ni-
types:{{ni_node_vpn_type }}</type>
         {% if ni_vpn_description is defined %}<description>{{ni_vpn_description}}</description>{%
endif %}
         {% if ni_rode_router_id is defined %}<router-id>{{ni_node_router_id}}</router-id>{% endif %}
         <route-distinguisher>{{ni_node_route_distinguisher}}</route-distinguisher>
         <enabled>true</enabled>
      </config>
      <encapsulation>
         <config>
            <encapsulation-type xmlns:oc-ni-types="http://openconfig.net/yang/network-instance-
types">oc-ni-types:MPLS</encapsulation-type>
            <label-allocation-mode xmlns:oc-ni-types="http://openconfig.net/yang/network-instance-
types">oc-ni-types:INSTANCE_LABEL</label-allocation-mode>
         </config>
      </encapsulation>
      {% endif %}
   </network-instance>
</network-instances>
```

- Define routing protocols used within L3-VPN network instance

```
<network-instances xmlns="http://openconfig.net/yang/network-instance">
  <network-instance>
    <name>{{ ni_node_vpn_name }}</name>
    <protocols>
      <protocol{% if operation is defined %} xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
nc:operation="{{operation}}"{% endif %}>
        <identifier xmlns:oc-pol-types="http://openconfig.net/yang/policy-types">oc-pol-
types:{{ni_node_policy_type}}</identifier>
        <name>{{ni_node_protocol_name}}</name>
        {% if operation is not defined or operation != 'delete' %}
        <config>
          <identifier xmlns:oc-pol-types="http://openconfig.net/yang/policy-types">oc-pol-
types:{{ ni_node_policy_type }}</identifier>
          <name>{{ ni_node_protocol_name }}</name>
        </config>
        {% if identifier=='BGP' %}
        <bgp>
          <global>
            <config>
              <as>{{ni_node_bgp_local_autonomous_system }}</as>
              <router-id>{{ni_routing_protocol_bgp_router_id }}</router-id>
            </config>
          </global>
        </bgp>
        {% endif %}
        {% endif %}
      </protocol>
    </protocols>
    {% if operation is not defined or operation != 'delete' %}

    <tables>
      <table>
        <protocol xmlns:oc-pol-types="http://openconfig.net/yang/policy-types">oc-pol-
types:{{ni_node_protocol_name}}</protocol>
        <address-family xmlns:oc-types="http://openconfig.net/yang/openconfig-types">oc-
types:IPV4</address-family>
        <config>
          <protocol xmlns:oc-pol-types="http://openconfig.net/yang/policy-types">oc-pol-
types:{{ni_node_protocol_name}}</protocol>
          <address-family xmlns:oc-types="http://openconfig.net/yang/openconfig-types">oc-
types:IPV4</address-family>
        </config>
      </table>
    </tables>
    {% endif %}
  </network-instance>
</network-instances>
```

- o Configure interfaces/subinterfaces L3 parameters

```
<interfaces xmlns="http://openconfig.net/yang/interfaces"
      xmlns:oc-ip="http://openconfig.net/yang/interfaces/ip" >
 <interface>
  <name>{{ni_network_access_interface_id}}.{{ni_network_access_termination _point}}</name>
  <config>
   <name>{{ni_network_access_interface_id}}.{{ni_network_access_termination _point}}</name>
   <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">ianaift:l3ipvlan</type>
   {% if ni_network_access_mtu is defined %}<mtu>{{ni_network_access_mtu}}</mtu>{% endif%}
   <enabled>true</enabled>
  </config>
  <subinterfaces>
   <subinterface>
    <index>0</index>
    <config>
     <index>0</index>
     <description>{{ni_network_access_description}}</description>
    {% if vendor=="ADVA" and vlan_id is not defined %}
     <untagged-allowed xmlns="http://www.advaoptical.com/cim/adva-dnos-oc-
interfaces">true</untagged-allowed>
     {% endif%}
    </config>
    {% if ni_network_access_vlan_id is defined %}
    <vlan xmlns="http://openconfig.net/yang/vlan">
     <match>
      <single-tagged>
       <config>
        <vlan-id>{{ni_network_access_vlan_id}}</vlan-id>
       </config>
      </single-tagged>
     </match>
    </vlan>
    {% endif%}
    {% if ni_network_access_address_ip is defined %}
    <oc-ip:ipv4>
     <oc-ip:addresses>
      <oc-ip:address>
       <oc-ip:ip>{{ni_network_access_address_ip}}</oc-ip:ip>
       <oc-ip:config>
        <oc-ip:ip>{{ni_network_access_address_ip}}</oc-ip:ip>
        <oc-ip:prefix-length>{{ ni_network_access_prefix}}</oc-ip:prefix-length>
       </oc-ip:config>
      </oc-ip:address>
     </oc-ip:addresses>
    </oc-ip:ipv4>
    {% endif%}
   </subinterface>
  </subinterfaces>
 </interface>
</interfaces>
```

- Add interfaces (endpoint) to L3-VPN network instance

```
<network-instances xmlns="http://openconfig.net/yang/network-instance">
  <network-instance>
    <name>{{ ni_node_vpn_name }}</name>
    <interfaces>
      <interface>
        <id>{{ni_network_access_interface_id}}.{{ni_network_access_termination _point }}</id>
        <config>
          <id>{{ni_network_access_interface_id}}.{{ni_network_access_termination _point}}</id>
          <interface>{{ni_network_access_interface_id}}.{{ni_network_access_termination
_point}}</interface>
          <subinterface>0}</subinterface>
        </config>
      </interface>
    </interfaces>
  </network-instance>
</network-instances>
```

- Create BGP Routing Policies Import/Export for a L3-VPN
- <edit-config> create a routing-policy

```
<routing-policy xmlns="http://openconfig.net/yang/routing-policy">
  <defined-sets>
    <bgp-defined-sets xmlns="http://openconfig.net/yang/bgp-policy">
      <ext-community-sets>
        <ext-community-set{% if operation is defined %}
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation="{{operation}}"{% endif %}>
          <ext-community-set-name>{{rp_match_ext_community_set_name}}</ext-community-
set-name>
          {% if operation is not defined or operation != 'delete' %}
            <config>
              <ext-community-set-name>{{rp_match_ext_community_set_name}}</ext-
community-set-name>
              <ext-community-member>{{rp_match_ext_community_member}}</ext-community-
member>
            </config>
          {% endif %}
        </ext-community-set>
      </ext-community-sets>
    </bgp-defined-sets>
  </defined-sets>
</routing-policy>
```

o <edit-config> create BGP match conditions and action

```
<routing-policy xmlns="http://openconfig.net/yang/routing-policy">
  <policy-definitions>
    <policy-definition {% if operation is defined %}
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation="{{operation}}"{% endif %}>
      <name>{{rp_set_name}}</name>
      {% if operation is not defined or operation != 'delete' %}
      <config>
        <name>{{rp_set_name}}</name>
      </config>
      <statements>
        <statement>
          <name>{{rp_set_statement_name}}</name>
          <config>
            <name>{{rp_set_statement_name}}</name>
          </config>
          <conditions>
            <config>
              <install-protocol-eq xmlns:oc-pol-types="http://openconfig.net/yang/policy-
types">oc-pol-types:DIRECTLY_CONNECTED</install-protocol-eq>
            </config>
            <bgp-conditions xmlns="http://openconfig.net/yang/bgp-policy">
              <config>
                <ext-community-set>{{rp_match_ext_community_set_name}}</ext-community-
set>
              </config>
            </bgp-conditions>
          </conditions>
          <actions>
            <config>
              <policy-result>{{rp_action_policy_result}}</policy-result>
            </config>
          </actions>
        </statement>
      </statements>
      {% endif %}
    </policy-definition>
  </policy-definitions>
</routing-policy>
```

o Apply BGP Import/export Policy (Route Target) to L3-VPN network instance

```
{% if operation is not defined or operation != 'delete' %}
<network-instances xmlns="http://openconfig.net/yang/network-instance">
   <network-instance>
      <name>{{ ni_node_vpn_name }}</name>
      <inter-instance-policies>
         <apply-policy>
            <config>
              {% if ni_node_import_policy is defined %}<import-policy>{{ni_node_import_policy}}</import-
policy>{% endif%}
              {% if ni_node_export_policy is defined %}<export-policy>{{ni_node_export_policy}}</export-
policy>{% endif%}
            </config>
         </apply-policy>
      </inter-instance-policies>
   </network-instance>
</network-instances>
{% endif %}
```

- Create protocol redistribution policies within a L3-VPN network instance

```
<network-instances xmlns="http://openconfig.net/yang/network-instance">
   <network-instance>
      <name>{{ ni_node_vpn_name }}</name>
      <table-connections>
         <table-connection{% if operation is defined %}
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation="{{operation}}"{% endif %}>
            <src-protocol xmlns:oc-pol-types="http://openconfig.net/yang/policy-types">oc-pol-
types:{{ni_routing_protocol_src}}</src-protocol>
            <dst-protocol xmlns:oc-pol-types="http://openconfig.net/yang/policy-types">oc-pol-
types:{{ ni_routing_protocol_dst}}</dst-protocol>
            <address-family xmlns:oc-types="http://openconfig.net/yang/openconfig-types">oc-
types:{{ ni_routing_protocol_af}}</address-family>
            {% if operation is not defined or operation != 'delete' %}
            <config>
               <src-protocol xmlns:oc-pol-types="http://openconfig.net/yang/policy-types">oc-pol-
types:{{ ni_routing_protocol_src}}</src-protocol>
               <dst-protocol xmlns:oc-pol-types="http://openconfig.net/yang/policy-types">oc-pol-
types:{{ ni_routing_protocol_dst}}</dst-protocol>
               <address-family xmlns:oc-types="http://openconfig.net/yang/openconfig-types">oc-
types:{{ ni_routing_protocol_af}}</address-family>
              <default-import-policy>ACCEPT_ROUTE</default-import-policy>
            </config>
            {% endif %}
         </table-connection>
      </table-connections>
   </network-instance>
</network-instances>
```

8.3. ACL

- Create ACL-SET

```
<acl xmlns="http://openconfig.net/yang/acl">
 <acl-sets>
   <acl-set{% if operation is defined %} xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
nc:operation="{{operation}}"{% endif %}>
    <name>{{acl_name}}</name>
    <type>{{acl_type}}</type>
    <config>
     <name>{{acl_name}}</name>
     <type>{{acl_type}}</type>
    </config>
    <acl-entries>
     <acl-entry>
      <sequence-id>{{acl_id}}</sequence-id>
      <config>
       <sequence-id>{{acl_id}}</sequence-id>
      </config>
      {% if operation is not defined or operation != 'delete' %}
      {% if acl_type=='ACL_L2' %}
      <l2>
       <config>
        {% if acl_match_source_mac is defined %}<source-mac>{{
acl_match_source_mac}}</source-mac>{% endif%}
        {% if acl_match_destination_mac is defined %}<destination-mac>{{
acl_match_destination_mac}}</destination-mac>{% endif%}
       </config>
      </l2>
      {% endif%}
      {% if type=='ACL_IPV4' %}
      <ipv4>
       <config>
        {% if acl_match_source_ipv4 is defined %}<source-address>{{
acl_match_source_ipv4}}</source-address>{% endif%}
        {% if acl_match_destination_ipv4 is defined %}<destination-address>{{
acl_match_destination_ipv4}}</destination-address>{% endif%}
        {% if acl_match_protocol is defined %}<protocol>{{ acl_match_protocol}}</protocol>{%
endif%}
        {% if acl_match_dscp is defined %}<d dscp>{{ acl_match_dscp}}</dscp>{% endif%}
        {% if acl_match_hop_limit is defined %}<limit>{{ acl_match_hop_limit}}</hop-limit>{%
endif%}
       </config>
      </ipv4>
      <transport>
       <config>
        {% if acl_match_source_port is defined %}<source-port>{{
acl_match_source_port}}</source-port>{% endif%}
        {% if acl_match_destination_port is defined %}<destination-port>{{
acl_match_destination_port}}</destination-port>{% endif%}
        {% if acl_match_tcp_flags is defined %}<tcp-flags>{{ acl_match_tcp_flags}}</tcp-flags>{%
endif%}
       </config>
      </transport>m
     {% endif%}
```

```
      {% if type=='ACL_IPV6' %}
       <ipv6>
         <config>
           {% if acl_match_source_ipv6 is defined %}<source-address>{{
acl_match_source_ipv6}}</source-address>{% endif%}
           {% if acl_match_destination_ipv6 is defined %}<destination-address>{{
acl_match_destination_ipv6}}</destination-address>{% endif%}
           {% if acl_match_protocol is defined %}<protocol>{{ acl_match_protocol}}</protocol>{%
endif%}
           {% if acl_match_dscp is defined %}<dscp>{{ acl_match_dscp}}</dscp>{% endif%}
           {% if acl_match_hop_limit is defined %}<hop-limit>{{ acl_match_hop_limit}}</hop-limit>{%
endif%}
         </config>
       </ipv6>
      {% endif%}
       <actions>
         <config>
           {% if acl_match_forwarding_action is defined %}<forwarding-action>{{
acl_match_forwarding_action}}</forwarding-action>{% endif%}
           {% if acl_match_log_action is defined %}<log-action>{{ acl_match_log_action}}</log-
action>{% endif%}
         </config>
       </actions>
       {% endif%}
     </acl-entry>
    </acl-entries>
  </acl-set>
 </acl-sets>
</acl>
```

- o Add ACL-ENTRY to ACL-SET (Same template as Create ACL_SET)
- o Associate the ACL to an interface

[1] Ingress

```
<acl xmlns="http://openconfig.net/yang/acl">
 <interfaces>
  <interface {% if operation is defined %}{% if all is defined %}
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation="{{operation}}"{% endif %} {%
endif %}>
   <id>{{ acl_interface _id}}</id>
   <config>
    <id>{{ acl_interface _id}}</id>
   </config>
   {% if interface is defined %}
   <interface-ref>
    <config>
     <interface>{{ acl_interface  }}</interface>
     {% if subinterface is defined %}<subinterface>{{ acl_interface_subinterface}}</subinterface>{%
endif%}
    </config>
   </interface-ref>
   {% endif%}
   <ingress-acl-sets>
    <ingress-acl-set {% if operation is defined %}{% if ingress is defined %}
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation="{{operation}}"{% endif %} {%
endif %}>
     <set-name>{{acl_interface _acl_set_name}}</set-name>
     <type>{{acl_interface _acl_set_type}}</type>
     <config>
      <set-name>{{ acl_interface _acl_set_name}}</set-name>
      <type>{{acl_iterface _acl_set_type }}</type>
     </config>
    </ingress-acl-set>
   </ingress-acl-sets>
  </interface>
 </interfaces>
</acl>
```

[2] Egress

```
<acl xmlns="http://openconfig.net/yang/acl">
 <interfaces>
  <interface {% if operation is defined %}{% if all is defined %}
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation="{{operation}}"{% endif %} {%
endif %}>
   <id>{{ acl_interface _id}}</id>
   <config>
    <id>{{ acl_interface _id}}</id>
   </config>
   {% if interface is defined %}
   <interface-ref>
    <config>
     <interface>{{ acl_interface  }}</interface>
     {% if subinterface is defined %}<subinterface>{{ acl_interface_subinterface}}</subinterface>{%
endif%}
    </config>
   </interface-ref>
   {% endif%}
   <egress-acl-sets>
    <egress-acl-set {% if operation is defined %}{% if ingress is defined %}
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation="{{operation}}"{% endif %} {%
endif %}>
     <set-name>{{acl_interface _acl_set_name}}</set-name>
     <type>{{acl_interface _acl_set_type}}</type>
     <config>
      <set-name>{{ acl_interface _acl_set_name}}</set-name>
      <type>{{acl_iterface _acl_set_type }}</type>
     </config>
    </egress-acl-set>
   </egress-acl-sets>
  </interface>
 </interfaces>
</acl>
```

8.4. Inventary

```
<get xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <filter>
    <components xmlns="http://openconfig.net/yang/platform">
      <component
      </component>
    </components>
  </filter>
</get>
```

References

[1] H2020 EU TeraFlow project, "MS3.1: Study of technical aspects of relevant SDN, Cloud-native and SDO solutions ", April 2021.

[2] H2020 EU TeraFlow project, "MS3.2: Code freeze for TeraFlow OS components (v1): Context Management, Monitoring, Auto Scaling, Load balancing, L0/L3 integration and Device Management, Automation and Policy Management, Slice Management ", October 2021.

[3] H2020 EU TeraFlow project, "MS3.3: Code freeze for TeraFlow OS release components (v2): Context Management, Monitoring, Auto Scaling, Load balancing, L0/L3 integration and Device Management, Automation and Policy Management, Slice Management ", September 2022.

[4] H2020 EU TeraFlow project, "D2.2: Final requirements, architecture design, business models and data models", Dec. 2022.

[5] H2020 EU TeraFlow project, "D3.1: Preliminary Evaluation of Life-cycle Automation and High Performance SDN Components", Dec. 2021.

[6] H2020 EU TeraFlow project, "D4.2: Final evaluation of TeraFlow security and B5G network integration", Dec 2022.

[7] H2020 EU TeraFlow project, "D5.2: Implementation of pilots and first evaluation", Dec 2022.

[8] H2020 EU TeraFlow project, "D5.3: Final demonstrators and evaluation report", June 2023.

[9] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. 38, 2 (April 2008), 69–74. DOI: https://doi.org/10.1145/1355734.1355746

[10]Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. SIGCOMM Comput. Commun. Rev. 44, 3 (July 2014), 87–95. DOI: https://doi.org/10.1145/2656877.2656890

[11]Open Networking Foundation (ONF): https://opennetworking.org/

[12]ONF Stratum OS: https://opennetworking.org/stratum/

[13]ONF Open Network Operating System (ONOS): https://opennetworking.org/onos/

[14]The P4.org Working Groups, Available: https://p4.org/working-groups/

[15]The P4.org Applications Working Group, Contributions from Alibaba, Arista, CableLabs, Cisco Systems, Dell, Intel, Marvell, Netronome, VMware, "In-band Network Telemetry (INT) Dataplane Specification", version 2.1, November 11, 2020, Available: https://p4.org/p4-spec/docs/INT_v2_1.pdf

[16]Qin Wu, Igor Bryskin, Henk Birkholz, Xufeng Liu, Benoit Claise, "A YANG Data model for ECA Policy Management", IETF draft NETMOD Working Group, February 19, 2021. Work in progress. Available from: https://datatracker.ietf.org/doc/html/draft-ietf-netmod-eca-policy

[17]Grafana K6: Load testing for engineering teams, Available: https://k6.io/

[18] CockroachDB, https://github.com/cockroachdb/cockroach. Accessed: 28/11/2022.

[19] S. Orlowski, M. Pioro, A. Tomaszewski, and R. Wessaly, "SNDlib 1.0–Survivable Network Design Library," in Proceedings of the 3rd International Network Optimization Conference (INOC 2007), Spa, Belgium, April 2007, http://sndlib.zib.de, extended version accepted in Networks, 2009.

Available: http://www.zib.de/orlowski/Paper/OrlowskiPioroTomaszewskiWessaely2007-SNDlib-INOC.pdf.gz

[20] Facebook Open Source, “https://facebook.github.io/prophet/,” Online, accessed in October 7, 2022.

[21] A. Alsharef, K. Aggarwal, M. Kumar, A. Mishra et al., “Review of ml and automl solutions to forecast time-series data,” Archives of Computational Methods in Engineering, pp. 1–15, 2022

[22] A. Farrel, et. al, “Framework for IETF Network Slices,” IETF, draft-ietf-teas-ietf-network-slices-16, 2022.

[23] B. Wu, et. al, “IETF Network Slice Service YANG Model,” IETF, draft-ietf-teas-ietf-network-slice-nbi-yang-03, 2022.

[24] Internet Engineering Task Force (IETF) RFC 6241, “Network Configuration Protocol (NETCONF),” 2011. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6241

[25] OpenConfig: Vendor-neutral, model-driven network management designed by users, [Online] Available: https://www.openconfig.net/

[26] QuestDB, https://github.com/questdb/questdb. Accessed: 23/12/2022