FEDERAL UNIVERSITY OF UBERLÂNDIA
FACULTY OF ELECTRICAL ENGINEERING

MURILO GABRIEL LEAL VIEIRA

Telemetry Performance Comparison between gNMI and SNMP in IP/MPLS Network Architecture

Uberlândia
2026

MURILO GABRIEL LEAL VIEIRA

Telemetry Performance Comparison between gNMI and SNMP in IP/MPLS Network Architecture

Undergraduate thesis submitted to the Faculty of Electrical Engineering (FEELT) of the Federal University of Uberlândia (UFU) in partial fulfillment of the requirements for the degree of Bachelor's in Electronic and Telecommunications Engineering.

Area of Concentration: Computer networks, telemetry.

Advisor: Dr. Éderson Rosa da Silva

Uberlândia

2026

MURILO GABRIEL LEAL VIEIRA

Telemetry Performance Comparison between gNMI and SNMP in IP/MPLS Network Architecture

Undergraduate thesis submitted to the Faculty of Electrical Engineering (FEELT) of the Federal University of Uberlândia (UFU) in partial fulfillment of the requirements for the degree of Bachelor's in Electronic and Telecommunications Engineering.

Area of Concentration: Computer networks, telemetry.

Uberlândia, 05/08/2026

Thesis Committee:

_____

Dr. Éderson Rosa da Silva

_____

Dr. André Luiz Aguiar da Costa

_____

Me. Gabriel Andrade Queiroz

ACKNOWLEDGMENTS

First, I would like to express my gratitude to the Federal University of Uberlândia for the opportunity to pursue my studies and for the academic learning provided throughout this journey. I am equally grateful to the Electronic and Telecommunications Engineering department and their coordination for continuous support and dedication.

A special thank you goes to my advisor, Professor Éderson Rosa, whose guidance and assistance was fundamental to the completion and publication of this work.

Finally, and above all, I express my gratitude to my family and girlfriend. Their constant encouragement, unconditional love, and faith in my potential were the cornerstones that made this achievement possible.

ABSTRACT

The exponential growth of data traffic in Service Provider environments demands highly accurate and real-time network observability. Traditionally, the Simple Network Management Protocol (SNMP) has been the standard for network monitoring; however, its pull-based polling architecture presents limitations in detecting transient anomalies. This study presents a comparative performance analysis between SNMP and the modern, push and Remote Procedure Call (RPC) based gRPC Network Management Interface (gNMI). By using Infrastructure as Code (IaC) principles, a multi-vendor routing environment comprising Juniper and Arista nodes was virtualized using the Containerlab orchestration framework. The methodology subjected the observability pipeline to deterministic control-plane and data-plane anomalies, including volumetric Distributed Denial of Service (DDoS) bursts, Border Gateway Protocol (BGP) route flapping, and Label Distribution Protocol (LDP) intermittency. The research concludes that gNMI provides high-fidelity visibility (better spikes view) and lower operational overhead (39%-50% less CPU usage) for modern autonomous networks, recommending a phased migration strategy for production environments.

Keywords: BGP; gNMI; MPLS; network automation; network telemetry.

RESUMO

O crescimento exponencial do tráfego de dados em ambientes de Provedores de Serviço exige uma observabilidade de rede altamente precisa e em tempo real. Tradicionalmente, o SNMP (do inglês, Simple Network Management Protocol) tem sido o padrão para o monitoramento de redes; contudo, sua arquitetura baseada em consultas periódicas (do inglês, pull-based polling) apresenta limitações na detecção de anomalias transitórias. Este estudo apresenta uma análise comparativa de desempenho entre o SNMP e a moderna interface baseada em envio contínuo de dados (do inglês, push-based) e RPC (do inglês, Remote Procedure Call), o gNMI (do inglês, gRPC Network Management Interface). A partir de princípios de Infraestrutura como Código (do inglês, Infrastructure as Code - IaC), um ambiente de roteamento multifabricante compreendendo nós Juniper e Arista foi virtualizado através do framework de orquestração Containerlab. A metodologia submeteu o pipeline de observabilidade a anomalias determinísticas no plano de controle e no plano de dados, incluindo rajadas volumétricas de Negação de Serviço Distribuída (do inglês, Distributed Denial of Service - DDoS), instabilidade de rotas Border Gateway Protocol (BGP route flapping) e intermitência do LDP (do inglês, Label Distribution Protocol). A pesquisa conclui que o gNMI fornece a visibilidade de alta fidelidade (melhor identificação de picos) e menor custo operacional (39%-50% uso reduzido de CPU) para redes autônomas modernas, recomendando uma estratégia de migração em fases para ambientes de produção.

Palavras-chave: automação de redes; BGP; gNMI; MPLS; SNMP; telemetria.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AS	Autonomous Systems
BGP	Border Gateway Protocol
DDoS	Distributed Denial of Service
DSCP	Differentiated Services Code Point
EBGP	External Border Gateway Protocol
ECMP	Equal-Cost Multipath
EXP	Experimental Bits
FDT	Failure Detection Time
FEC	Forwarding Equivalence Class
FEELT	Faculty of Electrical Engineering
gNMI	gRPC Network Management Interface
HTTP/2	Hypertext Transfer Protocol Version 2
IaC	Infrastructure as Code
IBGP	Internal Border Gateway Protocol
IETF	Internet Engineering Task Force
IGP	Internal Gateway Protocol
IP	Internet Protocol
ISP	Internet Service Provider
LDP	Label Distribution Protocol
LER	Label Edge Router
LFIB	Label Information Base
LSP	Label Switching Path
LSR	Label Switching Router
MED	Multi-Exit Discriminator
MIB	Management Information Base
MPLS	Multiprotocol Label Switching
NLRI	Network Layer Reachability Information
NMS	Network Management Station
NOS	Network Operating System

OID	Object Identifier
PDU	Protocol Data Unit
PE	Provider Edge
PHP	Penultimate Hop Popping
PromQL	Prometheus Query Language
Protobuf	Protocol Buffers
QoS	Quality-of-Service
RIB	Routing Information Base
RPC	Remote Procedure Call
RR	Route Reflector
RSVP-TE	Resource Reservation Protocol with Traffic Engineering
SNMP	Simple Network Management Protocol
SP	Service Provider
SPF	Shortest Path First
TCP	Transmission Control Protocol
TLV	Type-Length-Value
TTL	Time to Live
USM	User-based Security Model
UDP	User Datagram Protocol
UFU	Federal University of Uberlândia
VACM	View-based Access Control Model
VPN	Virtual Private Network
YAML	YAML Ain't Markup Language
YANG	Yet Another Next Generation

1 INTRODUCTION

The modern internet infrastructure and Service Providers (SPs) networks depend on protocols for external communication such as **Border Gateway Protocol** (**BGP)** and for internal traffic management as Multiprotocol Label Switching (**MPLS**). The BGP is a routing protocol that permits to exchange worldwide routing information among routers in different autonomous systems[1] (ASs). It includes the complete route to each destination [2] and allows providers to enforce policies and determine the best paths for worldwide traffic. On backbone, the MPLS is designed to optimize packet forwarding by a label-based transport underlay rather than performing complex routing queries at each hop [3]. However, the fundamental mechanism it provides for modern applications is service creation and traffic engineering.

The interest in monitoring the functionality of protocols and physical links is tremendous. As reference to the *2024 Observability Forecast* from New Relic, engineering teams devote 30% of their time addressing interruptions and the leading cause over the last two years is network failure (**35%**) [4]. In a Service Provider context, these are not generic failures; they are the concrete failure of a BGP peering session, a physical link in the Internal Gateway Protocol (IGP), or the collapse of an MPLS transport tunnel. This high dependency, coupled with the high cost of failure, makes the deep observability of BGP and MPLS not just an operational goal, but a core business requirement.

Network observability means the use of data sources to understand what is happening inside a network, which translates into actionable insights on performance, behavior and health of its internals [5]. For decades, the Simple Network Management Protocol (SNMP) has been the consolidated protocol to achieve this. It was designed to collect and report data from network devices based on its operations (SNMP GET, GETNEXT, GETBULK, etc.) under a foundational **Manager-Agent** (or client-server) architecture. The Network Management Station (NMS) acts as the Manager, which periodically polls the Agents (routers, switches, etc.) in order to maintain metrics on a database called Management Information Base (MIB).

This manager-driven, **pull-based** model provides a periodic and reactive snapshot of the network's health. It became the global standard for the majority of vendors and solutions, but its operational utility is intrinsically tied to the frequency of its poll.

In this context, streaming telemetry has been raised as an alternative. Based on Remote Procedure Calls (RPC), **gRPC Network Management Interface (gNMI)** was created by Google as part of the OpenConfig initiative to provide a standard, efficient protocol for network

[1] Unique identifier that indicates a set of routers under a single technical administration.

device configuration and telemetry, published as an IETF draft in 2018 [6]. It leverages gRPC over HTTP/2 to carry a payload that contains data structured by Yet Another Next Generation (**YANG)** schemas of intrinsically characteristics: tree organization and serialization on Protocol Buffers (**Protobuf**) [7]. This allows gNMI to manage data via its **Subscribe RPC** operation, establishing a stateful stream where the device can efficiently push updates either on-change (only when a value changes) or at a high-frequency sample interval.

1.1 Motivation

For this project, a comparison between SNMP and gNMI is built to measure the advantages of each implementation. A focus on Failure Detection Time (FDT) and operational resource costs will be part of the trade-offs against the requirements for data granularity and network scalability.

1.2 Related Works

In the academic field, several researchers have focused on presenting approaches for automating tasks such as device provisioning, real-time telemetry monitoring, and the integration of networks with artificial intelligence to enable autonomous optimization and failure prediction. This section discusses relational approaches and solutions for Internet Protocol (IP) and optical network automation, analyzing recent academic studies, simulation tools, and relevant architectures, highlighting how these works contribute to the evolution of automation practices and network monitoring.

The study presented by [1] describes the implementation of a network simulation to analyze the performance and trade-offs of gNMI streaming telemetry-based monitoring systems. Employing an approach similar to the current work, the authors used Containerlab to integrate a Docker-based network topology with a modern monitoring stack. The objective was to compare gNMI's resource utilization against traditional protocols like SNMP and sFlow on collector. The outcomes demonstrated that while metric retrieval had minimal impact on network latency and throughput, the gNMI system utilized disk write/read and Central Processing Unit (CPU) resources significantly more intensively, showing up to a 50% increase in CPU utilization and doubling the disk usage.

Additionally, the work proposed by [38] present novel extensions for a Cloud-native software-defined networking controller designed for partially disaggregated optical networks. The primary objective of the study was to enable the real-time monitoring of optical transponders using the gNMI protocol and OpenConfig data models. The research demonstrates

the feasibility of this architecture by successfully retrieving real-time telemetry measurements, such as pre-FEC bit error rates.

Finally, the article presented by [39] provides a comprehensive framework integrating telemetry, digital twins, and agentic Artificial Intelligence to establish the foundations for optical network automation. The objective of the paper was to consolidate how streaming telemetry acts as a sensory data layer, while Large Language Models and multi-agent systems coordinate complex operational decisions. The outcomes highlight a strong industry convergence toward intent-based operations, where standardized data models and distributed intelligence replace manual, reactive management. The study clearly points out that the future of network automation relies on the development of dynamic ecosystems of collaborative agents, advanced compression techniques for telemetry streams, distributed digital twins, and the implementation of strict safeguards to ensure the reliability of autonomous actions.

The analysis of the different approaches mentioned highlights how network automation and real-time telemetry have become effective solutions to network monitoring and responsive systems. Moreover, trade-offs and results are aimed to be measured on this work to test practical scenarios using gNMI and its comparison to SNMP in the demands of modern IP networks.

1.3 Objectives

The primary objective is to evaluate performance and observability on Service Provider networks which use BGP and MPLS. A controlled laboratory environment for data acquisition and fault simulation will serve as the methodological foundation for quantifying these metrics.

Ultimately, this study aims to provide a clear recommendation on which technology is better suited for production environments, accompanied by the initial implementation steps required for such migration.

In essence, this study aims to answer the following questions:

(i) What are the delays achievable for failure detection for a gNMI telemetry stream acquisition?

(ii) What are the determining factors for this delay?

(iii) How does it differ from standard SNMP?

1.4 Organization

To provide a clear and logical progression of the concepts, methodology, and analysis, this study is organized into five main chapters, followed by bibliographic references and supplementary appendices.

On Chapter 2, Technical Fundamentals, it will be established the theoretical foundation required to comprehend the experimental architecture. It objectively reviews core routing and forwarding architectures, specifically MPLS and the BGP. Furthermore, it contrasts legacy network management via the SNMP with modern model-driven telemetry using the gNMI, concluding with an introduction to the Containerlab orchestration framework.

Followed by Chapter 3 – Methodology, details of the practical design and execution of the experimental testbed will be shown. This chapter outlines the declarative network modeling in Containerlab, the automated node provisioning process leveraging Jinja2 and Python, the mechanisms for synthetic traffic and routing injection utilizing ExaBGP and *iperf3*, and the deployment of the observability pipeline using Telegraf and Grafana.

Finally, Chapter 4 – Results and Discussion and Chapter 5 – Conclusion presents, analyzes the data collected from the laboratory environment and evaluates the fulfillment of the proposed objectives. It systematically checks the network's behavior under synthetic volumetric traffic, examines Label Distribution Protocol (LDP) state transition delays and BGP prefix-count metrics, and formally assesses the computational and operational overhead induced by the monitoring protocols.

To guarantee experimental reproducibility, the Appendices compile all declarative configuration schemas, automation scripts, and deployment templates utilized throughout the study, including the primary Containerlab YAML file, Python orchestration scripts and the Telegraf collector configurations.

2 TECHNICAL FUNDAMENTALS

Before analysis of the performance differences between legacy and modern network management paradigms, it is essential to establish the technical foundation upon which this study is built. This section details the core protocols and virtualization tools that constitute a modern SP infrastructure.

The architecture of a scalable Internet Service Provider (ISP) backbone relies on the synergy between BGP and MPLS, for global routing and efficient internal service encapsulation. To simulate these protocols in a controlled yet realistic environment, this research employs Containerlab, a modern orchestration tool[2] for container-based networking labs [8].

Together, these technologies provide a high-fidelity platform to contrast the traditional pull-model of SNMP against the push-model of gNMI telemetry. The explanation of the mechanics of each component better quantifies the trade-offs between data granularity, failure detection latency, and the computational cost of observability.

2.1 Multiprotocol Label Switching

In traditional IP routing, every router in a path must perform a computationally intensive longest-prefix match against a large routing table for every incoming packet. This forwarding decision is made on a hop-by-hop basis, where each independent routing element must independently query its Routing Information Base (RIB) to determine the next hop. While this model provides an end-to-end delivery service that is independent of the underlying link-layer technology, it lacks the inherent ability to easily scale for advanced services.

However, as modern network demands evolved such seamless mobility, Virtual Private Networks (VPNs) and Quality-of-Service (QoS), the overhead for these features became architecturally complex. On this way, MPLS has a lighter forwarding plane because it is made only once on ingress node, where the IP destination is mapped to a fixed-length label in packet's header [9]. Within this domain, routers switch packets based on labels, avoiding RIB lookups.

2.1.1 MPLS Elements

Within an MPLS domain, routers use a label-forwarding table to establish a Label Switched Path (LSP) - a unidirectional tunnel toward a specific destination. The nodes along this path that perform label-swapping operations are known as Label Switching Routers (LSRs). Since the forwarding decision is done on entry (IP inspection), the LSP Ingress (or Ingress

Label Edge Router - LER) is the first element to map an incoming IP packet to an MPLS path, while the LSP Egress (or Egress LER) removes the label to deliver the original packet.

The mapping process could be defined on its intent as a Forwarding Equivalence Class (FEC). A FEC represents a group of IP packets that are managed in an identical manner - traverse the same path with the same forwarding treatment and assigned to the same label [9]. It can be a destination network prefix, a specific Quality of Service (QoS) requirement, or a Virtual Private Network (VPN) identifier.

The MPLS label has 32 bits (4 bytes) and is also called shim, because it is located between Layer 3 and 2 headers. Below a packet sniffer example on Figure 1:

Figure 1 – MPLS Label

```
1     Ethernet II, Src: MAC_P1_ge-2/0/3, Dst: MAC_P2_gi0/0/0/2
2         Type: MPLS label switched packet (0x8847)
3     MultiProtocol Label Switching Header
4         1111 0100 0010 0100 0010 .... .... .... = Label: 1000002
5         .... .... .... .... .... 000. .... .... = Traffic Class: 0
6         .... .... .... .... .... ...0 .... .... = Bottom of Stack: 1
7         .... .... .... .... .... .... 1111 1100 = MPLS TTL: 252
8     Internet Protocol Version 4, Src: 10.1.12.10, Dst: 10.2.34.30
9         Version: 4
10        Header Length: 20 bytes
11        Differentiated Services Field: 0x00
12        # IPv4 Packet Header Details and IPv4 Packet Payload
```

Source: [3].

The label is contained in 20 bits, followed by the Traffic Class (formerly known as Experimental Bits - EXP). It is functionally analogous to Differentiated Services Code Point (DSCP) on IPv4 header, which is used to manage traffic classes and prioritization during periods of congestion [3].

While the 20-bit label field supports over a million unique values, the Internet Engineering Task Force (IETF) strictly reserves the first 16 values (0-15) for special operations as shown in Table 1:

Table 1 – IETF Reserved Labels

LABEL VALUE	NAME	INSTRUCTION
0	IPv4 Explicit Null	Process TC bits before IPv4 routing
1	Router Alert	Send to control plane (MPLS ping and traceroute)
2	IPv6 Explicit Null	Preserve QoS marking for encapsulated IPv6 payloads
3	Implicit Null	Remove MPLS header for IP routing lookup

Source: Adapted from [10].

The Bottom of Stack flag (1 bit) is set to identify the last label until the immediate payload is the original IP packet, which follows a hierarchy that will be explained later.

In IP forwarding, a packet uses a field called Time to Live (TTL) to evaluate the number of hops it can pass through. On each node, it performs a unitary decrement on TTL value (8 bits). If it exceeds (reach zero), the packet gets discarded. In MPLS environment, the protocol must maintain the integrity of this mechanism while dealing with label hierarchy.

According to RFC 3031, the following procedures are mandatory for proper operation:

— TTL Ingress Propagation: On LSP Ingress, it initially loads the TTL field from the network layer header into the shim header.

— Hop Decrementing: At each LSR hop along the path, the TTL value is decremented to ensure the total number of hops traversed is accurately reflected.

— TTL Egress Propagation: The MPLS TTL value is copied back into the network layer header.

This consistent mechanism ensures that a packet emerging from a hierarchy of LSPs has the same TTL value it would have possessed if it had traversed the same sequence of routers via conventional hop-by-hop routing [9]. Beyond loop suppression, this behavior is vital for supporting network diagnostic tools like *traceroute* and for managing multicast scoping.

2.1.2 Forwarding Plane

The ingress Provider Edge (PE) is positioned at the boundary of the MPLS domain and performs two lookups:

— Forwarding Information Base (FIB) Lookup: It checks FIB to determine the FEC for the destination IP.

— Label Push: Once FEC is identified, the PE pushes a unique label onto the packet, determines the outgoing interface and the next-hop Provider (P) router.

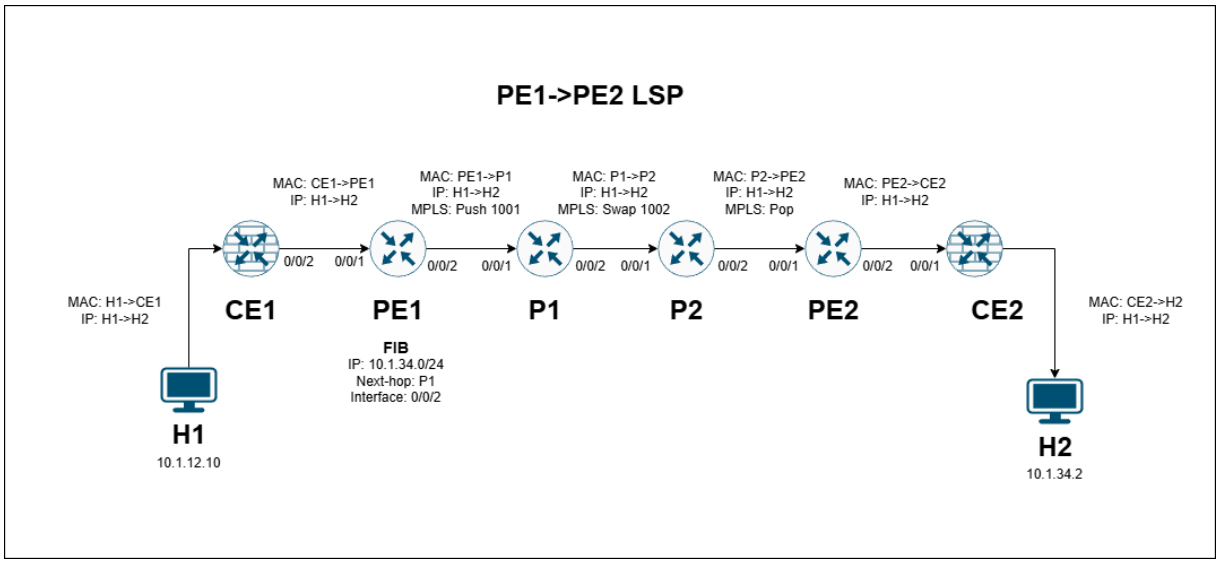
The P-routers constitute the core of the network and maintain the Label Forwarding Information Base (LFIB). At each hop, it ignores the IP header to lookup only at the incoming label to perform a swap operation toward the destination.

The penultimate LSR will perform Penultimate Hop Popping (PHP) on the label before forwarding the IP packet to the egress LSR [9]. During LSP formation, the egress PE node signals the penultimate LSR with a specific label—called the Implicit Null[2] (label value 3). Instead of encapsulating the packet with this label, the penultimate LSR interprets it as an instruction to pop the existing label stack before forwarding, delivering a native IP packet to the egress node. It was detailed under RFC 3032 ("MPLS Label Stack Encoding") [10].

This allows the egress node to immediately perform a standard IP lookup to deliver the packet to its final the destination or the customer-facing interface.

Below, the Figure 2 abstracts how it would work on a point-to-point topology:

Figure 2 – LSP Example



Source: Author.

1) H1 sends a packet destined to H2 through the gateway CE1.
2) CE1 identifies that for H2, it must send to provider domain to reach its IP address.

[2] In the late 1990s, router hardware lacked the processing capacity to perform both label and subsequent IP routing table lookup within a single forwarding cycle.

3) PE1 places the packet into PE1->PE2 LSP by pushing a new MPLS shim header between IPV4 and Ethernet headers of H1->H2 packet.
4) P1 receives, inspects and removes the original MPLS header. Then, it swaps to a locally scoped label towards P2.
5) P2 receives the labeled packet and consults its LFIB. Upon matching the incoming label with a pop operation, P2 removes the MPLS header and forwards the native IP packet to the egress router.
6) PE2 receives the IP packet without any MPLS headers. The usual RIB lookup based on the destination address is made towards H2.
7) H2 receives the IP packet.

While the forwarding plane efficiently executes these hop-by-hop operations, it is inherently reliant on a separate control plane to populate the FIB and LFIB. The labels utilized by PE1, P1, and P2 are not statically assigned; they are dynamically generated and distributed by signaling protocols such as the Label Distribution Protocol (LDP), which maps the underlying IGP routing topology to the label-switched paths.

2.1.3 Label Distribution Protocol

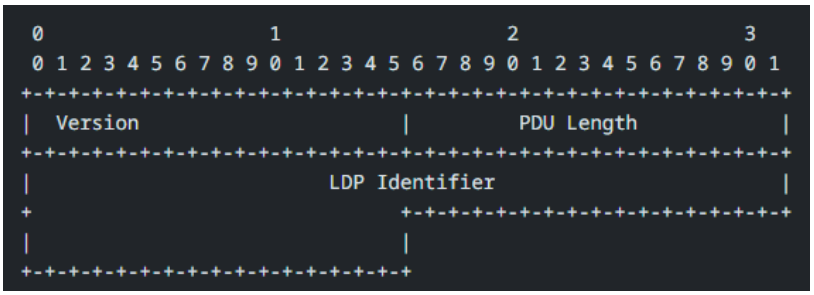
The forwarding plane relies on simple label-swapping operations, however the label decision for each FEC of an MPLS transit network is governed by the control plane signaling protocol. For complex traffic steering and manipulation, the Resource Reservation Protocol with Traffic Engineering (RSVP-TE) [11] is an option. It creates stateful, constraint-based tunnels (e.g., bandwidth reservations and Fast Reroute) for complex traffic steering.

However, on standard transit networks and for this study, the Label Distribution Protocol (LDP) serves as the primary signaling mechanism to map the IGP (Open Shortest Path First - OSPF or Intermediate System-to-Intermediate System - IS-IS) topology to LSPs based on shortest path calculation [10].

2.1.3.1 Architecture

LDP exchanges information by encapsulating messages within a specific Protocol Data Unit (PDU) structure. Unlike the IGP, which often runs directly over IP, LDP utilizes both User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) depending on the operation. Each LDP PDU is an LDP header followed by one or more messages. Below on Figure 3 is shown its format:

Figure 3 – LDP Header



Source: [10].

To describe each field:

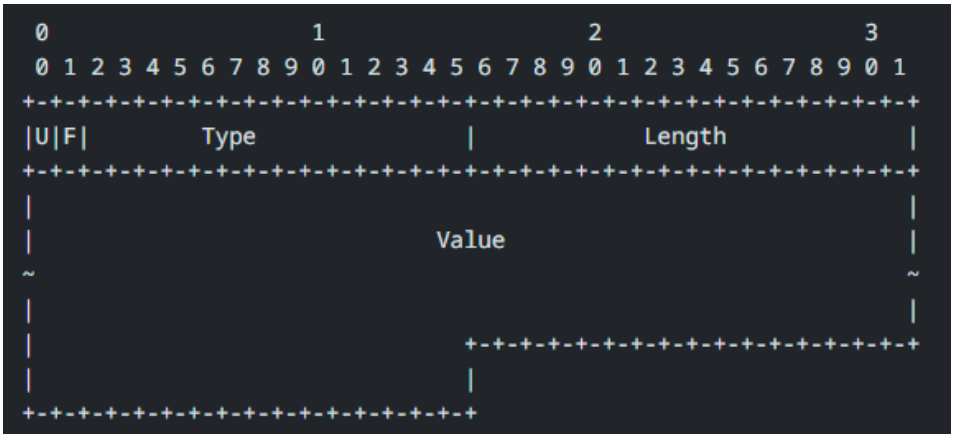
— Version: two octets containing LDP protocol version number (typically Version 1).

— PDU Length: two octets to specify the length excluding Version and PDU Length fields, the maximum allowable length is 4096 bytes, but it is negotiable during LDP session initialization.

— LDP Identifier: six octets that uniquely identifies the label space for LSR, with first four octets as a globally unique value and last two octets to identify a label space (per-platform or per-interface) within the LSR (often zero).

Each message within the PDU is structured using a Type-Length-Value (TLV) encoding scheme. The Figure 4Figure 4 – TLV Encoding above explains the scheme:

Figure 4 – TLV Encoding



Source: [10].

The message begins with an Unknown Bit (U-bit). If it is zero, a notification is returned to the originator and the message is ignored. Afterwards, the Forward Bit (F-bit) is evaluated

only if U-bit is set and data is to be processed. If it is zero, the message is not forwarded. These are fields for manipulation and decision-making, which allow TLVs to be propagated as opaque data for LSRs along the path even if they do not recognize it. Following these initial control flags, the header defines:

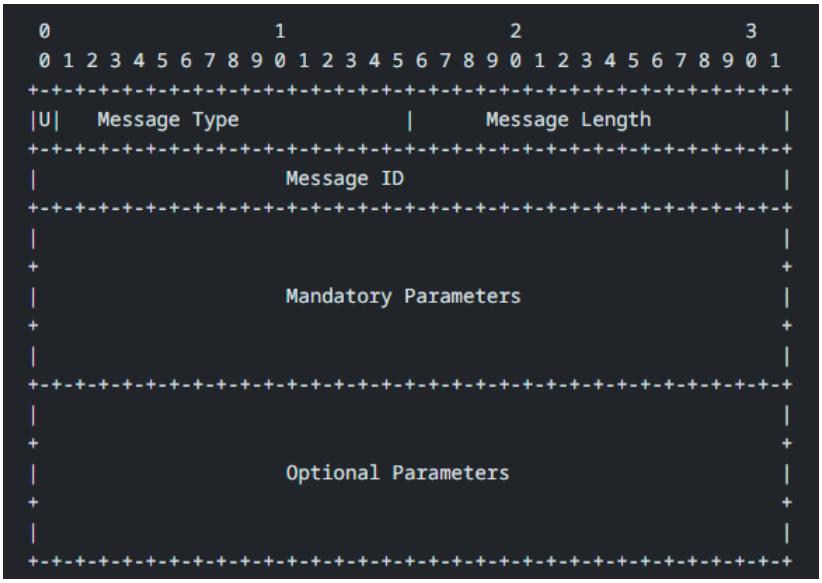
— Type: 14-bits unique identifier that specifies the exact nature of the parameter being transported (e.g., a Forwarding Equivalence Class, Hop Count, or IP Address).

— Length: 16-bit integer that explicitly declares the total size of the subsequent data payload in octets. It is crucial to inform the LSR exactly how many bytes to process.

— Value: actual variable-length data payload, because its exact dimensions are pre-declared by the Length field, the Value section can transport diverse routing attributes.

For LDP, the message has the following format as Figure 5 exhibits:

Figure 5 – LDP Message



Source: [10].

A Message Type, Message Length and a Message ID describe how Value should be interpreted followed by mandatory and optional parameters encoded as TLVs. This architecture makes LDP highly extensible but also creates complex data structures.

The protocol relies on four distinct categories of messages to maintain the LSPs:

— Discovery Messages: Used to announce and maintain the presence of an LSR in a network (e.g., Hello).

— Session Messages: Used to establish, maintain, and terminate sessions between LDP peers (e.g., Initialization, KeepAlive).

— Advertisement Messages: The core operational messages used to create, change, and delete label mappings for specific FECs. This includes Label Mapping, Label Request, Label Withdraw, and Label Release messages.
— Notification Messages: Used to provide advisory information and signal error conditions (e.g., Malformed PDU).

2.1.3.2 Discovery and Advertisement

Before routers can exchange labels, they must discover each other and establish a secure signaling session. An LSR actively discovers neighbors by periodically multicasting LDP Hello messages to the multicast address (224.0.0.2) over **UDP port 646**. These messages contain the transport address (usually the loopback IP) that the router intends to use for the TCP session.

Once a neighbor is discovered, the router with the higher transport IP address initiates a TCP connection to port 646 of the peer. Upon successful TCP handshake, the routers exchange Initialization messages to negotiate session parameters (e.g., protocol version, keepalive timers, label advertisement modes). If acceptable, KeepAlive messages are exchanged, and the session transitions to Operational state.

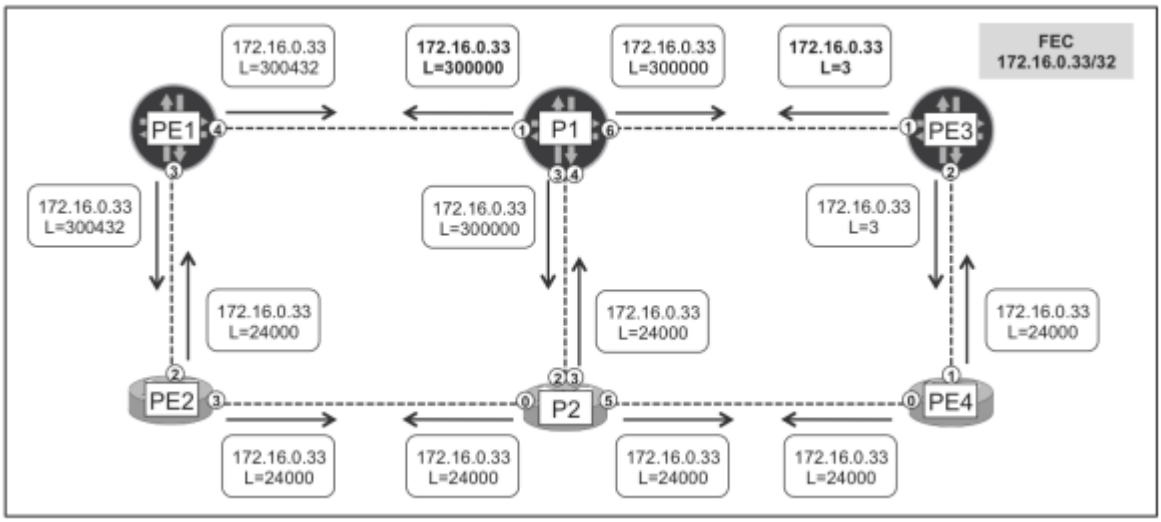
On operational state, the routers must distribute their label-to-FEC bindings. The protocol defines two primary advertisement modes that dictate when a router should advertise a label [12]:

Downstream Unsolicited: In this mode, an LSR actively distributes label bindings to its LDP peers for all routes in its routing table, without waiting for an explicit request. If a router knows how to reach a destination, it assigns a label and pushes that mapping to all neighbors. This is the default and most prevalent mode in modern Service Provider networks. It consumes more memory on the routers, as they must store labels for paths they might not currently use, but it drastically reduces convergence time during a failover, as backup labels are already pre-distributed and stored in the LFIB.

Downstream on Demand: In this mode, an LSR only distributes a label binding for a specific FEC when an upstream peer explicitly requests it via a Label Request message. While this conserves memory and control-plane bandwidth by strictly maintaining only actively used LSPs, it introduces significant latency during topology changes, as the routers must wait for a request-and-reply cycle before forwarding traffic.

To exemplify this, an LSP is constructed on Figure 6 across the transit core for loopback address 172.16.0.33/32 learned over IGP. The label distribution occurs in the reverse direction of data traffic flow:

Figure 6 – LSP FEC Creation



Source: [3].

The process initiates at egress PE3 router, which owns the 172.16.0.33/32 network. It instructs with a Label Mapping message to its upstream neighbors to forward traffic to this FEC using Implicit Null label for PHP operation. Upon receiving this mapping, the routers update their LFIB for a Pop operation on the specific FEC.

Since it is managed under downstream unsolicited messages, P1 and PE4 routers generate their own locally significant label and distribute a Label Mapping message again to its LDP neighbors (label 300000 and 240000 respectively). Simultaneously, remaining nodes perform the identical mechanism.

The signaling cycle concludes at the ingress LER, such as PE1. When PE1 receives the Label Mapping from P1 and PE2, it updates its FIB. For any incoming native IP packet destined for 172.16.0.33/32, it must travel under the unidirectional LSP (packet encapsulation with label 300000 or 240000). The LSP is now fully operational.

The theoretical mechanics of the LDP and the forwarding plane guarantee efficient transit routing, maintaining visibility into this core presents significant operational challenges. For example, when a physical link fails or an IGP topology change occurs, the MPLS control plane must rapidly withdraw and re-advertise label bindings. To accurately evaluate control plane convergence and Failure Detection Time, metrics such as LDP Session State, LFIB Churn (label bindings modifications), TCP-based LDP sessions transitions and interface operational status are critical [13].

2.2 Border Gateway Protocol

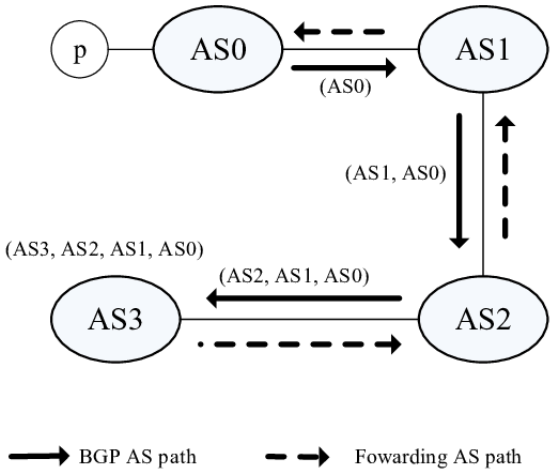
Core domain established to provide efficient hop-by-hop signaling transport, the network architecture requires a mechanism to ingest, manage, and distribute external customer routes. In a modern Service Provider environment, BGP is strictly segmented based on its peering relationships and operational scope [14], ensuring that external instability does not compromise the internal transit core.

The main purpose of a BGP speaking system is to exchange Network Layer Reachability Information (NLRI)[3] between nodes, essentially a routing prefix to a destination.

2.2.1 BGP Elements

This protocol is fundamentally a path vector protocol designed to carry routing information under BGP Messages between ASs. This designation derives from the fact that BGP routing information carries a strict sequence of AS numbers that a network prefix has traversed. This path information is the primary mechanism utilized by the protocol to enable loop prevention across the Internet [15], called AS_PATH attribute. Below on Figure 7 is visible the route advertisement between ASs and AS_PATH operation:

Figure 7 – BGP AS Path



Source: [16].

The AS_PATH attribute operates as an ordered sequence of AS numbers. As a BGP route advertisement propagates across different administrative domains, each participating autonomous domain prepends its own AS number to this list before forwarding.

[3] The Network Layer Reachability Information structure is highly extensible and designed to carry routing information for diverse address families beyond standard IPv4.

As illustrated on Figure 7 – BGP AS Path, the route for prefix *p* originates in AS0. As the advertisement travels outward, the AS_PATH expands sequentially becoming (AS1, AS0) at AS1, and eventually (AS3, AS2, AS1, AS0) by the time it reaches AS3. Beyond defining the routing path, this attribute serves as primary loop prevention mechanism: if a BGP router receives a message and detects its own AS number already present within the AS_PATH sequence, it immediately discards the route, assuming a routing loop has occurred.

For information exchange, routers running the BGP process must form a reliable transport connection under TCP port 179. By utilizing TCP, BGP offloads all transport reliability mechanisms, such as packet acknowledgment, sequencing, and retransmission, directly to the transport layer. This architectural decision simplifies BGP by removing the need to build complex reliability features into the routing protocol itself.

Two BGP speakers that successfully form a TCP connection are known as neighbors or peers [15]. The lifecycle and operational state of this peering relationship are governed by four distinct message types:

- OPEN Messages: Upon establishing the TCP connection, peers exchange OPEN messages to negotiate connection parameters. This includes communicating the BGP version number, the router ID, and supported capabilities.
- UPDATE Messages: During initial session establishment, peers exchange all candidate BGP routes. However, once this initial sync is complete, BGP sends only incremental updates as network information changes. It contains tuples indicating reachable destinations, their associated path attributes (such as AS_PATH and preference degree known as LOCAL_PREF attribute), and lists of any previously advertised routes that have become unreachable and must be explicitly withdrawn (WITHDRAWAL Message).
- KEEPALIVE Messages: In a steady-state network where no routing changes occur, peers exchange periodic KEEPALIVE messages to ensure the TCP connection remains active.
- NOTIFICATION Messages: BGP includes a graceful close mechanism to handle errors. If a configuration mismatch, incompatibility, or manual operator intervention occurs, a NOTIFICATION error message is transmitted. This ensures that all outstanding messages are delivered before the TCP session is definitively closed, preventing peers from wasting resources attempting to maintain a broken state.

For route selection, BGP selects the optimal path to a destination network by passing candidate routes through a deterministic, multi-step tie-breaking algorithm based on path attributes. When a BGP speaker receives multiple updates for the same NLRI, it sequentially

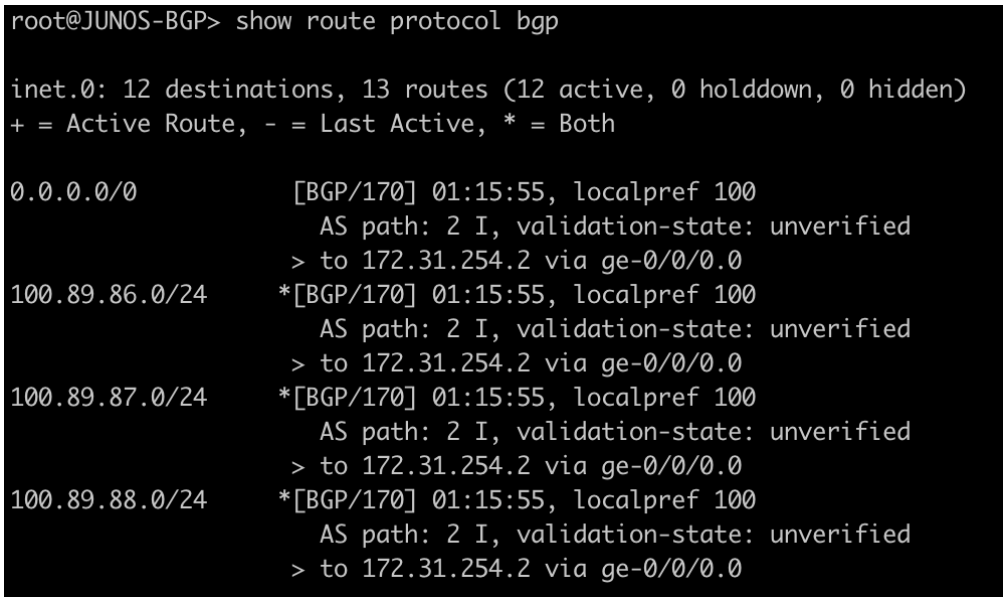
evaluates their attributes to select the single best route for installation into a unique table called Loc-RIB [14].

The standard selection order prioritizes routes with the highest Local Preference (an internal domain metric), followed by locally originated routes, and then the shortest AS_PATH length. If a tie persists, the algorithm sequentially prefers the lowest Origin code (IGP – code I - over Incomplete – code ? - for redistributed routes), the lowest Multi-Exit Discriminator (MED) for routes from the same neighboring AS, external (eBGP) paths over internal (iBGP) paths, and finally, the path with the lowest underlying IGP metric to the BGP next-hop address. Furthermore, each attribute is categorized as how BGP speakers must handle them:

- Well-known Mandatory: Must be recognized by all routers and included in all updates (e.g., AS_PATH, NEXT_HOP, ORIGIN).
- Well-known Discretionary: Recognized by all routers, but inclusion is optional (e.g., LOCAL_PREF).
- Optional Nontransitive: May not be recognized, and is not passed to other ASes (e.g., MED).

To exhibit how attributes are shown on a router CLI, on Figure 8 it is done under Junos system:

Figure 8 – BGP Routes on Junos



Source: [17].

As seen above, MED is not configured because it is not mandatory. For a relation between attribute and route selection order, the Table 2 is created:

Table 2 – BGP Route Selection Sequence and Attributes

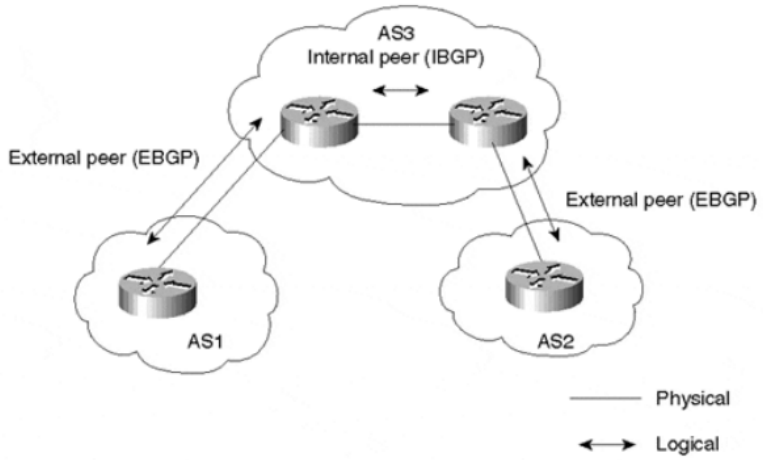
SELECTION ORDER	ATTRIBUTE NAME	TYPE CODE ON BGP HEADER	ATTRIBUTE CATEGORY
Prerequisite	NEXT_HOP	3	Well-known Mandatory
1st	LOCAL_PREF	5	Well-known Discretionary
2rd	AS_PATH	2	Well-known Mandatory
3rd	ORIGIN	1	Well-known Mandatory
4th	MULTI_EXIT_DISC	4	Optional Nontransitive

Source: Adapted from [15].

2.2.2 Internal BGP

Beyond its primary role in establishing a loop-free interdomain routing topology, BGP is equally important within the AS domain. When a peering session is established between two routers residing within the same administrative scope, the protocol is designated as Internal BGP (iBGP). In this context, the objective of iBGP is to seamlessly distribute external destination reachability information - learned at the network edge by External BGP (eBGP) peers - across the internal routing infrastructure. This relationship is described on Figure 9:

Figure 9 – iBGP and eBGP Topology



Source: [15].

The necessity of employing iBGP rather than relying on an Interior Gateway Protocol (IGP), such as OSPF or IS-IS, to carry this external information resides from fundamental

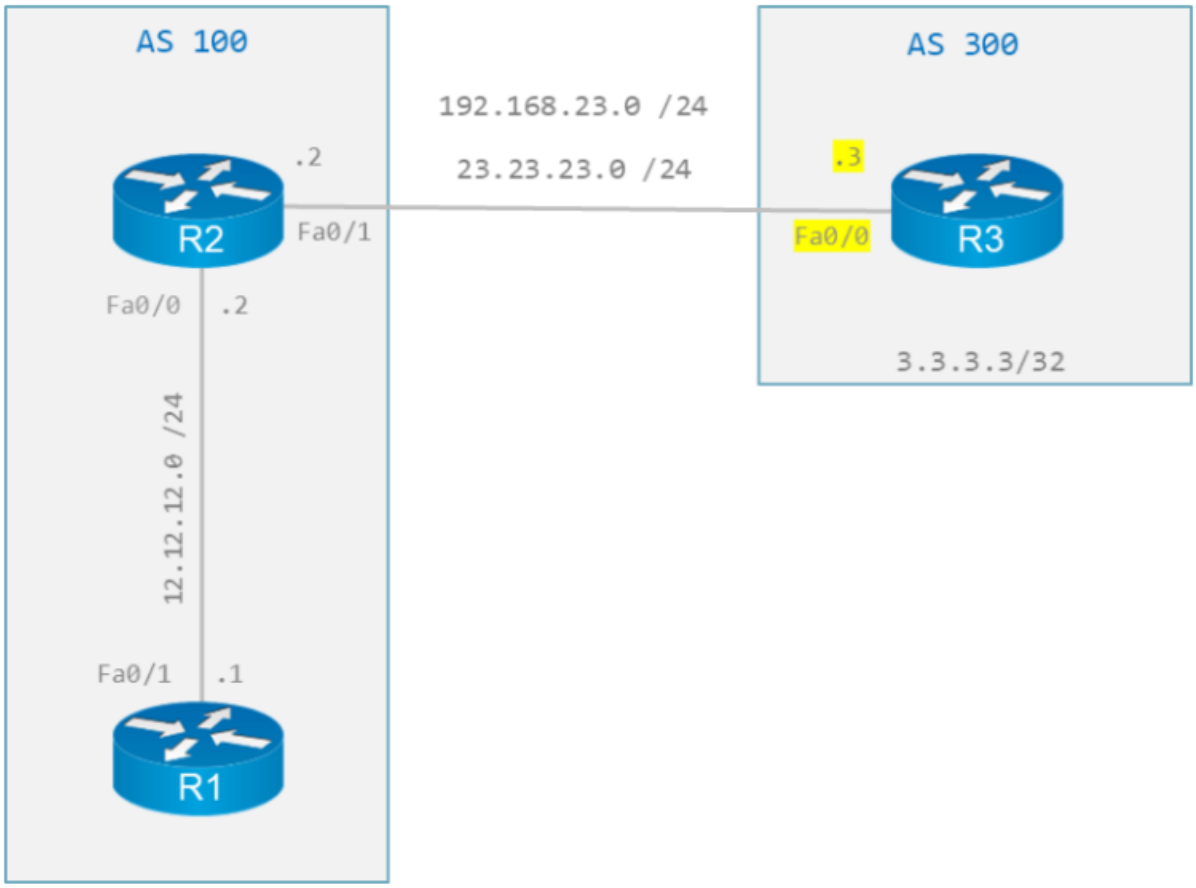
architectural differences in protocol scale and design. IGPs are optimized to maintain a highly dynamic link-state database of the internal infrastructure based on a computationally intensive algorithm called Dijkstra's Shortest Path First (SPF) to calculate optimal paths.

The redistribution of massive volumes of external customer prefixes or global Internet routes into an IGP would exponentially inflate this database, leading to severe CPU exhaustion and control plane instability during periods of route churn. Conversely, iBGP is designed as an overlay operating over a reliable TCP connection. It efficiently transports hundreds of thousands of prefixes across the backbone without continuous recalculations of the internal topology. This segregation restricts the IGP to its primary purpose: ensuring foundational reachability between the physical nodes and loopback interfaces of the provider network, while iBGP handles the customer routing.

A defining characteristic of iBGP is its internal loop prevention mechanism, known as the iBGP split-horizon rule. Unlike eBGP, which relies on the appending of AS numbers to the AS_PATH attribute to detect routing loops, routers within the same domain share the identical AS number, rendering the standard AS_PATH check ineffective.

Consequently, the protocol mandates that a route learned from one iBGP peer must never be advertised to another iBGP peer. This strict operational constraint traditionally necessitates a full-mesh peering topology, requiring a direct TCP connection between every BGP speaker in the network. Furthermore, iBGP does not modify the NEXT_HOP attribute of a propagated route by default [15]. The route propagation is illustrated on Figure 10 below:

Figure 10 – iBGP NEXT_HOP Operation



Source: [18].

In this scenario, R3 in AS 300 originates the prefix 3.3.3.3/32 and advertises it to R2 in AS 100 via their eBGP session. In this UPDATE message, the NEXT_HOP is recorded as R3's directly connected interface IP address (23.23.23.3). When R2 subsequently distributes this external prefix internally to R1 via iBGP, it passes the route with the exact same NEXT_HOP value (23.23.23.3). If it is not redistributed on any IGP, this route will not be active on R1 due to no match on its RIB.

In a Service Provider topology, the ingress edge router must actively manipulate this attribute (typically via a next-hop-self policy) before advertising the external route to the iBGP mesh. This ensures that the destination is resolved to the edge router's loopback address, allowing the internal routers to recursively map the BGP payload to the underlying IGP. A topology exchange for this is the use of a route distribution within iBGP domain called Route Reflector, a concept that will be discussed later.

2.2.3 External BGP

External BGP (eBGP) functions as the definitive gateway between completely distinct administrative domains. The operational mechanics of eBGP differ significantly from its internal counterpart, because eBGP operates across trust point-to-point boundaries. By default, eBGP packets are generated with an IP TTL value of 1. This ensures that the peering session can only be established across a direct, physical link, preventing malicious or accidental session establishment across multiple unauthorized routing hops.

In ISP architecture, the eBGP edge is the absolute enforcement point for routing policy and traffic engineering. When an eBGP edge router receives a prefix from an external entity, it modifies the NEXT_HOP attribute to its own interface IP address before propagating the route into the iBGP core. It is also at this strict boundary that engineers apply ingress and egress routing policies—manipulating attributes like LOCAL_PREF to dictate outbound traffic flows or sending custom MED values to influence how external ASs route traffic back into the provider network.

The most fundamental application of BGP policy is route filtering. By default, an eBGP peering session exchanges all known active routes. In the context of the global Internet, receiving the full BGP routing table (which exceeds one million prefixes) can quickly exhaust the memory and CPU resources of an edge router. To establish control, administrators implement strict filtering mechanisms, primarily using Prefix-Lists and AS_PATH regular expressions. For example, a policy is created on Junos system to allow distribution of a static route into BGP as the following Figure 11:

Figure 11 – BGP Policy Configuration on Junos

```
      set policy-options policy-statement send-static term 1 from protocol static
      set policy-options policy-statement send-static term 1 from route-filter
172.16.0.0/16 orlonger
      set policy-options policy-statement send-static term 1 then accept
      set routing-options static route 172.16.6.0/24 reject
      set routing-options router-id 172.16.1.1
      set routing-options autonomous-system 17
      set protocols bgp group external-peer type external
      set protocols bgp group external-peer peer-as 200
      set protocols bgp group external-peer export send-static
      set protocols bgp group external-peer neighbor 10.0.0.2
```

Source: Adapted from [2].

The BGP process does not simply receive a route and immediately install it into the router's hardware forwarding table. Instead, every prefix exchanged during a peering session must traverse three distinct components of the RIB: the Adj-RIB-In, the Loc-RIB, and the Adj-RIB-Out.

When an eBGP speaker receives an UPDATE Message from an external transit provider or customer, the raw routing data is first deposited into the Adj-RIB-In (Adjacent RIB Inbound). This database represents the absolute perspective of the external neighbor without any previous modifications. It contains every prefix the peer advertised, complete with their original path attributes, exactly as they were received over the TCP socket.

The transition from the Adj-RIB-In to the central Loc-RIB (Local RIB) is where the router's inbound policy engine is executed. The eBGP speaker sequentially processes the raw routes against configured inbound prefix-lists and route-maps. Prefixes that are explicitly denied (such as bogon IP space or unauthorized customer subnets) are discarded at this boundary as illustrated on Figure 11 – BGP Policy Configuration on Junos for prefix 172.16.6.0/24.

The permitted routes then pass through the BGP best-path selection algorithm and the optimal path for each destination is selected and installed into Loc-RIB and further to FIB. To query the FIB for a specific destination, the Figure 12 demonstrates it to a /32 prefix:

Figure 12 – FIB Query on Junos

```
user@host> show route forwarding-table destination 66.0.0.1
Internet:
Destination        Type RtRef Next hop           Type Index NhRef Netif
66.0.0.1/32        user     0                    indr 2097159     3
                                                 ulst 2097156     2
                              5.1.0.2           ucst   574     1 et-6/0/0.1
                              5.2.0.2           ucst   575     1 et-6/0/0.2
```

Source: [2].

It displays a query of the Junos FIB for the destination prefix 66.0.0.1/32 using the *show route forwarding-table* command. This specific output demonstrates a hierarchical resolution involving Equal-Cost Multipath (ECMP) load balancing. The first field defines the Type regarding the origin or classification of the route entry. For *user* designation, this route was

learned and installed by the route protocol process (rpd), distinguishing from system-level or permanent interface routes (perm).

The next field Route Reference (RtRef) indicates how many distinct routing table entries for this route are on forwarding structure followed by the Next hop IP address for the downstream gateway.

Further to the right, the hardware-level resolution is detailed on second Type column for structural classification of the next-hop in hardware memory: process beings with an indirect pointer (*indr*) that is mapped to a Unicast List (*ulst*) to facilitate load balancing for two distinct physical Unicast (*ucst*) paths. Each of these structural elements is assigned a unique Index for hardware memory pointer used by the ASIC to traverse the forwarding tree. The Next-Hop Reference (NhRef) field monitors how many internal routing structures rely on that index.

Finally, the Network Interface (NetIf) field displays the definitively resolved physical or logical egress interface out of which the packet will be transmitted.

2.2.4 Route Reflector

As previously established, the iBGP split-horizon rule dictates that a route learned from one internal peer cannot be advertised to another. This restriction traditionally mandates a full-mesh peering topology to ensure complete routing reachability across the domain. However, in full mesh, the number of required TCP sessions scales on the following relation:

$$p = \frac{N \cdot (N-1)}{2} \tag{1}$$

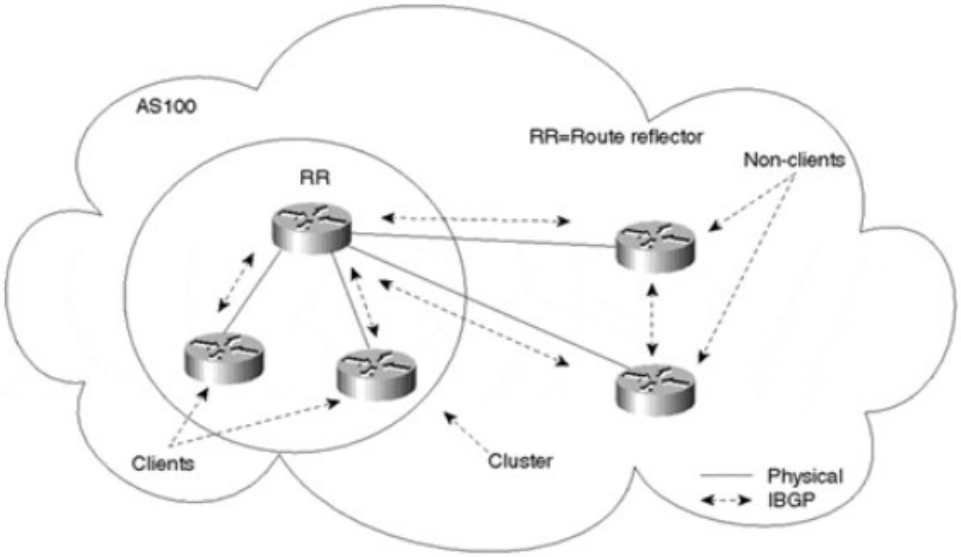
Where p is the required BGP internal sessions and N is the total number of nodes on the topology. If we add a node into the full mesh:

$$p = \frac{(N+1) \cdot (N+1-1)}{2} = \frac{N^2 + N}{2} = \frac{N \cdot (N-1)}{2} + N \tag{2}$$

Consequently, each node added increases the number of iBGP peerings to a factor of N. In large-scale networks containing hundreds of internal routers, maintaining this exponential number of concurrent peering sessions becomes administratively unmanageable and causes severe control-plane CPU exhaustion.

To resolve this architectural limitation, the IETF standardized the BGP Route Reflector (RR) on RFC4456 [19]. Instead of requiring every PE router to peer with each other, the edge routers are designated as RR Clients under a concentration router for internal BGP sessions. Furthermore, the relation between Client and Non-Client is used to overcome the constraint that a route learned via iBGP cannot be advertised to another iBGP speaker alongside special fields: ORIGINATOR_ID and CLUSTER_LIST [15]. Figure 13 illustrates each component:

Figure 13 – RR Components



Source: [15].

Within RR architecture, iBGP peers are strictly divided into two classifications: clients and non-clients. The RR router along with its designated clients, constitutes a logical routing domain known as a cluster. Any router peering outside of this specific grouping is classified as a non-client. Because non-clients function as conventional iBGP speakers, they remain bound by standard split-horizon rules.

Consequently, they must maintain a full peering mesh with one another and with the RR, though they are completely relieved of the burden of peering directly with the cluster's clients. Conversely, clients are architecturally restricted and must not establish iBGP sessions with any internal routers beyond their designated cluster.

When an RR receives a route from an IBGP peer, it first executes the standard BGP decision process to determine the single best path [19]. Once the optimal path is identified, the RR performs the reflection process. In this context, reflection is defined as the propagation of the route to other internal peers without modifying its fundamental path attributes (such as

NEXT_HOP, LOCAL_PREF, or AS_PATH) to ensure that it acts purely as a transparent control-plane entity. This mechanism depends on the hierarchical designation of the peer from which the route was received:

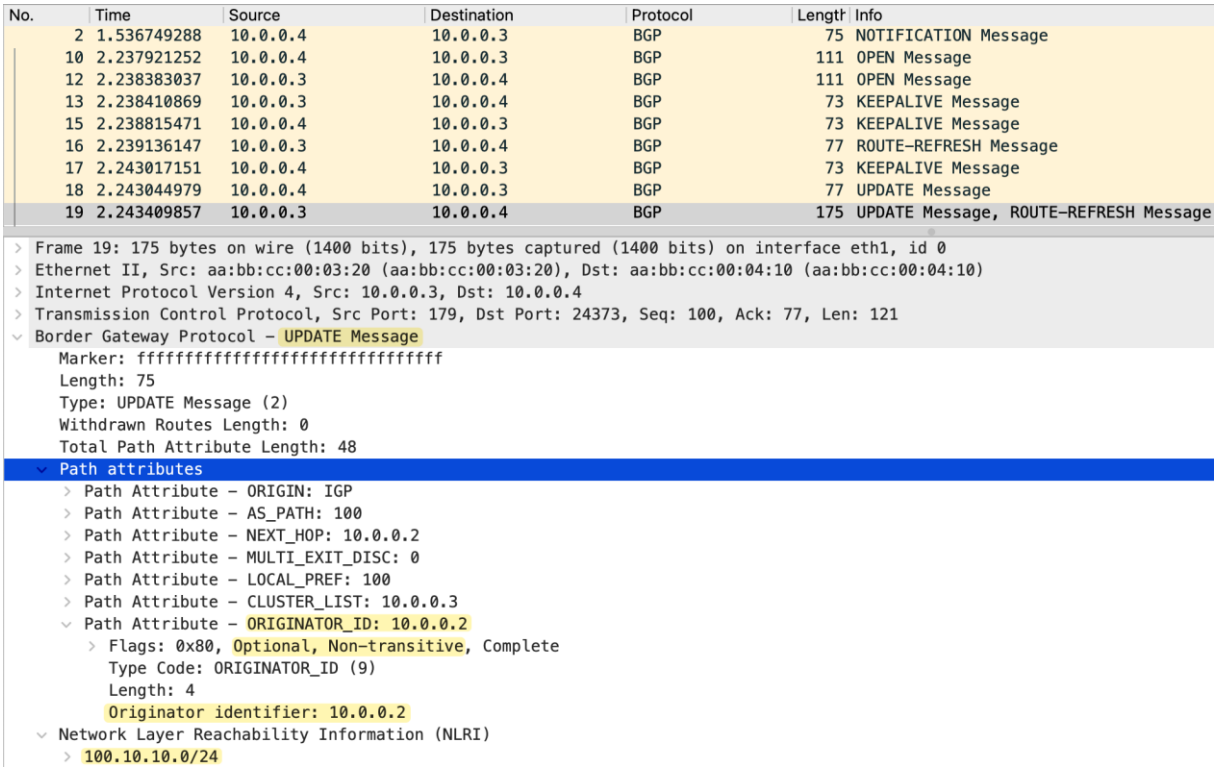
— From Non-Client peer: The RR reflects the route exclusively to its configured clients.

— From Client peer: The RR reflects the route to all other Clients within its cluster, as well as to all non-client peers.

— From eBGP peer: Routes received from an external domain are reflected to all internal peers, encompassing both clients and non-clients (standard BGP rule).

To safely execute this reflection without inducing control-plane routing loops, the RR is mandated to introduce two specialized, non-transitive attributes to the reflected BGP UPDATE messages [2]. For router ID identification of the BGP neighbor that originally advertised the route, the 4-byte attributed called ORIGINATOR_ID is attached only on first reflection. It allows that routes are discarded when the originator receives it. For packet structure, the Figure 14 shows it on a packet sniffer software:

Figure 14 – ORIGINATOR_ID on BGP UPDATE Message



Source: [20].

In this scenario, when router addressed with 10.0.0.2 router ID receives 100.10.10.0/24 route, it will discard to neutralize the loop.

To logically create the cluster architecture, each RR is assigned to a cluster ID. It allows multiple RR topologies and routes retention. A network design may dictate deploying multiple redundant RRs within the same cluster to serve a shared set of clients, or by contrast, segregating the backbone into distinct clusters with different RRs to distribute the control-plane processing load.

When a RR reflects a route, it adds its own cluster ID into the CLUSTER_LIST attribute. Similar to AS_PATH, if an RR is participant of a cluster already included on CLUSTER_LIST, it discards the route. Normally, the cluster ID is equal to the router ID of the RR router. On Figure 14 – ORIGINATOR_ID on BGP UPDATE Message, it illustrates that the RR under source IP 10.0.0.3 (which is used as cluster ID) prepended it on CLUSTER_LIST attribute to contain its identifier.

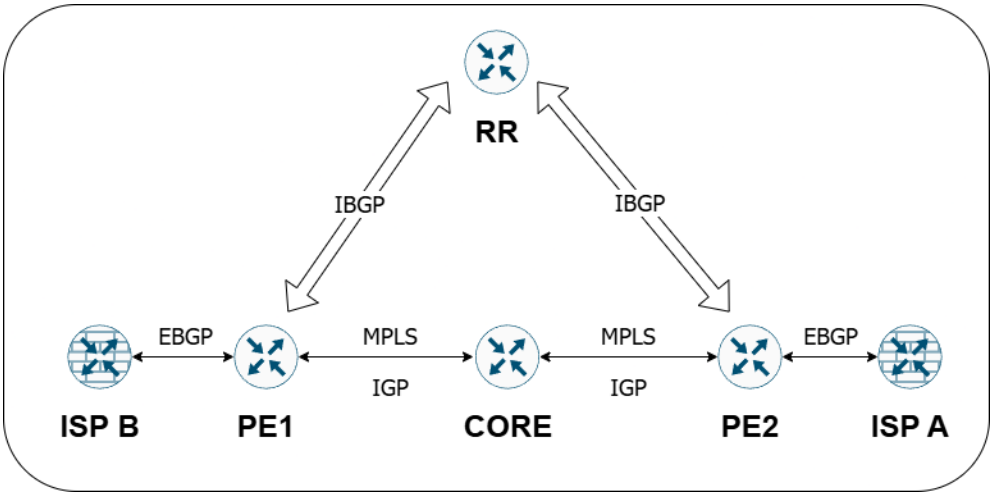
2.2.4.1 BGP-Free Core

On control-plane once addressed with Route Reflectors to overcome full-mesh topology, the architectural focus must shift to the data plane. To prevent internal backbone routers from being overwhelmed by the volume of global Internet routes due to iBGP messages, a paradigm known as BGP-Free Core is presented.

The concept is to compartmentalize routing responsibilities, decoupling the external BGP control plane from the internal transit data plane. It relies on iBGP peerings strictly on PE routers as demarcation point between external entities (customer or transit providers), while they maintain eBGP sessions, enforce routing policies and hold the full BGP RIB.

On core, the P-routers focus on MPLS label switching and IGP topology calculations. They are relieved to run BGP, since the routing decisions are preemptively resolved on Ingress PE with recursive route lookup and push label operation. For illustration, Figure 15 is shown:

Figure 15 – BGP Free Core topology



Source: Author.

This figure demonstrates the end-to-end encapsulation and forwarding process and BGP unawareness on core network. When an incoming IP packet destined for an external network arrives at the ingress PE, the router consults its Loc-RIB. It resolves the BGP NEXT_HOP attribute, encapsulates the native IP payload with an MPLS transport label corresponding to the egress PE's internal loopback address, and forwards it into the backbone. As the frame traverses the core, the intermediate P-routers perform rapid label swapping based exclusively on the outermost MPLS header. Upon reaching the egress PE, the transport label is popped, and the unencapsulated IP packet is routed toward its final external destination.

Consequently, the internal backbone is deliberately isolated from the global routing table and there is no BGP state on P-routers. The observability platform must focus on the crucial intersection between the external BGP control plane (on the PEs) and the internal MPLS/IGP transport plane, such as SPF calculations frequency, ECMP distribution, IGP-to-LDP synchronization (convergence to label bindings) and unresolvable next-hops.

2.3 Simple Network Management Protocol

The complex control and data plane architectures once established and required for inter-domain routing, it is imperative to address how these distributed systems are monitored and maintained. For over three decades, the standard for network observability and element management has been the Simple Network Management Protocol (SNMP). Initially developed

in the late 1980s[4] as a temporary mechanism to manage IP networks, SNMP achieved universal adoption due to its relative simplicity and hardware-agnostic design, becoming deeply embedded in nearly every enterprise and Service Provider network globally.

The architecture is fundamentally centralized and relies on client-server model. The three components are:

— Network Management System: server responsible for monitoring applications, collecting data and controlling network elements.
— SNMP Agent: software module within the network element that collects and translates operational data.
— Management Information Base: hierarchical database under a strict tree structure to define managed objects within a device, each metric is mathematically addressed using an Object Identifier (OID).

This model operates predominantly on a pull-based telemetry model. The NMS must actively query the agent using specific commands on UDP port 161. For normalization, communication between each element is defined on a set of Protocol Data Units (PDUs). Each message type defines the transactional capabilities between the NMS and the localized agent.

The fundamental polling message is a GET Request, which NMS queries the agent for the value of one or more specific OID. For example, on Figure 16 there is query for inbound traffic on octets (bytes):

Figure 16 – Junos SNMP GET Request

```
show snmp mib get ifInOctets.1599

ifInOctets.1599 = 786227836

show snmp mib get decimal 1.3.6.1.2.1.2.2.1.10.1600

ifInOctets.1600 = 1224709716
```

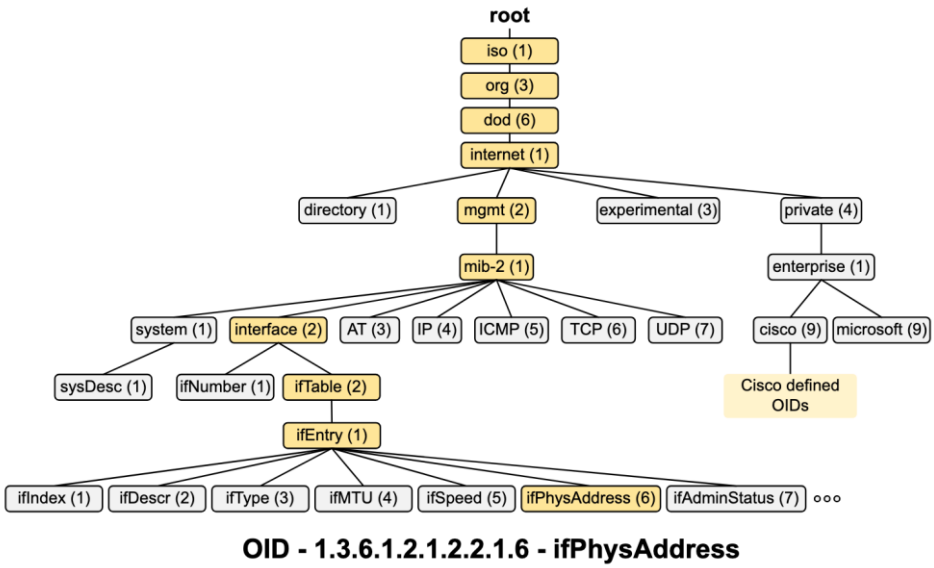
Source: Author.

For identification, the OID is built under a decimal number format known as dot-decimal notation, which maps to a strictly hierarchical, tree-like namespace managed by the

[4] SNMP was originally introduced merely as a temporary, short-term measure. However, the architectural complexity of its competitors ultimately resulted it as the permanent, industry-wide standard.

IANA [21]. Below on Figure 17 is shown a part of MIB tree on decimal representation for physical address:

Figure 17 – MIB Structure



Source: [26].

As illustrated in Figure 16, the NMS can query the agent using either the human-readable string (ifInOctets) or its exact numeric equivalent (1.3.6.1.2.1.2.2.1.10). The last indexes appended to the end of a tabular OID - such as the .1599 and .1600 – are known as instance identifiers (or IfIndex on interfaces group).

A message to query the next sequential OID in the hierarchical MIB tree is called GETNEXT. This is a traditional mechanism to dynamically request information through variable-length tables (interface index or routing table). On Figure 18, it is queried as “walk” on all routing table entries (for this example only one is shown):

Figure 18 – Junos SNMP GET Next

```
show snmp mib walk inetCidrRouteTable #or 1.3.6.1.2.1.4.24.4
inetCidrRouteIfIndex.1.4.0.0.0.0.0.0.2.0.0.0.0 = 0
```

Source: Author.

To explain each octet on this entry from left to right (1.4.0.0.0.0.0.2.0.0.0.0 = 0) [22]:

— 1: **Destination Type** indicates IPv4 (1 = IPv4, 2 = IPv6).

— 4: **Destination Length** indicates the IP address is 4 bytes long.

- 0.0.0.0: **Destination Prefix** indicates the actual network address (0.0.0.0).
- 0: **Prefix Length (CIDR)** indicates the subnet mask is /0.
- 2: **Policy Length** indicates the routing policy identifier is 2 digits long.
- 0.0: **Policy OID** indicates the standard SNMP identifier 0.0 (known as zeroDotZero), meaning no specific routing policy is attached to this entry.
- 0: **Next-Hop Type** indicates the address type of the next-hop (0 = Unknown, 1 = IPv4).
- 0: **Next-Hop Length** indicates he next-hop address is 0 bytes long (meaning there is no IP address for the next-hop).
- Value 0: **IfIndex** is 0, it means the route is not bounded to a physical or logical interface and should discard/reject forwarding packets.

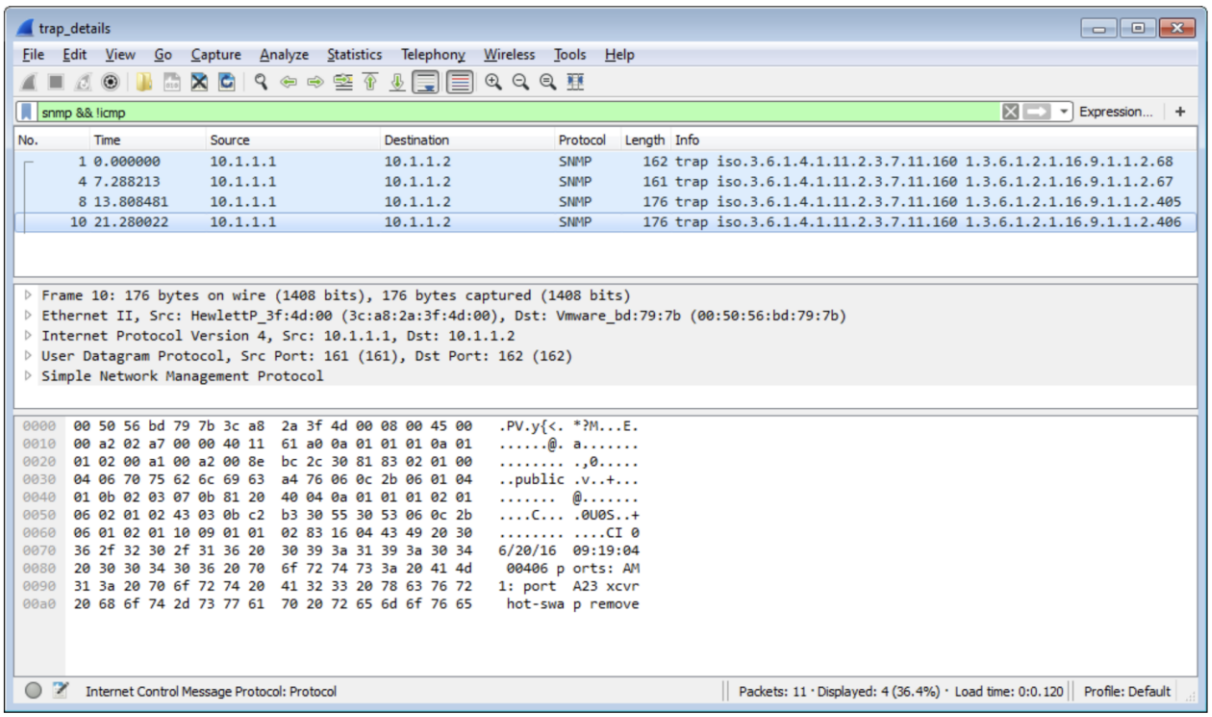
While this specific SNMP request successfully identifies a blackhole route, it exposes a critical architectural vulnerability within the protocol. The fundamental danger of the GETNEXT operation lies in its iterative, sequential design on a MIB tree and querying big data structures – such as global BGP routing table.

Consequently, it generates an overwhelming volume of transactional requests and rapidly exhausts the router's control-plane CPU, risking the delayed processing of vital protocol keepalives and potentially triggering routing instability across the node. To mitigate it, the IETF subsequently introduced a dedicated block-retrieval message type in later versions of the protocol [23], which will be examined in detail in the following section.

The primary administrative control mechanism on SNMP is managed by SET Request. On server solicitation, it transmits this message to modify a configuration variable or trigger an action on the network element, provided the manager possesses the necessary read-write authorization. However, the limitation of transactional integrity and automatic rollbacks that leads to unpredictable failure prevented its widespread adoption in production environments, driving the industry toward more robust protocols like NETCONF.

Conversely, the TRAP message fundamentally alters the standard pull-based polling model of SNMP by introducing an asynchronous, event-driven notification mechanism. Unlike traditional queries initiated by the NMS, a TRAP is an unsolicited PDU generated proactively by the agent to immediately alert the central manager of critical network state changes, such as a physical link failure, a BGP peer transition, or a threshold violation. Figure 19 illustrates a message for a removed transceiver

Figure 19 – SNMP TRAP Message on Wireshark



Source: [40].

As shown, TRAP alerts uniquely target UDP port 162 on the central management server, fundamentally diverging from standard SNMP polling which targets UDP port 161 on the local device [24]. The decoded hexadecimal payload at the bottom of the trace reveals the exact human-readable trigger for this specific trap: a physical hardware alteration (port A23 xcvr hot-swap remove). Moreover, the router's SNMP agent proactively constructed a trap containing specific OID (visible in the Info column as iso.3.6.1.4.1.11...), binding the alert directly to the affected hardware component.

By design, the first version of SNMP prioritized simplicity over robust control mechanisms. As the protocol matured, subsequent versions systematically addressed these early architectural vulnerabilities. Table 3 summarizes the five core message types that comprised this original standard:

Table 3 – SNMPv1 Messages

MESSAGE TYPE	DIRECTION	FUNCTIONAL DESCRIPTION
GET Request	Manager to Agent	Query the current value of one or more explicitly identified OIDs.
GETNEXT Request	Manager to Agent	An iterative and sequential query used to retrieve the OID immediately following the specified identifier
SET Request	Manager to Agent	The administrative control message utilized to modify device configurations or trigger operational states.
TRAP	Agent to Manager	An asynchronous, unacknowledged notification generated proactively to alert the management platform of an event or threshold violation.

Source: Author.

2.3.1 SNMPv2

To address the performance bottlenecks and reliability limitations inherent in the original architecture, the IETF introduced a new revision known as SNMPv2 in 1998. [25]. While preserving the foundational agent-manager polling methodology, this iteration was explicitly developed to enhance data retrieval efficiency and asynchronous alerting in complex network environments. The most critical operational enhancements were the introduction of the GETBULK Request message, which mitigated the control-plane processing exhaustion caused by iterative table traversing. We can simulate it on command line with *snmpwalk* and *snmlbulkwalk*, but it needs to explicitly inform the **community string** as seen in Figure 20:

Figure 20 – SNMP CLI for GETNEXT and GETBULK

```
snmpwalk -v2c -c public 10.1.1.1 SNMPv2-MIB::snmp #one entry per iteration
snmpbulkwalk -v2c -c public -Cr10 10.1.1.1 SNMPv2-MIB::snmp #10 entries per iteration
```

Source: Author.

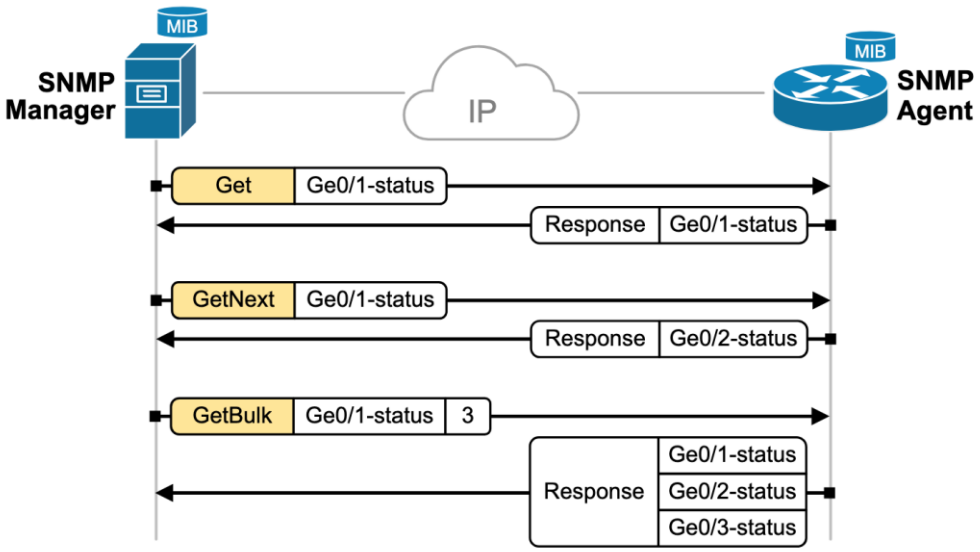
For security, community strings were developed in SNMPv1 as a simple community-based access control mechanism, and they were later reused in SNMPv2. It acts as a shared

plaintext password between the SNMP manager and the SNMP agent (public in this example). When the manager sends a request, it includes the community string in the SNMP packet. The agent checks whether that string matches one configured locally and then applies the permissions associated with it, such as read-only access for monitoring or read-write access for changing writable MIB objects. However, without encryption it fundamentally exposes a security and configuration vulnerability.

GETBULK is generally more efficient than GETNEXT because it allows an SNMP manager to retrieve multiple OID values in a single request/response exchange, while GETNEXT usually retrieves only the next object at a time and must be repeated many times during a walk. This reduces the total number of SNMP packets, lowers protocol overhead, and decreases the amount of repeated work the device must perform, such as parsing requests, checking permissions, looking up MIB objects and building responses.

As a result, GETBULK can reduce CPU load on the monitored device and speed up polling, especially for large tables like interface counters or routing information. An overview is shown on Figure 21 of each GET message:

Figure 21 – Agent Response for GET Messages

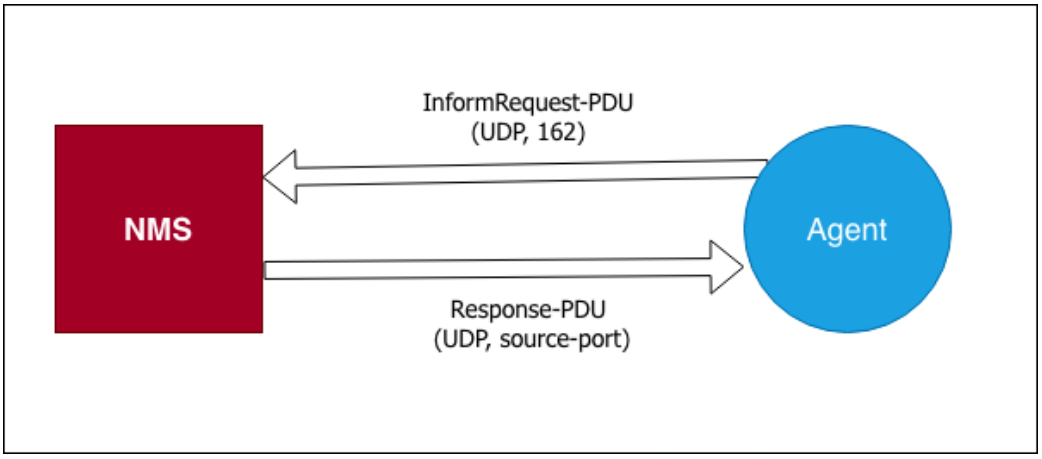


Source: [26].

On this version, it is also presented the INFORM Request, which provided a guaranteed delivery system for critical network notifications. Similarly to TRAP messages, by providing asynchronous, event-driven telemetry, it fundamentally alters the transport reliability model by enforcing a formal acknowledgment via a standard SNMP Response PDU. If the originating

agent does not receive this explicit acknowledgment within a predefined timeout window, it assumes the packet was lost to transient network congestion and actively retransmits the notification. The dynamics are outlined in the Figure 22 below:

Figure 22 – INFORM Request



Source: Author.

It highlights the main characteristic of an SNMP INFORM: unlike a normal trap, it requires an acknowledgement from the NMS. When the agent sends an *InformRequest-PDU* to the NMS, usually toward UDP port 162, it does not immediately assume the notification was successfully delivered. Instead, the agent waits for the NMS to send back a *Response-PDU*, which serves as the acknowledgement [25]. If this Response-PDU is received, the agent considers the INFORM successfully delivered. However, if the acknowledgement does not arrive before the timeout expires, the agent treats the INFORM as failed or unconfirmed and retransmits it according to the retry configuration.

This acknowledgement and retry behavior make INFORM more reliable than a TRAP, but it also consumes more resources because the agent must keep state for each pending INFORM until it is acknowledged or abandoned.

2.3.2 SNMPv3

SNMPv3 was developed to solve the main security limitations of SNMPv1 and SNMPv2c, especially their dependence on plaintext community strings for access control. Unlike earlier versions, SNMPv3 introduces a security framework that provides user-based authentication, message integrity, privacy through encryption and granular access control [27]. Its architecture separates message processing, security, and access control functions, mainly

through the User-based Security Model (USM) and the View-based Access Control Model (VACM). With USM, SNMPv3 can verify that a message came from an authorized user and was not modified in transit, while optional privacy mechanisms encrypt the SNMP payload to protect sensitive management data. For configuration overview on Junos, Figure 23 is shown:

Figure 23 – USM Configuration on Junos

```
set snmp v3 usm local-engine user snmpuser authentication-sha256 authentication-
key "example”
set snmp v3 usm local-engine user snmpuser privacy-aes128 privacy-key "example"
```

Source: Author.

As illustrated, the practical implementation of the USM framework requires the explicit definition of cryptographic parameters for each administrative entity. In this specific deployment scenario, a management user (*snmpuser*) is instantiated and bound to the local SNMP engine. The configuration enforces strict message integrity and origin authentication by applying the SHA-256 cryptographic hashing algorithm. Furthermore, to guarantee the confidentiality of the management payload against packet interception, the Advanced Encryption Standard (AES) with a 128-bit key is deployed as the privacy protocol. This approach ensures that all telemetry polling and subsequent responses associated with this user profile are securely transported across the network infrastructure.

To completely secure the management plane, the architecture requires a mechanism to define exactly what an authenticated user is permitted to see or modify. This function is exclusively managed by the VACM as **views** to define which OID trees are allowed attached to the USM user.

2.4 gRPC Network Management Interface

While the successive iterations of SNMP successfully addressed critical security vulnerabilities, the inherent inefficiencies of its pull-based polling architecture and unstructured OIDs proved inadequate for scalability. To resolve this limitation, the OpenConfig operator working group introduced the gNMI. Built upon the high-performance gRPC remote procedure call framework and HTTP/2 transport layer, gNMI represents a paradigm shift toward subscription model which leads to continuously stream data to centralized collectors [28].

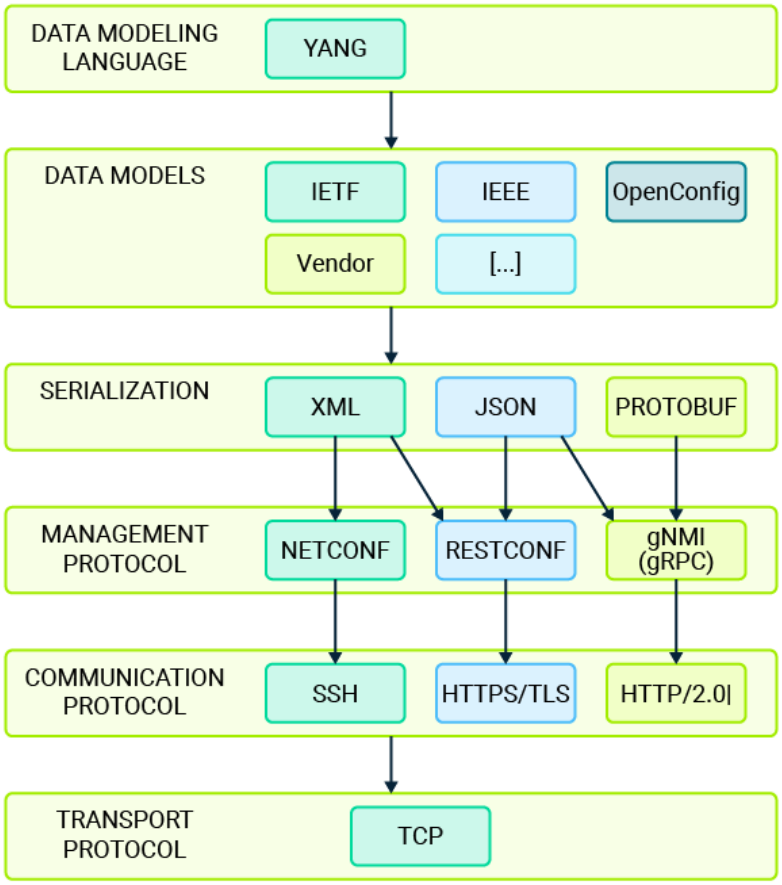
Unlike legacy architectures that separate configuration management from asynchronous alerting (such as SNMP TRAPs), gNMI provides a single, unified interface for state retrieval,

configuration modification, and continuous, push-based telemetry streaming. By natively integrating standardized YANG data models and utilizing Protobuf for efficient binary serialization, gNMI reduces control-plane computational overhead from JSON/XML parsing and RESTful endpoints.

2.4.1 Architecture

This architecture abstraction operates on a multi-layered stack. The data is manipulated to comply with standards and vendors documentation. Each step is demonstrated on Figure 24:

Figure 24 – gNMI Stack



Source: [29].

This model-driven telemetry transitions from abstract data definitions down to physical transport. Once structurally defined, the data is computationally packaged at the serialization layer before being handled by a designated management protocol (such as NETCONF, RESTCONF, or gNMI). Finally, these protocols utilize secure communication channels to transport the telemetry over standard TCP connections.

2.4.1.1 YANG

For data organization, the schema model emerged in 2010 as YANG dictates a semantic hierarchy and modular language. In this context, a schema maps the structure, type of data and constraints as shown on Figure 25:

Figure 25 – YANG Schema Model

Source: [29].

To ensure global uniqueness and prevent structural collisions across different vendors and standards, each module is strictly identified by a Uniform Resource Identifier (URI) defined within its namespace statement. For Junos, it is illustrated on Figure 26 for interface configuration:

Figure 26 – YANG Module

```
module junos-conf-interfaces {
   namespace "http://yang.juniper.net/junos/conf/interfaces";
   prefix jc-interfaces;
   import junos-conf-root {
        prefix jc;
   }
   grouping interfaces-group {
        container interfaces {
            description "Container for all interface configurations";
            list interface {
                key "name";
                description "List of individual interfaces";
                leaf name {
                    type string;
                    description "The physical interface name";
                }
            }
        }
    }
   augment /jc:configuration {
     uses interfaces-group;
   }
}
```

Source: Author.

The vendor-specific URI establishes the authoritative domain for the data structure. The prefix serves as representation of the namespace. For a configuration task, import parameters and data definitions from external modules to inherit its data (requires less compilation time due to tooling scope).

On grouping statement, it defines a reusable structural template. However, it is not immediately instantiated until explicitly invoked by a downstream command called uses.

For long schemas, subschemas are categorized and can be defined on container statement. Within this branch, the schema models data using a structural paradigm: singular scalar variables (leaf), arrays of simple data types (leaf-list), and complex tabular collections (list) which strictly require unique identifiers (key) to distinguish individual entries.

Finally, the augment mechanism allows external YANG modules to non-destructively graft new, highly specific data nodes into an existing conceptual tree without altering the original source file. For example, on interfaces YANG module implementation, the schema targets the external root container *junos-conf-root/configuration* the template *interfaces-group* with a defined collection of nodes (containers, lists or leaves).

Specifically for Junos, its YANG modules are shared on a public repository with all its versions and models. As shown below on Figure 27, the module is compiled as a structural tree implementation, and it is the actual information processed by gNMI after Protobuf derivation.

Applying this architecture to an interface-level YANG model, as its schema is illustrated on Figure 27, the container would group all interface configurations. Within it, a list would instantiate specific interfaces keyed by their names (e.g., "ge-0/0/0"), while the elements on grouping statement would dictate their specific attributes.

Figure 27 – YANG Compiled Tree

```
#pyang ietf-interfaces.yang -f tree
module: ietf-interfaces
  +--rw interfaces
  |  +--rw interface* [name]
  |     +--rw name                          string
  |     +--rw description?                  string
  |     +--rw type                          identityref
  |     +--rw enabled?                      boolean
  |     +--rw link-up-down-trap-enable?     enumeration {if-mib}?
  +--ro interfaces-state
     +--ro interface* [name]
        +--ro name                string
        +--ro type                identityref
        +--ro admin-status        enumeration {if-mib}?
        +--ro oper-status         enumeration
        +--ro last-change?        yang:date-and-time
        +--ro if-index            int32 {if-mib}?
        +--ro phys-address?       yang:phys-address
        +--ro higher-layer-if*    interface-state-ref
        +--ro lower-layer-if*     interface-state-ref
        +--ro speed?              yang:gauge64
        +--ro statistics
           +--ro discontinuity-time    yang:date-and-time
           +--ro in-octets?            yang:counter64
           +--ro in-unicast-pkts?      yang:counter64
           +--ro in-broadcast-pkts?    yang:counter64
           +--ro in-multicast-pkts?    yang:counter64
           +--ro in-discards?          yang:counter32
           +--ro in-errors?            yang:counter32
           +--ro in-unknown-protos?    yang:counter32
           +--ro out-octets?           yang:counter64
           +--ro out-unicast-pkts?     yang:counter64
           +--ro out-broadcast-pkts?   yang:counter64
           +--ro out-multicast-pkts?   yang:counter64
           +--ro out-discards?         yang:counter32
           +--ro out-errors?           yang:counter32
```

Source: Author.

2.4.1.2 Protobuf

While YANG establishes a strict structural blueprint for network data, it does not natively dictate how this data is encoded for transmission over the network. Historically, management protocols such as NETCONF and RESTCONF relied on human-readable text formats, specifically XML and JSON, to serialize this structured data.

However, the verbose nature of text-based encoding introduces significant computational overhead and consumes excessive bandwidth, rendering it inadequate for continuous data streaming. To resolve this performance bottleneck, modern telemetry frameworks like gNMI employ Protocol Buffers (Protobuf).

Developed by Google, Protobuf is a highly optimized, language-neutral, and platform-neutral binary serialization format [7]. By computationally compressing structured data into a dense binary payload, Protobuf drastically reduces message size and minimizes the CPU cycles required for serialization and deserialization at the network element [6]. This efficiency is the critical mechanism that allows modern routers to stream massive volumes of telemetry data at microsecond intervals without degrading core routing performance.

To make the YANG blueprint usable by software, developers use compiler tools (like ygot - YANG Go Tools) to automatically translate the *.yang* schema into Protobuf definition files (*.proto*).

- If the OpenConfig YANG model says*: leaf mtus { type uint16; }*
- The compiler translates this into a Protobuf message: *uint32 mtus = 1;*

Once the *.proto* files are generated, the router's operating system uses them to compress the actual telemetry data (e.g., the actual MTU value of 1500) into a dense, binary payload which is transmitted via gRPC.

As an example, the wrapper *ygot* is used to create a Protobuf file based on Figure 26. To perform this, the compiler *proto_generator* from *ygot* repository on Figure 28 is shown:

Figure 28 – Compiling YANG Module with *ygot proto_generator*

```
proto_generator -path=./yang-modules -output_dir=./proto \
-package_name=junos_example device.yang
```

Source: Author.

Since it uses external module junos-conf-root, the YANG module parsing is made on placing the *.yang* file on the path yang-modules folder. The resulted *.proto* file is illustrated below on Figure 29:

Figure 29 – Protobuf File

```
      // junos_example.junos_interfaces_test is generated by proto_generator as
a protobuf
      // representation of a YANG schema.
      //
      // Input schema modules:
      //  - device.yang
      // Include paths:
      //   - yang-modules/...
      syntax = "proto3";

      package junos_example.junos_interfaces_test;

      import "github.com/openconfig/ygot/proto/yext/yext.proto";

      message Interfaces {
        message Interface {
        }
        message InterfaceKey {
          string name = 1 [(yext.schemapath) = "/interfaces/interface/name"];
          Interface interface = 2;
        }

        repeated  InterfaceKey  interface  =  493598066  [(yext.schemapath)  =
"/interfaces/interface"];
}
```

Source: Author.

On Figure 26 the interface list uses *key "name"*, however Protobuf does not contain this native concept. To preserve the relationship, it generates a wrapper message called InterfaceKey to act as an array which pairs the key (*string name*) together with the actual data payload (*Interface interface*) invoked as repeated.

The compiler hashes the YANG path string to generate unique field number as shown as 493598066 to prevent tag collisions when combining or augmenting multiple modules. Furthermore, the bracketed items *(yext.schemapath) = "/interfaces/interface"* are Protobuf extensions that store metadata and maps the fields back to the YANG model.

For Python code compilation, the protobuf-compiler library is needed on Linux as well *ygot .proto* files to get the binary *protoc*. Once installed, it can be used as Figure 30 exhibits:

Figure 30 – Compiling to Python code with *ygot protoc*

```
sudo apt install protobuf-compiler
git clone https://github.com/openconfig/ygot.git deps/github.com/openconfig/ygot
protoc --python-out=. -I=. -I=./deps proto/junos_interfaces_test.proto
```

Source: Author.

Once compiled, the junos_interfaces_test_pb2.py file is created and ready to be imported as library. The data objects can be instantiated and abstracted as Python classes to interact with the router. In summary, the data flow would proceed as follows:

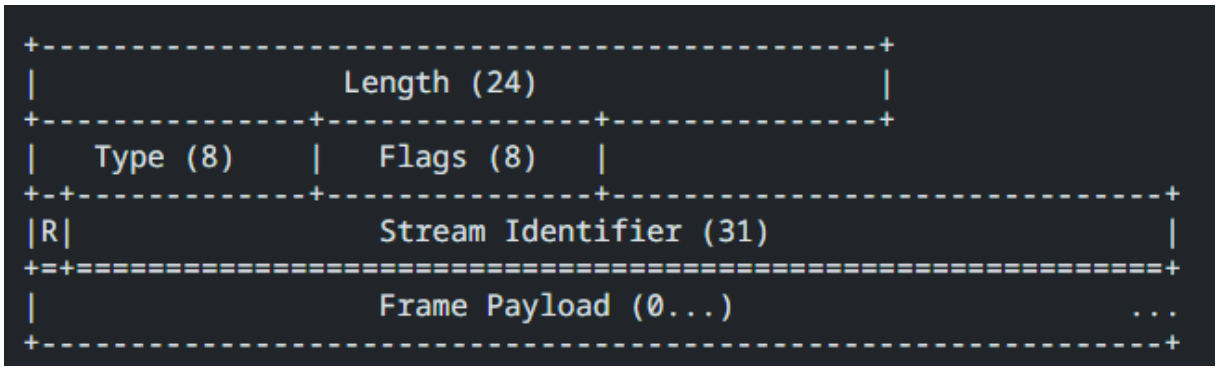
- The vendor (e.g., Juniper, Cisco, or OpenConfig) takes their YANG models.
- Protobuf files compilation with a tool like *proto_generator*.
- Compilation into Python/Go libraries for gNMI agents encoding with a tool like *protoc*.
- On host side, it is the same *.proto* files that are also compiled into libraries for decoding.

2.4.1.3 HTTP/2

While Protobuf efficiently compresses the telemetry payload into a dense binary format, the continuous delivery of these payloads requires a transport protocol capable of handling high-frequency, concurrent data streams. Historically, network management protocols - such as NETCONF over SSH or RESTCONF over HTTP/1.1 - suffered from head-of-line blocking and the computational overhead of establishing discrete TCP connections for sequential requests. To overcome these transport bottlenecks, modern telemetry frameworks, including gNMI, mandate the use of Hypertext Transfer Protocol Version 2 (HTTP/2) as the foundational communication protocol.

HTTP/2 introduces a binary framing layer and supports full multiplexing, allowing a network element to stream thousands of distinct telemetry updates simultaneously over a persistent TCP connection [30]. Instead of sending data as continuous text, it breaks the payload into distinct frame tagged with a unique Stream ID as Figure 31 illustrates:

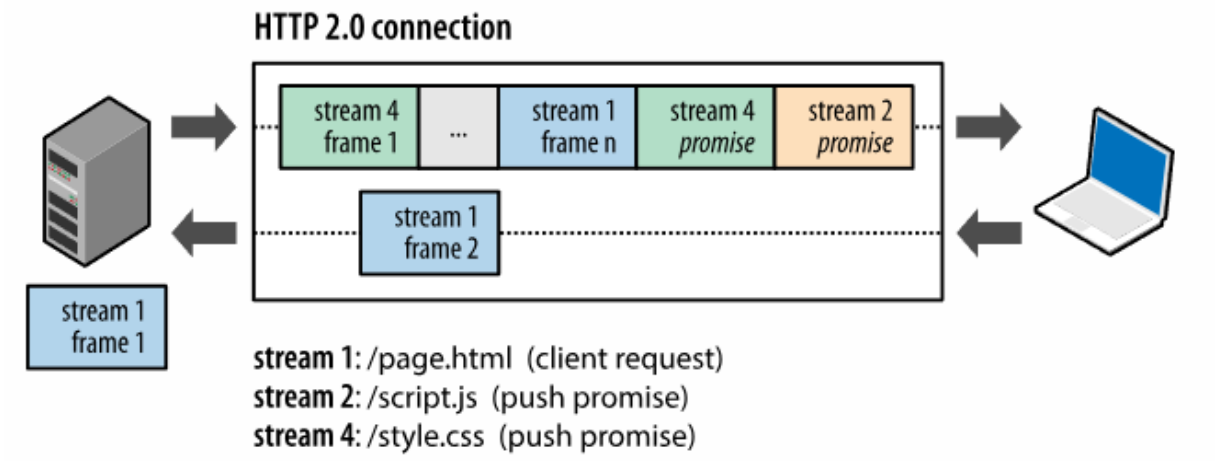
Figure 31 – HTTP/2 Frame Layout



Source: [30].

Moreover, the server push operation was created and enables the parallel preemptive delivery of anticipated client data without client request, deliberately trading increased network overhead for optimized latency performance [31]. Consequently, the router can interleave all the frames together on a single TCP session. The end host receives the mixed stream of frames and uses the Stream ID to sort them back into their original messages, since each stream is independent of each other as Figure 32 exhibits:

Figure 32 – HTTP/2 Streams



Source: [31].

By leveraging HTTP/2's multiplexing capabilities, gNMI allows distinct telemetry paths to operate simultaneously (e.g., BGP neighbor states, interface optical levels, and QoS queue drops) into a single, persistent TCP/TLS connection to the device. This reduces the CPU and

memory overhead on the network element, preserving those resources for core routing calculations.

2.4.2 Operations

The gNMI specification deliberately restricts network interaction to a tightly defined set of remote procedure calls. Rather than providing an expansive API, gNMI enforces architectural standardization by defining four Remote Procedure Calls (RPCs) to manage the entire lifecycle of a network element's YANG datastore.

The Capabilities RPC functions as the initial discovery mechanism between the telemetry collector and the network element. Because modern routers support a vast matrix of vendor-specific and vendor-neutral data models, the client must ascertain the device's capabilities before initiating data retrieval.

When a client executes this RPC, the network element returns a comprehensively structured message detailing:

- Supported YANG Models: A complete list of all supported schemas, including the specific organization (e.g., OpenConfig or native) and the exact semantic versioning.
- Supported Encodings: The serialization formats the device can process (e.g., JSON_IETF, PROTO, ASCII).
- gNMI Version: The protocol version currently running on the device agent.

For example, the Figure 33 shows these features via *gnmic* CLI for an Arista cEOS device:

Figure 33 – Arista cEOS gNMI Capabilities

```
gnmic -a 10.10.10.30:6030 -u admin -p admin --insecure capabilities
gNMI version: 0.7.0
supported models:
        - arista-interfaces-rates, Arista Networks <http://arista.com/>,
0.1.0
        - arista-classification-controller-field-set-open-config, Arista
Networks <http://arista.com/>, 1.0.0
        - arista-l1-open-config-optical-channel-model-augment, Arista
Networks <http://arista.com/>, 1.0.0
        - gnsi-authz, Google LLC,
        - openconfig-openflow, OpenConfig working group, 0.1.2
        - gnsi-certz, Google LLC,
        - openconfig-spanning-tree, OpenConfig working group, 0.3.1
        - openconfig-ospf, OpenConfig working group, 0.1.1
       […]
supported encodings:
        - JSON
        - JSON_IETF
        - ASCII
```

Source: Author.

As shown in the output, the device supports gNMI version 0.7.0 and successfully exposes both Arista-native and OpenConfig YANG models, utilizing JSON and ASCII encodings in addition to the underlying Protobuf transport.

To execute a synchronous, request-response operation that queries the network element's datastore at a specific point in time, the Get RPC is utilized, functionally mirroring the SNMP GET operation. While gNMI is heavily optimized for continuous streaming, the Get RPC remains architecturally necessary for discrete operational checks. Figure 34 shows this operation, retrieving the discrete LDP state from the device datastore:

Figure 34 – Arista cEOS gNMI Get

```
gnmic -a 10.10.10.30:6030 -u admin -p admin --insecure get --path
"eos_native:/Sysdb/mpls/ldp/ldpStatusColl/status/default/ldpSessionStatusColl/se
ssionStatus/" --format flat | grep "sessionState/sessionState"
```

Source: Author.

Conversely, the exclusive mechanism for mutating the configuration of the network element is called **Set** RPC. To ensure transactional integrity across complex service provider fabrics, the Set RPC processes operations atomically; if a multi-step configuration fails at any point, the entire transaction is rolled back.

Ultimately, the foundational driver for modern streaming telemetry is the **Subscribe** RPC. It fundamentally replaces SNMP polling by allowing devices to push data periodically (SAMPLE mode) or immediately upon a state transition (ON_CHANGE mode). For on-change subscription initialization targeting physical interface states, Figure 35 is shown:

Figure 35 – Subscribe ON_CHANGE Mode via gnmic

```
gnmic -a 10.10.10.20:32767 -u admin -p admin --insecure sub --path
/interfaces/interface/state --stream-mode on-change
```

Source: Author.

Because it accurately mirrors the real-time observability requirements of production network environments, the Subscribe RPC will serve as the primary data collection mechanism evaluated within the experimental testbed laboratory.

2.5 Containerlab

Historically, network emulation relied on heavyweight Virtual Machines (VMs) running within hypervisors (such as ESXi or KVM) or dedicated simulation platforms (like GNS3 or EVE-NG). While accurate, VMs require significant compute and memory resources, limiting the scale of the laboratory.

The foundational infrastructure of the laboratory was orchestrated utilizing Containerlab, an open-source, container-native network emulation framework. Unlike legacy network emulators that rely on dedicated simulation platforms (like GNS3 or EVE-NG), Containerlab rigidly adheres to Infrastructure as Code (IaC) principles and designed to specifically deploy containerized Network Operating Systems (NOS) [32].

To instruct Containerlab on how to build the network, the topology must be defined in a *.yaml* file. The YAML Ain't Markup Language (YAML) is a human-readable data serialization standard. Instead of imperative CLI sequential instructions (e.g., create interface, connect cable), YAML is declarative.

A text file describes the final desired state of the laboratory—listing the nodes (routers, switches, telemetry collectors) and the links (virtual cables) connecting them. Containerlab

parses this YAML file and automatically executes the underlying Docker networking commands to manifest that exact topology.

Figure 36 illustrates a simplified example of Containerlab YAML file that deploys an Arista router connected to a Linux-based telemetry collector:

Figure 36 – Arista cEOS and gNMI YAML Deployment

```
name: telemetry-testbed
topology:
      nodes:
          router-ceos:
            kind: ceos
            image: arista/ceos:4.28.0F
      gnmi-collector:
            kind: linux
            image: alpine:latest
      links:
           -  endpoints: ["router-ceos:eth1", "gnmi-collector:eth1"]
```

Source: Author.

Upon execution of the deployment directive, the orchestration engine parses the topology file, retrieves the specified container images, and instantiates the computational nodes via the Docker daemon.

During the deployment phase, Containerlab automatically creates a host-level bridge, attaches to every container and assigns management IP addresses. To interact with the nodes, Secure Shell (SSH), Telnet and Docker runtime execution (*docker exec*) are available to use.

2.6 Final Considerations

In this chapter, the theoretical principles underlying the experimental network architecture and telemetry monitoring were presented. The review of MPLS and the BGP detailed how dynamic routing information is exchanged and how data-plane traffic is efficiently forwarded across complex topologies. Furthermore, the underlying mechanics of LDP tunnel creation over IGP calculation and BGP prefix propagation are essential on service provider environments. This advanced routing infrastructure serves as the necessary foundation upon which the subsequent telemetry and network monitoring evaluations are built and tested.

The introduction of Containerlab highlighted the modern shift towards Infrastructure-as-Code (IaC), enabling the lightweight and reproducible virtualization of these complex network environments through Docker containers. The theoretical integration between these advanced routing protocols and modern telemetry frameworks is crucial for validating real-time network observability and performance. In the following chapter, the practical implementation of these concepts will be detailed, outlining the deployment of the experimental testbed, the automated provisioning of the routing nodes, and the architecture of the synthetic testing and visualization pipeline.

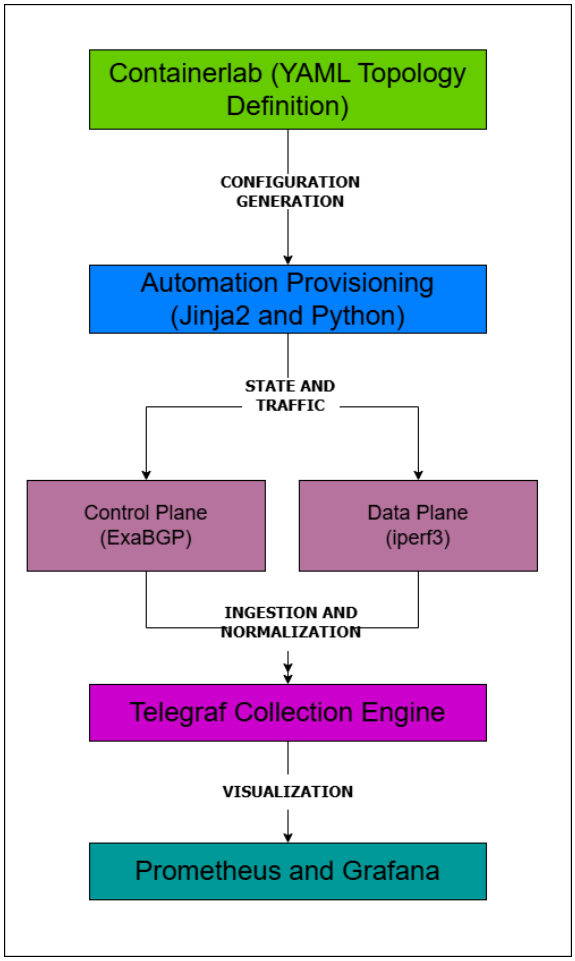
3 METHODOLOGY

This chapter details the methodology and architectural design utilized to construct, automate, and evaluate a multi-vendor network telemetry testbed. The primary objective of this environment was to simulate a realistic provider network, enabling a comparative analysis between SNMP polling and modern, push-based Model-Driven Telemetry (MDT) via gNMI and OpenConfig.

To achieve a reproducible, scalable, and hardware-independent environment, Infrastructure as Code (IaC) principles were applied. Container orchestration was utilized to deploy virtualized network operating systems from Juniper and Arista alongside synthetic traffic generators. Configuration management tools were employed to dynamically provision routing adjacencies, decouple configuration data from operational logic, and eliminate manual command line configuration.

Below on Figure 37 is the logical progression of the lab deployment:

Figure 37 – Activities Flowchart



Source: Author.

3.1 Network Design on Containerlab

The evaluation of modern telemetry protocols, such as gNMI and OpenConfig, requires a highly dynamic and reproducible testing environment. By utilizing operating system-level virtualization (containerization) rather than traditional hypervisor-based virtual machines, the testbed achieves near-native performance with significantly reduced computational overhead. This architecture enables the simultaneous deployment of multiple commercial network operating systems on a single computational host.

On Containerlab page it is available the platforms supported, they are declared into a predefined statement called kinds on YAML structure. Crucially, there are vendors which have official containerized NOS (Arista cEOS and Nokia SR Linux) and others that essentially are QEMU VM packaged in a Docker format (Juniper vMX and Cisco Nexus). The following Table 4 exhibits each image and version used for the laboratory:

Table 4 – Nodes Information

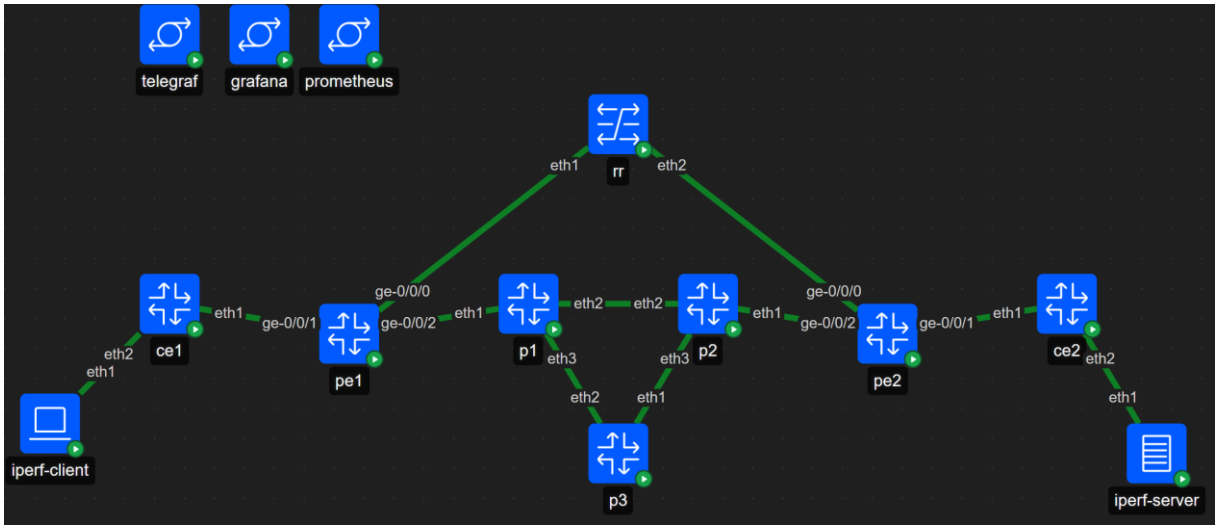
DEVICE	TYPE	SYSTEM	IMAGE
Juniper vJunos-router	QEMU	JunOS	juniper_vjunos-router:25.4R1.12
Arista cEOS	Container	Arista EOS	ceos:4.36.0F
Telegraf	Container	Linux	telegraf:1.38.4
Prometheus	Container	Linux	prom/prometheus:v3.12.0
Grafana	Container	Linux	grafana/grafana:13.0.2
ExaBGP	Container	Linux	pierky/exabgp:4.2.11
iperf3	Container	Linux	wbitt/network-multitool:3.22.2

Source: Author.

The deployment will follow the BGP-Free Core design standard. On control plane, OSPF acts as IGP for baseline IP reachability across the loopback interfaces of all provider-managed nodes (PE, P and RR). Atop IGP foundation, the LDP provisions LSPs across the core provider routers to actively forward packets based on MPLS label swapping.

To manage end-customer routing state, iBGP is established between PE routers and RR to avoid the requirement for a full-mesh topology. Finally, eBGP sessions are configured at the network perimeter between the PE routers and the simulated Customer Edge nodes (CE1 and CE2). Below Figure 38 illustrates the logical placement of this deployment:

Figure 38 – Network Topology



Source: Author.

A design requirement of the laboratory testbed was the inclusion of a multi-vendor routing environment. Historically, network operators relied on homogeneous, single-vendor architectures to simplify management. However, modern Service Provider networks universally adopt multi-vendor strategies to mitigate supply chain risks, avoid vendor lock-in, and leverage the specific architectural strengths of different NOS. To accurately reflect this industry standard, the testbed's roles were explicitly divided between Juniper as PE routers and Arista as internal core and RR:

— *pe1* and *pe2*: Juniper Provider Edge routers.
— *p1*, *p2* and *p3*: Arista Provider routers.
— *rr*: Arista Route Reflector router.
— *ce1* and *ce2*: ExaBGP Customer Edge routers.
— *iperf-client* and *iperf-server*: iperf3 hosts.

For IP assignment, Table 5 shows the customer and eBGP neighbor addresses used:

Table 5 – Customer IP Allocation

NODE	INTERFACE	PREFIX	ROLE	PEER
ce1	eth2	10.50.0.1/24	Site 1 LAN gateway	iperf-client (eth1)
ce2	eth2	10.60.0.1/24	Site 2 LAN gateway	iperf-server (eth1)
ce1	eth1	192.168.11.2/24	Site 1 WAN	pe1 (ge-0/0/1)
ce2	eth1	192.168.12.2/24	Site 2 WAN	pe2 (ge-0/0/1)
pe1	ge-0/0/1	192.168.11.1/24	WAN Peer (eBGP)	ce1 (eth1)
pe2	ge-0/0/1	192.168.12.1/24	WAN Peer (eBGP)	ce2 (eth1)
iperf-client	eth1	10.50.0.100/24	LAN Host	ce1 (eth2)
iperf-server	eth1	10.60.0.100/24	LAN Host	ce2 (eth2)

Source: Author.

On IGP core, Table 6 exhibits each interface IP address configuration to establish point-to-point connectivity:

Table 6 – Point-to-Point IP Allocation

NODE A	IP ADDRESS	NODE B	IP ADDRESS
pe1 (ge-0/0/0)	10.1.10.1/31	rr (Ethernet1)	10.1.10.0/31
pe2 (ge-0/0/0)	10.1.20.1/31	rr (Ethernet2)	10.1.20.0/31
pe1 (ge-0/0/2)	10.1.1.0/31	p1 (Ethernet1)	10.1.1.1/31
pe2 (ge-0/0/2)	10.1.2.0/31	p2 (Ethernet1)	10.1.2.1/31
p1 (Ethernet2)	10.1.12.0/31	p2 (Ethernet2)	10.1.12.1/31
p1 (Ethernet3)	10.1.13.1/31	p3 (Ethernet2)	10.1.13.0/31
p2 (Ethernet3)	10.1.23.1/31	p3 (Ethernet1)	10.1.23.0/31

Source: Author.

Finally, Table 7 illustrates the node loopbacks used to establish iBGP relationship and LDP binding:

Table 7 – Loopback IP Allocation

NODE	IP ADDRESS	ROLE
pe1	10.0.0.10/32	iBGP Client and Router ID
pe2	10.0.0.20/32	iBGP Client and Router ID
rr	10.0.0.100/32	iBGP Server
p1	10.0.0.1/32	Router ID
p2	10.0.0.2/32	Router ID
p3	10.0.0.3/32	Router ID

Source: Author.

The entire testbed is defined declaratively through a single YAML configuration file. An effective approach to managing complex networks is to separate the underlying network topology from the generalized device configurations [33]. This file serves as this strict topological blueprint, defining the management networks, virtual nodes, container images, and the point-to-point physical interconnects. For reference, the full *.yml* file content is presented on Appendix A. It initializes as follows by Figure 39:

Figure 39 – Management Network Initialization

```
name: gnmi-lab
prefix: ""

mgmt:
  network: core-mgmt
  ipv4-subnet: 10.10.10.0/24
  ipv6-subnet: 2001:db8::/64
```

Source: Author.

The topology initialization begins by defining the global namespace (*gnmi-lab*) and the out-of-band management network. Containerlab automatically provisions a dedicated Docker bridge network (*core-mgmt*) operating on 10.10.10.0/24. Every container deployed in the topology automatically receives a management interface attached to this bridge. This isolates administrative traffic - such as SSH access and telemetry scraping - from the simulated data-plane and control-plane traffic.

The *kinds* block specifies the exact container images required for the virtualized routers. By defining the target versions for Juniper (vJunos-router) and Arista (cEOS) globally, the script ensures environmental consistency. This eliminates version-mismatch errors across the multi-vendor architecture and ensures the laboratory accurately reflects a standardized production environment. Below on Figure 40 it is an example for Juniper and Arista images:

Figure 40 – Juniper vJunos Image Definition

```
topology:
  kinds:
    juniper_vjunosrouter:
      image: vrnetlab/juniper_vjunos-router:25.4R1.12
    arista_ceos:
      image: ceos:4.36.0F
```

Source: Author.

The next section described on Figure 41 provisions the centralized monitoring stack: Telegraf (the collector), Prometheus (the time-series database), and Grafana (the visualization dashboard). These nodes are deployed as lightweight Linux containers and static management IPs. Crucially, the *binds* array maps the configuration files from the host hypervisor directly into the containers, allowing the telemetry pipeline to be modified and restarted without destroying the containers.

Figure 41 – Observability Stack

```
nodes:
  # Monitoring pool
  telegraf:
    #collector
    kind: linux
    image: telegraf:latest
    mgmt-ipv4: 10.10.10.201
    binds:
      - configs/telegraf.conf:/etc/telegraf/telegraf.conf:ro

  prometheus:
    #database
    kind: linux
    image: prom/prometheus:latest
    mgmt-ipv4: 10.10.10.202
    binds:
      - configs/prometheus.yml:/etc/prometheus/prometheus.yml:ro
    ports:
      - 9090:9090

  grafana:
    #dashboard
    kind: linux
    image: grafana/grafana:latest
    mgmt-ipv4: 10.10.10.203
    ports:
      - 3000:3000
    env:
      GF_SECURITY_ADMIN_USER: admin
      GF_SECURITY_ADMIN_PASSWORD: admin
    binds:
      - configs/grafana/datasources:/etc/grafana/provisioning/datasources:ro
      - configs/grafana/dashboards:/etc/grafana/provisioning/dashboards:ro
```

Source: Author.

Finally, the block referenced by Figure 42 instantiates the core Service Provider infrastructure. Each node is assigned a specific role (Provider Edge, Route Reflector, or core Provider), receives its respective start-up configuration file and Linux commands to run during boot cycle if needed. The injection of the *startup-config* allows the routers to boot natively into a fully converged state with all IGP (OSPF) and LDP neighbors established.

Figure 42 – Control and Data Plane Nodes

```
# Nodes pool
   #control plane
   rr:
     kind: arista_ceos
     mgmt-ipv4: 10.10.10.100
     startup-config: configs/rr.cfg
   pe1:
     kind: juniper_vjunosrouter
     mgmt-ipv4: 10.10.10.10
     startup-config: configs/pe1.cfg
```

Source: Author.

The final section of the YAML manifest is the *links* array, as illustrated in Figure 43, which dictates the point-to-point wiring of the simulated environment. In network emulation, a topology graph is utilized to explicitly represent the relationships and physical interconnects between network components. Containerlab reads these endpoints and instructs the Linux kernel to provision isolated virtual Ethernet pairs between the specified containers.

Figure 43 – Interconnections under *links* Statement

```
 links:
   # Route Reflector to PEs
   - endpoints: [ "rr:eth1", "pe1:ge-0/0/0" ]
     ipv4: [ "10.1.10.0/31", "10.1.10.1/31" ]
   # iperf3 to Customer Edge
   - endpoints: ["ce1:eth2", "iperf-client:eth1"]
     ipv4: ["10.50.0.1/24", "10.50.0.100/24"]
```

Source: Author.

To deploy the YAML file, the following Figure 44 command is used:

Figure 44 – Containerlab Deploy Command

```
containerlab deploy -t <clab.yml file>
```

Source: Author.

The status and result of the deployment show at the end of the command output. For configuration file generation, the next section is presented.

3.2 Provisioning via Jinja2 and Python

A fundamental challenge in multi-vendor network emulation is managing the heterogeneous syntax of different NOS. In this testbed, the Provider Edge nodes (Juniper vJunos) utilize a hierarchical, block-based configuration structure, the core nodes (Arista cEOS) utilize a flat, IOS-like syntax, and the Customer Edge nodes (ExaBGP) require a daemon-specific process configuration file. Manually generating these configurations is not only time-consuming but highly prone to human error.

To resolve this, the configuration generation process was abstracted using Jinja2, a templating engine for Python, for alignment with the concept of topology-driven configuration [33]. A configuration template acts as an incomplete structural blueprint that contains vendor-specific syntax interwoven with variable placeholders (e.g., *{{ router_id }}*, *{{ local_as }}*) rather than hardcoded values.

The *pe-router.j2* and *p-router.j2* templates provisions Juniper PE routers and Arista P routers respectively, which both are detailed on Appendix B. To demonstrate the deterministic templating mechanism utilized in the provisioning pipeline, the following Figure 45 illustrate how routing protocols were generalized.

This template explicitly standardizes the Provider Edge routing policy. It permanently establishes an iBGP relationship with the RR while dynamically mapping the eBGP adjacency to the Customer Edge. Furthermore, each configuration line is attached to a variable which eliminates the need to maintain multiple data models, adhering to strict software engineering efficiency and scalability.

Figure 45 – BGP Configuration Template

```
            bgp {
                group IBGP-RR {
                    type internal;
                    local-address {{ loopback_ip | replace("/32", "") }};
                    family inet {
                        unicast;
                    }
                    family inet-vpn {
                        unicast;
                    }
                    export NEXT-HOP-SELF;
                    neighbor {{ rr_ip }};
                }
                group EBGP {
                    type external;
                    peer-as {{ ce_peer_as }};
                    neighbor {{ ce_peer_ip }};
                }
            }
}
```

Source: Author.

To orchestrate the generation of these vendor-specific templates, a custom Python script (*build.py*) was developed and detailed on Appendix D. With Jinja2 library, it starts by parsing the template files as Figure 46 shows:

Figure 46 – Jinja2 Templates Parsing

```
from jinja2 import Environment, FileSystemLoader
env = Environment(loader=FileSystemLoader('.'))
pe_template = env.get_template('pe-router.j2')
p_template = env.get_template('p-router.j2')
```

Source: Author.

The core topological state of the laboratory is abstracted into two Python lists containing dictionaries (*pe_routers* and *p_routers*). This data model acts as the absolute source of truth and node definition as illustrated by Figure 47:

Figure 47 – Provider Edge Provisioning

```
# Map nodes as JSON into the template
pe_routers = [
{
        "hostname": "pe1",
        "loopback_ip": "10.0.0.10/32",
        # to RR
        "rr_iface": "ge-0/0/0",
        "rr_iface_ip": "10.1.10.1/31",
        "rr_ip": "10.0.0.100",
        # to P-router
        "core_iface": "ge-0/0/2",
        "core_ip": "10.1.1.0/31",
        # to CE
        "ce_iface": "ge-0/0/1",
        "ce_ip": "192.168.11.1/24",
        "ce_peer_ip": "192.168.11.2",
        "ce_peer_as": "65001"
    },
    {
        "hostname": "pe2",
        "loopback_ip": "10.0.0.20/32",
        # to RR
        "rr_iface": "ge-0/0/0",
        "rr_iface_ip": "10.1.20.1/31",
        "rr_ip": "10.0.0.100",
        # to P-router
        "core_iface": "ge-0/0/2",
        "core_ip": "10.1.2.0/31",
        # to CE
        "ce_iface": "ge-0/0/1",
        "ce_ip": "192.168.12.1/24",
        "ce_peer_ip": "192.168.12.2",
        "ce_peer_as": "65002"
    }
]
```

Source: Author.

In the final execution phase, the script iterates through the structured data models as Figure 48 exhibits. For every dictionary (router) in the list, it passes the data into the *.render()* function of the respective Jinja2 template. The engine substitutes all placeholders with the targeted variables and exports a syntactically complete configuration artifact (e.g., *pe1.cfg*, *p1.cfg*).

Figure 48 – Jinja2 Rendering

```
for router in pe_routers:
    config = pe_template.render(router)
    filename = f"{router['hostname']}.cfg"

    with open(filename, 'w') as file:
        file.write(config)
    print(f"File {filename} generated")

for router in p_routers:
    config = p_template.render(router)
    filename = f"{router['hostname']}.cfg"

    with open(filename, 'w') as file:
        file.write(config)
    print(f"File {filename} generated")
```

Source: Author.

Essentially, with YAML file and templates defined, the configuration files are generated via command line. Figure 49 illustrates as an output example for the laboratory topology:

Figure 49 – Configuration Files Generation

```
$ python3 build.py
       File pe1.cfg generated
       File pe2.cfg generated
       File p1.cfg generated
       File p2.cfg generated
       File p3.cfg generated
```

Source: Author.

For Route Reflector configuration, due to its individualities in configuration, it was manually built and is available in Appendix C. Crucially, the client's routes reflection feature must be declared in addition to the neighbor IPs for iBGP as shown by Figure 50 :

Figure 50 – iBGP Route Reflector

```
router bgp 65000
   router-id 10.0.0.100

   ! iBGP Group
   neighbor IBGP-PE peer group
   neighbor IBGP-PE remote-as 65000
   neighbor IBGP-PE update-source Loopback0
   neighbor IBGP-PE route-reflector-client
   neighbor IBGP-PE send-community extended

   ! Peering with PE1 and PE2 Loopbacks
   neighbor 10.0.0.10 peer group IBGP-PE
   neighbor 10.0.0.20 peer group IBGP-PE

   ! Enable VPNv4 Reflection
   address-family ipv4
      neighbor IBGP-PE activate
!
```

Source: Author.

3.3 Synthetic Tests with ExaBGP and iperf3

The primary objective of this testbed is to evaluate the performance, latency, and accuracy of Model-Driven Telemetry (gNMI) against legacy polling (SNMP) during periods of network instability. A static network is insufficient for telemetry validation, as it does not trigger event-driven updates or expose the aliasing limitations of pull-based polling. Therefore, synthetic tests were engineered to independently stress both the network control plane (routing state) and the data plane (interface throughput).

ExaBGP is a software-defined BGP injector that allows custom applications to interface with network routing tables via an API. As defined in the configuration file (*ce1.conf* and *ce2.conf*) as referenced on Figure 51, the CE nodes establish standard eBGP peering with the PE while injecting a static route (10.50.0.0/24 and 10.60.0.0/24). It ensures the TCP session remains established for *iperf3* throughput test.

Figure 51 – Site 1 ExaBGP Configuration

```
process route-leak {
    run /usr/bin/python3 /run/route_leak.py 200 192.168.11.2 65001 15;
    encoder text;
}
neighbor 192.168.11.1 {
    router-id 192.168.11.2;
    local-address 192.168.11.2;
    local-as 65001;
    peer-as 65000;

    static {
        route 10.50.0.0/24 next-hop 192.168.11.2;
    }

    api {
        processes [ route-leak ];
    }
}
```

Source: Author.

Simultaneously, the API invokes the *dynamic_routes.py* process as exhibited on Figure 52 to simulate BGP instability. Rather than relying on static prefix advertisements, a custom Python script was engineered to simulate rapid route flapping and shared on Appendix F. Essentially, it performs BGP Updates for route announcement or withdraw between a random interval while *routes_states* maintain the current state (announced or withdrawn) of each prefix.

Figure 52 – BGP Updates via ExaBGP

```
 while time.time() < end_time:
        for p in prefixes:
            sys.stdout.write(f"announce route {p} next-hop {NEXT_HOP} as-path
{AS_PATH}\n")
        sys.stdout.flush()

        time.sleep(FLAP_INTERVAL)
        for p in prefixes:
            sys.stdout.write(f"withdraw route {p} next-hop {NEXT_HOP}\n")
        sys.stdout.flush()

        time.sleep(FLAP_INTERVAL)
```

Source: Author.

In production ISP environments, route leaks frequently occur when a multi-homed customer or peer inadvertently misconfigures their routing policies, advertising massive blocks of learned routes back into the provider core. This results in thousands of prefixes being abruptly injected, withdrawn, or rerouted. In this way, the script *dynamic_routes.py* simulates this phenomenon.

While the ExaBGP integration successfully induces control-plane instability, routing updates alone do not generate significant data-plane payload traffic. To accurately correlate control-plane routing states with physical interface throughput, synthetic data-plane traffic is generated. Furthermore, modern observability pipelines must be validated against their ability to detect anomalous bandwidth exhaustion, such as Distributed Denial of Service (DDoS) attacks, which remain a primary threat vector in Service Provider networks.

To achieve this, the network testing tool *iperf3* was used. Recognized in academic literature as a standard utility for end-to-end performance measurement and throughput prediction [34], *iperf3* allows for the controlled generation of specific traffic profiles. Rather than placing the traffic generators within the core, the *iperf3* client and server containers were deployed at the perimeter of the testbed (behind *ce1* and *ce2*, respectively).

This architectural placement ensures that all synthetic payloads must successfully traverse the dynamically injected ExaBGP routes, the Provider Edge RIBs, and the MPLS core. Consequently, the interface bandwidth metrics captured by the telemetry collectors directly correlate with the successful end-to-end propagation of the BGP prefixes.

Because a standard *iperf3* execution simply measures maximum achievable bandwidth, a custom Python orchestrator (*iperf.py*) was developed to simulate a volumetric DDoS profile and it is available on Appendix E. Bursts of UDP traffic is explicitly utilized in this simulation because, unlike TCP, it lacks congestion control and windowing mechanisms, allowing it to mathematically force link saturation.

To control the *iperf3* binary remotely, the Python script uses the native subprocess module to issue *docker exec* commands directly to the container's namespace as Figure 53 demonstrates:

Figure 53 – Docker Subprocess Command Injection

```
def run_iperf(bandwidth, duration, phase_name):
    """Executes a blocking iperf3 command for a specific duration."""
    print(f"[{time.strftime('%H:%M:%S')}] Starting {phase_name} Phase:
{bandwidth}bps for {duration}s")
    cmd = f"docker exec {CLIENT_NODE} iperf3 -c {SERVER_IP} -u -b {bandwidth}
-t {duration}"

    # Block the script until the iperf3 timer finishes
    subprocess.run(cmd, shell=True, stdout=subprocess.DEVNULL,
stderr=subprocess.DEVNULL)
```

Source: Author.

The *run_iperf* function dynamically constructs a string instructing the Docker daemon to execute the iperf3 application inside the *CLIENT_NODE*. The command parameters dictate the exact behavior of the traffic:

— *-c {SERVER_IP}:* Instructs the application to operate in client mode, targeting the server IP residing behind the opposite Customer Edge router.

— *-u*: Forces the use of UDP payloads, which is mandatory for simulating volumetric bandwidth exhaustion without TCP windowing interference.

— *-b {bandwidth}* and *-t {duration}*: Injects the specific target bitrates and time limits.

By utilizing *subprocess.run*, the Python script execution halts until the exact duration expires inside the container. This ensures synchronization and prevents overlapping traffic generation processes. Finally, Figure 54 dictates a loop with traffic profiles alternates between two states:

Figure 54 – Traffic Profiles Injection

```
    while True:
        # Normal Traffic
        run_iperf(BASELINE_BW, BASELINE_TIME, "BASELINE")

        # Volumetric Spike
        run_iperf(PEAK_BW, PEAK_TIME, "ATTACK PEAK")
```

Source: Author.

3.4 Telegraf Pipeline and Visualization on Grafana

To bridge the gap between the network infrastructure and the time-series database, an intermediate collection engine is required. In this architecture, the plugin-driven server agent

InfluxData's Telegraf was selected and designed for collecting, parsing, and normalizing metric data.

As opposed to deploying disparate, protocol-specific collectors (e.g., an SNMP exporter and a separate gNMI exporter), Telegraf acts as a unified ingestion gateway. More importantly, Telegraf executes an internal processing pipeline, manipulating the raw, vendor-specific payloads into a standardized metric schema before exposing the data to Prometheus for time-series storage and metrics retention.

The whole Telegraf configuration file (*telegraf.conf*) is shown on Appendix G. Its foundation dictates the operational polling cadence, data egress methodology and instructions for Prometheus plugin. On Figure 55, this is demonstrated:

Figure 55 – Pull-based Plugins Configuration

```
[agent]
  interval = "10s"
  round_interval = true
  metric_batch_size = 1000
  metric_buffer_limit = 10000

# Prometheus integration
[[outputs.prometheus_client]]
  listen = ":9273"
  expiration_interval = "60s"
  export_timestamp = true
  metric_version = 2
```

Source: Author.

Because Prometheus operates on a pull-based scraping model, Telegraf does not actively push data to the database. Instead, the *prometheus_client* output plugin instructs the collector to open a local web server on TCP port 9273. Telegraf holds the normalized metrics in memory, translating them into the Prometheus exposition format until the time-series server performs its periodic HTTP scrape.

On Prometheus side, it is imperative to configure the Telegraf endpoint as well. Figure 56 illustrates the YAML file to point to the exposed TCP port from collector:

Figure 56 – Prometheus YAML File

```yaml
global:
  scrape_interval: 10s
  evaluation_interval: 10s

scrape_configs:
  - job_name: 'telegraf-collector'
    static_configs:
      - targets: ['10.10.10.201:9273']
```

Source: Author.

Since SNMP processing is highly CPU-intensive (specially for SNMP Walk operations) for network operating systems, the polling interval was explicitly overridden to 30s [35]. The configuration uses the *inputs.snmp.table* structure to walk standardized MIBs, such as *ifHCInOctets* for interface bandwidth, and were collected uniformly across vendors. However, protocol-specific data required querying fragmented, vendor-proprietary trees, such as the *jnxBgpM2PrefixOutPrefixes* tree for Juniper BGP state and the *aristaBgp4V2PrefixOutPrefixes* tree for Arista. For interface traffic tree, Figure 57 shows the SNMP polling configuration for all core and edge devices.

A fundamental limitation discovered during the normalization pipeline design was the mechanism by which legacy SNMP handles complex table indices. Unlike gNMI, which encapsulates metadata as native JSON strings (e.g., {"peer_ip": "192.168.11.2"}), SNMP dynamically embeds row identifiers directly into the Object Identifier (OID) tree.

When an SNMP table is indexed by a string or an IP address (such as a BGP Peer IP or an LDP Router ID), the SNMP agent converts each character or byte of that string into its Decimal-ASCII or Hexadecimal equivalent, appending it to the end of the OID.

Consequently, when the Telegraf collector executed the index_as_tag = true directive to capture the peer identifiers, it was incapable of direct ASCII translation. Instead of capturing a human-readable IP address, Telegraf captured the raw, decimal-encoded OID suffix (e.g., 1.4.192.168.11.2.1.1). If exported directly to the Prometheus database, this data would be completely illegible for dashboard visualization.

Figure 57 – Interface Traffic SNMP Polling

```
[[inputs.snmp]]
  agents = [
    "pe1:161",  # PE1
    "pe2:161",  # PE2
    "p1:161",  # P1
    "p2:161",  # P2
    "p3:161",  # P3
    "rr:161"  # RR
  ]
  version = 2
  community = "public"
  interval = "30s"
  timeout = "5s"
  retries = 1

  # Define OIDs

  [[inputs.snmp.table]]
    name = "interface"

    [[inputs.snmp.table.field]] # interface name as tag
      name = "ifDescr"
      oid = ".1.3.6.1.2.1.2.2.1.2"
      is_tag = true

    [[inputs.snmp.table.field]] # Up/Down state
      name = "ifOperStatus"
      oid = ".1.3.6.1.2.1.2.2.1.8"

    [[inputs.snmp.table.field]]
      name = "ifHCInOctets"
      oid = "1.3.6.1.2.1.31.1.1.1.6"

    [[inputs.snmp.table.field]]
      name = "ifHCOutOctets"
      oid = "1.3.6.1.2.1.31.1.1.1.10"
```

Source: Author.

To resolve this lack of direct ASCII support, the normalization pipeline was engineered with Telegraf processors to operate over the OID strings. For example, the *processors.regex* plugin was deployed against the Arista SNMP BGP tables as exhibited by Figure 58. A custom regular expression was engineered to mathematically parse the raw index string, isolate the four octets representing the IPv4 address, and cast them into a clean *PeerNeighbor* string tag.

Figure 58 – Regex Neighbor IP Extraction

```
[[processors.regex]]
  namepass = ["snmp_arista_bgp_prefixes"]

  [[processors.regex.tags]]
    key = "index"
    pattern = "^\\d+\\.1\\.4\\.(\\d+\\.\\d+\\.\\d+\\.\\d+)\\.\\d+\\.\\d+$"
    replacement = "${1}"
    result_key = "PeerNeighbor"
```

Source: Author.

For Juniper nodes, the BGP Peer IP addresses and the prefix counters were isolated in entirely different SNMP tables, linked only by a raw decimal identifier. Because Telegraf could not natively resolve this arbitrary integer into an ASCII IP address, an embedded Starlark - dialect of Python supported by Telegraf - script was created. This script operated as an in-memory dictionary, actively translating the raw integer indices into human-readable BGP Peer IPs before appending them to the metric payload. Below on Figure 59 is the full script to operate over different trees and to correlate the neighbor's IP address:

Figure 59 – Starlark Logic for Relational Mapping

```
[[processors.starlark]]
  namepass = ["snmp_juniper_bgp_peers*", "snmp_juniper_bgp_prefixes*"]
  source = '''
def apply(metric):
    if "peer_map" not in state:
        state["peer_map"] = {}

    peer_map = state["peer_map"]
    if metric.name.startswith("snmp_juniper_bgp_peers"):
        p_neighbor = metric.tags.get("PeerNeighbor")
        agent = metric.tags.get("agent_host")

        for field_name, field_value in metric.fields.items():
            p_index = str(int(field_value))

            if agent and p_index and p_neighbor:
                # Create a unique key per router (e.g., "pe1:1")
                cache_key = agent + ":" + p_index
                peer_map[cache_key] = p_neighbor

        return metric
    elif metric.name.startswith("snmp_juniper_bgp_prefixes"):
        agent = metric.tags.get("agent_host")
        raw_index = metric.tags.get("index") # e.g., "0.1.1"

        if agent and raw_index:
            # Extract the first element from the index tag (e.g., "0" or "1")
            parts = raw_index.split(".")
            p_index = parts[0]

            # Reconstruct the unique router key (e.g., "pe1:1")
            cache_key = agent + ":" + p_index

            if cache_key in peer_map:
                metric.tags["PeerNeighbor"] = peer_map[cache_key]

        return metric
'''
```

Source: Author.

To evaluate the capabilities of modern observability, the Telegraf collector was also configured to ingest gNMI metrics. Unlike SNMP, which relies on opaque numerical OIDs, gNMI data are defined on schema-backed paths defined by YANG models (e.g., /interfaces/interface/state/counters). An architectural advantage leveraged in this testbed was gNMI's native support for discrete subscription modes. As defined in the *telegraf.conf* file and

commented on Figure 60, the collection pipeline explicitly partitioned telemetry based on its volatility:

Figure 60 – gNMI Subscription Mode

```
[[inputs.gnmi]]
  addresses = [
    "pe1:32767",  # PE1
    "pe2:32767",  # PE2
    "rr:6030"   # RR
  ]
  [[inputs.gnmi.subscription]]
    name = "gnmi_bgp_messages"
    origin = "openconfig"
    path = "/network-instances/network-
instance/protocols/protocol/bgp/neighbors/neighbor/state/messages"
    subscription_mode = "sample"
    sample_interval = "10s"

  [[inputs.gnmi.subscription]]
    name = "gnmi_bgp_state"
    origin = "openconfig"
    path = "/network-instances/network-
instance/protocols/protocol/bgp/neighbors/neighbor/state/session-state"
    subscription_mode = "on_change"
```

Source: Author.

For BGP messages, sample mode under 10 seconds interval was used. However, for BGP states the subscription model ON_CHANGE is preferrable.

While OpenConfig YANG models theoretically standardize telemetry across disparate hardware, a significant challenge discovered during this research was vendor support. Although Juniper successfully streamed LDP session states using standard OpenConfig paths, the Arista cEOS nodes lacked OpenConfig support for LDP.

To circumvent this limitation and maintain observability parity, the pipeline was forced to subscribe to Arista's proprietary internal database architecture (*origin = "eos_native"*), targeting the */Sysdb/mpls/ldp/* path.

This deviation introduced a severe formatting challenge. Rather than returning clean, nested JSON structures, the *eos_native* telemetry path returned flattened, concatenated strings containing historical log entries and dynamically injected IP addresses (e.g., *0_10_0_0_10_ldpSessionState_sessionState*). If passed directly to Prometheus, this

unstructured output would cause database cardinality instability and render the data entirely ungraphable. Once again, an embedded Starlark script was engineered directly into the Telegraf pipeline to extract the LDP peer IP address as demonstrated on Figure 61:

Figure 61 – LDP Peer IP Extraction via Starlark

```
[[processors.starlark]]
  namepass = ["gnmi_ldp_state_arista"]
  source = '''
def apply(metric):
    keys_to_remove = []

    for k, v in metric.fields.items():
        if type(v) == "string" and "sessionstate" in k.lower() and "logentry"
not in k.lower():
            keys_to_remove.append(k)
            if "/" in k:
                # Handles Arista format:
"0_0_10_0_0_10/sessionState/sessionState"
                prefix = k.split("/")[0]
            else:
                prefix =
k.split("_ldpSessionState")[0].split("_sessionState")[0]

            if prefix.startswith("_"):
                prefix = prefix[1:]

            parts = prefix.split("_")
            if len(parts) >= 4:
                # Grab the last 4 elements (the octets) and join them with
dots
                peer_ip = ".".join(parts[-4:])
            else:
                peer_ip = prefix

            metric.tags["ldp_peer"] = peer_ip
            metric.fields["session_state_raw"] = v

    for k in keys_to_remove:
        metric.fields.pop(k)

    return metric
'''
```

Source: Author.

The final challenge in normalizing the gNMI stream was resolving the inherent limitation of time-series database. Protocols like BGP and LDP transmit their operational statuses as human-readable strings (e.g., *"ESTABLISHED", "IDLE", "ACTIVE").* However, Prometheus cannot mathematically aggregate or graph string values. The *processors.enum* plugin was implemented to translate control-plane strings into universally standardized integers to overcome this limitation. Figure 62 illustrates the implementation:

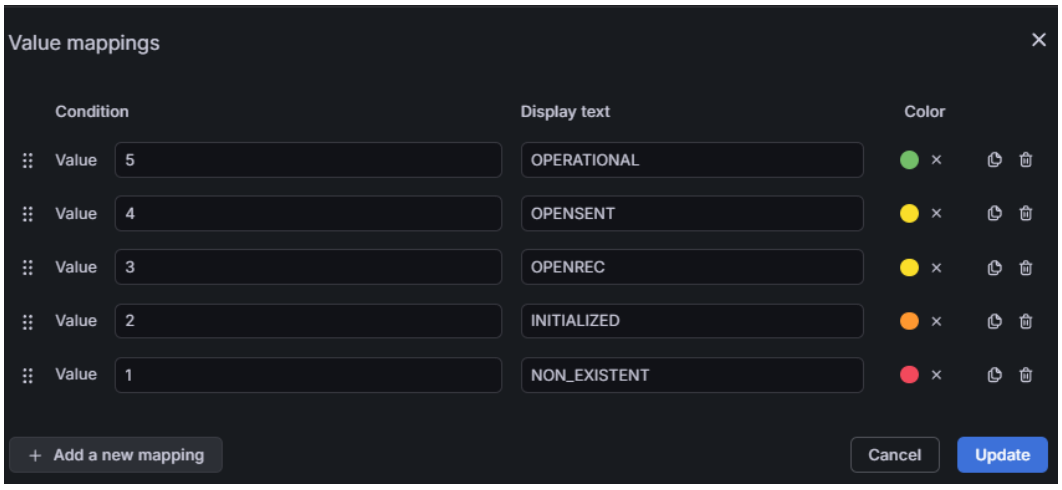
Figure 62 – BGP States Enumeration

```
[[processors.enum]]
  namepass = ["gnmi_bgp_state"]
  [[processors.enum.mapping]]
    field = "session_state"
    dest  = "session_state_int"
    [processors.enum.mapping.value_mappings]
      "IDLE" = 1
      "CONNECT" = 2
      "ACTIVE" = 3
      "OPENSENT" = 4
      "OPENCONFIRM" = 5
      "ESTABLISHED" = 6
```

Source: Author.

This processor forces an index relation between the BGP states. By translating "ESTABLISHED" to 6 and "IDLE" to 1, the observability dashboard gains the ability to plot routing anomalies as discrete, mathematical step functions. On Grafana as Figure 63 demonstrates, it is possible to map back again to BGP states for visualization:

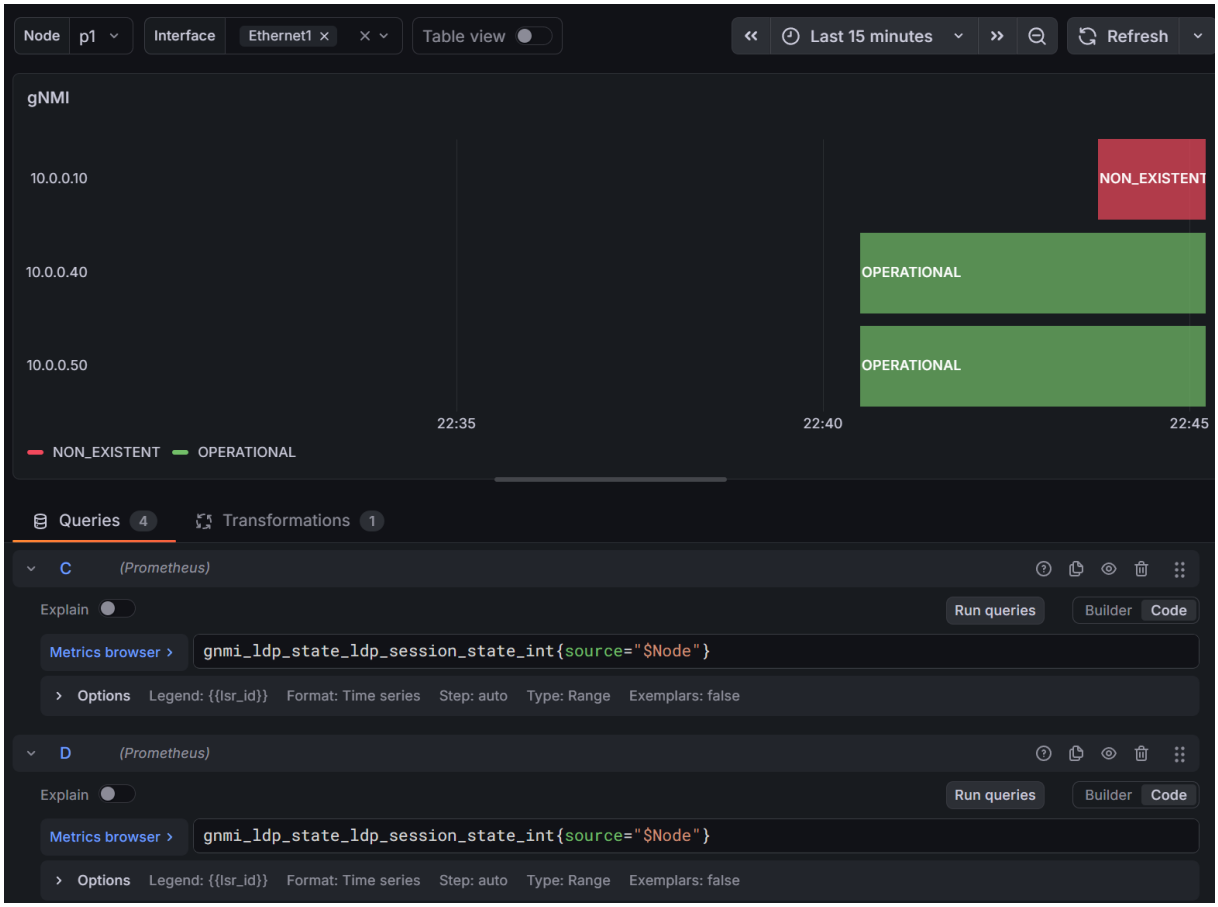
Figure 63 – Index Values to BGP States on Grafana



Source: Author.

To expose the operational comparison for SNMP and gNMI, specific Grafana panels were engineered utilizing the Prometheus Query Language (PromQL). For example, below on Figure 64 it is observed a panel to monitor LDP state under gNMI standard:

Figure 64 – Juniper and Arista LDP State Panel



Source: Author.

3.5 Final Considerations

The development of this experimental testbed successfully merges modern network virtualization (Containerlab) with Infrastructure as Code (Jinja2 and Python automation). By decoupling the routing topology from physical hardware limitations and programmatically generating the entire network state, the methodology guarantees reproducible baseline.

Crucially, the validity of the subsequent results relies on the deterministic nature of the synthetic simulations and the out-of-band telemetry collection. These architectural decisions provides the isolation necessary to investigate the performance disparities between SNMP polling and modern model-driven gNMI telemetry.

4 RESULTS AND DISCUSSION

The analysis in this research involves observing the interface bandwidth during the volumetric *iperf3* bursts, route leaking during BGP announcement and LDP status monitoring. The resulting metrics were ingested, normalized, and stored within a centralized Prometheus time-series database. To facilitate a rigorous comparative analysis, a declarative observability dashboard was engineered utilizing Grafana. The dashboard was structured into side-by-side comparative panels, ensuring that both legacy and modern telemetry pipelines were evaluated against the exact same temporal axis and network events.

Moreover, SNMP MIBs between Juniper and Arista has proven inconsistency. To monitor identical metrics on both Arista EOS and Juniper vJunos routers, the polling engine had to be configured to traverse entirely distinct, vendor-proprietary OIDs. This lack of standardization inevitably leads to fragmented collector configurations, requiring manual MIB translation and complex data normalization for each specific operating system.

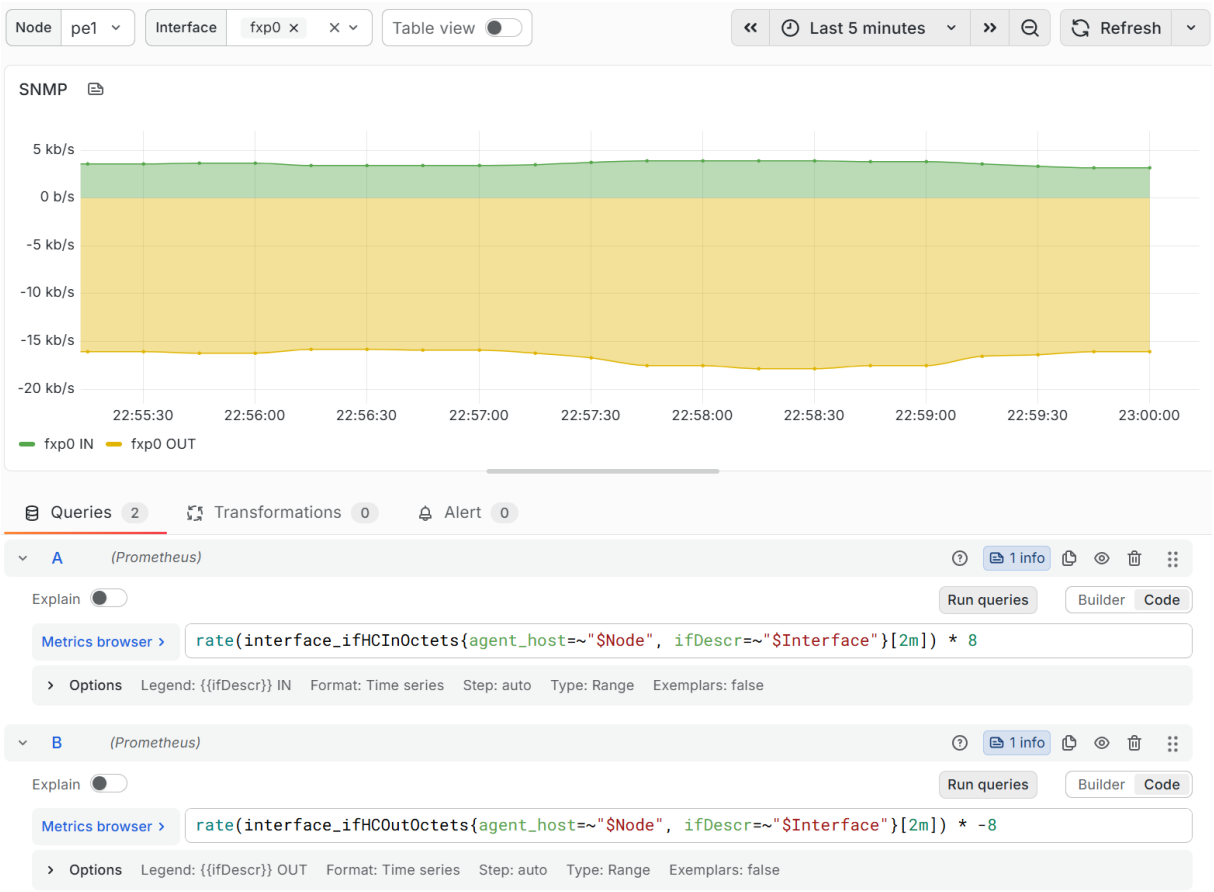
In contrast, the integration of gNMI paired with OpenConfig data models demonstrated unified, vendor-agnostic behavior. A single gNMI subscription successfully mapped to the same telemetry endpoints on both Arista and Juniper devices, considering minor result differences. This shared data modeling abstracted the underlying OS differences, simplifying the collector's configuration while guaranteeing mathematically consistent data structures across the multi-vendor topology.

4.1 Synthetic Volumetric Traffic

For data-plane observability, the testbed was subjected to a repeating volumetric traffic profile generated by *iperf3*. The network maintained a baseline throughput of 10 Mbps for 45 seconds, immediately followed by a simulated UDP micro-burst of 200 Mbps lasting exactly 15 seconds. The most critical operational requirement during a volumetric anomaly is accurately detecting the peak magnitude (bandwidth exhaustion) and the exact duration of the attack.

To visualize the SNMP telemetry pipeline, the Grafana dashboard queried the standardized *ifHCInOctets* and *ifHCOutOctets* SNMP MIBs. Since the SNMP collector was hard-limited to a 30-second polling interval to prevent router CPU starvation, the PromQL derivative function was mathematically constrained. As defined in the dashboard configuration, the rate calculation utilized a two-minute sliding window as illustrated on Figure 65:

Figure 65 – SNMP Interface Rate Traffic



Source: Author.

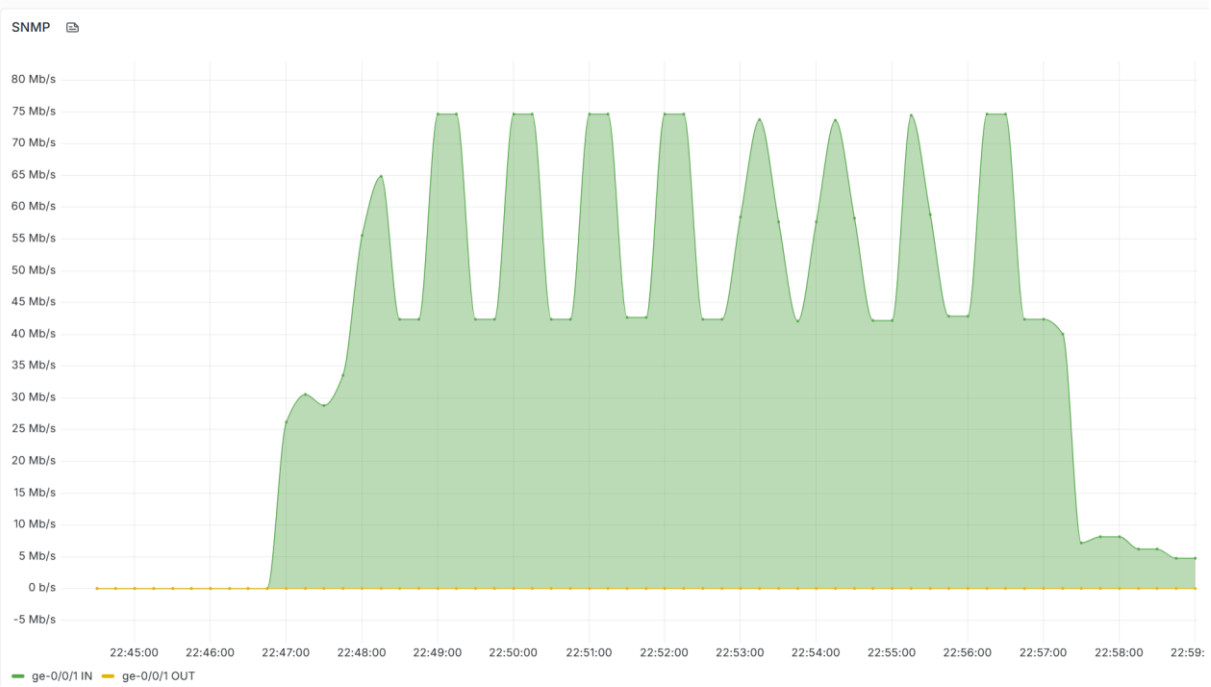
When subjected to the 15-second, 200 Mbps volumetric spike, the limitations of this legacy architecture became immediately apparent. The empirical data generated by the testbed perfectly replicated a known vulnerability in legacy anomaly detection systems. Academic literature corroborates that SNMP is inherently vulnerable to volumetric DDoS variants specifically designed to circumvent fixed threshold triggers [36], because SNMP calculates the average utilization over a fixed polling period (e.g., 30 seconds), an attacker can easily bypass detection by reducing the attack length rather than the total volume.

For example, a short 10-second burst at absolute link saturation (100% utilization) will not significantly impact the average bandwidth calculation when mathematically diluted across a standard 30-second polling window. Consequently, the SNMP architecture is fundamentally unaware of such attacks unless the micro-bursts cumulatively add up to an average that breaks the alarming threshold.

This exact phenomenon - known as telemetry aliasing - was visibly proven on the testbed's Grafana dashboard. Since the 15-second anomaly was significantly shorter than the

PromQL rate window, the mathematical derivative averaged the 200 Mbps burst across the entire timeframe. On the visualization panel shown on Figure 66, the massive volumetric spike rendered as a low, heavily diluted wave, peaking far below the actual 200 Mbps threshold.

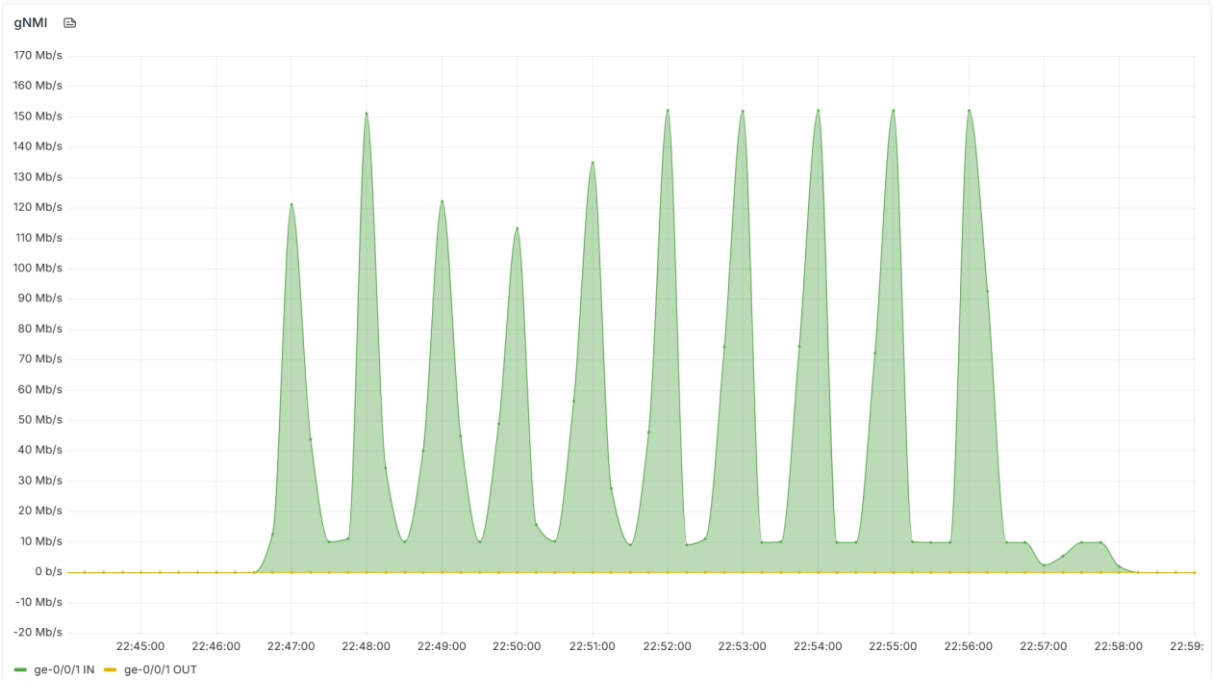
Figure 66 – SNMP DDoS Detection



Source: Author.

For comparison, the gNMI telemetry pipeline was explicitly configured to use the SAMPLE subscription mode with a 10-second interval for all interface traffic counters, as this mode is the most suitable for continuous, high-frequency bandwidth monitoring. When observing the exact same physical router interface during the identical 15-second micro-burst, the gNMI panel yielded a different visual result as Figure 67:

Figure 67 – gNMI DDoS Detection



Source: Author.

The dashboard rendered a nearly perfect, vertical square wave that accurately reflected the abrupt transition from the 10 Mbps baseline to the 200 Mbps ceiling.

4.2 LDP State Transition Delay

The mechanical limitations of polling are not restricted to data-plane bandwidth observability; they fundamentally impair control-plane state visibility. As demonstrated in Section 4.1, short-lived volumetric anomalies were mathematically obfuscated by SNMP's rigid 30-second polling cycle. To empirically validate whether this equally affects internal routing protocols, a high-volume statistical analysis was executed against the LDP states to simulate link outages.

The LDP event-driven detection comparison was based on Junos enterprise-specific SNMP notifications and an OpenConfig gNMI ON_CHANGE subscription due to deprecated MPLS-LDP-MIB support. On the SNMP side, an LDP session-down notification was identified by the Juniper enterprise trap OID *1.3.6.1.4.1.2636.4.4.0.4*. The notification included the *jnxMplsLdpSesState* object, whose base OID is *1.3.6.1.4.1.2636.3.36.1.3.3.1.2*.

On the gNMI side, the corresponding state was obtained through the OpenConfig path */network-instances/network-instance/mpls/signaling-protocols/ldp/neighbors/neighbor/state/session-state,* using a streaming ON_CHANGE

subscription. The event was classified as down when the gNMI path reported NON_EXISTENT (LSP down).

A Netmiko script was created to repeatedly induce intermittency OSPF neighborship failures across the core network infrastructure, resulting in 100 discrete *NON_EXISTENT* network events. Essentially, it deactivates OSPF instances and starts them over again:

Figure 68 – Juniper OSPF Intermittency Simulation

```
deactivate protocols ospf
commit confirmed
```

Source: Author.

The objective was to quantify via *gnmic* and *snmpget* the relative detection latency between the event-driven OpenConfig architecture (gNMI) and the polling architecture (SNMP).

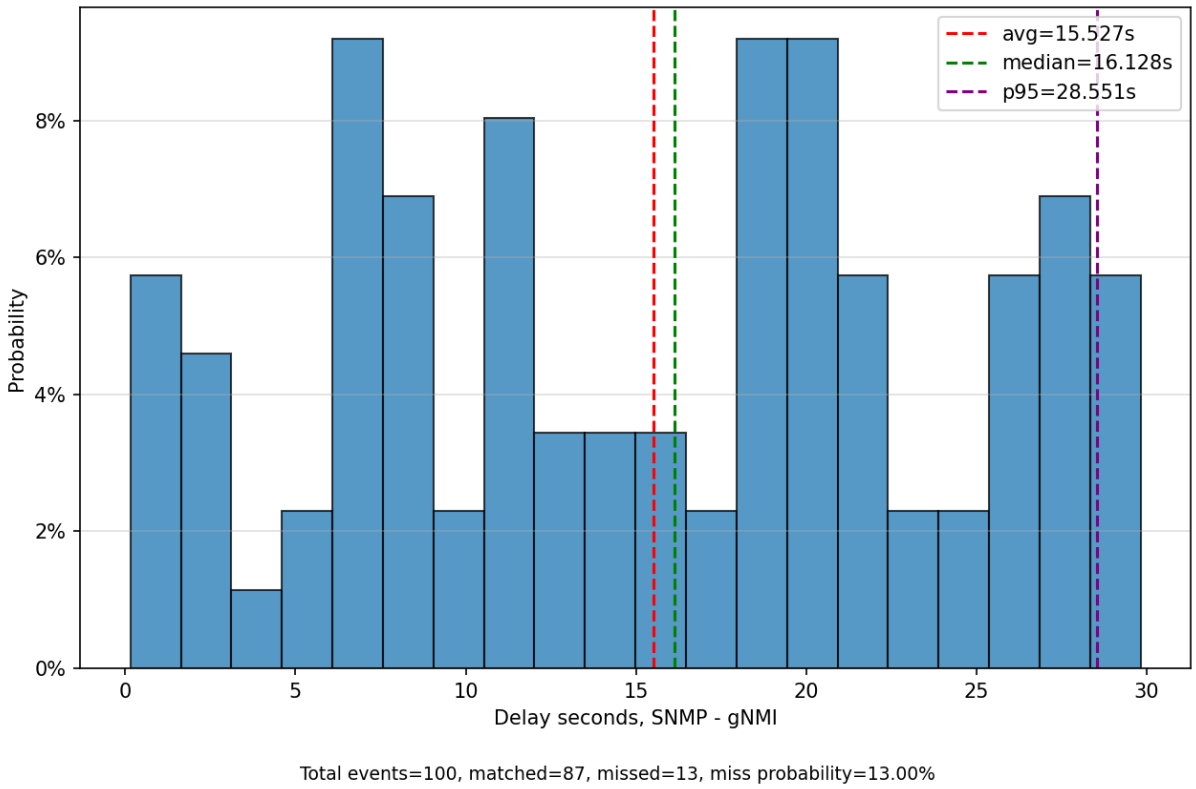
The histogram on Figure 69 represents the measured delay between the moment when gNMI detected the LDP session entering *NON_EXISTENT* state and the moment when SNMP detected the same state. In other words, difference in seconds when it states that the LDP session is down. The delay is calculated as:

$$\Delta t = gNMI_{timestamp} - SNMP_{timestamp}\ seconds \quad (3)$$

Therefore:

— $\Delta t < 0$: SNMP metric arrived before gNMI notification
— $\Delta t > 0$: gNMI arrived before SNMP
— $\Delta t = 0$: both arrived at the same instant

Figure 69 - LDP NON_EXISTENT State Detection Delay Histogram



Source: Author.

Therefore, the values mean that SNMP detected the LDP failure after gNMI. In this experiment, 100 LDP down events were generated. gNMI detected all 100 events, while SNMP detected 87 of them. The remaining 13 events were missed by SNMP, producing a miss probability of 13%.

This is an important result because it shows that, under intermittent failure conditions, SNMP polling may not only detect events later than gNMI. However, may completely miss short-lived failures.

Additionally, the bars show the probability of distribution of the delay for the 87 events that were detected by both protocols. The x-axis shows the delay in seconds, while the y-axis shows the probability of a detection falling into each delay interval. The distribution is spread between approximately 0 and 30 seconds, which indicates that SNMP detection latency varies significantly depending on where the failure occurs inside the SNMP polling cycle. Events that happen shortly before an SNMP poll are detected with low delay, while events that happen just after a poll may only be detected at the next polling cycle, resulting in a larger delay.

To get the average discrete events, the incongruence is used to calculate the Mean Time to Detect (MTTD). Each delay per event is measured, the total seconds of delay per number of network events consolidate the metric:

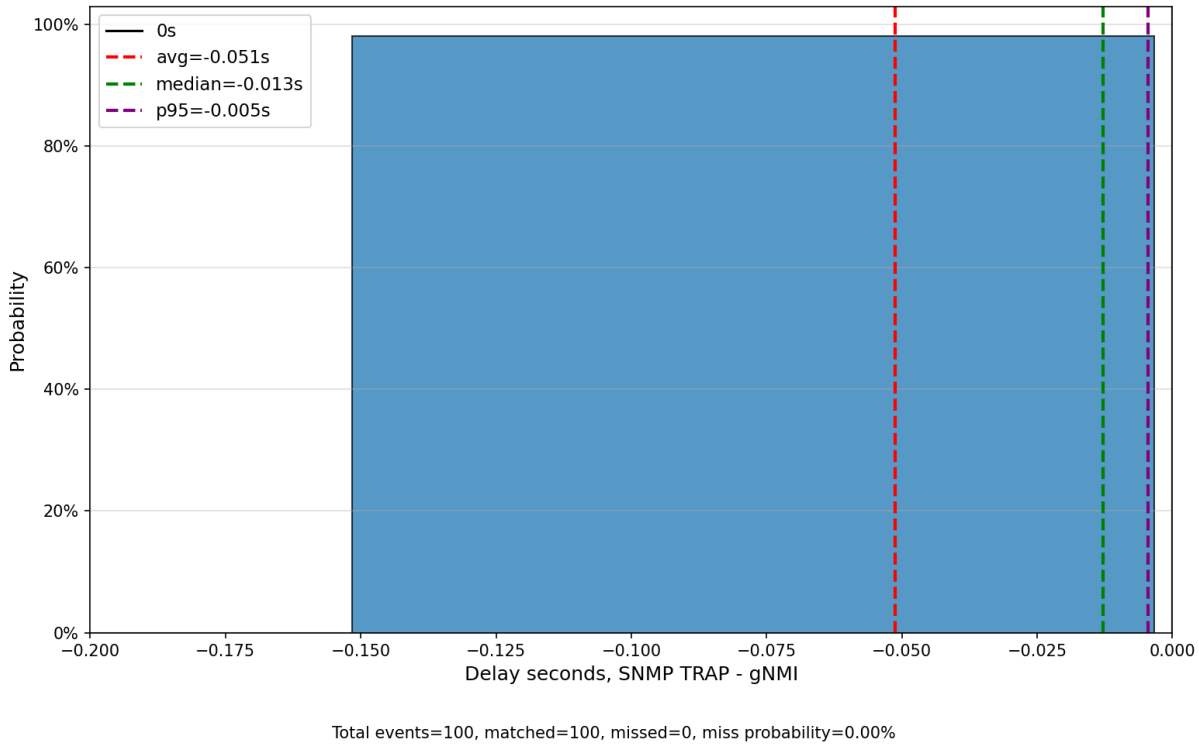
$$MTTD = \frac{\sum_{i=0}^{N} \Delta t_i}{N} \quad (4)$$

Finally, the MTTD measured for the 87 detected events is 15.527 seconds. In practical terms, for the matched events, SNMP typically detected the LDP failure around 15–16 seconds after gNMI. This behavior is consistent with a polling-based monitoring model, where the expected detection delay is strongly influenced by the polling interval and the random timing of the failure relative to the next poll.

The 95th percentile delay resulted in 28.551 seconds. This means that 95% of the SNMP detections occurred within approximately 28.6 seconds after gNMI detection, while the slowest 5% took longer. This percentile is especially relevant for operational monitoring because it represents a near-worst-case detection delay. In a production environment, this means that even when SNMP does detect the LDP failure, there is a non-negligible probability that the detection will occur close to 30 seconds after the event was already visible through gNMI telemetry.

For representability, SNMP TRAPs were measured as well compared to gNMI ON_CHANGE subscription model. The histogram Figure 70 presents the distribution of detection-time differences between notifications for the same 100 induced LDP state transitions:

Figure 70 – SNMP TRAP LDP State Transition Delay



Source: Author.

The median of -0.013 seconds indicates that, for a typical event, the SNMP trap arrived approximately 13 milliseconds before the gNMI notification. The 95th-percentile marker at −0.005 seconds means that 95% of the observed signed delays were less than or equal to approximately −5 milliseconds. Because the distribution is negative, this indicates that all observations showed an SNMP TRAP lead over gNMI, although the lead was usually very small. Operationally, both mechanisms detected the state transition almost simultaneously.

The average delay of -0.051 seconds is more negative than the median because the distribution contains a small number of large negative observations. The visible outliers around -0.4 and -3.0 seconds pull the arithmetic mean away from the dense group near zero. Consequently, the median better characterizes the typical event, whereas the mean reflects the influence of rare delayed gNMI observations.

Both mechanisms are event-driven, but they do not necessarily share the same internal processing path. On SNMP side, it does update the native state on MIB and generate a SNMP TRAP event, while gNMI notification requires an OpenConfig state update and model translation before encoding and export.

The slight SNMP advantage is plausibly associated with differences in the device and collector processing pipelines. SNMP TRAPs are asynchronous notifications generated by the

SNMP notification subsystem, as specified in RFC 3413 and RFC 3416. gNMI *ON_CHANGE*, defined by the OpenConfig specification, also provides event-driven updates, but the device may first need to map its native protocol state into the OpenConfig model and then encode and transmit the notification over gRPC. Therefore, it does not guarantee immediate or zero-latency delivery. The collector receives a structured notification only after the target has detected the change and produced the corresponding model update [6].

In addition, the measurement implementation timestamps gNMI after processing by the *gnmic* client and JSON parser, whereas the SNMP trap is timestamped directly after UDP reception. The measured differences therefore represent end-to-end collector visibility rather than pure protocol transport latency.

4.3 BGP Prefix-Count

In the architecture of Service Provider networks, the BGP dictates the flow of traffic between autonomous systems. Consequently, monitoring the Active BGP Prefix Count (the number of reachable network destinations advertised by a peer) is one of the most critical telemetry metrics for ensuring control-plane stability.

Anomalies in prefix counts are primary indicators of severe network events:

— Route Leaks: A sudden spike in received prefixes often indicates that a peer has accidentally advertised its entire routing table (e.g., a customer leaking the global internet table to an ISP), which can easily overwhelm the memory of smaller receiving routers.

— Network Partitions: A sudden drop in prefixes indicates lost reachability to downstream networks.

— Route Flapping: When a router rapidly advertises and withdraws the same set of prefixes, it causes route intermittency. Every single BGP update forces all receiving routers to recalculate their best-path algorithms and update their FIB. Excessive route flapping severely exhausts router CPU resources and can trigger systemic instability [37].

To adequately evaluate the latency and accuracy of monitoring protocols, it is necessary to subject the network monitoring system to a severe, deterministic route flapping scenario by artificially inducing a high-frequency control-plane anomaly.

The testbed required a stimulus generator that could inject a sudden peak of 200 routes into the core network, hold them active, and then entirely withdraw them exactly 15 seconds later. By repeating this 15-second cycle continuously, the generator creates a deterministic "square wave" of routing updates. This strict 15-second interval was chosen specifically to test

the limitations of standard 30-second SNMP polling intervals and demonstrate the mathematical aliasing effect.

For active prefix counts collection, the polling engine had to abandon the standard BGP4-MIB (OID *1.3.6.1.2.1.15*). The standard MIB, standardized decades ago, lacks native counters to distinguish BGP prefix counts across distinct Address Family Identifiers (AFI) and Subsequent Address Family Identifiers (SAFI), such as IPv4/IPV6 and VRFs. To overcome this limitation and retrieve the 200 flapping routes, the polling engine had to traverse enterprise MIB table on Arista Route Reflector under object *aristaBgp4V2PrefixInPrefixes* (*.1.3.6.1.4.1.30065.4.1.1.8.1.3*).

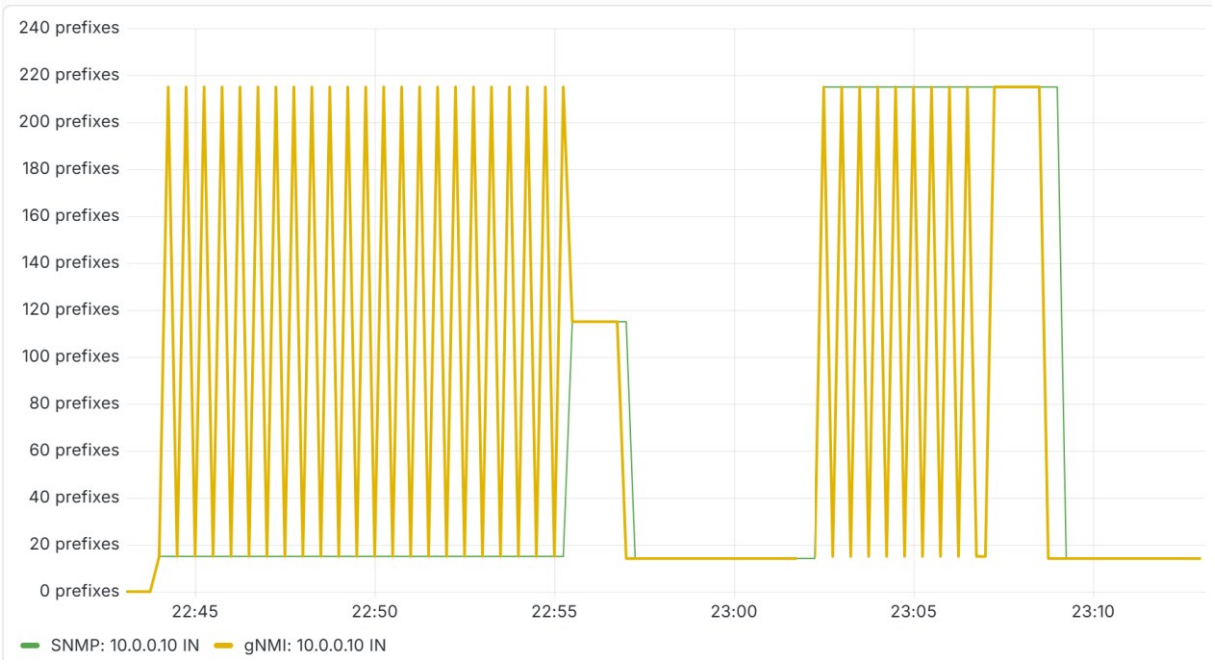
On gNMI side, it leverages the vendor-neutral OpenConfig path */network-instances/network-instance/protocols/protocol/bgp/neighbors/neighbor/afi-safis/afi-safi/state/prefixes*, which differentiates per SAFI and AFI codes.

To achieve sub-second precision in generating this anomaly, a Python script (*route-leak.py* on Appendix F) was developed to act as an automated prefix generator. The script executes within the namespace of the ExaBGP container, writing BGP announcement and withdrawal commands directly into ExaBGP's standard input pipe.

This Figure 71 demonstrates the limitation of interval-based polling (SNMP) when observing dynamic control-plane anomalies like BGP route flapping. Since the frequency of the network anomaly (the route flap) exceeded the frequency of the SNMP polling interval, the monitoring system suffered to data aliasing.

Between approximately 22:44 and 22:55, gNMI captures the repeated transitions between the low and high prefix-count states. The gNMI (yellow series) records each advertisement and withdrawal, clearly exposing the continuous route oscillation. SNMP, by contrast, remains predominantly at the lower level of approximately 15 prefixes. In this period, the SNMP polling instants aligned mainly with the withdrawn state. The collector consequently presented a false stable withdrawal condition, even though the route set was repeatedly being announced and withdrawn between polls.

Figure 71 – BGP Route Intermittency



Source: Author.

Finally, between approximately 22:57 and 23:02, both series remain near 15 prefixes, indicating a genuinely stable low-prefix state. Unlike the oscillating intervals, the agreement between the two mechanisms during this period indicates that SNMP polling represents persistent states adequately when those states last longer than the polling interval.

A second high-frequency oscillation begins around 23:02. gNMI again records repeated changes between approximately 15 and 215 prefixes. However, the SNMP line rises to approximately 215 prefixes and stays there throughout most of the oscillation. In this case, SNMP polls aligned with the announcement peaks. The resulting green line represents **false** stability at the high-prefix state: the dashboard suggests that all leaked routes remained continuously present, while gNMI shows that they were repeatedly withdrawn and re-advertised.

The graph demonstrates that the principal limitation of SNMP in this scenario is not necessarily an incorrect MIB value. Rather, it is temporal aliasing caused by periodic observation. When the BGP state changes several times between SNMP polls, the collector sees only the value present at the sampling instant. Depending on the phase alignment between polling and route oscillation, SNMP can produce opposite views.

Thus, SNMP can understate both the frequency and duration of BGP instability. More than that, the graph shows that event-driven telemetry can preserve the dynamic behavior of the

leak, while periodic polling may collapse many state transitions into one apparently stable value.

4.4 Operational Overhead

After evaluating detection latency and visibility of transient network events, the analysis was extended to the computational cost imposed on the telemetry collector. This aspect is relevant because increasing collection frequency improves observability but also increases the rate at which the collector must retrieve, decode, transform, and export monitoring data. A telemetry architecture must therefore be evaluated not only by its ability to detect events, but also by the resources required to sustain the desired temporal resolution.

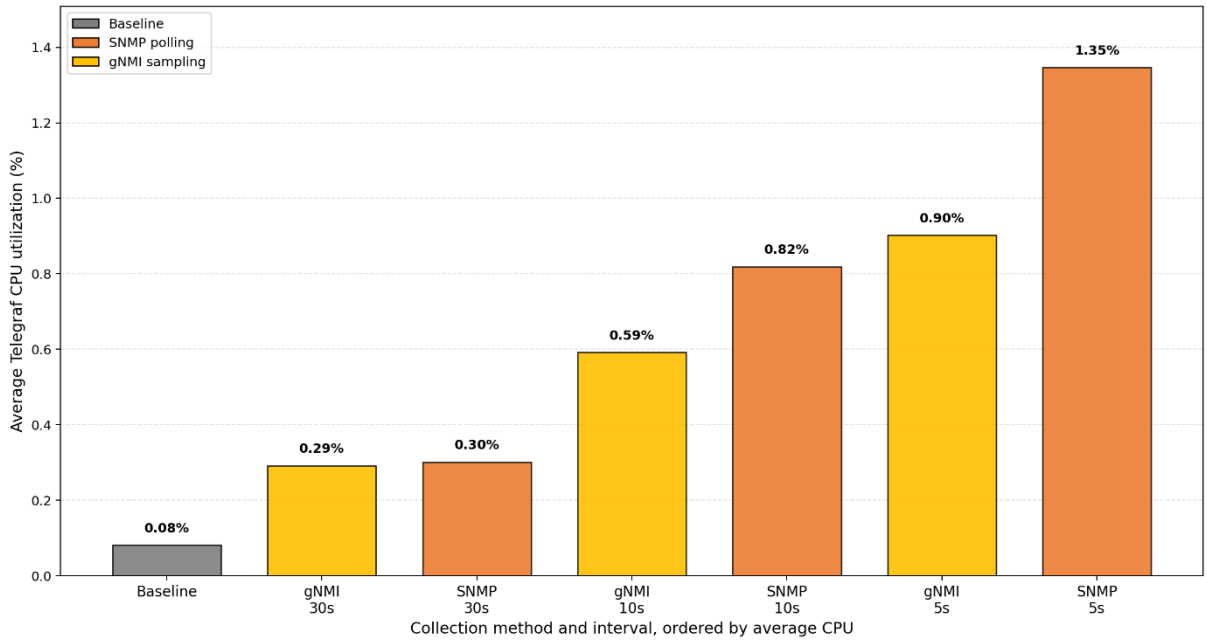
Telegraf was used as the common collector for both SNMP and gNMI, both to collect same metrics as well processing needed to normalize the data. Since these mechanisms follow different processing paths, their collector overhead was measured empirically. The objective was to determine how the average Telegraf CPU utilization changed under 15-minutes interval when the same functional monitoring scope was collected using SNMP polling and gNMI sampling at intervals of 30, 10, and 5 seconds.

The Telegraf collector was executed in a Docker container and monitored externally using Docker *cgroup* CPU statistics. A baseline phase with no active SNMP or gNMI collection was followed by independent SNMP and gNMI phases. During gNMI phases, all evaluated subscriptions used SAMPLE mode so that the configured retrieval interval could be compared directly with SNMP polling. The network workload and collector output configuration were held constant across all phases.

This procedure quantified the collector cost associated with increasing telemetry frequency and provided evidence for the scalability of SNMP and gNMI under the evaluated configuration.

Figure 72 presents the average Telegraf CPU utilization obtained from the baseline and from each SNMP and gNMI interval configuration on ascendant order:

Figure 72 – Average CPU Usage per Sampling and Polling Interval



Source: Author.

With no gNMI or SNMP active, the baseline was 0.08% average CPU usage. For gNMI, reducing the interval from 30 to 10 seconds increased average CPU usage from 0.29% to 0.59%, approximately doubling it. Reducing the interval again to 5 seconds increased it to 0.90%. SNMP followed the same general trend but increased more sharply: 0.30% at 30 seconds, 0.82% at 10 seconds, and 1.35% at 5 seconds.

This behavior is expected because shorter intervals cause the collector to process more telemetry cycles per unit of time. A 30-second interval generates approximately two collection cycles per minute, while 10- and 5-second intervals generate approximately six and twelve cycles per minute, respectively. The relationship is not perfectly proportional because Telegraf has fixed background costs and because protocol processing, metric transformation, buffering, and output handling may be shared or amortized across collection cycles.

At the 30-second interval, the two methods produced nearly identical average CPU utilization. The absolute difference was only 0.01% point, so this result should be interpreted as practical equivalence unless repeated experiments show that the difference is statistically stable.

At 10 seconds, SNMP required more than gNMI approximately:

$$\frac{0.82 - 0.59}{0.59} \approx 39\% \tag{5}$$

At 5 seconds, SNMP required even more:

$$\frac{1.35 - 0.90}{0.90} = 50\% \tag{6}$$

The result is consistent with the architectural differences between the two collection mechanisms. SNMP polling is based on repeated manager-initiated request/response transactions. For each polling cycle, Telegraf must query the configured MIB objects and tables, receive one or more responses, decode them, process table indexes, apply processors, and create output metrics. Larger tables may require multiple GETNEXT or GETBULK exchanges.

By contrast, gNMI sampling normally uses long-lived gRPC subscription streams. Once the subscriptions are established, the collector receives periodic notifications over persistent connections. This can reduce repeated connection and request orchestration, although gNMI still incurs protobuf decoding, path processing, field mapping, and metric generation costs.

The graph therefore suggests that the persistent streaming model scales more efficiently on the collector as the sampling frequency increases.

5 CONCLUSION

The experiments demonstrate that the relative performance of SNMP and gNMI depends strongly on the retrieval model being evaluated. SNMP polling, SNMP TRAPs, gNMI sampling, and gNMI *ON_CHANGE* should not be treated as equivalent mechanisms. Polling and sampling periodically retrieve information, whereas TRAPs and ON_CHANGE subscriptions are event-driven. Consequently, the results show different latency, reliability, and resource-consumption characteristics according to the operational scenario.

During the volumetric DDoS simulations, the limitations of pull-based monitoring were severely exposed. Since the SNMP was artificially constrained to a 30-second polling interval as standard between resources exhaustion, the protocol suffered from telemetry aliasing. It mathematically diluted the 15-second, 200 Mbps micro-bursts, completely failing to represent the true magnitude of the bandwidth exhaustion. In contrast, gNMI sampling under 10-seconds interval successfully captured the exact square-wave profile of the attack.

This aliasing vulnerability extended to the control plane during the BGP Prefix-Count experiment. When subjected to 15-second route flapping intervals, SNMP polling exhibited false stability by aligning with peaks, and later falsely perceived complete route withdrawals. However, gNMI event-driven subscription bypassed this entirely, maintaining a 1:1 synchronization with the router's ground-truth state.

When evaluating LDP state transitions, SNMP polling missed 13% of the intermittent failures entirely. For the 87% of events it did detect, SNMP exhibited a Mean Time to Detect (MTTD) of 15.527 seconds behind gNMI, with 95th-percentile delays approaching 30 seconds. This confirms that polling not only incurs significant latency but is fundamentally unreliable for capturing short-lived control-plane anomalies.

However, when comparing event-driven mechanisms directly (SNMP TRAPs vs. gNMI ON_CHANGE), both protocols successfully detected 100% of the LDP state transitions. Under these specific conditions, the legacy SNMP TRAP arrived at the collector with a median lead of 13 milliseconds over the gNMI notification. This marginal advantage is likely due to the internal processing pipeline; SNMP traps are native notifications, whereas gNMI requires the device to map native states into OpenConfig models and encode them via Protocol Buffers before transmission.

Moreover, the resource experiment on collector demonstrated that operational overhead is influenced not only by the telemetry protocol, but also by the configured collection interval. Both SNMP polling and gNMI sampling increased Telegraf CPU utilization as the interval was reduced. On higher frequency intervals, SNMP could require 50% more CPU than gNMI. These

results indicate that the persistent streaming model used by gNMI can provide better collector-side efficiency as telemetry frequency increases, while the repeated request-response and MIB-processing operations associated with SNMP become progressively more costly.

Ultimately, to fulfill the objectives of this study, gNMI is recommended as the definitive standard for modern Service Provider environments. While SNMP remain a viable, low-latency mechanism for hardware alerts, gNMI has proven maturity for high-frequency data-plane metrics, complex control-plane tracking, granularity and resource usage.

5.1 Future Work

Although this study successfully quantifies the operational differences between SNMP and gNMI in a software-virtualized environment, several opportunities for future research remain.

First, because this testbed utilized Containerlab and software-based routing nodes, the computational overhead observed on the devices does not perfectly mirror physical network elements. Future work should transition from software virtualization to real-world scenarios running on physical hardware that natively supports the gNMI streaming protocol. Evaluating the exact CPU and Disk I/O offloading benefits of gNMI streaming on physical commercial ASICs will provide a definitive resource trade-off analysis.

Second, the high-fidelity observability provided by gNMI streaming telemetry establishes the necessary foundation for next-generation autonomous network operations. Future research should explore the integration of continuous telemetry streams with Large Language Models and Agentic Artificial Intelligence as initially done by [39]. Streaming telemetry can act as a real-time sensory data layer for distributed agents, allowing models to translate high-level operator intents into structured, automated tasks. By feeding the precise gNMI anomaly data captured in this study into an LLM-driven control loop, networks could achieve autonomous fault localization, real-time alarm refinement, and dynamic BGP traffic engineering without human intervention.

Finally, the simulated architecture could be expanded into a comprehensive digital twin framework. By securely linking continuous gNMI telemetry streams to a dynamic virtual replica of the physical network, future studies could allow AI agents to safely simulate and validate complex recovery steps before executing those configuration changes on the live production network.

6 REFERENCES

[1] ARIEFPUTRA, K. F; MULYANA, E. *Performance Analysis of gNMI Streaming Telemetry Based Monitoring Systems Using Containerlab Network Simulation*. Jurnal Nasional Teknik Elektro dan Teknologi Informasi, Bandung, v. 13, n. 2, p. 101-106, may 2024. ISSN 2460-5719. DOI 10.22146/jnteti.v13i2.10185.

[2] JUNIPER, 2025. *BGP Overview*. Available: https://www.juniper.net/documentation/us/en/software/junos/bgp/topics/topic-map/bgp-overview.html.

[3] SÁNCHEZ-MONGE, A.; SZARKOWICZ, K., G. *MPLS in the SDN Era: Interoperable Scenarios to Make Networks Scale to New Services*. Sebastopol: O'Reilly Media, 2015. ISBN 978-1491905456.

[4] NEW RELIC, 2024. *Observability Forecast.* Available: https://newrelic.com/sites/default/files/2024-10/new-relic-2024-observability-forecast-report_0.pdf.

[5] IBM. *What is network observability?.* Available: https://www.ibm.com/think/topics/network-observability.

[6] OPENCONFIG. *gNMI - gRPC Network Management Interface*. IETF, 2018. Available: https://datatracker.ietf.org/doc/html/draft-openconfig-rtgwg-gnmi-spec-01.

[7] GOOGLE. *Protobuf Documentation*. Available: https://protobuf.dev/

[8] NOKIA. *Containerlab: A free and simple way to build your complex virtual test lab.* Available: https://www.nokia.com/blog/containerlab-a-free-and-simple-way-to-build-your-complex-virtual-test-lab/.

[9] ROSEN, E. VISWANATHAN, A. CALLON, R. *Multiprotocol Label Switching Architecture*, RFC 3031, The Internet Society, January 2001. Available: http://www.ietf.org/rfc/rfc3031.txt.

[10] ANDERSSON, L. MINEI, I. THOMAS, B. *LDP Specification*. RFC 5036. IETF, Oct. 2007. Available: https://datatracker.ietf.org/doc/rfc5036/.

[11] AWDUCHE, D. BERGER, L. GAN, D. LI, T. SRINIVASAN, V. SWALLOW, G. *RSVP-TE: Extensions to RSVP for LSP Tunnels*, RFC 3209, Dec. 2001. Available: https://datatracker.ietf.org/doc/rfc3209/.

[12] MINEI, I.; LUCEK, J. *MPLS-Enabled Applications: Emerging Developments and New Technologies*. 3. ed. Chichester: John Wiley & Sons, 2011.

[13] CUCCHIARA, C.; SJOSTRAND, H.; LUCIANI, J. *Definitions of Managed Objects for the Multiprotocol Label Switching (MPLS), Label Distribution Protocol (LDP)*. RFC 3815. IETF, Jun. 2004. Available: https://datatracker.ietf.org/doc/rfc3815/.

[14] REKHTER, Y.; LI, T.; HARES, S. *A Border Gateway Protocol 4 (BGP-4).* RFC 4271. IETF, Jan. 2006. Available: https://datatracker.ietf.org/doc/rfc4271/.

[15] HALABI, S.; MCPHERSON, D. *Internet Routing Architectures*. 2. ed. Indianapolis: Cisco Press, 2000.

[16] ZHANG, S.; LIU, Y.; PEI, D. *A measurement study on BGP AS path looping (BAPL) behavior*. In: 23RD INTERNATIONAL CONFERENCE ON COMPUTER COMMUNICATION AND NETWORKS (ICCCN), 2014. IEEE, Aug. 2014. DOI: 10.1109/ICCCN.2014.6911734.

[17] MYERS, K. *Juniper to MikroTik – BGP commands*. StubArea51.net, 24 Jan. 2021. Available: https://stubarea51.net/2021/01/24/juniper-to-mikrotik-bgp-commands/.

[18] NETWORKWALKS ACADEMY. *BGP (Border Gateway Protocol).* Available: https://networkwalks.com/bgp-border-gateway-protocol/.

[19] BATES, T.; CHEN, E.; CHANDRA, R. *BGP Route Reflection: An Alternative to Full Mesh Internal BGP*. RFC 4456. Internet Engineering Task Force (IETF), Apr. 2006. Available: https://datatracker.ietf.org/doc/html/rfc4456.

[20] VINASITHTHAMBY, S. *BGP Route Reflectors, Originator ID and Cluster ID.* Packetswitch, Sep. 23, 2025. Available: https://www.packetswitch.co.uk/bgp-route-reflectors-originator-id-and-cluster-id/.

[21] INTERNET ASSIGNED NUMBERS AUTHORITY (IANA). *Structure of Management Information (SMI) Numbers (MIB Module Registrations).* 24 Apr. 2026. Available: https://www.iana.org/assignments/smi-numbers.

[22] HABERMAN, B. *IP Forwarding Table MIB*. RFC 4292. Internet Engineering Task Force (IETF), Apr. 2006. Available: https://datatracker.ietf.org/doc/html/rfc4292.

[23] CASE, J.; MUNDY, R.; PARTAIN, D.; STEWART, B. *Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP)*. RFC 3416. Internet Engineering Task Force, Dec. 2002. Available: https://datatracker.ietf.org/doc/html/rfc3416.

[24] CASE, J.; FEDOR, M.; SCHOFFSTALL, M.; DAVIN, J. *A Simple Network Management Protocol (SNMP)*. RFC 1157. Internet Engineering Task Force (IETF), May 1990. Available: https://datatracker.ietf.org/doc/html/rfc1157.

[25] CASE, J.; MUNDY, R.; PARTAIN, D.; STEWART, B. *Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP)*. RFC 3416. Internet Engineering Task Force (IETF), Dec. 2002. Available: https://datatracker.ietf.org/doc/html/rfc3416.

[26] NETWORK ACADEMY. *Simple Network Management Protocol (SNMP)*. NetworkAcademy.io. Available: https://www.networkacademy.io/ccna/network-services/simple-network-management-protocol-snmp.

[27] CASE, J.; MUNDY, R.; PARTAIN, D.; STEWART, B. *Introduction to Version 3 of the Internet-standard Network Management Framework*. RFC 2570. Internet Engineering Task Force (IETF), Apr. 1999. Available: https://www.rfc-editor.org/rfc/rfc2570.html.

[28] CLEMM, A.; VOIT, E. *Subscription to YANG Datastores*. RFC 8641. Internet Engineering Task Force (IETF), Sep. 2019. Available: https://datatracker.ietf.org/doc/html/rfc8641.

[29] CODILIME. *YANG: a Data Model Approach to Network Management*. Aug. 2024. Available: https://codilime.com/blog/yang-data-modeling-approach-to-network-management/.

[30] BELSHE, M.; PEON, R.; THOMSON, M. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540. Internet Engineering Task Force (IETF), May 2015. Available: https://datatracker.ietf.org/doc/html/rfc7540.

[31] GRIGORIK, I. HTTP/2. In: GRIGORIK, I. *High Performance Browser Networking*. Sebastopol: O'Reilly Media, 2013.

[32] SÁNCHEZ, E.; GODOY, H. A. de; FIGUEROA, D. A. *Container-Based Virtualization for Network Environments*. ReVITA, 2024.

[33] KOBAYASHI, S.; SHIIBA, R.; MIWA, S.; MIYACHI, T.; FUKUDA, K. *Topology-Driven Configuration of Emulation Networks With Deterministic Templating*. IEEE Transactions on Network and Service Management, v. 22, n. 5, p. 3933-3946, 2025.

[34] ZIELIŃSKI, B. *Assessment of iPerf as a Tool for LAN Throughput Prediction*. INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, v. 69, n. 3, p. 523-528, 2023. DOI: 10.24425/ijet.2023.146501.

[35] ROQUERO, P.; ARACIL, J. *On Performance and Scalability of Cost-Effective SNMP Managers for Large-Scale Polling*. IEEE Access, 2021. DOI: 10.1109/ACCESS.2021.3049310.

[36] AUN, Y.; KHAW, Y. J.; GAN, M. L.; PONNUSAMY, V. *Adaptive Polling Rate for SNMP for Detecting Elusive DDOS*. Journal of Cyber Security, v. 4, n. 1, p. 17-28, 2022. DOI: 10.32604/jcs.2022.027524.

[37] VILLAMIZAR, C.; CHANDRA, R.; GOVINDAN, R. *BGP Route Flap Damping*. RFC 2439. Internet Engineering Task Force (IETF), Nov. 1998. Available: https://datatracker.ietf.org/doc/html/rfc2439.

[38] VILALTA, R.; MANSO, C.; YOSHIKANE, N.; MUÑOZ, R.; CASELLAS, R.; MARTÍNEZ, R. *Telemetry-enabled Cloud-native Transport SDN Controller for Real-time Monitoring of Optical Transponders Using gNMI*. In: 2020 European Conference on Optical Communications (ECOC), p. 1-4, 2020. DOI: 10.1109/ECOC48923.2020.9333143.

[39] CRUZES, S. *Telemetry and Agentic AI: Foundations for Optical Network Automation*. TechRxiv, Dec. 2025. DOI: 10.36227/techrxiv.176539548.84734375/v2.

[40] HPE, 2024. *SNMPv1 trap examples*. Available: https://arubanetworking.hpe.com/techdocs/AOS-S/16.11/MCG/KB/content/kb/snm-tra-cap-exa.htm

APPENDIX A – CONTAINERLAB YAML FILE

This YAML file is used for network initialization and definition of nodes on Containerlab.

```yaml
name: gnmi-lab
prefix: ""

mgmt:
  network: core-mgmt
  ipv4-subnet: 10.10.10.0/24
  ipv6-subnet: 2001:db8::/64

topology:
  kinds:
    juniper_vjunosrouter:
      image: vrnetlab/juniper_vjunos-router:25.4R1.12
    arista_ceos:
      image: ceos:4.36.0F

  nodes:
    # Monitoring pool
    telegraf:
      #collector
      kind: linux
      image: telegraf:latest
      mgmt-ipv4: 10.10.10.201
      binds:
        - configs/telegraf.conf:/etc/telegraf/telegraf.conf:ro

    prometheus:
      #database
      kind: linux
      image: prom/prometheus:latest
      mgmt-ipv4: 10.10.10.202
      binds:
        - configs/prometheus.yml:/etc/prometheus/prometheus.yml:ro
      ports:
        - 9090:9090

    grafana:
      #dashboard
      kind: linux
      image: grafana/grafana:latest
      mgmt-ipv4: 10.10.10.203
      ports:
        - 3000:3000
      env:
```

```
        GF_SECURITY_ADMIN_USER: admin
        GF_SECURITY_ADMIN_PASSWORD: admin
      binds:
        -
configs/grafana/datasources:/etc/grafana/provisioning/datasources:ro
        - configs/grafana/dashboards:/etc/grafana/provisioning/dashboards:ro

    # Nodes pool
    #control plane
    rr:
      kind: arista_ceos
      mgmt-ipv4: 10.10.10.100
      startup-config: configs/rr.cfg
    pe1:
      kind: juniper_vjunosrouter
      mgmt-ipv4: 10.10.10.10
      startup-config: configs/pe1.cfg
      env:
        CLAB_MGMT_PASSTHROUGH: "true"
    pe2:
      kind: juniper_vjunosrouter
      mgmt-ipv4: 10.10.10.20
      mgmt-ipv6: 2001:db8::20
      startup-config: configs/pe2.cfg
      env:
        CLAB_MGMT_PASSTHROUGH: "true"
    #data plane
    p1:
      kind: arista_ceos
      mgmt-ipv4: 10.10.10.30
      startup-config: configs/p1.cfg
    p2:
      kind: arista_ceos
      mgmt-ipv4: 10.10.10.40
      startup-config: configs/p2.cfg
    p3:
      kind: arista_ceos
      mgmt-ipv4: 10.10.10.50
      startup-config: configs/p3.cfg
    #customer edges/hosts
    ce1:
      kind: linux
      image: pierky/exabgp
      binds:
        - configs/exabgp/ce1.conf:/etc/exabgp/exabgp.conf:ro
        - configs/exabgp/dynamic_routes.py:/etc/exabgp/dynamic_routes.py:ro
      cmd: exabgp /etc/exabgp/exabgp.conf
      exec:
        - sysctl -w net.ipv4.ip_forward=1
```

```
      - ip addr add 10.50.0.1/24 dev eth2
      - ip addr add 192.168.11.2/24 dev eth1
      - ip route del default
      - ip route add default via 192.168.11.1 dev eth1
  ce2:
    kind: linux
    image: pierky/exabgp
    binds:
      - configs/exabgp/ce2.conf:/etc/exabgp/exabgp.conf:ro
      - configs/exabgp/route_leak.py:/run/route_leak.py:rw
    cmd: exabgp /etc/exabgp/exabgp.conf
    exec:
      - sysctl -w net.ipv4.ip_forward=1
      - ip addr add 10.60.0.1/24 dev eth2
      - ip addr add 192.168.12.2/24 dev eth1
      - ip route del default
      - ip route add default via 192.168.12.1 dev eth1
  iperf-client:
    kind: linux
    image: wbitt/network-multitool
    exec:
      - ip addr add 10.50.0.100/24 dev eth1
      - ip route del default
      - ip route add default via 10.50.0.1
  iperf-server:
    kind: linux
    image: wbitt/network-multitool
    exec:
      - ip addr add 10.60.0.100/24 dev eth1
      - ip route del default
      - ip route add default via 10.60.0.1
      - iperf3 -s -D
links:
  # Route Reflector to PEs
  - endpoints: [ "rr:eth1", "pe1:ge-0/0/0" ]
    ipv4: [ "10.1.10.0/31", "10.1.10.1/31" ]
  - endpoints: [ "rr:eth2", "pe2:ge-0/0/0" ]
    ipv4: [ "10.1.20.0/31", "10.1.20.1/31" ]
  # PE to P-Router Uplinks
  - endpoints: [ "pe1:ge-0/0/2", "p1:eth1" ]
    ipv4: [ "10.1.1.0/31", "10.1.1.1/31" ]
  - endpoints: [ "pe2:ge-0/0/2", "p2:eth1" ]
    ipv4: [ "10.1.2.0/31", "10.1.2.1/31" ]
  # Core P-Router Mesh
  - endpoints: [ "p1:eth2", "p2:eth2" ]
    ipv4: [ "10.1.12.0/31", "10.1.12.1/31" ]
  - endpoints: [ "p3:eth1", "p2:eth3" ]
    ipv4: [ "10.1.23.0/31", "10.1.23.1/31" ]
  - endpoints: [ "p3:eth2", "p1:eth3" ]
```

```
    ipv4: [ "10.1.13.0/31", "10.1.13.1/31" ]
  # Provider Edge to Customer Edge
  - endpoints: [ "pe1:ge-0/0/1", "ce1:eth1" ]
    ipv4: [ "192.168.11.1/24", "192.168.11.2/24" ]
  - endpoints: [ "pe2:ge-0/0/1", "ce2:eth1" ]
    ipv4: [ "192.168.12.1/24", "192.168.12.2/24" ]
  # iperf3 to Customer Edge
  - endpoints: ["ce1:eth2", "iperf-client:eth1"]
    ipv4: ["10.50.0.1/24", "10.50.0.100/24"]
  - endpoints: ["ce2:eth2", "iperf-server:eth1"]
    ipv4: ["10.60.0.1/24", "10.60.0.100/24"]
```

APPENDIX B – JINJA2 TEMPLATES

Template for Juniper Provider Edge routers configuration file provisioning:

```
# Jinja2 template for PE-Routers
# System
system {
    host-name {{ hostname }};

    root-authentication {
        encrypted-password "{{ 'admin' | junos_hash }}";
    }

    login {
        user admin {
            uid 2000;
            class super-user;
            authentication {
                encrypted-password "{{ 'admin' | junos_hash }}";
            }
        }
    }
# gNMI Configuration
    services {
        extension-service {
            request-response {
                grpc {
                    routing-instance mgmt_junos;
                    clear-text {
                        port 32767;
                    }
                }
            }
        }
    }
}

# Interface Configuration
interfaces {
    lo0 {
```

```
            unit 0 {
                family inet {
                    address {{ loopback_ip }};
                }
                family mpls;
            }
        }
        {{ rr_iface }} {
            description "to RR";
            unit 0 {
                family inet {
                    address {{ rr_iface_ip }};
                }
            }
        }
        {{ core_iface }} {
            description "to P-router";
            unit 0 {
                family inet {
                    address {{ core_ip }};
                }
                family mpls;
            }
        }
        {{ ce_iface }} {
            description "to CE";
            unit 0 {
                family inet {
                    address {{ ce_ip }};
                }
            }
        }
    }

    policy-options {
        policy-statement NEXT-HOP-SELF {
            then {
                next-hop self;
                accept;
```

```
            }
        }
    }
    # Protocols Configuration
    routing-options {
        router-id {{ loopback_ip | replace("/32", "") }};
        autonomous-system 65000;
    }

    protocols {
        ospf {
            area 0.0.0.0 {
                interface lo0.0 {
                    passive;
                }
                interface {{ core_iface }}.0 {
                    interface-type p2p;
                }
                interface {{ rr_iface }}.0 {
                    interface-type p2p;
                }
            }
        }
        mpls {
            interface {{ core_iface }}.0;
        }
        ldp {
            track-igp-metric;
            interface {{ core_iface }}.0;
            interface lo0.0;
        }
        bgp {
            group IBGP-RR {
                type internal;
                local-address {{ loopback_ip | replace("/32", "") }};
                family inet {
                    unicast;
                }
                family inet-vpn {
```

```
                        unicast;
                    }
                    export NEXT-HOP-SELF;
                    neighbor {{ rr_ip }};
                }
                group EBGP {
                    type external;
                    peer-as {{ ce_peer_as }};
                    neighbor {{ ce_peer_ip }};
                }
            }
        }
        # SNMP Configuration
        snmp {
            community public {
                authorization read-only;
                routing-instance mgmt_junos;
            }
            trap-group TRAPS {
                trap-group TRAPS {
                version v2;
                destination-port 1620;
                routing-instance mgmt_junos;
                targets {
                    10.10.10.1;
                    {{ nms_ip }};
                    }
                }
            }
            routing-instance-access;
}
```

Template for Arista Provider routers configuration file provisioning:

```
        ! Jinja2 template for P-Routers
        hostname {{ hostname }}
        username admin privilege 15 secret admin
        ip routing
        service routing protocols model multi-agent
```

```
! Telemetry
management api gnmi
   transport grpc default
   provider eos-native
snmp-server community public ro
snmp-server host {{ nms_ip }} traps version 2c public udp-port 1620
snmp-server enable traps mpls-ldp

! LDP Configuration
mpls ip
mpls ldp
   router-id interface Loopback0
   transport-address interface Loopback0
   no shutdown
! Interfaces Configuration
interface Ethernet1
   no switchport
   ip address {{ eth1_ip }}
   ip ospf network point-to-point
   ip ospf area 0.0.0.0
   mpls ip
   mpls ldp interface
!
interface Ethernet2
   no switchport
   ip address {{ eth2_ip }}
   ip ospf network point-to-point
   ip ospf area 0.0.0.0
   mpls ip
   mpls ldp interface
!
interface Ethernet3
   no switchport
   ip address {{ eth3_ip }}
   ip ospf network point-to-point
   ip ospf area 0.0.0.0
   mpls ip
   mpls ldp interface
!
```

```
interface Loopback0
   ip address {{ loopback_ip }}
   ip ospf area 0.0.0.0
   mpls ip
   mpls ldp interface
!
! OSPF Underlay
router ospf 1
   router-id {{ loopback_ip | replace("/32", "") }}
   passive-interface Loopback0
!
 end
```

APPENDIX C – ROUTE REFLECTOR STARTUP FILE

Arista cEOS configuration file for Route Reflector:

```
! RR Configuration

hostname rr
username admin privilege 15 secret admin
service routing protocols model multi-agent
ip routing

! Telemetry
! Enable the gNMI on the default port and SNMP

management api gnmi
   transport grpc default
   provider eos-native
snmp-server community public ro

!
! Interface Configuration
!
interface Ethernet1
   description to PE1
   no switchport
   ip address 10.1.10.0/31
   ip ospf network point-to-point
   ip ospf area 0.0.0.0
!
interface Ethernet2
   description to PE2
   no switchport
   ip address 10.1.20.0/31
   ip ospf network point-to-point
   ip ospf area 0.0.0.0
!
interface Loopback0
   description BGP RouterID
   ip address 10.0.0.100/32
   ip ospf area 0.0.0.0

! 3. OSPF Underlay

router ospf 1
   router-id 10.0.0.100
   passive-interface Loopback0
   ! Avoid RR as transit path
   max-metric router-lsa
```

```
! BGP Overlay (AS 65000)

router bgp 65000
   router-id 10.0.0.100

   ! iBGP Group
   neighbor IBGP-PE peer group
   neighbor IBGP-PE remote-as 65000
   neighbor IBGP-PE update-source Loopback0
   neighbor IBGP-PE route-reflector-client
   neighbor IBGP-PE send-community extended

   ! Peering with PE1 and PE2 Loopbacks
   neighbor 10.0.0.10 peer group IBGP-PE
   neighbor 10.0.0.20 peer group IBGP-PE

   ! Enable VPNv4 Reflection
   address-family ipv4
      neighbor IBGP-PE activate
!
end
```

APPENDIX D – *BUILD.PY* SCRIPT

Automation provisioning based on Jinja2 templates:

```
import os
from passlib.hash import md5_crypt
from jinja2 import Environment, FileSystemLoader

def hash_junos_password(password):
    return md5_crypt.hash(password, salt="salt")

env = Environment(loader=FileSystemLoader('.'))
env.filters['junos_hash'] = hash_junos_password

pe_template = env.get_template('pe-router.j2')
p_template = env.get_template('p-router.j2')

# Map nodes as JSON into the template
pe_routers = [
{
        "hostname": "pe1",
        "loopback_ip": "10.0.0.10/32",
        "nms_ip": "10.10.10.201",
        # to RR
        "rr_iface": "ge-0/0/0",
        "rr_iface_ip": "10.1.10.1/31",
        "rr_ip": "10.0.0.100",
        # to P-router
        "core_iface": "ge-0/0/2",
        "core_ip": "10.1.1.0/31",
        # to CE
        "ce_iface": "ge-0/0/1",
        "ce_ip": "192.168.11.1/24",
        "ce_peer_ip": "192.168.11.2",
        "ce_peer_as": "65001"
    },
    {
        "hostname": "pe2",
        "loopback_ip": "10.0.0.20/32",
        "nms_ip": "10.10.10.201",
        # to RR
        "rr_iface": "ge-0/0/0",
        "rr_iface_ip": "10.1.20.1/31",
        "rr_ip": "10.0.0.100",
        # to P-router
        "core_iface": "ge-0/0/2",
        "core_ip": "10.1.2.0/31",
        # to CE
        "ce_iface": "ge-0/0/1",
```

```
        "ce_ip": "192.168.12.1/24",
        "ce_peer_ip": "192.168.12.2",
        "ce_peer_as": "65002"
    }
]

p_routers = [
    {
        "hostname": "p1",
        "loopback_ip": "10.0.0.30/32",
        "nms_ip": "10.10.10.201",
        "eth1_ip": "10.1.1.1/31",    # to PE1
        "eth2_ip": "10.1.12.0/31",   # to P2
        "eth3_ip": "10.1.13.1/31"    # to P3
    },
    {
        "hostname": "p2",
        "loopback_ip": "10.0.0.40/32",
        "nms_ip": "10.10.10.201",
        "eth1_ip": "10.1.2.1/31",    # to PE2
        "eth2_ip": "10.1.12.1/31",   # to P1
        "eth3_ip": "10.1.23.1/31"    # to P3
    },
    {
        "hostname": "p3",
        "loopback_ip": "10.0.0.50/32",
        "nms_ip": "10.10.10.201",
        "eth1_ip": "10.1.23.0/31",   # to P2
        "eth2_ip": "10.1.13.0/31",   # to P1
        "eth3_ip": "127.0.0.1/8"     # dummy config
    }
]

for router in pe_routers:
    config = pe_template.render(router)
    filename = f"{router['hostname']}.cfg"

    with open(filename, 'w') as file:
        file.write(config)
    print(f"File {filename} generated")

for router in p_routers:
    config = p_template.render(router)
    filename = f"{router['hostname']}.cfg"

    with open(filename, 'w') as file:
        file.write(config)
    print(f"File {filename} generated")
```

APPENDIX E – *IPERF.PY* SCRIPT

Traffic simulation and Docker commands injection with *iperf3* tool:

```
import subprocess
import time
import sys


CLIENT_NODE = "iperf-client"
SERVER_IP = "10.60.0.100"

# Traffic profile
BASELINE_BW = "10M"
BASELINE_TIME = 45

PEAK_BW = "200M"
PEAK_TIME = 15

def run_iperf(bandwidth, duration, phase_name):
    print(f"[{time.strftime('%H:%M:%S')}] Starting {phase_name} Phase:
{bandwidth}bps for {duration}s")
    cmd = f"docker exec {CLIENT_NODE} iperf3 -c {SERVER_IP} -u -b {bandwidth}
-t {duration}"

    # Block the script until the iperf3 timer finishes
    subprocess.run(cmd, shell=True, stdout=subprocess.DEVNULL,
stderr=subprocess.DEVNULL)
try:
    while True:
        # Normal Traffic
        run_iperf(BASELINE_BW, BASELINE_TIME, "BASELINE")

        # Volumetric Spike
        run_iperf(PEAK_BW, PEAK_TIME, "ATTACK PEAK")

except KeyboardInterrupt:
    print("\nSimulator stopped by user.")

    cleanup_cmd = f"docker exec {CLIENT_NODE} pkill iperf3"
    subprocess.run(cleanup_cmd, shell=True, stdout=subprocess.DEVNULL,
stderr=subprocess.DEVNULL)
    sys.exit(0)
```

APPENDIX F – *ROUTE_LEAK.PY* SCRIPT

Simulation of BGP routes manipulation with ExaBGP:

```
import sys
import time
import random

if len(sys.argv) != 4:
    sys.stderr.write("Usage: dynamic_routes.py <base_octet> <next_hop>
<local_as>\n")
    sys.exit(1)

# Map the arguments
base_octet = sys.argv[1]
next_hop = sys.argv[2]
local_as = sys.argv[3]

# Generate 50 unique dynamic routes
DYNAMIC_ROUTES = [f"10.{base_octet}.{i}.0/24" for i in range(1, 51)]

# Dictionary to track the state of each route
route_states = {route: False for route in DYNAMIC_ROUTES}

def send_to_exabgp(command):
    sys.stdout.write(command + "\n")
    sys.stdout.flush()

time.sleep(10)

try:
    while True:
        target = random.choice(DYNAMIC_ROUTES)

        if route_states[target]:
            send_to_exabgp(f"withdraw route {target} next-hop {next_hop}")
            route_states[target] = False
        else:
            send_to_exabgp(f"announce route {target} next-hop {next_hop} as-
path [ {local_as} ]")
            route_states[target] = True

        time.sleep(random.uniform(0.1, 2.0))

except KeyboardInterrupt:
    pass
```

APPENDIX G – TELEGRAF CONFIGURATION

Telegraf configuration file for SNMP and gNMI entries:

```
# Telegraf collection
[agent]
  interval = "10s"
  round_interval = true
  metric_batch_size = 1000
  metric_buffer_limit = 10000

# Prometheus integration
[[outputs.prometheus_client]]
  listen = ":9273"
  expiration_interval = "90s"
  export_timestamp = true
  metric_version = 2

[[inputs.internal]]
  collect_memstats = false

##################################### SNMP Input
[[inputs.snmp]]
  agents = [
    "pe1:161",  # PE1
    "pe2:161",  # PE2
    "p1:161",  # P1
    "p2:161",  # P2
    "p3:161",  # P3
    "rr:161"  # RR
  ]
  version = 2
  community = "public"
  interval = "30s"
  timeout = "5s"
  retries = 1

  # Define OIDs

  [[inputs.snmp.table]]
    name = "interface"

    [[inputs.snmp.table.field]] # interface name as tag
      name = "ifDescr"
      oid = ".1.3.6.1.2.1.2.2.1.2"
      is_tag = true

    [[inputs.snmp.table.field]] # Up/Down state
      name = "ifOperStatus"
```

```
    oid = ".1.3.6.1.2.1.2.2.1.8"

  [[inputs.snmp.table.field]]
    name = "ifHCInOctets"
    oid = "1.3.6.1.2.1.31.1.1.1.6"

  [[inputs.snmp.table.field]]
    name = "ifHCOutOctets"
    oid = "1.3.6.1.2.1.31.1.1.1.10"

[[inputs.snmp.field]]
name = "hrProcessorLoad"
oid = "1.3.6.1.2.1.25.3.3.1.2.1"  # CPU

[[inputs.snmp.table]]
  name = "snmp_bgp"

  [[inputs.snmp.table.field]]
    name = "bgpPeerRemoteAddr"
    oid = ".1.3.6.1.2.1.15.3.1.7" # IP address of the peer
    is_tag = true

  [[inputs.snmp.table.field]]
    name = "bgpPeerState"
    oid = ".1.3.6.1.2.1.15.3.1.2" # 1=Idle, 6=Established

  [[inputs.snmp.table.field]]
    name = "bgpPeerFsmEstablishedTime"  # resets on flap
    oid = ".1.3.6.1.2.1.15.3.1.16"

  [[inputs.snmp.table.field]]
    name = "bgpPeerInUpdates"
    oid = ".1.3.6.1.2.1.15.3.1.10"

  [[inputs.snmp.table.field]]
    name = "bgpPeerOutUpdates"
    oid = ".1.3.6.1.2.1.15.3.1.11"

[[inputs.snmp.table]]
  name = "snmp_arista_bgp_prefixes"
  index_as_tag = true

  [[inputs.snmp.table.field]]
    name = "aristaBgp4V2PrefixInPrefixes"
    oid = ".1.3.6.1.4.1.30065.4.1.1.8.1.3"

  [[inputs.snmp.table.field]]
    name = "aristaBgp4V2PrefixInPrefixesAccepted"
    oid = ".1.3.6.1.4.1.30065.4.1.1.8.1.4"
```

```
    [[inputs.snmp.table.field]]
      name = "aristaBgp4V2PrefixOutPrefixes"
      oid = ".1.3.6.1.4.1.30065.4.1.1.8.1.5"

  [[inputs.snmp.table]]
    name = "snmp_juniper_bgp_prefixes"
    index_as_tag = true

    [[inputs.snmp.table.field]]
      name = "AFI" #AFI (1 = IPv4, 2 = IPv6, etc.)
      oid = ".1.3.6.1.4.1.2636.5.1.1.2.6.2.1.1"
      is_tag = true

    [[inputs.snmp.table.field]]
      name = "SAFI" #SAFI (1 = Unicast, 128 = VPN, etc.)
      oid = ".1.3.6.1.4.1.2636.5.1.1.2.6.2.1.2"
      is_tag = true

    [[inputs.snmp.table.field]]
      oid = ".1.3.6.1.4.1.2636.5.1.1.2.1.1.1.14"
      name = "PeerIndex"
      is_tag = true

    [[inputs.snmp.table.field]]
      name = "jnxBgpM2PrefixInPrefixes"
      oid = ".1.3.6.1.4.1.2636.5.1.1.2.6.2.1.7"

    [[inputs.snmp.table.field]]
      name = "jnxBgpM2PrefixInPrefixesAccepted"
      oid = ".1.3.6.1.4.1.2636.5.1.1.2.6.2.1.8"

    [[inputs.snmp.table.field]]
      name = "jnxBgpM2PrefixOutPrefixes"
      oid = ".1.3.6.1.4.1.2636.5.1.1.2.6.2.1.10"

    [[inputs.snmp.table.field]]
      name = "jnxBgpM2PrefixInPrefixesActive"
      oid = ".1.3.6.1.4.1.2636.5.1.1.2.6.2.1.11"

  [[inputs.snmp.table]]
    name = "snmp_juniper_bgp_peers"
    index_as_tag = true

    [[inputs.snmp.table.field]]
      oid = "1.3.6.1.4.1.2636.5.1.1.2.1.1.1.14"
      name = "jnxBgpM2PeerIndex"

  [[inputs.snmp.table]]
```

```
    name = "snmp_juniper_ldp"
    index_as_tag = true # captures the peer ID string

    [[inputs.snmp.table.field]]
      name = "jnxMplsLdpSesState"
      oid = ".1.3.6.1.4.1.2636.3.36.1.3.3.1.2"

    [[inputs.snmp.table.field]]
      name = "jnxMplsLdpSesStateLastChange"
      oid = ".1.3.6.1.4.1.2636.3.36.1.3.3.1.1"

  [[inputs.snmp.table]]
    name = "snmp_mpls" #label table
    index_as_tag = true

    [[inputs.snmp.table.field]]
      name = "mplsLdpSessionState"
      oid = ".1.3.6.1.2.1.10.166.4.1.3.3.1.2"

    [[inputs.snmp.table.field]]
      name = "mplsLdpSessionStateLastChange"
      oid = ".1.3.6.1.2.1.10.166.4.1.3.3.1.1"

    [[inputs.snmp.table.field]]
      name = "mplsL3VpnVrfOperStatus"
      oid = ".1.3.6.1.2.1.10.166.11.1.2.2.1.6"

[[inputs.snmp_trap]] # SNMP trap
  service_address = "udp://:1620"
  [inputs.snmp_trap.tags]
    mechanism = "snmp_trap"

################################### gNMI Input over OpenConfig paths

[[inputs.gnmi]]
  addresses = [
    "pe1:32767",  # PE1
    "pe2:32767",  # PE2
    "p1:6030",    # P1
    "p2:6030",    # P2
    "p3:6030",    # P3
    "rr:6030"    # RR
  ]

  username = "admin"
  password = "admin"
  insecure_skip_verify = true

  [[inputs.gnmi.subscription]]
```

```
    name = "interfaces_telemetry"
    origin = "openconfig"
    path = "/interfaces/interface/state/counters"
    subscription_mode = "sample"
    sample_interval = "10s"

  [[inputs.gnmi.subscription]]
    name = "gnmi_cpu"
    origin = "openconfig"
    path = "/components/component/cpu/utilization/state/instant"
    subscription_mode = "sample"
    sample_interval = "10s"

  [[inputs.gnmi.subscription]]
    name = "gnmi_ldp_state"
    origin = "openconfig"
    path = "/network-instances/network-instance/mpls/signaling-
protocols/ldp/neighbors/neighbor/state/session-state"
    subscription_mode = "on_change"



[[inputs.gnmi]] #Arista-only
  addresses = [
    "p1:6030",   # P1
    "p2:6030",   # P2
    "p3:6030",   # P3
    "rr:6030"   # RR
  ]

  username = "admin"
  password = "admin"
  insecure_skip_verify = true


  [[inputs.gnmi.subscription]]
    name = "gnmi_ldp_state_arista"
    origin = "eos_native"
    path =
"/Sysdb/mpls/ldp/ldpStatusColl/status/default/ldpSessionStatusColl/sessionStat
us"
    subscription_mode = "on_change"


[[inputs.gnmi]]
  addresses = [
    "pe1:32767",  # PE1
    "pe2:32767",  # PE2
    "rr:6030"   # RR
```

```
  ]

  username = "admin"
  password = "admin"
  insecure_skip_verify = true

  tagexclude = ["path"]

  [[inputs.gnmi.subscription]] #BGP
    name = "gnmi_bgp_prefixes"        # gNMI-exclusive capability
    origin = "openconfig"
    path = "/network-instances/network-
instance/protocols/protocol/bgp/neighbors/neighbor/afi-safis/afi-
safi/state/prefixes"
    subscription_mode = "sample"
    sample_interval = "10s"

  [[inputs.gnmi.subscription]]
    name = "gnmi_bgp_messages"
    origin = "openconfig"
    path = "/network-instances/network-
instance/protocols/protocol/bgp/neighbors/neighbor/state/messages"
    subscription_mode = "sample"
    sample_interval = "10s"

  [[inputs.gnmi.subscription]]
    name = "gnmi_bgp_state"
    origin = "openconfig"
    path = "/network-instances/network-
instance/protocols/protocol/bgp/neighbors/neighbor/state/session-state"
    subscription_mode = "on_change"


# Processors
[[processors.enum]]
  namepass = ["gnmi_bgp_state"]
  [[processors.enum.mapping]]
    field = "session_state"
    dest  = "session_state_int"
    [processors.enum.mapping.value_mappings]
      "IDLE" = 1
      "CONNECT" = 2
      "ACTIVE" = 3
      "OPENSENT" = 4
      "OPENCONFIRM" = 5
      "ESTABLISHED" = 6

[[processors.enum]]
  namepass = ["gnmi_ldp_state"]
```

```
  [[processors.enum.mapping]]
    field = "session_state"
    dest  = "ldp_session_state_int"
    [processors.enum.mapping.value_mappings]
      "NON_EXISTENT" = 1
      "INITIALIZED" = 2
      "OPENREC" = 3
      "OPENSENT" = 4
      "OPERATIONAL" = 5

[[processors.starlark]]
  namepass = ["gnmi_ldp_state_arista"]
  source = '''
def apply(metric):
    keys_to_remove = []

    # Iterate through the incoming fields
    for k, v in metric.fields.items():

        # 1. Target session states but explicitly IGNORE historical LogEntries
        if type(v) == "string" and "sessionstate" in k.lower() and "logentry"
not in k.lower():
            keys_to_remove.append(k)

            # 2. Extract the dynamic IP/Peer prefix
            if "/" in k:
                # Handles Arista format:
"0_0_10_0_0_10/sessionState/sessionState"
                prefix = k.split("/")[0]
            else:
                # Handles flattened format:
"0_10_0_0_10_ldpSessionState_sessionState"
                prefix =
k.split("_ldpSessionState")[0].split("_sessionState")[0]

            # Clean up leading underscores if present
            if prefix.startswith("_"):
                prefix = prefix[1:]

            # 3. Convert "0_10_0_0_10" to standard IP "10.0.0.10"
            parts = prefix.split("_")
            if len(parts) >= 4:
                # Grab the last 4 elements (the octets) and join them with
dots
                peer_ip = ".".join(parts[-4:])
            else:
                # Fallback just in case the format is unexpected
                peer_ip = prefix
```

```
            # 4. Set the clean IP tag and field
            metric.tags["ldp_peer"] = peer_ip
            metric.fields["session_state_raw"] = v

    for k in keys_to_remove:
        metric.fields.pop(k)

    return metric
'''

[[processors.enum]]
  namepass = ["gnmi_ldp_state_arista"]

  [[processors.enum.mapping]]
    field = "session_state_raw"
    dest  = "ldp_session_state_int"

    [processors.enum.mapping.value_mappings]
      "stateNonExist" = 1
      "stateInitialized" = 2
      "stateOpenRecv" = 3
      "stateOpenSent" = 4
      "stateOperational" = 5

[[processors.regex]]
  namepass = ["snmp_arista_bgp_prefixes"]

  [[processors.regex.tags]]
    key = "index"
    pattern = "^\\d+\\.1\\.4\\.(\\d+\\.\\d+\\.\\d+\\.\\d+)\\.\\d+\\.\\d+$"
    replacement = "${1}"
    result_key = "PeerNeighbor"

  [[processors.regex.tags]]
      key = "index"
      pattern =
"^\\d+\\.1\\.4\\.(\\d+\\.\\d+\\.\\d+\\.\\d+)\\.(\\d+)\\.(\\d+)$"
      replacement = "${2}"
      result_key = "AFI"

  [[processors.regex.tags]]
      key = "index"
      pattern =
"^\\d+\\.1\\.4\\.(\\d+\\.\\d+\\.\\d+\\.\\d+)\\.(\\d+)\\.(\\d+)$"
      replacement = "${3}"
      result_key = "SAFI"

[[processors.regex]]
  namepass = ["snmp_juniper_bgp_peers"]
```

```
  [[processors.regex.tags]]
    key = "index"
    pattern = ".*\\.(\\d+\\.\\d+\\.\\d+\\.\\d+)$"
    replacement = "${1}"
    result_key = "PeerNeighbor"

[[processors.starlark]]
  namepass = ["snmp_juniper_bgp_peers*", "snmp_juniper_bgp_prefixes*"]

  source = '''

def apply(metric):
    if "peer_map" not in state:
        state["peer_map"] = {}

    peer_map = state["peer_map"]
    if metric.name.startswith("snmp_juniper_bgp_peers"):
        p_neighbor = metric.tags.get("PeerNeighbor")
        agent = metric.tags.get("agent_host")

        for field_name, field_value in metric.fields.items():
            p_index = str(int(field_value))

            if agent and p_index and p_neighbor:
                # Create a unique key per router (e.g., "pe1:1")
                cache_key = agent + ":" + p_index
                peer_map[cache_key] = p_neighbor

        return metric

    elif metric.name.startswith("snmp_juniper_bgp_prefixes"):
        agent = metric.tags.get("agent_host")
        raw_index = metric.tags.get("index") # e.g., "0.1.1"

        if agent and raw_index:
            # Extract the first element from the index tag (e.g., "0" or "1")
            parts = raw_index.split(".")
            p_index = parts[0]

            # Reconstruct the unique router key (e.g., "pe1:1")
            cache_key = agent + ":" + p_index

            if cache_key in peer_map:
                metric.tags["PeerNeighbor"] = peer_map[cache_key]

        return metric
'''
```