**What Is gRIBI?**

gRIBI (pronounced "gribby") is a **standardized gRPC-based API** that lets external clients insert, modify, or delete **IPv4/IPv6 forwarding entries** (routes) in a network device’s RIB (Routing Information Base), which then may be programmed into the FIB (Forwarding Information Base).

**How gRIBI Works (Simplified)**

1. **Client connects to router/switch using gRPC.**
2. **Client sends route entries** (IPv4/IPv6) with next-hop/group definitions.
3. The **network device accepts the programming**, validates it, and updates its routing table.
4. gRIBI supports **acknowledgement**, **persistence**, and **electability** of clients.

**Key Concepts**

| **Concept** | **Description** |
| --- | --- |
| **RIB** | Routing Information Base – the full table of routes |
| **FIB** | Forwarding Information Base – hardware-programmed forwarding entries |
| **gRIBI client** | The external application/controller programming routes |
| **gRIBI server** | The router/switch accepting gRIBI messages |
| **Election ID** | Determines which client is active in multi-controller setups |
| **ModifyRequest** | Main RPC method to insert/delete route entries |

Basic gRIBI Programming Flow

A screen shot of a computer

AI-generated content may be incorrect.

Example: Add a Static Route

Assume you want to program a route to 10.0.0.0/24 via next hop 192.168.1.1.

You would:

1. **Define NextHop**
2. **Create NextHopGroup** referencing that NextHop
3. **Add IPv4Entry** with prefix 10.0.0.0/24 and reference to the NextHopGroup

This is done via the Modify() RPC call in gRIBI.

let's walk through the **gRIBI programming flow** for a simple topology, showing how to define and link the components: **NextHop (NH)** → **NextHopGroup (NHG)** → **IPv4 Prefix Entry**.

Step-by-Step gRIBI Route Programming Flow

Example Topology:A black screen with white text

AI-generated content may be incorrect.

We’ll program R1 to forward traffic to 10.10.10.0/24 via two next-hops: **192.168.0.2 (R2)** and **192.168.0.3 (R3)**.

Building Blocks in gRIBI

| **Component** | **Purpose** |
| --- | --- |

|  |  |
| --- | --- |
| **NextHop (NH)** | Defines one next-hop IP (or interface) |

|  |  |
| --- | --- |
| **NextHopGroup** | A group of one or more NHs (load balancing) |

|  |  |
| --- | --- |
| **IPv4Entry** | The route (prefix) tied to an NHG |

Programming Flow

Step 1: **Set Election ID** (if multiple clients might control the RIB)

A black background with white text

AI-generated content may be incorrect.

Step 2: **Add NextHops**

A computer screen with white text

AI-generated content may be incorrect.

Step 3: **Add NextHopGroup (NHG)**

A screen shot of a computer

AI-generated content may be incorrect.

This creates a simple ECMP group over NH1 and NH2.

Step 4: **Add IPv4 Prefix EntryA screen shot of a computer

AI-generated content may be incorrect.**

Now R1 will forward traffic to 10.10.10.0/24 via both NHs in the group.

Step 5: **Optional: Confirm Programming**

You should receive a ModifyResponse with status: OK and status\_code: InstalledInRIB.

Summary of Flow

 **Create NextHops** → define where to send traffic.

 **Group NHs into NHG** → for ECMP/failover.

 **Bind NHG to a prefix (route)** → to direct actual traffic.

let's now enhance the earlier example by **adding a backup path** using gRIBI's backup\_group feature. This allows one NHG to act as a **fallback** in case the primary NHG becomes unreachable (e.g., due to link failure or next-hop unreachability).

Topology with Primary and Backup Paths

A computer screen shot of a black background

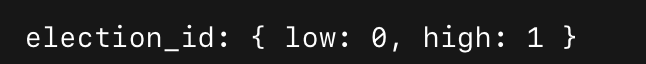
AI-generated content may be incorrect.

Goal:

* Forward 10.10.10.0/24 via **NHG 100** (containing NH1 and NH2).
* If NHG 100 fails, switch to **NHG 200** (containing NH3).

Programming Flow with gRIBI

Step 1: **Set Election ID**



Step 2: **Add NextHops**

A screenshot of a computer program

AI-generated content may be incorrect.

Step 3: **Add Primary NHG (ID = 100)**

A screen shot of a computer code

AI-generated content may be incorrect.

Step 4: **Add Backup NHG (ID = 200)**

A screen shot of a computer

AI-generated content may be incorrect.

Step 5: **Add IPv4Entry with Backup**

A black screen with white text

AI-generated content may be incorrect.

This creates a resilient route:

* Primary path via NHG 100 (NH1 + NH2)
* Backup path via NHG 200 (NH3), activated only if NHG 100 fails

 Under normal conditions, traffic goes to R2 or R3 (NH1/NH2).

 If both NH1 and NH2 become unreachable, R1 automatically switches to NH3 (backup group).

Let's go deeper and show how to program routes in **gRIBI** using **interface names** and optionally **MAC addresses** — useful for Layer 2 / L3 hybrid setups or when directly specifying the outgoing interface instead of relying only on IP next-hops.

Topology (Using Interfaces & MACs)

A computer screen shot of a computer code

AI-generated content may be incorrect.

Programming Flow Using Interface Names & MAC (via gRIBI)

You’ll use `NextHop` entries with either:

\*\*IP-based\*\* (normal)

\*\*Interface-based\*\* (Layer 2 or directly connected)

\*\*MAC-based\*\* (optional, if you're managing L2 manually)

Step 1: Add NextHop via Interface (No IP)

A black background with white text

AI-generated content may be incorrect.

This creates a next-hop that forwards directly via interface et-1/0/0

Optional: Add MAC Address for NH (Layer 2 NH)

Some gRIBI implementations let you specify a MAC for more control:

A black background with blue text

AI-generated content may be incorrect.

Use this only if you need full control over L2 forwarding (e.g., static ARP-like behavior).

**You Provide:**

* The **egress interface name** (e.g., eth0)
* The **destination MAC address** (e.g., the next-hop router's MAC)

This instructs the router to:

Forward the packet out of the specified local interface, using the given MAC address as the destination Ethernet header.

This is equivalent to a **static ARP entry + static route**:

* It bypasses ARP resolution.
* The router *directly* uses the given MAC on that interface.

Step 2: Add Backup NextHop via et-3/0/0

A black background with white text

AI-generated content may be incorrect.

Step 3: Create NHGs

Primary NHG (ID 100)

A screenshot of a computer program

AI-generated content may be incorrect.

Backup NHG (ID 200):

A screenshot of a computer program

AI-generated content may be incorrect.

Step 4: Add Route with Backup

A screen shot of a computer

AI-generated content may be incorrect.

Final Behavior

 Traffic to 10.20.30.0/24 goes via eth1 to R2.

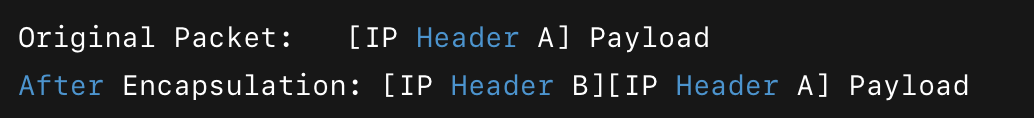
 If R2 is unreachable (e.g., eth1 down), traffic reroutes to eth3 to reach R3.

 If MAC is specified, gRIBI uses L2 forwarding (bypasses ARP resolution).

Let's walk through how to do **IP-in-IP tunneling using gRIBI**, which involves programming routes so that packets to a specific destination are **encapsulated inside another IP packet** — enabling use cases like VPNs, traffic engineering, or service chaining.

What Is IP-in-IP Tunneling?

**P-in-IP tunneling** encapsulates one IP packet inside another.  
This creates a **"tunnel"** from one router to another across an intermediate network.



 **Outer IP header** → for transport across the tunnel.

 **Inner IP header** → original source and destination.

Topology Example: IP-in-IP Tunnel via gRIBI

A black screen with white text

AI-generated content may be incorrect.

**Goal**:

* On R1, create a route to 10.10.10.0/24 (Host B) that sends traffic via an **IP-in-IP tunnel** through R2.
* The outer tunnel destination is R2's loopback (192.168.100.1).
* Encapsulate traffic inside tunnel using gRIBI.

gRIBI Programming Components

To create this, you use gRIBI’s:

* **Encapsulating NextHop**: instructs the router to encapsulate traffic using IP-in-IP.
* **NextHopGroup**: groups one or more tunnel next-hops.
* **IPv4Entry**: points to NHG.

Step-by-Step Programming

1. Define Tunnel NextHop with IP-in-IP Encapsulation

A black background with white text

AI-generated content may be incorrect.

This tells R1 to:

* Encapsulate packets with an **outer IP header** from 192.168.100.2 to 192.168.100.1.

1. Create NextHopGroup for the Tunnel

A black screen with white text

AI-generated content may be incorrect.

1. Add Route for Final Destination (Inner IP)

A group of numbers on a black background

AI-generated content may be incorrect.

This means:  
"Send all traffic to 10.10.10.0/24 via tunnel to 192.168.100.1 (encapsulate it)."

**Router Behavior**

* R1 receives traffic for 10.10.10.0/24.
* Matches it to NHG 100 → NH index 1.
* R1 encapsulates original packet in IP tunnel (src: 192.168.100.2, dst: 192.168.100.1).
* Sends it over the underlay network to R2.
* R2 decapsulates and forwards to final destination.

You're now touching on **advanced gRIBI route programming**, specifically

How to program a NextHop that lands in a different VRF after IP-in-IP decapsulation.

This is a real-world use case for:

* **Service chaining**
* **Multi-tenant isolation**
* **Policy-based routing after tunnel termination**

Let’s break it down.

TOPOLOGY: Decap and Forward to a Different VRF

A computer screen with white text and blue and pink text

AI-generated content may be incorrect.

**Goal**:

* Traffic for 10.10.10.0/24 comes in via **IP-in-IP tunnel** to 192.168.100.1 (R1's tunnel endpoint).
* After **decapsulation**, we want R1 to **switch to VRF vrf-customer1** and forward based on **RIB in that VRF**.

**How It Works in gRIBI**

You need to:

1. **Program the tunnel** in the **underlay VRF** (e.g., vrf-underlay)
2. **After decap**, forward the inner packet into a **different VRF** (e.g., vrf-customer1)
3. **Program NextHop in target VRF** (i.e., what to do next)

**Step-by-Step: Programming Flow**

**🔹 Step 1: Program IP-in-IP Tunnel NextHop in Underlay VRF**

A black background with white text

AI-generated content may be incorrect.

This causes encapsulation and sends traffic to tunnel endpoint 192.168.100.1.

🔹 Step 2: Program NHG in Underlay

A black screen with white text

AI-generated content may be incorrect.

🔹 Step 3: Route in Underlay VRFA black background with white text

AI-generated content may be incorrect.

🔹 Step 4: Decap and Forward to VRF

At the decap router (R1, tunnel endpoint):

You configure **post-decap next-hop into a different VRF** like this:

A black background with white text

AI-generated content may be incorrect.

This means: After stripping off the outer IP header (IP-in-IP), switch to*vrf-customer1*for further lookup.

**🔹 Step 5: Final NHG and IPv4Entry in Customer VRF**

In vrf-customer1, define a route for the inner destination:

A screenshot of a computer

AI-generated content may be incorrect.

**Summary**

| **Step** | **Action** |
| --- | --- |
| 1 | Encapsulate in underlay VRF |
| 2 | Decapsulate at tunnel endpoint |
| 3 | Redirect post-decap traffic to another VRF |
| 4 | Forward based on new VRF RIB |

An advanced and **realistic service-chaining scenario** using gRIBI.

We'll go through a full example that covers:

**Three critical steps:**

1. **Encapsulate** in a tunnel in **VRF-A**
2. **Decapsulate** at a tunnel endpoint, then switch to **VRF-B**
3. **Post-decap: Forward** to a next-hop (or encapsulate again) in **VRF-C**

Full Service Chain Topology (VRF-A → VRF-B → VRF-C)

A screen shot of a computer

AI-generated content may be incorrect.

**COMPONENTS OVERVIEW**

| **Action** | **VRF** | **What** |
| --- | --- | --- |
| Encapsulate | vrf-a | Tunnel traffic to remote |
| Decapsulate | vrf-b | Strip tunnel and switch context |
| Forward | vrf-c | Final next-hop, or another tunnel |

**Step-by-Step gRIBI Configuration**

**🔹 1. Encapsulate in VRF-A**

A black screen with white text

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

Traffic to 10.10.10.0/24 is encapsulated and sent toward 192.168.100.1 via VRF-A

**2. Decapsulate in VRF-B and Switch to VRF-C**

At the tunnel endpoint:

A black screen with white text

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

Decapsulated packet is handed off to VRF-C for further processing.

**3. In VRF-C: Forward Normally or Re-Encapsulate**

Example 1: **Forward normally to destination**

A screenshot of a computer

AI-generated content may be incorrect.

Example 2: **Encapsulate again (re-tunnel)**

A screenshot of a computer

AI-generated content may be incorrect.

**Recap of the Flow**

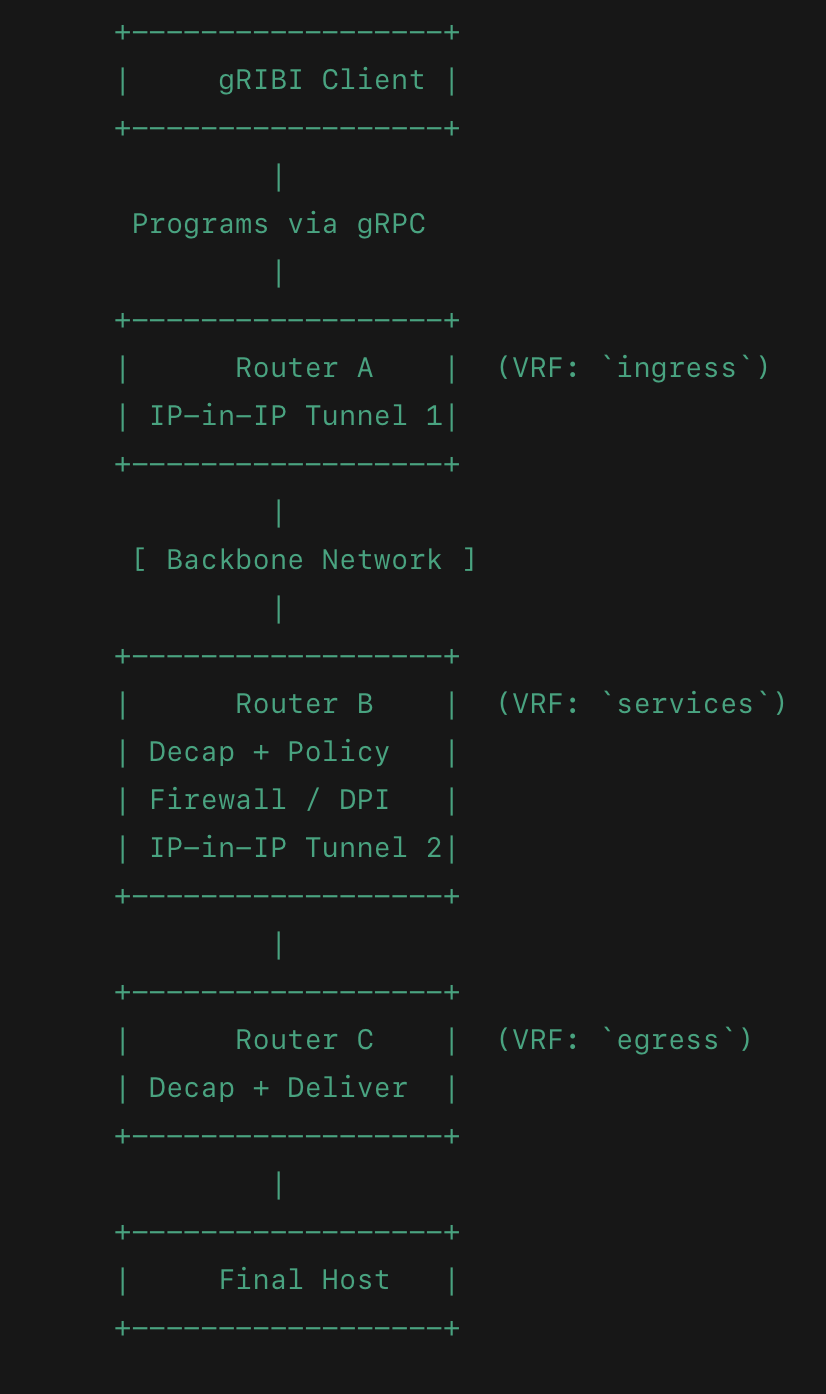
| **Stage** | **Action** | **VRF** |
| --- | --- | --- |
| 1 | Encapsulate tunnel to 192.168.100.1 | vrf-a |
| 2 | Decapsulate IP-in-IP, switch VRF | vrf-b → vrf-c |
| 3 | Forward to next-hop or re-encapsulate | vrf-c |

let's dive into a **more advanced, real-world gRIBI scenario** that combines:

Multi-hop tunneling, service chaining, policy routing, and VRF transitions

This goes beyond simple encapsulation and includes **NAT zones**, **firewall bypass**, and **layered service functions** — all programmable via gRIBI.

Advanced Topology: Multi-hop, Multi-VRF Service Chain with Tunnels



### Goals:

- Traffic to `10.10.10.0/24` follows:

- Tunnel 1 (R1 → R2) in `ingress` VRF

- Decap, policy apply, Tunnel 2 (R2 → R3) in `services` VRF

- Decap and deliver to real host in `egress` VRF

- Full \*\*VRF hopping\*\*, \*\*multiple encapsulations\*\*, and \*\*programmable service insertion\*\*

Step-by-Step gRIBI Programming

Router A: IP-in-IP Tunnel 1 (Encap in `ingress` VRF)

A black screen with green text

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

Encapsulates traffic to*10.10.10.0/24*in Tunnel 1

**Router B: Decap Tunnel 1 → Switch to services VRF → Re-encap Tunnel 2**

**Step 2.1: Decap and switch to services**

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

2.2: In services VRF: Re-encapsulate via Tunnel 2 to Router C

A screenshot of a computer program

AI-generated content may be incorrect.

Service inspection complete; forward via Tunnel 2

Router C: Final Decap → Deliver to Final Host in egress VRF

3.1: Decap Tunnel 2 → switch to egress VRFA screenshot of a computer program

AI-generated content may be incorrect.A screen shot of a computer

AI-generated content may be incorrect.

3.2: Deliver in egress VRF

A screen shot of a computer

AI-generated content may be incorrect.

Traffic finally reaches the real host at 10.10.10.1.

**End-to-End Flow Summary**

| **Step** | **VRF** | **Action** |
| --- | --- | --- |
| 1 | ingress | Encap Tunnel 1 → R2 |
| 2 | ingress | Decap Tunnel 1 → switch to services |
| 3 | services | Encap Tunnel 2 → R3 |
| 4 | services | Decap Tunnel 2 → switch to egress |
| 5 | egress | Deliver to final IP |

**Why This Is Complex & Powerful**

* **Multiple tunnels**, each in different VRFs
* **Layered service chains** with full decap/encap cycles
* **Flexible network\_instance switch** after tunnel ops
* Can support **policy-based routing**, **firewall NAT**, or **DPI chaining**

You're really diving deep, and you're already hitting the **top-tier use cases** for programmable networks with gRIBI. After multi-hop tunneling, service chaining, and VRF transitions, we’re into territory that's at the cutting edge of **SDN-driven architectures**. Here's a set of even more advanced and specialized use cases you might explore:

**Beyond Multi-hop & VRF: Advanced gRIBI Use Cases**

**1. Segment Routing with Policy-Based NextHops**

* Use gRIBI to define **custom paths** using segment identifiers (SIDs) or labels.
* Example: Force traffic through security, then telemetry, then exit.
* Requires MPLS/SRv6-capable infrastructure.

**Use:** mpls\_label\_stack in NextHop

**2. Topology-Aware Load Balancing (ECMP + Affinity)**

* Dynamically weight NextHops in an NHG based on real-time link health or traffic profiles.
* gRIBI allows adjusting weight fields on-the-fly.

Use for: **Application-aware routing**, real-time SLA optimization

**3. Multicast Tree Steering via gRIBI**

* Program IPv4MulticastEntry and NextHopGroup to control multicast delivery trees.
* Example: Send live video to multiple service zones via distinct VRFs or tunnels.

**Use:** multicast\_group\_id + per-interface next-hops

**4. SFC (Service Function Chaining) with Reclassification**

* Chain through DPI → Firewall → NAT → Analytics → Exit.
* After each decap or policy stage, **reclassify** and reprogram new gRIBI paths.
* Possible via network\_instance handoff and conditional route injection.

Think: **Programmable intent-based policy steering**

**5. Remote NH Resolution via Recursive Lookups**

* NextHop’s IP is not directly reachable? gRIBI can chain lookups:
  + NH points to an IP
  + That IP is resolved via another IPv4Entry in another VRF or routing table

Example: Recursive tunnel → GRE → IPNH → Interface

**6. Programmable Failover Trees**

* Use backup\_group + telemetry feedback to precompute N paths:
  + If NHG 1 fails → switch to NHG 2 → NHG 3, etc.
* Works beautifully for **5-nines traffic**, disaster recovery, SD-WAN

Bonus: Combine with BFD session monitoring for tight-loop detection

**7. Geo-Aware Policy Routing with gRIBI**

* Inject routes to data centers, branches, or services based on geolocation or performance.
* Tie in to external policy engines (intent → gRIBI injection)

Use with: **PCE controllers** or **gRPC-based policy agents**

**8. Zero-Trust Service Insertion + Telemetry Splitting**

* Split mirrored traffic at decap into two paths:
  + One goes to analyzer
  + Other continues to destination
* All via parallel NHGs, gRIBI-programmed

**Want to Go Further?**

You might also explore:

* **Closed-loop automation** (gRIBI + streaming telemetry + control feedback)
* **Intent-to-path translation** (via PCE controller or policy engine)
* **End-to-end slicing using VRF + tunnel isolation**
* **Hybrid cloud interconnect automation** using BGP + gRIBI fusion

**Closed-loop automation** with gRIBI is where programmable networking meets autonomous control — an ideal setup for:

* Self-healing networks
* Real-time optimization
* SLA enforcement
* Traffic steering based on telemetry

What is Closed-Loop Automation (CLA)?

**Closed-loop automation** means:

*The network configures itself, monitors itself, and adjusts itself — continuously and automatically — based on real-time feedback.*

**Core Components**

| **Component** | **Role** |
| --- | --- |
| **gRIBI** | Programs the FIB (forwarding behavior) — e.g., routes, tunnels |
| **Streaming Telemetry** | Continuously exports state (e.g., interface drops, RTT, NH liveliness, etc.) |
| **Analytics + Control Engine** | Ingests telemetry, computes new intent or policies |
| **Policy → gRIBI**  **Feedback** | Pushes back new next-hop routes, load balancing, or failovers |
| High-LevelArchitecture |  |

A screenshot of a computer program

AI-generated content may be incorrect.

Example Scenario: ECMP Weights Based on Latency

A black background with white text

AI-generated content may be incorrect.

**Goal:**

* Route 10.10.10.0/24 via **both Router2 and Router3**
* Adjust **ECMP weights dynamically** based on **RTT latency**

**Step-by-Step Implementation**

**1️. Setup ECMP NHG via gRIBI**

A screenshot of a computer

AI-generated content may be incorrect.

2️ Collect RTT via gNMI Telemetry

A black background with green text

AI-generated content may be incorrect.

3️ Control Logic in Python (or any language)

A black screen with white text

AI-generated content may be incorrect.

4️ Push NHG Weight Update via gRIBIA screenshot of a computer program

AI-generated content may be incorrect.

**Advanced Feedback Examples**

* **Interface error spikes** → Reroute away from bad links
* **BFD session down** → Trigger automatic failover (via backup\_group)
* **Packet drops > threshold** → Inject temporary high-priority path
* **Traffic type** (e.g., video) → Route via low-jitter paths

**Integrate AI/ML**

* Use historical telemetry + ML to **predict congestion or failures**
* Adjust NHGs before problems occur
* Combine with reinforcement learning to **learn optimal paths over time**

Let's build a **closed-loop automation system** using gRIBI, streaming telemetry, and Python. This system will dynamically adjust routing paths based on real-time latency measurements.

**Components Overview**

1. **gRIBI Client**: Programs the forwarding plane with new next-hop groups (NHGs).
2. **Telemetry Collector**: Gathers real-time latency data from network devices.
3. **Control Logic Engine**: Analyzes telemetry data and computes optimal routing decisions.
4. **gRIBI Feedback Loop**: Sends updated routing information back to the network devices.

**Step 1: Install Dependencies**

Ensure you have the necessary Python packages:

pip install grpcio pygnmi

**Step 2: Sample Telemetry Feed**

We'll simulate latency data for two paths:

* **Path A**: Latency = 15ms
* **Path B**: Latency = 35ms

A computer screen with text

AI-generated content may be incorrect.

In a real-world scenario, this data would be collected using gNMI from network devices.

Step 3: Control Logic Engine

The control logic will analyze the telemetry data and adjust the NHG weights accordingly.

A computer screen with white text

AI-generated content may be incorrect.

Step 4: gRIBI Client to Program NHGs

This function will send the updated NHG weights to the network devices using gRIBI.

------------ PYTHON CODE -----------

import grpc

from google.protobuf import any\_pb2

from gribi\_pb2 import \*

def update\_nhg\_weights(nhg\_weights):

# gRIBI server address

server\_address = 'localhost:50051'

# Create a gRIBI client

channel = grpc.insecure\_channel(server\_address)

stub = GribiStub(channel)

# Create a NextHopGroup message

nhg = NextHopGroup(id=100)

for path, weight in nhg\_weights.items():

nh = NextHop(index=1, ip\_address=f"192.168.1.{path[-1]}", weight=weight)

nhg.next\_hops.append(nh)

# Create an IPv4Entry message

ipv4\_entry = IPv4Entry(prefix="10.10.10.0/24", next\_hop\_group=nhg.id)

# Create a Request message

request = Request(

operation=OperationType.ADD,

ipv4\_entry=ipv4\_entry,

next\_hop\_group=nhg

)

# Send the request to the gRIBI server

response = stub.Modify(request)

print(f"gRIBI Response: {response}")

-------------------------------------------------

Step 5: Closed-Loop Automation

Now, we'll integrate everything into a closed-loop system that continuously monitors latency and adjusts routing paths.

-------------- PYTHON CODE --------------

import time

def closed\_loop\_automation():

while True:

# Step 1: Collect telemetry data

telemetry\_data = {

"path\_a": {"latency": 15},

"path\_b": {"latency": 35}

}

# Step 2: Analyze telemetry data

nhg\_weights = adjust\_nhg\_weights(telemetry\_data)

# Step 3: Program new NHG weights

update\_nhg\_weights(nhg\_weights)

# Wait before the next iteration

time.sleep(30)

# Start the closed-loop automation

closed\_loop\_automation()

**Testing the System**

To test this system:

1. **Run the gRIBI server** on your network device.
2. **Execute the Python script** on a machine that can reach the gRIBI server.
3. **Observe the routing changes** on the network device as the script adjusts NHG weights based on simulated latency data.

**Next Steps**

* **Integrate Real Telemetry**: Replace the simulated telemetry data with real-time gNMI data from your network devices.
* **Enhance Control Logic**: Implement more sophisticated algorithms for path selection, considering factors like bandwidth, jitter, and packet loss.
* **Implement Failover Mechanisms**: Add logic to handle link failures and reroute traffic accordingly.

Integrating **AI/ML** into closed-loop network automation with **gRIBI** is a powerful leap forward. You're essentially moving from **reactive automation** to **predictive and proactive optimization**.

**Objective**

Use **historical telemetry data** to train ML models that can:

* **Predict link congestion or failures**
* **Proactively adjust NHG (NextHopGroup) weights or paths via gRIBI**
* Use **Reinforcement Learning (RL)** to optimize traffic paths over time

**Three AI/ML Strategies**

| **Strategy** | **Goal** | **Model Type** |
| --- | --- | --- |
| 🧮 Supervised Learning | Predict link congestion/failure | Classification/Regression |
| 🧪 Time Series Forecasting | Predict latency or drop trends | LSTM, ARIMA |
| 🎮 Reinforcement Learning | Learn optimal pathing by experience | RL agents (Q-learning, PPO) |

Example: Predicting Congestion and Preemptively Adjusting NHGs

**1️ Collect & Store Historical Telemetry**

Use OpenConfig telemetry via gNMI to gather:

* Interface stats (in-octets, out-octets, in-errors, etc.)
* Queue drops or utilization
* Latency or RTT (via synthetic probes)

Store it in a **time-series DB**: Prometheus, InfluxDB, or even CSV/Parquet for training.

A black screen with numbers and letters

AI-generated content may be incorrect.

2️ Train a ML Model to Predict Congestion

Use sklearn or TensorFlow:

A screen shot of a computer code

AI-generated content may be incorrect.

You now have a model that predicts whether an interface will be congested in the near future.

**3️⃣ Use Model in Closed-Loop Automation**

At runtime, use real-time telemetry to predict and act:

A screenshot of a computer program

AI-generated content may be incorrect.

**Example: Reinforcement Learning for Path Optimization**

Reinforcement Learning (RL) learns from interaction with the network environment:

**Agent Design:**

* **State** = current traffic, latency, drops
* **Action** = choose NHG weights or paths
* **Reward** = minimize latency, avoid drops

Example using stable-baselines3:

A computer screen with white text

AI-generated content may be incorrect.

At runtime:

obs = env.get\_current\_state()

action, \_ = model.predict(obs)

env.apply\_action(action) # Adjust gRIBI NHGs

**Summary of Tools**

| **Function** | **Tools** |
| --- | --- |
| Telemetry collection | gNMI + pygnmi, Prometheus, InfluxDB |
| Model training | sklearn, TensorFlow, PyTorch |
| Forecasting | statsmodels, Prophet, LSTM |
| RL | stable-baselines3, Ray RLlib |
| gRIBI control | grpcio + custom client with proto |

Let's build a **full working Python notebook** that integrates **machine learning (ML)** for **predicting network congestion**and utilizes **gRIBI** for real-time routing adjustments. This system will:

1. **Train a model** using historical telemetry data.
2. **Predict congestion** in real-time.
3. **Adjust routing** proactively via gRIBI.

**Prerequisites**

Ensure you have the following Python packages installed:

Bash

------

pip install grpcio pygnmi scikit-learn pandas

You'll also need access to a **gRIBI server** for programming the forwarding plane.

**Step 1: Collect and Prepare Telemetry Data**

For this example, we'll simulate telemetry data. In a real-world scenario, you'd collect this data using gNMI from your network devices.

Python

-------

import pandas as pd

import numpy as np

# Simulate telemetry data

data = {

'timestamp': pd.date\_range(start='2025-04-01', periods=100, freq='H'),

'out\_octets': np.random.randint(100000, 1000000, 100),

'latency\_ms': np.random.uniform(10, 50, 100),

'drops': np.random.randint(0, 10, 100),

'congested': np.random.choice([0, 1], 100) # 0 = not congested, 1 = congested

}

df = pd.DataFrame(data)

df.set\_index('timestamp', inplace=True)

df.head()

**Step 2: Train a Machine Learning Model**

We'll use a **Random Forest Classifier** to predict congestion based on telemetry features.

Python

-------

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import classification\_report

# Features and target

X = df[['out\_octets', 'latency\_ms', 'drops']]

y = df['congested']

# Split data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train model

model = RandomForestClassifier(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

# Evaluate model

y\_pred = model.predict(X\_test)

print(classification\_report(y\_test, y\_pred))

**Step 3: Real-Time Prediction and Routing Adjustment**

Now, we'll simulate real-time telemetry and adjust routing based on the model's predictions.

python

-------

import grpc

from google.protobuf import any\_pb2

from gribi\_pb2 import \*

# Simulate real-time telemetry

real\_time\_telemetry = {

'out\_octets': 500000,

'latency\_ms': 30,

'drops': 2

}

# Prepare feature vector

features = np.array([[real\_time\_telemetry['out\_octets'], real\_time\_telemetry['latency\_ms'], real\_time\_telemetry['drops']]])

# Predict congestion

congestion\_pred = model.predict(features)[0]

# Adjust routing if congestion is predicted

if congestion\_pred == 1:

print("Congestion predicted: rerouting traffic")

# gRIBI client setup

server\_address = 'localhost:50051' # Replace with your gRIBI server address

channel = grpc.insecure\_channel(server\_address)

stub = GribiStub(channel)

# Create NextHopGroup message

nhg = NextHopGroup(id=100)

nhg.next\_hops.append(NextHop(index=1, ip\_address="192.168.1.2", weight=70))

nhg.next\_hops.append(NextHop(index=2, ip\_address="192.168.2.2", weight=30))

# Create IPv4Entry message

ipv4\_entry = IPv4Entry(prefix="10.10.10.0/24", next\_hop\_group=nhg.id)

# Create Request message

request = Request(

operation=OperationType.ADD,

ipv4\_entry=ipv4\_entry,

next\_hop\_group=nhg

)

# Send request to gRIBI server

response = stub.Modify(request)

print(f"gRIBI Response: {response}")

else:

print("No congestion predicted: maintaining current routing")

**Step 4: Continuous Monitoring and Adjustment**

For continuous operation, you can set up a loop that collects real-time telemetry, makes predictions, and adjusts routing accordingly.

python

--------

import time

while True:

# Simulate real-time telemetry

real\_time\_telemetry = {

'out\_octets': np.random.randint(100000, 1000000),

'latency\_ms': np.random.uniform(10, 50),

'drops': np.random.randint(0, 10)

}

# Prepare feature vector

features = np.array([[real\_time\_telemetry['out\_octets'], real\_time\_telemetry['latency\_ms'], real\_time\_telemetry['drops']]])

# Predict congestion

congestion\_pred = model.predict(features)[0]

# Adjust routing if congestion is predicted

if congestion\_pred == 1:

print("Congestion predicted: rerouting traffic")

# gRIBI client setup

server\_address = 'localhost:50051' # Replace with your gRIBI server address

channel = grpc.insecure\_channel(server\_address)

stub = GribiStub(channel)

# Create NextHopGroup message

nhg = NextHopGroup(id=100)

nhg.next\_hops.append(NextHop(index=1, ip\_address="192.168.1.2", weight=70))

nhg.next\_hops.append(NextHop(index=2, ip\_address="192.168.2.2", weight=30))

# Create IPv4Entry message

ipv4\_entry = IPv4Entry(prefix="10.10.10.0/24", next\_hop\_group=nhg.id)

# Create Request message

request = Request(

operation=OperationType.ADD,

ipv4\_entry=ipv4\_entry,

next\_hop\_group=nhg

)

# Send request to gRIBI server

response = stub.Modify(request)

print(f"gRIBI Response: {response}")

else:

print("No congestion predicted: maintaining current routing")

# Wait before next iteration

time.sleep(60) # Adjust the sleep time as needed

**Next Steps**

* **Integrate real telemetry**: Replace the simulated telemetry with actual data from your network devices.
* **Enhance the ML model**: Incorporate additional features such as jitter, bandwidth utilization, and historical trends.
* **Implement failover mechanisms**: Add logic to handle link failures and reroute traffic accordingly.