GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

NEPTUNE
Network Prefix Translation Unified Environment

Autor: **Lucas Carrasco Domínguez**

Director: **Atilano Ramiro Fernández-Pacheco Sánchez-Migallón**

Madrid

Declaración de originalidad

Declaro bajo mi responsabilidad que el Proyecto presentado con el título **NEPTUNE: Network Prefix Translation Unified Environment** de la ETS de Ingeniería – ICAI de la Universidad Pontificia Comillas en el curso académico **4º** es de mi autoría y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Uso de Inteligencia Artificial[1]

Declaro bajo mi responsabilidad que (indicar la opción correcta):

ϒ No he utilizado Inteligencia Artificial en la elaboración del presente documento.

X He utilizado Inteligencia Artificial en la elaboración del presente documento y/o del Anexo B siempre en las condiciones permitidas por la Universidad Pontificia Comillas, es decir, aplicando el Nivel 2 de la Escala de Evaluación de Perkins et al. (2024): "*La IA puede utilizarse para actividades previas a la tarea, como la lluvia de ideas, la descripción y la investigación inicial. Este nivel se centra en el uso de la IA para la planificación, las síntesis y la generación de ideas, pero las evaluaciones deben hacer hincapié en la capacidad de desarrollar y refinar estas ideas de forma independiente*". En concreto, las Inteligencia Artificial ha sido empleada para:

- Realizar lluvias de ideas preliminares sobre la estructura general del proyecto y la organización de los capítulos.
- Apoyar en la exploración inicial del estado del arte, sugiriendo posibles líneas de investigación, tecnologías relevantes y fuentes normativas (RFCs, documentación técnica, etc.) y ayudando en la descripción de los mismos en capítulos de documentación de tecnologías existentes.
- Asistir en la revisión y adaptación lingüística y estilística de fragmentos del texto en inglés, mejorando la claridad, la corrección gramatical y la coherencia formal.
- Contribuir a la síntesis de conceptos complejos, ayudando a resumir ideas técnicas para su posterior desarrollo por parte del autor.
- Proponer estructuras alternativas de redacción y organización de contenidos, cuya selección, adaptación y validación final han sido realizadas por el autor.
- Facilitar la generación de ideas preliminares sobre posibles escenarios de evaluación y casos de uso, que posteriormente han sido diseñados, implementados y analizados de forma autónoma.
- Asistir en la conceptualización del uso de código y simulaciones como herramientas de apoyo para la validación de la propuesta, entendiendo que el objetivo del proyecto no es la implementación de un protocolo operativo listo para su despliegue en entornos reales, sino la definición y evaluación de una arquitectura técnica, en un enfoque comparable al proceso de especificación seguido por organismos de estandarización como el IETF.

En todos los casos, el autor ha sido responsable de la validación técnica, desarrollo conceptual y redacción final del contenido presentado en este documento.

Firmado Digitalmente por CARRASCO DOMINGUEZ LUCAS - ***4945** el día 28/05/2026 con un certificado emitido por AC FNMT Usuarios

Firmado (alumno): Lucas Carrasco Domínguez

[1] Esta declaración se refiere al uso de la Inteligencia Artificial generativa para realizar los documentos del Proyecto (Anexo B y Memoria). No aplica a Proyectos donde, por su naturaleza, deban emplear inteligencia artificial como parte de los mismos (aplicación de técnicas de aprendizaje automático, redes neuronales, análisis de datos…)

Fecha: 28/05/2026

Autorización para la entrega del Proyecto

El Director del Proyecto	El co-Director del Proyecto (si aplica)
71216314 B ATILANO RAMIRO FERNÁND Signed by: 71216314B ATILANO RAMIRO FERNÁNDEZ-PACHECO Date: 2026-05-29 20:36:53 CEST	
Fdo: Atilano Ramiro Fernández-Pacheco Sánchez-Migallón	Fdo:
Fecha: 28/05/2026	Fecha:

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

NEPTUNE
Network Prefix Translation Unified Environment

Autor: **Lucas Carrasco Domínguez**

Director: **Atilano Ramiro Fernández-Pacheco Sánchez-Migallón**

Madrid

Agradecimientos

En primer lugar, quería agradecer a mi Director de Proyecto, Atilano Ramiro Fernández-Pacheco Sánchez-Migallón, por su orientación y apoyo a lo largo del desarrollo de este trabajo, así como por ayudarme a entender qué se esperaba de mi en cada etapa del proyecto y cómo enfocar correctamente tanto la parte técnica como la documentación asociada.

También quería agradecer a mi profesor de Arquitectura de Redes, Alejandro García San Luis, por despertar mi interés por el mundo de las redes y transmitir una visión especialmente práctica de esta disciplina, que ha terminado siendo una parte fundamental de mi formación.

Del mismo modo, quería agradecer a mi profesor de Conmutación y Transmisión, Carlos Javier Monedero Martínez, por contribuir a mi formación técnica en el ámbito de las redes de telecomunicaciones y por ayudarme durante el desarrollo de su asignatura a comprender muchos de los conceptos sobre los que posteriormente he construido este proyecto.

Finalmente, quería agradecer a mi familia, amigos y compañeros por el apoyo y la paciencia durante todos estos años de carrera y especialmente durante el desarrollo del presente trabajo.

NEPTUNE: Network Prefix Translation Unified Environment

Autor: **Carrasco Domínguez, Lucas**
Director: **Fernández-Pacheco Sánchez-Migallón, Atilano Ramiro**
Entidad Colaboradora: **ICAI – Universidad Pontificia Comillas**

RESUMEN DEL PROYECTO

NEPTUNE propone un framework operativo para escenarios de multihoming IPv6 edge que hacen uso de direccionamiento Provider-Aggregatable. El sistema integra mecanismos de policy routing, persistencia de flujos y traducción de prefijos mediante NPTv6 para proporcionar balanceo de tráfico y failover sin recurrir a BGP. La propuesta ha sido implementada y validada experimentalmente sobre un entorno de red virtualizado en Linux.

Palabras clave: IPv6, Multihoming, NPTv6, Policy Routing, Failover

1. Introducción

IPv6 [1] fue originalmente diseñado para resolver las limitaciones de IPv4 mediante la introducción de un espacio de direccionamiento mucho mayor, restableciendo el principio de conectividad de extremo a extremo. Sin embargo, casi tres décadas después de su especificación, su tasa de adopción sigue siendo moderada, dado que muchos escenarios siguen presentando una complejidad de implementación significativa.

Concretamente, los entornos de multihoming basados en el direccionamiento Provider-Aggregatable siguen requiriendo la combinación de múltiples mecanismos independientes, como el encaminamiento basado en políticas, la traducción de prefijos y la persistencia de flujos, cuya integración operativa sigue estando fragmentada y depende en gran medida de implementaciones específicas y de configuraciones ad-hoc.

Este proyecto se centra en esos escenarios mediante el desarrollo de NEPTUNE, un marco operativo experimental diseñado para proporcionar capacidades de multihoming IPv6, conmutación en caso de error, balanceo de carga y encaminamiento basado en políticas sin depender de los enfoques tradicionales basados en BGP. El sistema propuesto integra la selección dinámica de rutas, la persistencia de flujos y la traducción de prefijos basada en NPTv6 [2] dentro de una arquitectura unificada basada en Linux.

2. Definición del proyecto

Es habitual que las redes empresariales utilicen múltiples circuitos de Internet de distintos proveedores de servicios con el fin de mejorar la disponibilidad, la resiliencia y la capacidad de ancho de banda disponible.

En las implementaciones de IPv4, la traducción de direcciones de red (NAT) simplifica los escenarios de multihoming al permitir que varios hosts internos compartan un número reducido de direcciones públicas, y facilita mecanismos relativamente sencillos de balanceo de carga y de conmutación por error. En la práctica, las redes pequeñas y medianas han recurrido habitualmente a enfoques basados en NAT, mientras que los entornos de grandes empresas y operadores suelen utilizar BGP y direccionamiento independiente del proveedor para lograr capacidades avanzadas de ingeniería de tráfico.

IPv6, sin embargo, cambia significativamente este modelo operativo. El papel menos extendido de los mecanismos NAT tradicionales, combinado con la complejidad operativa asociada al despliegue de BGP, hace que los escenarios de multihoming agregables por el proveedor sean considerablemente más difíciles de implementar.

Aunque ya existen mecanismos como el enrutamiento por políticas, la traducción de prefijos y la persistencia de flujos dentro de las pilas TCP/IP modernas, actualmente no existe un marco operativo independiente del proveedor ampliamente adoptado capaz de integrar estos mecanismos en una arquitectura unificada y gestionable.

En respuesta a este problema, se propone NEPTUNE (Network Prefix Translation Unified Environment) como un marco operativo experimental diseñado para proporcionar capacidades de multihoming IPv6 mediante la integración de encaminamiento basado en políticas, dirección dinámica de flujos y mecanismos de gestión de prefijos basados en NPTv6.

3. Descripción del sistema

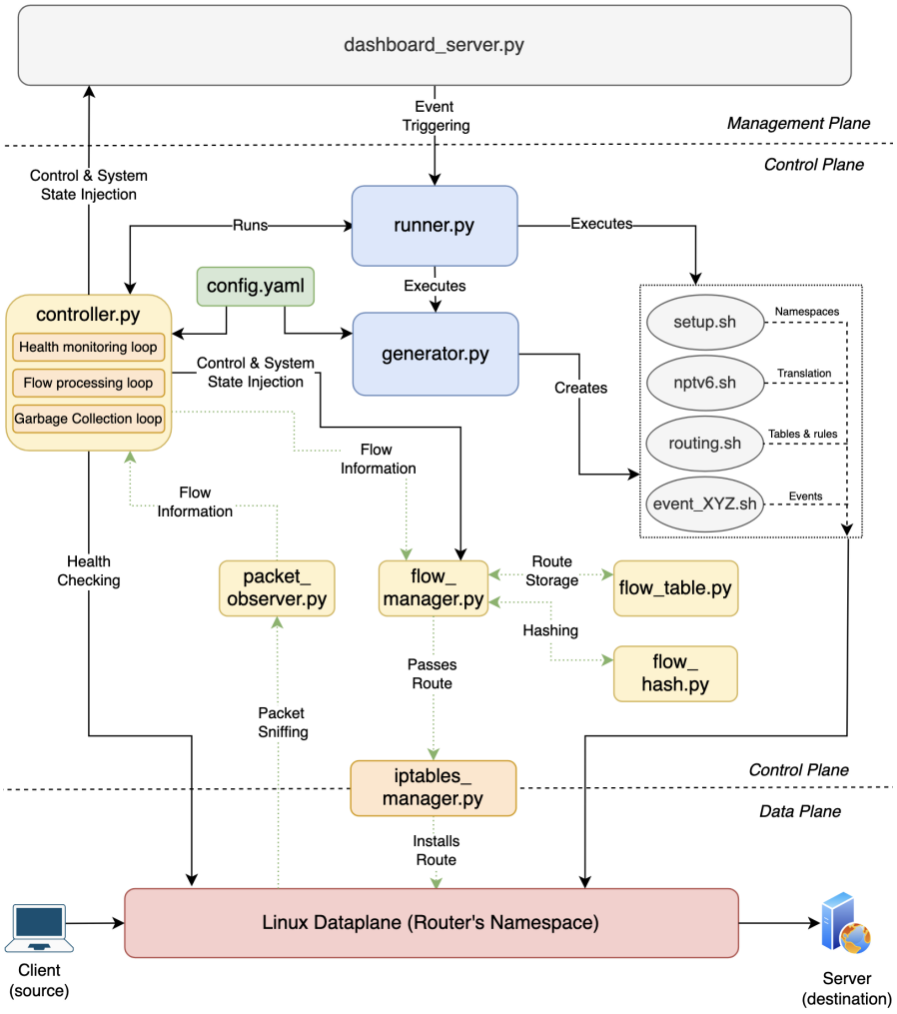


Figura 1: Diagrama de Componentes del Entorno NEPTUNE– Fuente: creación propia

El marco desarrollado sigue una arquitectura en capas que separa las responsabilidades de los planos de administración, control y datos. La Figura 1 ilustra la arquitectura del modelo, mostrando sus principales componentes y las interacciones entre los mismos.

En el nivel más alto, el sistema se rige por un modelo de configuración YAML declarativo que describe la topología, los ISP disponibles, la información de direccionamiento, las políticas de balanceo y los parámetros del entorno experimental. Esta configuración es procesada por el subsistema generador, que crea automáticamente los scripts de implementación y eventos necesarios para instanciar el laboratorio virtual.

En el nivel del plano de control, el controlador se encarga de supervisar la conectividad de los ISP, procesar nuevos flujos de comunicación y actualizar dinámicamente el estado de enrutamiento del sistema. Los nuevos flujos se detectan a través de mecanismos de observación de paquetes que operan en el plano de datos de Linux, lo que permite al controlador asignar rutas de forma dinámica de acuerdo con las políticas configuradas.

Una vez seleccionada una ruta, las reglas de reenvío se inyectan en el plano de datos de Linux [3]. Este diseño permite que el reenvío de paquetes siga siendo gestionado principalmente por el propio kernel, minimizando la sobrecarga de procesamiento introducida por el plano de control y reduciendo el impacto en el rendimiento del sistema.

Además de la gestión del tráfico, el despliegue de NEPTUNE desarrollado implementa:

- Traducción de prefijos basada en NPTv6 para admitir escenarios de multihoming agregables por el proveedor, eliminando la necesidad de usar BGP
- Generación de hashes por conexión, que garantizan la continuidad de la ruta de salida a nivel de flujo, evitando la ruptura de la comunicación por cambio de dirección IP origen en protocolos de transporte orientados a conexión como TCP
- Presentación de una capa de visualización operativa ligera diseñada para mejorar la observabilidad durante la validación experimental

La arquitectura resultante proporciona un entorno totalmente virtualizado y reproducible capaz de simular escenarios de multihoming dinámicos utilizando las capacidades de red nativas de Linux, al tiempo que mantiene una separación modular entre la orquestación, la lógica de control, el reenvío del plano de datos y la visualización operativa.

4. Resultados

La validación experimental ha demostrado la capacidad del sistema para gestionar el tráfico con varios proveedores y reaccionar ante cambios de conectividad. Durante las pruebas realizadas, el sistema llevó a cabo con éxito la selección dinámica de rutas, la redistribución del tráfico y la conmutación automática por error entre múltiples ISPs.

En condiciones normales de funcionamiento, los flujos asociados a políticas de enrutamiento utilizaban el ISP definido en la configuración, mientras que los escenarios de balanceo de carga distribuían dinámicamente los nuevos flujos entre los ISPs activos.

Durante los eventos de fallo de los ISP, el controlador detectó automáticamente la pérdida de conectividad, actualizó el conjunto de balanceo y redirigió los nuevos flujos de tráfico hacia proveedores activos sin requerir intervención manual.

Una vez que los enlaces degradados volvieron a estar disponibles, el sistema los reincorporó dinámicamente al conjunto de rutas activas.

También se validó que los mecanismos de hashing de flujos garantizaron que las comunicaciones mantuvieran asignaciones de ruta estables durante su vida útil.

La Figura 2 muestra uno de los escenarios de validación de conmutación por error, incluyendo la monitorización en tiempo real de los ISP, la visualización de flujos activos y el seguimiento de eventos a través del panel de administración desarrollado.

Figura 2: NEPTUNE Realizando Conmutación por Error – Fuente: creación propia

Dado que las operaciones de reenvío siguen siendo gestionadas principalmente por el kernel, la sobrecarga observada introducida por los mecanismos de control es mínima. En general, los resultados experimentales demuestran la viabilidad técnica de la arquitectura propuesta para escenarios de multihoming IPv6 sin necesidad de usar BGP.

5. Conclusiones

Este proyecto ha presentado NEPTUNE, un marco operativo experimental diseñado para abordar escenarios de multihoming IPv6 mediante el uso de direccionamiento agregable por el proveedor, sin depender de las arquitecturas tradicionales basadas en BGP.

La validación experimental ha demostrado la capacidad del sistema para gestionar conexiones de múltiples proveedores y adaptarse dinámicamente a los cambios de conectividad, manteniendo la continuidad operativa de los flujos de tráfico.

Aunque la arquitectura propuesta permanece limitada a un entorno experimental y virtualizado, el trabajo realizado demuestra la viabilidad técnica del enfoque propuesto, estableciendo una base funcional para futuras líneas de investigación y posibles evoluciones operacionales en entornos IPv6 multihomed.

6. Referencias

[1] Deering, S.; Hinden, R. “Internet Protocol, Version 6 (IPv6) Specification”, RFC 8200, IETF, 2017.

[2] Wasserman, M.; Carpenter, B. “IPv6-to-IPv6 Network Prefix Translation”, RFC 6296, IETF, 2011.

[3] Wright, G.; et al. “Policy Routing Using Linux”, Linux Journal, 2000.

NEPTUNE: Network Prefix Translation Unified Environment

Author: **Carrasco Domínguez, Lucas**
Supervisor: **Fernández-Pacheco Sánchez-Migallón, Atilano Ramiro**
Collaborating Entity: **ICAI – Universidad Pontificia Comillas**

ABSTRACT

NEPTUNE proposes an operational framework for IPv6 edge multihoming scenarios that utilize Provider-Aggregatable addressing. The system integrates policy routing, flow persistence, and prefix translation mechanisms using NPTv6 to provide traffic balancing and failover without relying on BGP. The proposal has been implemented and experimentally validated in a virtualized network environment running on Linux.

Keywords: IPv6, Multihoming, NPTv6, Policy Routing, Failover

1. Introduction

IPv6 [1] was originally designed to address the limitations of IPv4 by introducing a much larger address space, thereby restoring the principle of end-to-end connectivity. However, nearly three decades after its specification, its adoption rate remains modest, as many scenarios still present significant implementation challenges.

Specifically, multihoming environments based on Provider-Aggregatable addressing still require the combination of multiple independent mechanisms, such as policy-based routing, prefix translation, and flow persistence, whose operational integration remains fragmented and relies heavily on specific implementations and ad-hoc configurations.

This project focuses on these scenarios by developing NEPTUNE, an experimental operational framework designed to provide IPv6 multihoming, failover, load balancing, and policy-based routing capabilities without relying on traditional BGP-based approaches. The proposed system integrates dynamic route selection, flow persistence, and prefix translation based on NPTv6 [2] within a unified Linux-based architecture.

2. Project definition

It is common for enterprise networks to use multiple Internet circuits from different service providers in order to improve availability, resilience, and available bandwidth.

In IPv4 deployments, Network Address Translation (NAT) simplifies multihoming scenarios by allowing multiple internal hosts to share a small number of public addresses, and facilitates relatively simple load balancing and failover mechanisms. In practice, small and medium-sized networks have typically relied on NAT-based approaches, while large enterprise and carrier environments often use BGP and provider-independent addressing to achieve advanced traffic engineering capabilities.

IPv6, however, significantly changes this operational model. The diminished role of NAT, combined with the operational complexity associated with deploying BGP, makes provider-aggregated multihoming scenarios considerably more difficult to implement.

Although mechanisms such as policy-based routing, prefix translation, and flow persistence already exist within modern TCP/IP stacks, there is currently no widely adopted provider-independent operational framework capable of integrating these mechanisms into a unified and manageable architecture.

In response to this problem, NEPTUNE (Network Prefix Translation Unified Environment) is proposed as an experimental operational framework designed to provide IPv6 multihoming capabilities by integrating policy-based routing, dynamic flow steering, and NPTv6-based prefix management mechanisms.

3. Description of the system

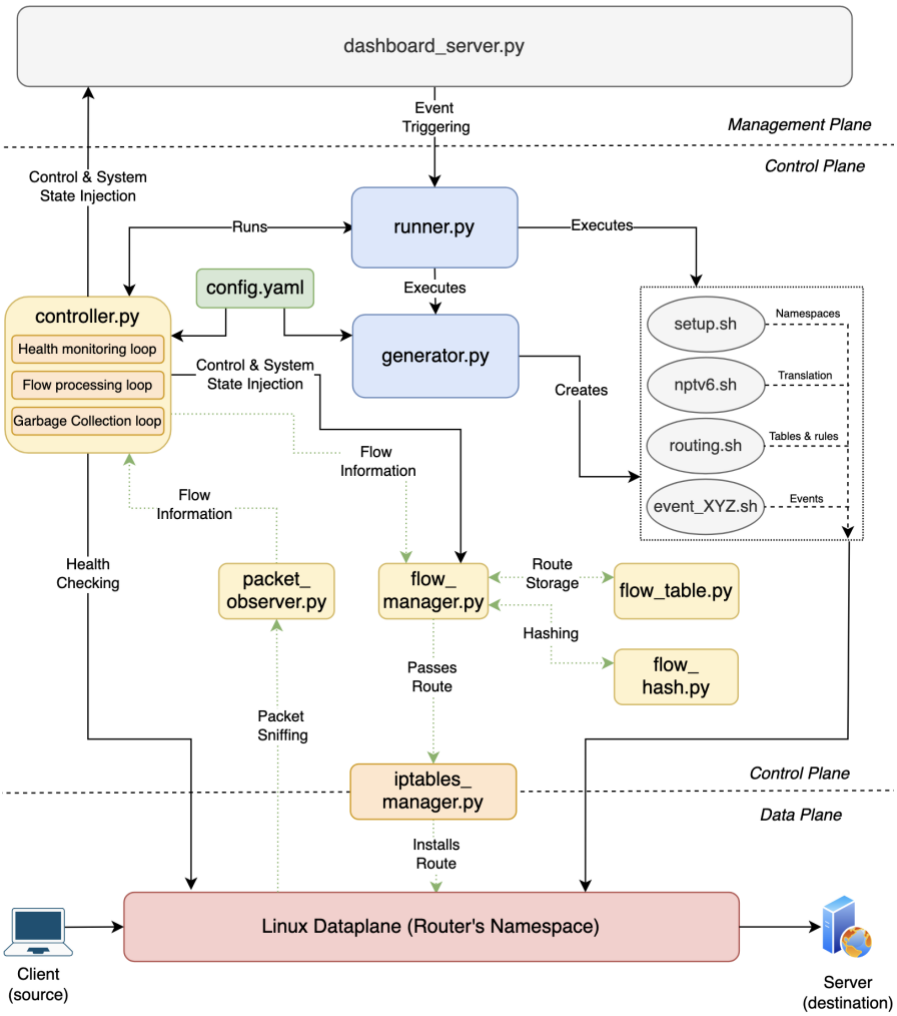


Figure 1: Diagram of the NEPTUNE Environment Components – Source: self-made

The framework developed follows a layered architecture that separates the responsibilities of the management, control, and data planes. Figure 1 illustrates the model's architecture, showing its main components and the interactions between them.

At the highest level, the system is governed by a declarative YAML configuration model that describes the topology, available ISPs, routing information, load-balancing policies, and experimental environment parameters. This configuration is processed by the generator subsystem, which automatically creates the deployment and event scripts necessary to instantiate the virtual lab.

At the control plane level, the controller is responsible for monitoring ISP connectivity, processing new communication flows, and dynamically updating the system's routing state. New flows are detected through packet observation mechanisms operating in the Linux data plane, allowing the controller to dynamically assign routes according to configured policies.

Once a route is selected, forwarding rules are injected into the Linux data plane [3]. This design allows packet forwarding to remain primarily managed by the kernel itself, minimizing the processing overhead introduced by the control plane and reducing the impact on system performance.

In addition to traffic management, the deployed NEPTUNE implementation features:

- NPTv6-based prefix translation to support provider-aggregated multihoming scenarios, eliminating the need for BGP
- Connection-based hash generation, which ensures the continuity of the outgoing path at the flow level, preventing communication disruption due to changes in the source IP address in connection-oriented transport protocols such as TCP
- Provision of a lightweight operational visualization layer designed to improve observability during experimental validation

The resulting architecture provides a fully virtualized and reproducible environment capable of simulating dynamic multihoming scenarios using native Linux networking capabilities, while maintaining a modular separation between orchestration, control logic, data plane forwarding, and operational visualization.

4. Results

Experimental validation has demonstrated the system's ability to manage traffic across multiple providers and respond to changes in connectivity. During testing, the system successfully performed dynamic route selection, traffic redistribution, and automatic failover among multiple ISPs.

Under normal operating conditions, traffic flows associated with routing policies used the ISP defined in the configuration, while load-balancing scenarios dynamically distributed new traffic flows among active ISPs.

During ISP failure events, the controller automatically detected the loss of connectivity, updated the load-balancing set, and redirected new traffic flows to active providers without requiring manual intervention.

Once the degraded links became available again, the system dynamically reincorporated them into the set of active routes.

It was also validated that the flow hashing mechanisms ensured that communications maintained stable route assignments throughout their lifetime.

Figure 2 shows one of the failover validation scenarios, including real-time monitoring of ISPs, visualization of active flows, and event tracking via the developed management dashboard.

Figure 2: NEPTUNE Performing Failover Switching – Source: self-made

Since forwarding operations are still primarily handled by the kernel, the observed overhead introduced by the control mechanisms is minimal. Overall, the experimental results demonstrate the technical feasibility of the proposed architecture for IPv6 multihoming scenarios without the need for BGP.

5. Conclusions

This project has introduced NEPTUNE, an experimental operational framework designed to address IPv6 multihoming scenarios by using provider-aggregated addressing, without relying on traditional BGP-based architectures.

Experimental validation has demonstrated the system's ability to manage connections from multiple providers and dynamically adapt to connectivity changes, maintaining the operational continuity of traffic flows.

Although the proposed architecture remains limited to an experimental and virtualized environment, the work carried out demonstrates the technical feasibility of the proposed approach, establishing a functional foundation for future lines of research and potential operational developments in multihomed IPv6 environments.

6. References

[1] Deering, S.; Hinden, R. "Internet Protocol, Version 6 (IPv6) Specification", RFC 8200, IETF, 2017.
[2] Wasserman, M.; Carpenter, B. "IPv6-to-IPv6 Network Prefix Translation", RFC 6296, IETF, 2011.
[3] Wright, G.; et al. "Policy Routing Using Linux", Linux Journal

Report's Index

Figure Index

Table Index

Chapter 1. INTRODUCTION

1.1 PROJECT MOTIVATION

The deployment of IPv6 across the Internet has experienced substantial growth in recent years, driven by the exhaustion of the IPv4 addressing space.

The original IP architecture assumed the availability of globally unique addressing for each Internet-connected host. However, the large growth of connected devices during the last two decades constitutes a situation not envisaged by the original specification from 1981 (RFC 791) [1], making this addressing scheme simply not possible using IPv4.

Therefore, as a temporary patch, Internet Service Providers (ISPs) started using Network Address Translation (NAT) in fixed broadband plans. NAT, as used in such deployments, consists of delegating public addressing (one or a few addresses) to the customer's home/business router, and assigning private addressing to the end devices inside the local network, enabling Internet connectivity to multiple devices using a few addresses by translating back and forth from private to public addressing [3].

Even with this address provisioning scheme, IPv4 scarcity continued escalating, leading to the use Carrier-Grade NAT (CG-NAT), performing address translation at ISP level, sharing public addressing among various customers [4], both in fixed and mobile broadband.

The lack of scalability of this situation increased the efforts in transitioning towards IPv6, the evolution to IPv4. IPv6 has a larger addressing space compared to IPv4, particularly 2^{128} (approximately 340 undecillion) unique addresses [2], compared to the 2^{32} (approximately four billion) of IPv4, restoring the possibility of coming back to the original IP protocol's approach of assigning a globally routable address to each host connected to the Internet.

1.2 PROBLEM STATEMENT

IPv6's larger addressing space raises inter-domain routing scalability concerns due to its potential for fine-grained prefix allocation, as global routing tables could experience unprecedented growth if addressing were heavily disaggregated into small blocks. Therefore, as a containment measure, IPv6 deployments follow a hierarchical allocation model designed to preserve aggregation by minimizing prefix fragmentation.

This operational principle leads to a preference for the use of Provider-Aggregatable (PA) addressing [5]. PA consists of addressing provided to the customer by the ISP, who is typically in control of a big set of adjacent blocks, being able to announce them to the Internet as a single aggregated large block, therefore constituting a single route announcement [6].

This situation heavily limits a very common scenario present in current IPv4 networks, where companies in the need for public addressing can acquire a Provider-Independent (PI) network prefix and sign up a connectivity agreement with one or more ISPs, being able to change their upstream provider down the line without changing their internal addressing. In IPv6 networks, this scenario is, therefore, discouraged in favor of using PA addressing.

The issue might not seem evident at first glance, as those companies could sign up for a broadband plan from an ISP offering an addressing space that suits their needs. However, when considering that businesses, particularly those whose operations heavily rely on Internet connectivity, tend to use multihoming, the practice of connecting to the Internet through more than one upstream ISP [7], the limitations of PA addressing become evident.

In such multihomed deployments, the use of Provider-Aggregatable addressing introduces a structural dependency between the assigned prefix and the upstream provider from whose allocation the prefix is derived. As a result, the customer network cannot advertise its assigned prefix through multiple upstream providers without requiring prefix disaggregation, which increases the size of the global routing table and prevents independent control over the selection of inbound and outbound traffic paths.

1.3 SCOPE OF THE PROJECT

This work focusses on the analysis of IPv6 multihoming in environments where globally routable Provider-Aggregatable addressing is deployed and its implications, as well as existing solutions, particularly for small to medium-size enterprise networks.

In addition to studying the matter of IPv6 multihoming, it needs to be considered that, to date, most IPv6-capable connectivity services include IPv4 connectivity alongside IPv6 for interoperability purposes with legacy networks and services. This IPv4 connectivity, whose multihoming is already supported through existing operational practices, is also considered as part of the deployment assumptions of the proposed framework.

This work does not consider the addition of new protocols to the TCP/IP stack in any of its layers. TCP/IP already counts with some of the required mechanisms to enable the set-up of most scenarios with a certain level of functionality, but lacks an integrated operational model that enables scalable multihoming under Provider-Aggregatable addressing constraints. The state of the art of the present solutions will be considered and reviewed.

The project is neither concerned with the advantages and disadvantages from the different IPv6 deployment methods, transition mechanisms nor ISP's practices.

1.4 CONTRIBUTIONS

This work provides an operational framework to enable scalable IPv6 multihoming in environments where Provider-Aggregatable (PA) addressing is deployed.

The TCP/IP protocol suite already includes mechanisms that can support IPv6 multihoming under certain deployment models, albeit with limited operational flexibility. However, the lack of an integrated operational model that enables scalable multihoming under Provider-Aggregatable addressing constraints remains an open deployment challenge.

The proposed framework introduces an abstraction layer between network administrators and existing control-plane mechanisms, enabling coordinated traffic engineering across multiple upstream providers without requiring Provider-Independent addressing or modifications to the TCP/IP protocol stack. In addition, considering that most IPv6-capable broadband services include IPv4 connectivity alongside IPv6 for interoperability purposes with legacy networks and services, IPv4 multihoming, already supported through established practices, is also considered within the deployment assumptions of the proposed framework.

The proposed framework is designed to operate on top of existing IPv6 deployment models and transition mechanisms, while also integrating IPv4 multihoming capabilities to ensure interoperability with legacy services.

The main contributions of this work can be summarized as follows:

- A deployment model for IPv6 multihoming under Provider-Aggregatable addressing constraints
- A unified operational framework for IPv4 and IPv6 multihoming
- Traffic engineering capabilities for inbound and outbound traffic across multiple upstream providers
- Compatibility with heterogeneous ISP IPv6 deployment architectures

1.5 DOCUMENT STRUCTURE

This document is structured as follows:

Chapter 2 reviews the technological foundations required to understand the problem addressed in this work, including the working principles of the Internet protocol suite, routing mechanisms, IPv6 addressing models, host configuration procedures and router architecture.

Chapter 3 presents the current state of the art regarding multihoming practices in IPv4 and IPv6 networks, as well as the operational limitations introduced by Provider-Aggregatable addressing in IPv6 deployments.

Chapter 4 formally defines the problem addressed in this project and outlines the objectives and methodology followed throughout the work.

Chapter 5 introduces the proposed operational framework for scalable IPv6 multihoming, including its architectural design and deployment model.

Chapter 6 evaluates the obtained results and discusses the implications of the proposed approach in real-world deployment scenarios.

Chapter 7 summarizes the main conclusions of this work and outlines potential directions for future research.

Chapter 8 contains all the references highlighted along the project in the IEEE format

Chapter 2. TECHNOLOGIES DESCRIPTION

This chapter is focused on introducing the reader to a set of technologies, protocols and concepts that will be used along the project, and their working principles, in order to understand their use and role in the final designed project.

2.1 PACKET DELIVERY ON THE INTERNET

2.1.1 TCP/IP PROTOCOL SUITE

TCP/IP constitutes a set of rules and mechanisms that every host in a computer network must follow in order to enable communication and transmit data across the network [8].

It is not a protocol itself, but an architecture definition, integrated by different protocols that accomplish different functions inside the networking stack of each host.

TCP/IP is structured in five layers [8], and each of them handles a different part of the communication process. A brief description of the layers is given below:

Application Layer (Layer 5)
The application layer constitutes the highest level in the TCP/IP stack and integrates the different types of services that can run and transmit information through the network.

It establishes how the different applications generate and structure the information that will be exchanged through the network. It defines the formatting and semantics of the data, but it does not handle reliability, data fragmentation nor data transmission.

Protocols that operate in this layer are, for example, HTTP for hypertext transmission, FTP for file sharing, or SMTP for email transfer, among others.

Transport Layer (Layer 4)

The transport layer establishes end-to-end communications between processes, establishing how data is packetized.

It can guarantee the reliable transmission of data, being able to mitigate network congestion, packet loss and other errors that may occur during transmission.

However, it is not aware of the underlying topology of the network nor recognize how to route the data through it.

Network Layer (Layer 3)

The network layer, often referred to as the *Internet Layer*, enables the interconnection of heterogeneous networks, and routes the information through them based in an addressing system defined by the IP protocol, in their versions 4 (IPv4) and 6 (IPv6) [1][2].

It is completely agnostic to the underlying infrastructure and therefore to the link layer, in such a way that, as long as the two endpoints in the communication process use the same layer 5, 4 and 3 protocols, a link can be established, regardless of the technologies used in layers 2 and 1.

Link Layer (Layer 2)

The link layer allows data delivery inside the same physical link and manages the access in the event of a shared media, for example in wireless communications, where multiple devices access the same radio spectrum.

Physical Layer (Layer 1)

The physical layer constitutes the media through which the data will be transmitted, and it is tightly related to the link layer above it, being both often presented as a single layer called the *Network Access Layer*.

Examples of physical layers are copper cable, optic fiber or radiofrequencies.

<table>
<tr><td rowspan="2">Processed only by the endpoints (typically)</td><td>Application Layer (L5)</td><td rowspan="2">Must use the same protocol</td></tr>
<tr><td>Transport Layer (L4)</td></tr>
<tr><td rowspan="3">Processed by endpoints and intermediate hosts</td><td>Network Layer (L3)</td></tr>
<tr><td>Link Layer (L2)</td><td rowspan="2">Protocols can be heterogeneous</td></tr>
<tr><td>Physical Layer (L1)</td></tr>
</table>

Figure 2.1.1.1: The TCP/IP Protocol Stack - Source: self-made

Both link and physical layer protocols can and tend to vary along the path data follows to its destination. A packet can be sent from a computer connected to the Internet via a copper cable, travel through an optic fiber across the ocean and arrive to a laptop connected via 5G. Both the physical media and the transmission technologies can be heterogeneous, and that way networks can be upgraded by segments with ease.

On the other hand, both application and transport layers need to support and use the same protocols end-to-end for the communication to function properly. However, as the headers from these layers are only processed by the endpoints (transmitter and receiver of data), only end-user equipment need to be updated to incorporate new L4 and L5 protocols, without the need to touch the underlying network infrastructure.

The network layer is processed by both endpoints and intermediate hosts (mainly routers), like layers 1 and 2, but it also requires the protocol used to be the same end-to-end, like layers 4 and 5. For these reasons, the number of protocols used in layer 3 should be minimal, and such protocols must barely be modified, as changes would require updates in each and every device, making them complex at both technical and economic levels.

Layer 3 mainly uses the Internet Protocol (IP), in their versions 4 (IPv4) and 6 (IPv6).

2.1.2 IP ADDRESSING: IPV4 AND IPV6

The IP protocol enables communication across heterogeneous, independent networks [1][2]. Its design stems from the need to disengage end-to-end communications from the physical characteristics of the underlying network. It is the main protocol used in Layer 3 routing.

Other additional protocols exist in this layer for control messaging (ICMP, Internet Control Message Protocol) [9] and security (IPsec, Internet Protocol Secure) [10], but they all use IP-compatible addressing, and are built as complements rather than alternatives to IP.

Its working principle resides on addresses, used to uniquely identify hosts across the network. The transmitter sends packets with the address of the target receiver, and the intermediate routers use that address in order to forward the packet to the next intermediate router, following the best suitable path according to different routing policies.

The first official version, IP version 4 (IPv4) offered an addressing space of 2^{32} (around four billion) unique addresses [1]. IPv4 addresses are expressed in decimal format, for example: **21.11.124.251**. With the growth in the number of connected devices, IPv4 addresses became increasingly scarce, creating the need for a successor that could solve IPv4's deficiencies.

IPv6 was proposed as an evolution to IPv4, offering a much greater addressing space of 2^{128} (around 340 undecillion) unique addresses [2], therefore being more suitable for the requirements of modern networks. IPv6 addresses are expressed in hexadecimal format, for example: **2001:0db8:85a3:0000:0000:8a2e:0370:7334**.

However, IPv4 and IPv6 are different, non-interoperable protocols. As mentioned before, layer 3 requires the protocol used for communication to be the same end-to-end and supported by both the endpoints and all the intermediate hosts in between, creating a need for transition mechanisms from IPv4 to IPv6 in order to enable a progressive upgrade.

Before exploring how the transition mechanisms work, we need to understand the architecture of the Internet and the roles that each component plays on it.

2.1.3 INTERNET ARCHITECTURE

The Internet is a distributed system built upon the interconnection of autonomous networks that collectively provide global connectivity [11]. Rather than operating as a single, centrally managed entity, it emerges from the interconnection of independently administered network domains that exchange information among them.

The scale and complexity of this interconnection make it complex to reason about the Internet as a monolithic system. Instead, it is useful to adopt an abstract view in which the network is divided into logical areas that accomplish differentiated roles.

The following diagram illustrates this architectural model, applied to an Internet Service Provider Network, in a simplified form:

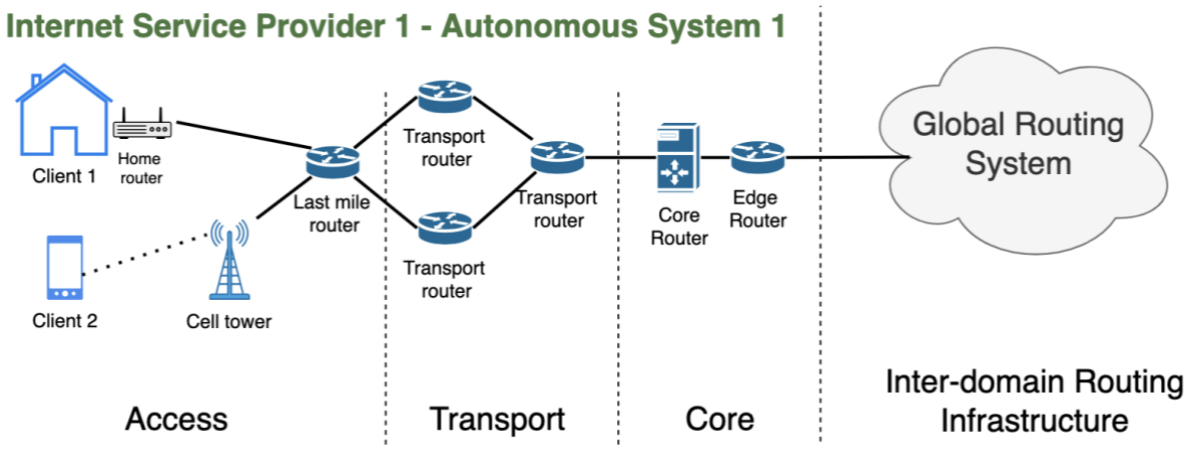


Figure 2.1.3.1.: Simplified architecture of an ISP – Source: self-made

An ISP network can be structured in three differentiated domains:

1. The **access network** (also known as **last mile**) comprehends the network segment that connects the last provider's router (last mile router) with the client premises equipment (CPE from now on), which might be a home router in the case of fixed Internet access, or a cell phone in the case of mobile networks.

2. The **aggregation network** (also known as **transport** or **backhaul**) integrates the set of routers (transport routers) that carry the user's traffic from the last mile to the ISP's network core across long distances.
3. The **network core** comprehends a set of routers and servers, typically centralized in a small set of physical locations, that process the user's traffic and send it to its destination, which can be inside or outside of the provider's network. The traffic that goes outside the provider's network passes through a router called edge router, which, depending on the ISP's configuration, can be the core router itself.

This set of physical (routers and servers) and logical (addressing and routing) resources under the control of a single administrative authority (typically an ISP or a big company) is known as an *Autonomous System* (AS from now on) [12]. Each AS delimitates a routing domain, that is, a network region with the same routing policy and routing control system.

When traffic does not need to go outside the AS, for example in a voice call between clients from the same ISP, it will be routed through network segments strictly inside the ISP's network until it reaches the destination's CPE.

However, in most cases, traffic needs to travel towards a third-party server located in another AS. This includes web browsing; texting, audio and video calling using a third-party service; video and music on-demand services and email, among many others.

In such cases, packet forwarding must rely on routing relationships established between different Autonomous Systems through the Global Routing System (GRS) [13]. The GRS is a distributed inter-domain routing infrastructure that enables independently administered networks to exchange reachability information and route traffic across administrative boundaries.

With this architecture in mind, we can explore how the different IPv6 transition mechanisms work, and how they integrate into the Internet's infrastructure and architecture.

2.1.4 IPV4-IPV6 INTEROPERABILITY: TRANSITION MECHANISMS

In order to overcome the inherent complexity of a global transition, a direct replacement of IPv4 by IPv6 was never considered a viable option. At first, some ISPs, Internet-based companies and network administrators of different institutions started experimenting with incorporating IPv6 to their networks, or to parts of them.

However, as described previously, TCP/IP requires the layer 3 (IP) protocol used to be the same end-to-end. Therefore, even if some network segments do support IPv6, until the two endpoints and all the intermediate hosts in between are IPv6-enabled, communication using IPv6 will not be possible.

When trying to establish communication over a network, we can encounter three scenarios:

1. **Legacy case scenario:** both endpoints and the network in between support IPv4, but at least one element does not support IPv6. Communication is established using IPv4.

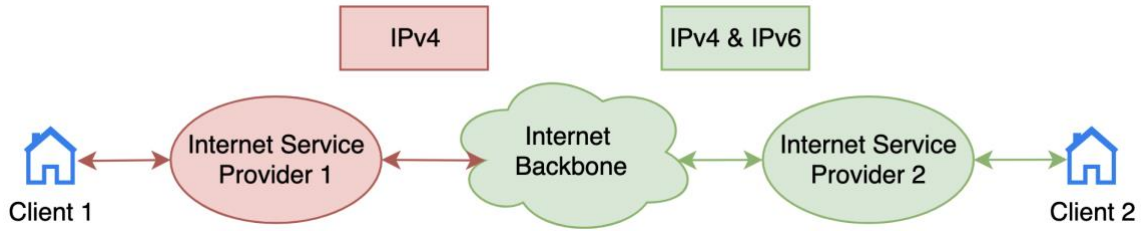


Figure 2.1.4.1.: Communication over IPv4 – Source: self-made

2. **Best case scenario:** both endpoints and the network in between support IPv6. Communication is established using IPv6.

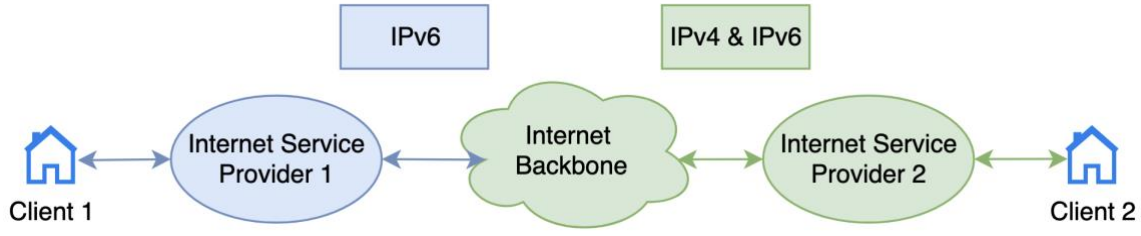


Figure 2.1.4.2.: Communication over IPv6 – Source: self-made

3. **Worst case scenario:** some elements in the communication (endpoints or network segments) only support IPv4, and others only IPv6. **Communication cannot occur**.

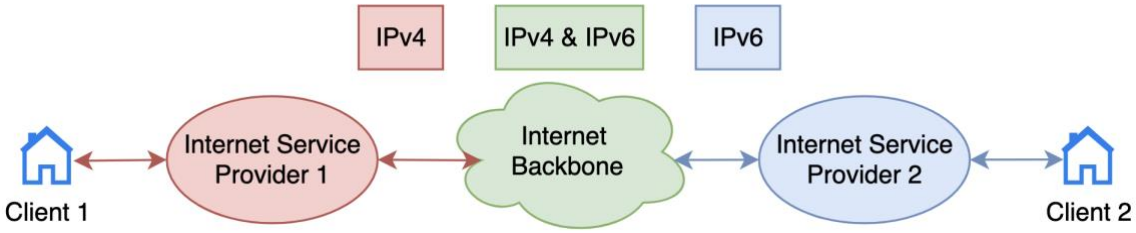


Figure 2.1.4.3.: Communication not possible – Source: self-made

In order to avoid situations such as the one illustrated in *Figure 2.1.4.3*, where communication cannot be established due to incompatible IP versions across network segments, transition mechanisms were introduced to enable interoperability between IPv4 and IPv6 domains.

During a communication flow, different types of packet adaptation may be required. Broadly speaking, two transition scenarios can be identified:

- **Transporting IPv6 traffic over an IPv4 network**, which was historically used to provide IPv6 connectivity across legacy IPv4 infrastructures.
- **Transporting IPv4 traffic over an IPv6 network,** which is currently the more common approach, allowing modern networks that have already transitioned to IPv6-only to maintain compatibility with IPv4-only services.

During the process of proposing IPv6 transition mechanisms to address these scenarios, engineers came up with a wide range of solutions and methods. However, the ones that generated the greatest interest were those which were proposed as an IPv6-first approach with IPv4 compatibility rather than the other way around. Three of them stand out greatly by their wide adoption: **Dual Stack**, **Dual Stack Lite** and **464XLAT**.

Below, a brief of the most common transition mechanisms is shown. They are classified by their working principle: **native access**, **address translation** and **packet encapsulation**.

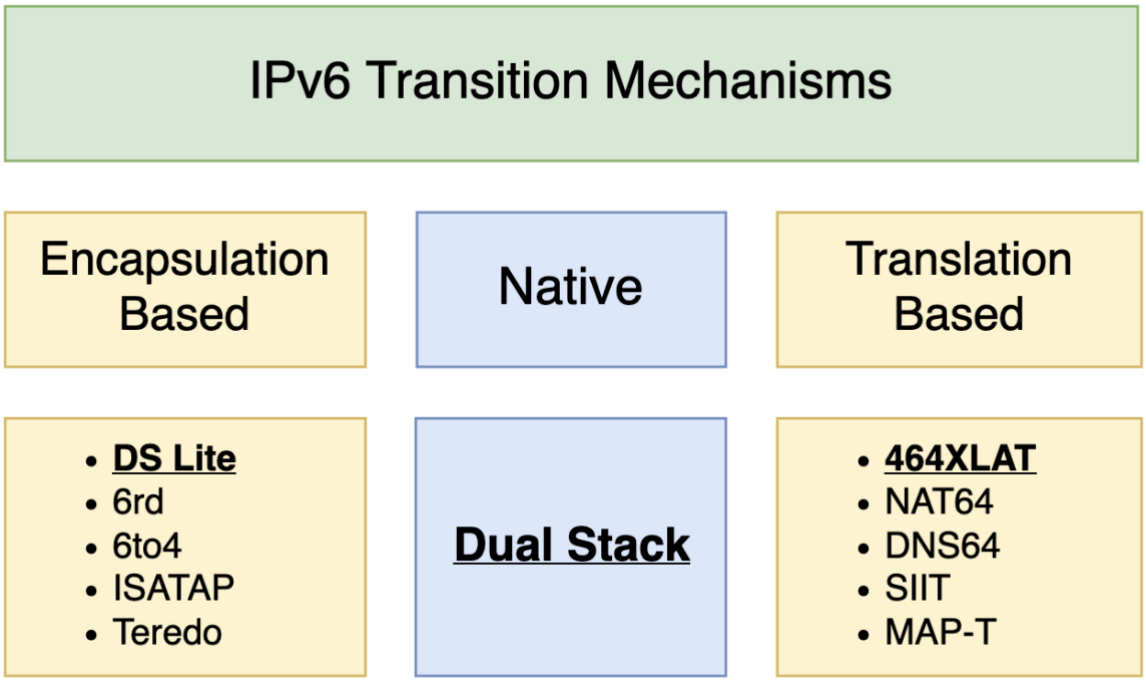


Figure 2.1.4.4.: IPv6 Transition Mechanisms – Source: self-made

2.1.4.1 Native transition mechanisms: Dual Stack

DS: Dual Stack (RFC 4213) [14]

Dual Stack emerges as the most robust, complete, and at the same time simple mechanism for IPv6 transitioning. It consists of deploying both IPv4 and IPv6 networks in parallel, running over the same hardware, but being two completely isolated logical networks.

The Internet Service Provider (ISP) deploys IPv4 and IPv6 networking stacks from the last mile (end users) up to the network edge, where information is exchanged with other Autonomous Systems from other ISPs and content delivery companies.

An oversimplified, fictional diagram of the Internet is presented, where some parts of the network (in green) are running Dual Stack, while others (in red) are running only IPv4:

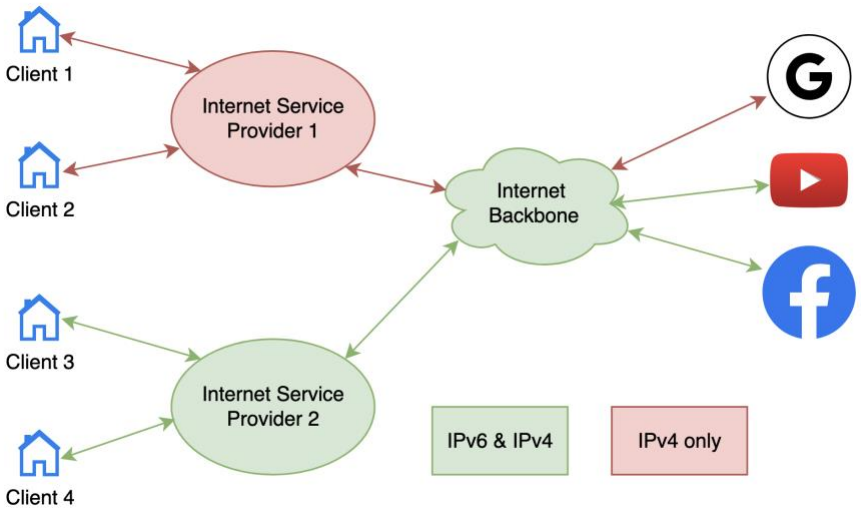


Figure 2.1.4.1.1.: Dual Stack – Source: self-made

In this example, clients from *Internet Service Provider 2* will be able to access YouTube and Facebook via IPv6 and IPv4, and Google via IPv4 only, while clients from *Internet Service Provider 1* will access all of them via IPv4.

That way, the user will be able to access such services using IPv6 as long as those service companies have implemented IPv6, and if they have not, they still have the ability to access them via IPv4. Therefore, as more services start supporting IPv6, its usage will increase.

Dual Stack's main drawback is its high deployment cost. Routers have a limited amount of memory and computing capacity, and having two parallel networking stacks deployed requires the provider to oversize their equipment to accommodate this dual infrastructure.

The Global Routing System started evolving towards a Dual Stack model throughout the 2000s, as Autonomous Systems progressively enabled native IPv6 support alongside existing IPv4 connectivity.

To mitigate the elevated associated with Dual Stack deployment, alternative transition approaches enable communication via both IPv4 and IPv6, even when some segments of the network are IPv4-only or IPv6-only. In practice, the latter is more common, due to the long-term approach being the gradual migration of every network towards IPv6.

This interoperability can be achieved via **address translation** or **packet encapsulation**. In both cases, the savings in required resources across the whole network come at the same expense: an increased complexity at the endpoint routers of the IPv6-only network segment.

2.1.4.2 Encapsulation-based transition mechanisms

Encapsulation operates by taking the original IPv4 packet, including its Layer 3 header, and using it as payload for an IPv6 packet, effectively adding another Layer 3 header.

This encapsulation is carried out by the endpoints of the IPv6-only segment, while the intermediate routers inside the network are not aware of this condition, and therefore, will treat the traffic as if it was composed of native IPv6 packets.

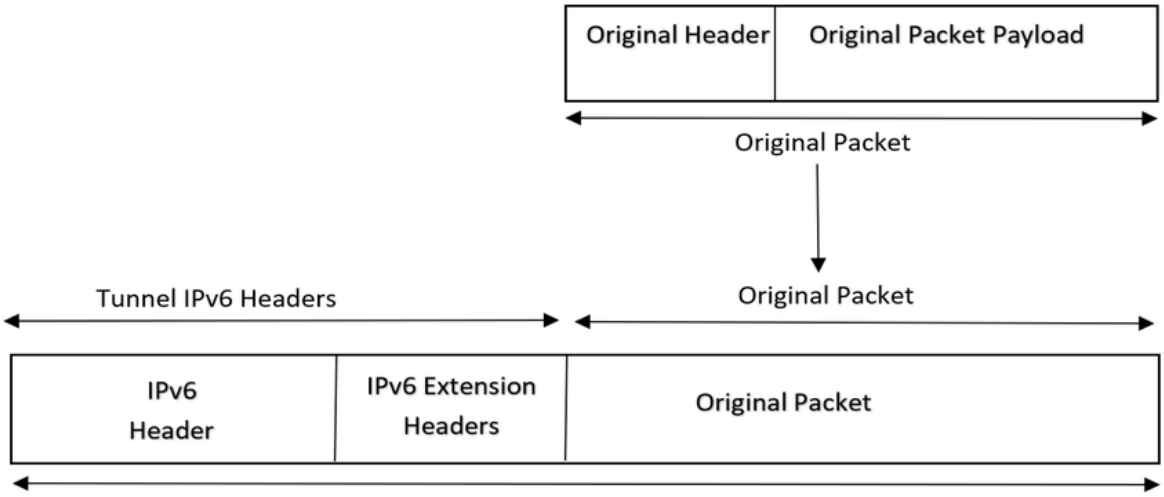


Figure 2.1.4.2.1.: IPv4 in IPv6 Encapsulation – Source: researchgate.net

The most widely deployed, encapsulation-based transition mechanism is **Dual Stack Lite**.

DS-Lite: Dual Stack Lite (RFC 6333) [15]

Dual Stack Lite was created as a more affordable IPv6 transition mechanism, particularly from the ISP's standpoint. Instead of deploying two networking stacks across the whole network, in a DS-Lite setup, the whole provider's network is IPv6-only, and encapsulation elements are placed at the CPE and within the ISP's network.

These encapsulation elements allow IPv4 traffic generated by the client to travel towards the IPv4 Internet. These components are the **B4 element** and the **AFTR**:

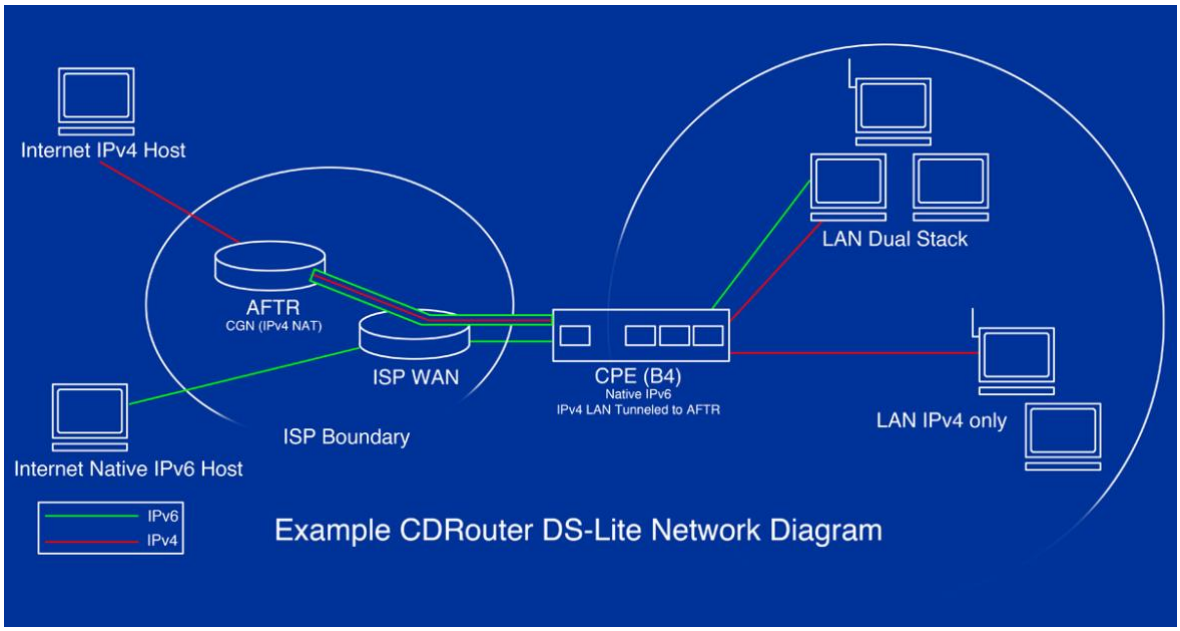


Figure 2.1.4.2.2.: Dual Stack Lite Architecture – Source: support.qacaffe.com

- **Basic Bridging BroadBand element** (**B4 element**): it is located at the CPE (typically the ISP's provided gateway). Its role is to both encapsulate outgoing IPv4-in-IPv6 packets and decapsulate incoming IPv6-in-IPv4 packets.
- **Address Family Transition Router** (**AFTR**): it is placed inside the ISP network. In addition to the encapsulation and decapsulation carried out also by the B4 element, it also translates the private IPv4 address of the client device to a public IPv4 address from an address pool, performing a carrier-grade network address translation (CG-NAT from now on) for the packet to be routable towards the IPv4 Internet.

The local area network can be IPv4-only or mixed IPv6-IPv4. If it's the former, users will not have connectivity towards the IPv6 Internet, but the ISP will still save resources in terms of IPv4 addressing to be purchased for the network to operate.

Dual Stack Lite is most commonly deployed in fixed broadband environments. Its two main disadvantages are:

- Maximum transmission unit (MTU) overhead: adding an additional header to each packet effectively reduces the effective path MTU.
- Carrier-grade network address translation side-effects: each IPv4 address in the pool is shared across multiple clients, therefore making it complicated for inbound traffic to reach a certain client's network, for example in self-hosted services. CG-NAT also increases the stateful processing requirements for the ISP's equipment.

Other encapsulation-based transition mechanisms such as 6rd, 6to4, ISATAP or Teredo do exist [16], but their current implementation in production infrastructure is more limited.

2.1.4.3 Translation-based transition mechanisms

Unlike encapsulation-based approaches, which work by adding an additional network-layer header, translation-based transition mechanisms operate by modifying the packet's IP header to convert it from one protocol version to another.

This process enables communication between IPv4-only and IPv6-only endpoints by mapping address spaces and adapting protocol semantics at the network edge. As a result, translation mechanisms allow native IPv6 infrastructure to provide connectivity towards IPv4 services without requiring dual-stack support across the entire network.

The process of translating an IPv4 address to an IPv6 address is called NAT46 (Network Address Translation 4-to-6), while the reverse process is called NAT64 [17].

464XLAT: 4-6-4 Extended Translation (RFC 6877) [18]

464XLAT is one of the most widely deployed translation-based IPv6 transition mechanisms, particularly in mobile broadband environments, such as 4G and 5G deployments.

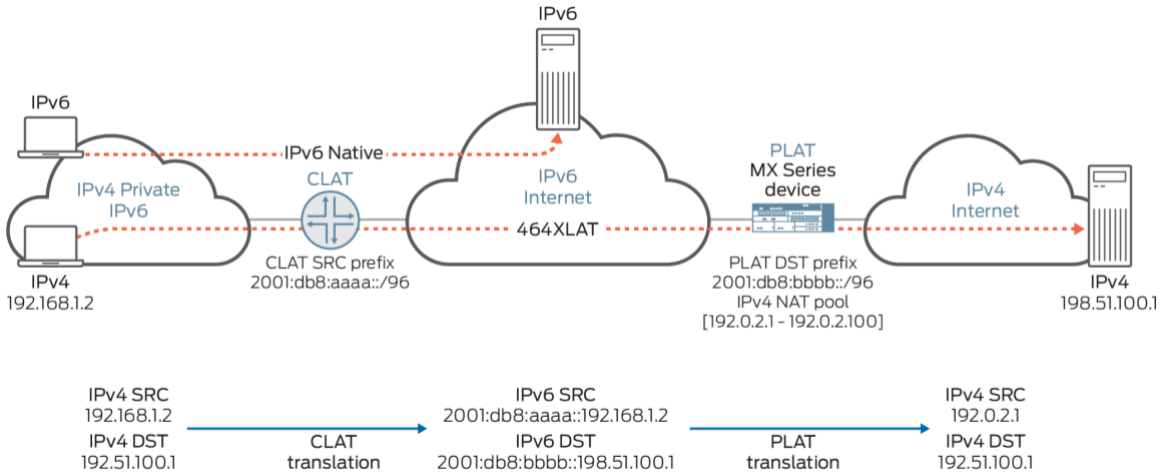


Figure 2.1.4.3.1.: 464XLAT – Source: juniper.net

In 464XLAT, as in DS-Lite, the ISP's network is IPv6-only. In this case, translation elements are placed both at the CPE and within the ISP's network to allow the flow of IPv4 traffic over the IPv6-only network segment.

These elements are the **CLAT** and the **PLAT**:

- **Customer-side translator** (**CLAT**): it is located in the CPE for fixed access, and in the client's terminal itself for mobile broadband, and converts traffic from IPv4 to IPv6 by applying NAT46.
- **Provider-side translator** (**PLAT**): it is located within the ISP's network and converts outgoing traffic from IPv6 to IPv4 (NAT64), and vice versa for returning traffic.

464XLAT's main disadvantage, particularly in mobile deployments where it has seen greater adoption, is the requirement for customer devices to support CLAT functionality. Legacy terminals lacking this capability may experience limited connectivity for services that do not support IPv6, a situation which may require the user to update or upgrade them.

2.2 ROUTING AND REACHABILITY

The previous section has described the architectural and technological foundations of the Internet that enable packet delivery across interconnected networks. However, effective communication also requires mechanisms to determine how packets are routed across such networks. The following section introduces the routing principles that govern reachability both within and between Autonomous Systems, as well as the policy constraints that influence traffic forwarding decisions when different suitable paths are available.

First off, it is important to define what a router is. A router is a network component integrated by multiple physical or logical interfaces associated with different network segments or routing domains [34], enabling the **forwarding of traffic between different networks**.

The logic inside a router can be modelled into 3 planes with differentiated functionality [35]:

- **Data plane**: performs the forwarding based on the control plane's decisions
- **Control plane**: makes the forwarding decisions the data plane follows
- **Management plane**: offers a user interface to configure the control plane

Routing is TCP/IP layer 3's main function, and consists of determining the paths that packets should follow towards their destination [8]. The process can be divided into two parts:

- **Forwarding**, which is performed by the data plane according to the routing table: the router internally stores a table where IP networks are associated with the address of the next hop in the path towards such networks.
- The population of the **routing table**, performed by the control plane according to routing protocols (also known as gateway protocols) and configured policies. Routing protocols enable routers to exchange reachability information, and enable participating routers to compute their tables in such a way that the best path is selected considering the configured policies. Routing protocols can be classified into **interior** gateway protocols (**IGPs**), used for routing within the same AS, and **exterior** gateway protocols (**EGPs**) for exchanging routes between different ASs.

2.2.1 ROUTING DOMAINS AND AUTONOMOUS SYSTEMS

As defined in previous sections, an Autonomous System is a set of physical and logical network resources under the control of a single entity. Autonomous Systems are identified by an Autonomous System Number (ASN) [12]. An AS also delimitates a routing domain, that is, a finite network region across which the routing policies are homogeneous.

Within a routing domain, Interior Gateway Protocols (IGPs) are implemented, and all the routers in the network make routing decisions under a unified criterion and policies.

The main reason that the Internet has not converged into a single routing domain is that the multiple networks that integrate it have different requirements in terms of security, performance, costs and commercial agreements.

However, the fact that Autonomous Systems are independently managed sets of networks but at the same time need to exchange information with one another poses the need for a model that makes this inter-domain communication possible, while at the same time allowing Autonomous Systems to manage their internal routing themselves.

Therefore, the External Gateway Protocol (EGP) used by ASs to exchange routes needs to be the same across the whole Internet. The protocol used is called Border Gateway Protocol (BGP) [13], and it is run on the edge routers of the different Autonomous Systems.

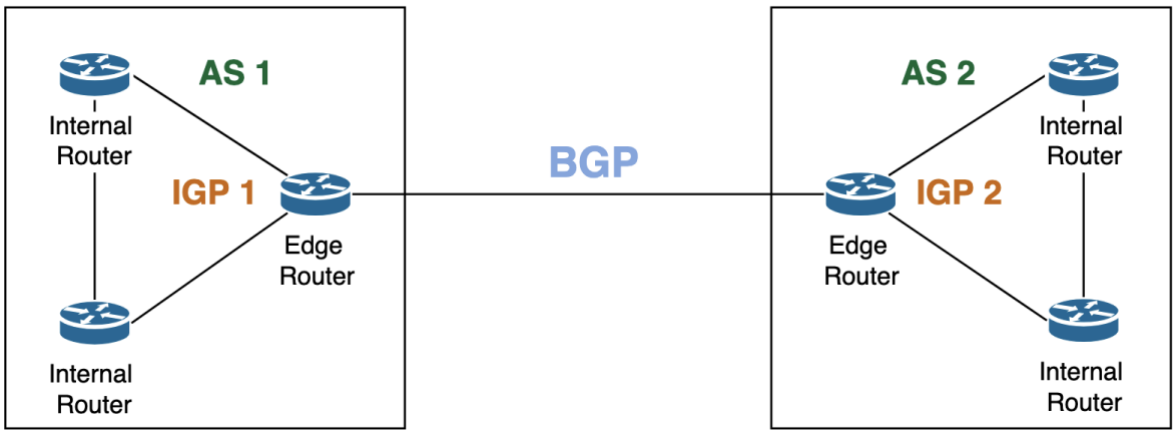
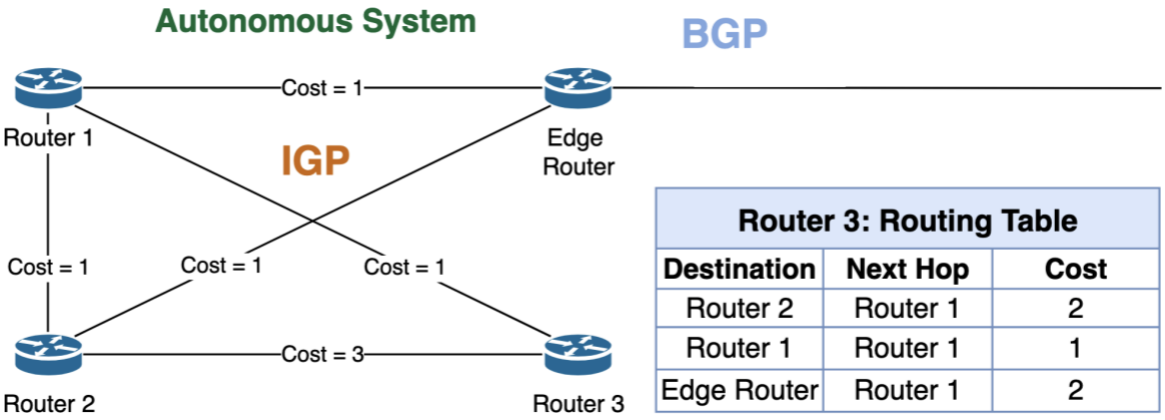


Figure 2.2.1.1.: BGP & IGP Roles – Source: self-made

2.2.2 INTRA-DOMAIN ROUTING

Routing decisions within a routing domain are performed using Interior Gateway Protocols (IGPs) [19], which enable routers inside an Autonomous System to exchange reachability information and select the optimal paths towards internal destinations.

IGPs operate under a common administrative authority and are designed to ensure efficient and reliable packet delivery within the network by selecting paths based on predefined performance metrics such as link cost, delay or bandwidth. Intra-domain routing primarily focuses on optimizing network performance and fast convergence to topology changes.



Router 3: Routing Table		
Destination	**Next Hop**	**Cost**
Router 2	Router 1	2
Router 1	Router 1	1
Edge Router	Router 1	2

Figure 2.2.2.1.: Interior Gateway Protocols – Source: self-made

In the example above, all links have a cost of 1 except for the one that connects routers 3 and 2, which has a cost of 3. The total cost of the path from router 3 to router 2 when using router 1 as intermediate hop is 2, the lowest among all possible options, and therefore, router 1 will be stored as next hop in router 3's routing table for the destination "router 2".

While simple, this example illustrates how routing protocols operate, and how routing preferences can be influenced by parameters such as link cost.

Some examples of IGPs are Open Shortest Path First (OSPF), Intermediate System to Intermediate System (IS-IS), or Routing Information Protocol (RIP).

2.2.3 INTER-DOMAIN ROUTING

Edge routers from different Autonomous Systems exchange routing information via the BGP protocol through the Global Routing System (GRS) [13], a distributed inter-domain routing infrastructure that enables reachability between independently administered networks. The subset of routers that maintain complete routing information without relying on a default route is commonly referred to as the Default-Free Zone (DFZ) [20].

The resources required to store these routes make it impossible for every host to do so, posing a need for a simpler approach to enable communication between each possible pair of endpoints. This is where the concept of the *default route* arises.

End hosts typically forward packets destined to external networks to a default gateway located within the local routing domain. From that point onwards, packet forwarding decisions are performed by routers based on inter-domain routing information.

When traffic needs to travel outside the AS, two approaches might be used:

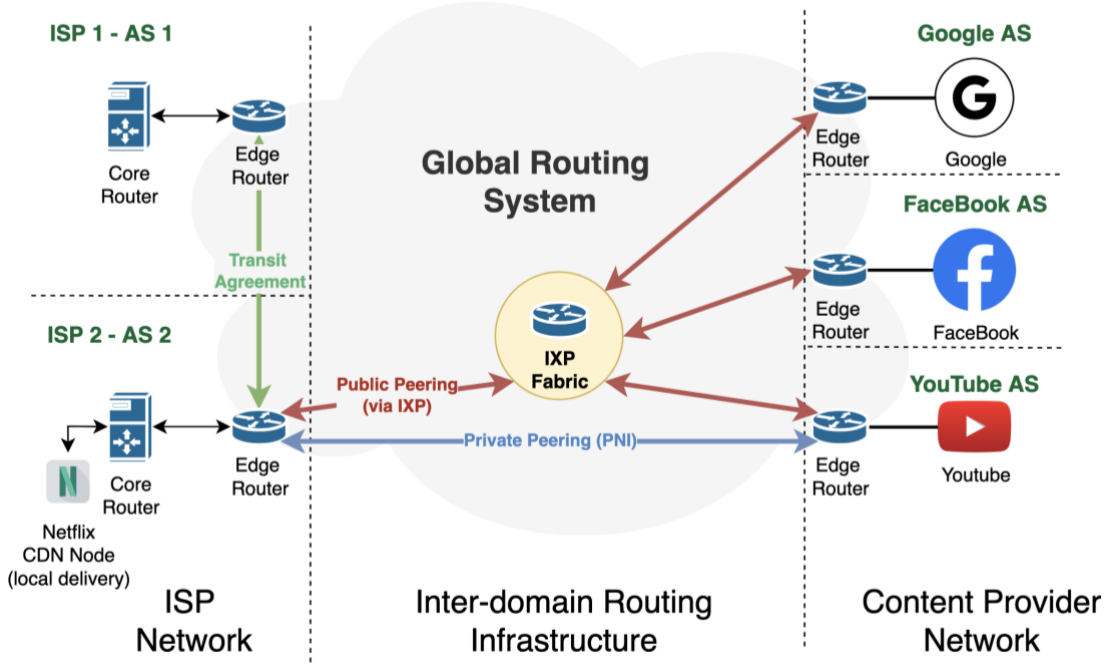


Figure 2.2.3.1.: Intra-domain Routing Architecture – Source: self-made

1. **Transit** traffic is a commercial arrangement between an ISP or a company acting as a client, and another ISP acting as the provider. The provider ISP has connectivity to the rest of the Internet, and advertises both its routes and those ones learned from third-party networks to the client via BGP, allowing it to reach external destinations beyond its routing domain.

   Transit's main drawback are the operational costs, being typically higher than its main alternative, peering, especially when the amount of forwarded traffic grows.

   In Figure 2.2.3.1., ISP 1 uses ISP 2 as a transit provider in order to obtain access to the Internet and exchange traffic with third parties.

2. **Peering** is a traffic exchange agreement between two or more Autonomous Systems that allows them to exchange traffic destined to their own networks and those of their respective customers, without providing connectivity to any third-party network. Contrary to transit, peering is an arrangement that all participants benefit from, therefore having typically lower running costs. It can be either public or private.

   Public peering enables interconnection at Internet Exchange Points (IXPs) [21]. An IXP constitutes a location where a concentration of routers is managed either by a connectivity provider or by an association of ASs, allowing traffic exchange.

   In Figure 2.2.3.1, interconnection between ISP 2 and Google will be possible thanks to the peering agreement established through the displayed IXP.

   Private peering consists of a physical or logical link that connects the premises of the two agreement participants, ensuring a dedicated path between the two Autonomous Systems, with dedicated bandwidth capacity and predictable latencies.

   In Figure 2.2.3.1, traffic between ISP 2 and YouTube will benefit from the Private Network Interconnect (PNI) through the dedicated link displayed in blue, establishing a private link between both.

In both variants, routing information is exchanged using the BGP protocol.

To further optimize traffic exchange, some content providers can install their own servers within the ISP's network, establishing an agreement known as co-location. This is particularly beneficial for services where a single copy of the data can be delivered to multiple users, such as video-on-demand.

That way, all traffic required for this service will remain inside the ISP's network, saving costs to both the content provider and the ISP itself. This deployment model is commonly used in content delivery networks (CDNs from now on) [22].

In the example diagram provided, traffic from ISP 2 clients to Netflix will remain inside the ISP's network thanks to the CDN co-location displayed.

For these reasons, big ISPs prefer establishing peering agreements and offer co-locations to CDN providers, when possible, in order to ensure enough bandwidth capacity and low latency in their communications, while also reducing their operational expenditures.

Unlike intra-domain routing protocols, which select paths based on performance-related metrics such as link cost or delay, BGP operates as a policy-driven protocol [13].

Route selection decisions are not only based on topological proximity, but also on a set of configurable path attributes that allow network operators to influence how traffic enters or leaves their network under different circumstances.

This enables Autonomous Systems to implement routing policies that reflect economic agreements, traffic engineering objectives or redundancy requirements, rather than strictly selecting the shortest available path as Interior Gateway Protocols.

2.2.4 ADDRESS INDEPENDENCE (PA VS PI)

After evaluating how routing is performed both within and across Autonomous Systems, it is important to consider the ownership of the addresses used. Internet connectivity can be provided using address space assigned either by the ISP or directly to the client:

- **Provider-Aggregatable** addressing (**PA**) [23]: the ISP provides Internet connection via addressing owned by itself. Common provisioning approaches include:
    - An addressing pool shared across multiple clients. It is common in budget domestic broadband plans, due to the IPv4 address exhaustion.
    - A single dedicated public IP address. It is common in premium domestic and small business broadband.
    - An address range. It is common in medium to large size businesses, due to the needs of such companies, and the higher economic cost that allocating IPv4 implies. It is also the most common approach in IPv6 deployments.

  Provider-Aggregatable addresses are owned by the ISP, typically being part of a larger addressing block that is announced as a whole, which generally prevents such prefixes from being advertised through another upstream provider.
- **Provider-Independent** addressing (**PI**) [24]: an IP addressing range is either owned by the client or allocated to it by a Regional Internet Registry (RIR), in both cases being independent from the addressing space owned by the ISP. One or more ISPs may provide transit services to the client, that is, a BGP session through which the customer both announces its routes and receives routes advertised by the ISP and third-party networks. It is mainly used by large companies and institutions.

While PI addressing was widely adopted in IPv4 to enable provider independence, IPv6 addressing practices have favoured Provider-Aggregatable (PA) allocations [24] in order to preserve routing scalability and limit the growth of the Default-Free Zone (DFZ). Due to the IPv6 addressing space being notably larger, it is costly in terms of resources to maintain a full table of spare routes to a great amount of small, non-aggregatable addressing blocks.

2.2.5 MULTIHOMING AND TRAFFIC ENGINEERING

Multihoming refers to the practice of connecting a network to the Internet using multiple upstream Internet Service Providers [25]. It can serve multiple purposes, such as increasing the uptime of the connection, by not relying on a single upstream connection, or increasing the available bandwidth towards the Internet by performing load-balancing techniques.

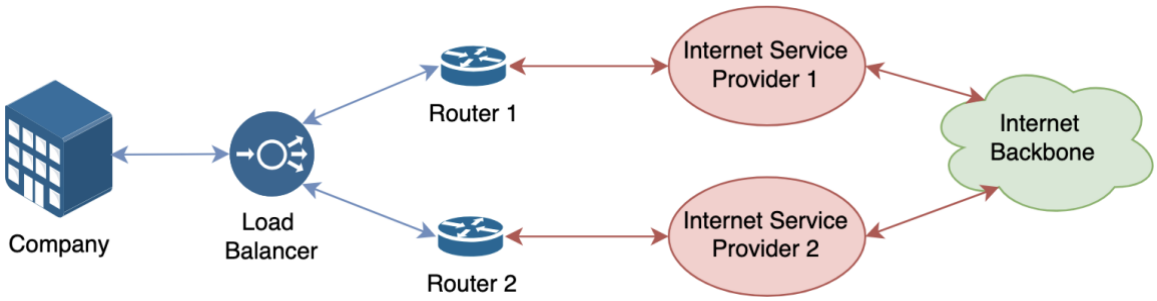


Figure 2.2.5.1.: Equal-Cost Multi-Path Multihoming – Source: self-made

Traffic engineering is the aspect of network engineering dedicated to the performance optimization of a network [26]. In the case of multihoming, it can be applied to enhance the efficiency of the network by performing selective load balancing depending on the source, the destination, the type of traffic and the state of the links at any given moment. For example, prioritizing voice traffic over a lower latency upstream, or file transfers over a higher bandwidth connection.

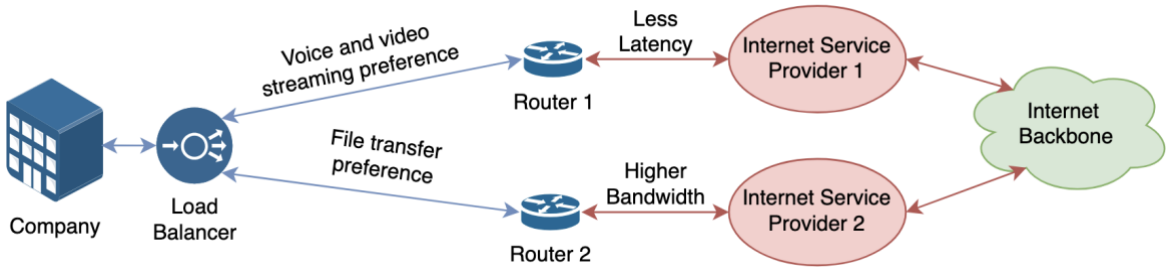


Figure 2.2.5.2.: Multihoming with policies – Source: self-made

When setting up a multihomed network environment, two different scenarios appear:

- The customer uses private addressing inside its network. In such case, each upstream will provide public connectivity to the client, and Network Address Translation (NAT) from the private addresses to the public addressing provided will be used to reach the Internet.
- The customer uses public addressing inside its network. In that case, the client will typically need a Provider-Independent addressing block, and transit agreements with various ISPs, in order to announce its routes to the Internet through the different upstreams and gain access to the Internet.

In both scenarios, outbound traffic can be distributed across multiple upstream interfaces to provide redundancy and performance improvements. However, the level of traffic engineering capabilities depends on the addressing model used, as inbound traffic control typically requires the announcement of customer-owned address space through multiple providers.

However, these approaches fail when IPv6 is brought into the equation, not because of IPv6 itself, but because of the deployment mechanisms that are typically used with it. The limiting factor is the fact that IPv6 addressing is Provider-Aggregatable [24], therefore making it complex to announce a globally routable address block through different upstreams to which such block does not belong to.

A solution to this issue could be distributing the prefixes delegated by each ISP inside the company's network. However, this setup leaves the upstream election decision in the hands of the end-user devices, making it complex for the network administrators to perform traffic engineering.

To date, the approaches to this issue are typically ad-hoc solutions, mostly implemented by the network engineers of big companies, but there is not a widely deployed architecture to approach IPv6 multihoming at scale.

2.3 HOST CONFIGURATION AND NAME RESOLUTION

While the previous sections have described how interconnected networks exchange traffic and how routing decisions are made across Autonomous Systems, end-to-end communication also depends on the correct configuration of the hosts inside the network.

In order to communicate over IP networks, hosts must be assigned addressing parameters and be able to resolve domain names into routable addresses. This section introduces the mechanisms that enable host configuration and name resolution, allowing end-user devices to obtain connectivity and interact with services across the Internet.

2.3.1 DOMAIN RESOLUTION

Domain Name System (DNS from now on) is an Application Layer protocol that allows network hosts to resolve domain names into resource records [27]. A domain name is a set of characters separated by one or more dots that uniquely identifies a node within the Domain Name System hierarchy, for example: *www.example.com*.

DNS's main purpose is to decouple a service's identity from its network's location by mapping human-readable domain names to *resource records* (typically routable IP addresses). This abstraction allows companies to restructure or upgrade their underlying network infrastructure without affecting the access of clients to their services.

It uses a server-client architecture, and resolutions are based on query-response exchanges:

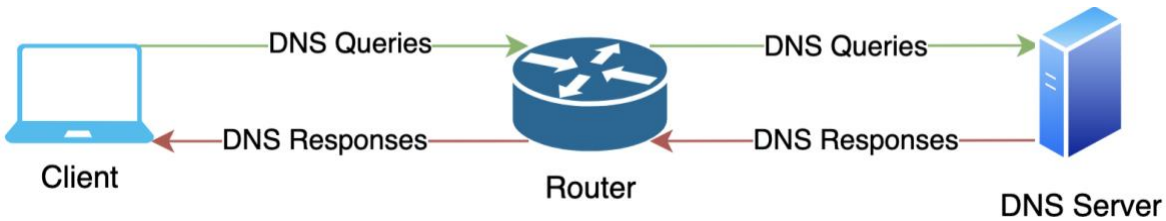


Figure 2.3.1.1.: Domain Name System – Source: self-made

DNS operates as a globally distributed hierarchical database [27][28] in which authoritative servers are responsible for specific portions of the namespace. Rather than maintaining a globally synchronized dataset, DNS resolution is performed iteratively by querying different levels of authority, with intermediate results being cached to improve efficiency.

Hosts typically keep a local DNS cache table, and query a remote server when they need to resolve a domain name which is not present in the cache, storing it for future use.

DNS entries are called records, classified by the type of mapping they perform. In order not to expand beyond the scope of this project, we will focus on two of them:

- "**A**" records map a domain name to an IPv4 address
- "**AAAA**" records map a domain name to an IPv6 address [29]

Aside from its basic functionality, DNS also has further uses, such as:

- Enforcing content filtering policies
- Providing geographically-aware domain resolution
- Enabling server load-balancing for a service accessed via a domain name
- **Translating IPv4 (A) to IPv6 (AAAA) records via DNS64**

DNS64 is a DNS-based mechanism [17][30] that allows communication between IPv6-only clients and IPv4-only services by synthesizing IPv6 addresses from IPv4 records. This synthesized address can then be used by the client to establish communication through a NAT64 gateway, which translates the traffic between IPv6 and IPv4 networks.

The configuration of recursive DNS resolvers on a host can be done manually, or automatically, as described in the next subsection.

2.3.2 ADDRESS ASSIGNMENT

The network configuration of a host is primarily integrated by four parameters:

- The **IP address** of the host, used as source address for outgoing packets and as destination address for incoming ones. It identifies the host's network interface.
- The **subnet mask**, which defines the address range of the network the host is connected to. It identifies the network.
- The IP address of the **default gateway**, used for traffic that needs to travel outside the network the host is connected to. It identifies the default next-hop.
- The **DNS** addresses, used for queries in order to resolve domain names into routable IP addresses. They identify the DNS servers.

These parameters can be configured manually in the network configuration of the host itself, or automatically via autoconfiguration protocols inside the TCP/IP protocol suite.

2.3.2.1 IPv4 Network Configuration: DHCP

Dynamic Host Configuration Protocol (DHCP) [31] is an Application Layer protocol that uses a server-client architecture to configure a host's network settings in a four-step process:

- **Discover**: the client broadcasts a discover message into the network
- **Offer**: the DHCP server receives the message and sends an offer that includes:
  - IP proposal (proposed address + network's subnet mask)
  - The router announces itself as default gateway
  - DNS server(s)
  - Lease time
- **Request**: the client accepts the offer
- **Acknowledgement**: the server confirms the new DHCP assignment and the IP address is reserved during the lease time, after which a DHCP renewal will be needed

DHCP can be used both at an ISP level to assign the IP addressing to the client, and inside the customer network to assign public or private IP addresses to the end hosts.

2.3.2.2 IPv6 Network Configuration: SLAAC

Stateless Address Autoconfiguration (SLAAC) [32] is an address configuration protocol that allows hosts to configure their network settings themselves, instead of relying on a server.

When a host joins a network, the local router sends a Router Advertisement (RA) message, which contains the following information:

- The network prefix (network address + prefix length)
- The default gateway
- The DNS servers (optionally)

The host combines the network prefix with an interface identifier (typically a randomly generated number) and generates its IP address. Before starting using such address, the host sends a Neighbor Solicitation (NS) message. If another host responds, it means that the address was already being used, and therefore it needs to generate a new address.

2.3.2.3 IPv6 Network Configuration: DHCPv6

DHCPv6 [33] follows similar working principles to DHCP, but using IPv6 addresses.

Its main advantage over SLAAC is statefulness (keeping record of the assigned addressing), allowing for a more organized deployment, at the expense of the requirement for a server to enable the assignment and storage of addresses. The DHCPv6-PD (Prefix Delegation) mode allows the server to assign network prefixes instead of single IPv6 addresses.

DHCPv6-PD is commonly used by ISPs to delegate a network prefix to the customer, while end hosts inside the network typically use SLAAC to obtain their address configuration. Mobile deployments using 464XLAT typically rely on SLAAC for address configuration.

Chapter 3. STATE OF THE ART

This chapter focusses on explaining the existing solutions used to perform IPv6 multihoming, both on paper and at production environments, reviewing their working principles and understanding their real-world limitations.

Getting a brief of the current state of the art gives a key insight needed for understanding the operational requirements and limitations of the final developed system.

3.1 ROUTING-BASED MULTIHOMING

3.1.1 PROVIDER-INDEPENDENT ADDRESSING AND BGP

As reviewed in section 2.2.4, Provider-Independent addressing is allocated directly to customers by a competent institution, the RIR (Regional Internet Registry) [24].

These blocks are not associated with any ISP, and therefore their announcement to the Internet can be done through different ISPs, at the expense of an increase in the number of routes present in the Default-Free Zone (DFZ). These route announcements are made using the Border Gateway Protocol (BGP) [13], as explained in section 2.2.3.

3.1.2 OPERATIONAL CHARACTERISTICS

The use of BGP with Provider-Independent addressing provides several advantages when setting up a multihomed network, such as:

- **Complete control of both outbound and inbound traffic**, as routes are announced directly by the customer and not by the ISP
- **ISP independence**, being able to switch upstreams in the future without altering the addressing used by the company
- **Universal compatibility** with ISPs and network equipment

3.1.3 LIMITATIONS

Although BGP-based multihoming provides significant advantages, it comes with trade-offs that need to be considered.

First, acquiring an allocation for an IP address block is expensive, particularly for IPv4, due to the scarcity of such addresses. Aside from the cost, addressing allocation typically comes with administrative barriers, leading to a tedious and sometimes long process.

Second, the use of Provider-Independent addressing requires to establish a peering or transit agreement with ISPs for them to advertise the customer's routes to the Internet, as well as advertising third-party routes to the customer. These kinds of connectivity agreements come at a higher price than standard residential or small business broadband plans, sometimes even orders of magnitude higher.

Finally, if we take into account that the use of Provider-Independent addressing in IPv6 networks is discouraged as a common practice because of its impact in the DFZ growth [20], BGP IPv6 multihoming is relegated to companies big enough to be able to justify the use of a PI addressing block, and at the same time financial capacity for establishing agreements with ISPs for such block to be announced to the Internet.

These limitations have a great impact in small businesses, which might require multihoming for their operations but cannot afford the required connectivity agreements nor justify the use of Provider-Independent IPv6 addressing.

Therefore, BGP-based multihoming is fully functional, but not widely available nor commonly deployed in IPv6 network environments.

3.2 HOST-BASED MULTIHOMING

While inter-domain multihoming based on Provider-Independent addressing and BGP provides full control over routing decisions, its operational and economic barriers make it unsuitable for many small and medium-sized deployments.

As an alternative, several approaches move the decision of which upstream path to use from the routing system to the end hosts themselves. These mechanisms are commonly referred to as host-based multihoming.

3.2.1 MULTIPLE PREFIX ADVERTISEMENT IN LAN

One approach to host-based multihoming is advertising multiple Provider-Aggregatable prefixes delegated by the ISPs in the Local Area Network (LAN) through Router Advertisements (RA) [32]. Hosts will, therefore, automatically configure multiple IPv6 addresses from different upstreams using SLAAC, as explained in section 2.3.2.2.

When hosts try to access the Internet, they will use Source Address Selection, defined in RFC 6724 [40]. The algorithm will try to choose:

1. An address compatible with the destination (IPv6-IPv6, IPv4-IPv4)
2. An address with the same scope (public-public, private-private)
3. An address with the most appropriate prefix (e.g. within the same prefix if possible)

In practice, this approach severely limits the control that network administrators can apply to traffic flow, as the decisions become distributed across end hosts, a circumstance completely unwanted in enterprise environments.

Furthermore, if one of the upstreams is preferred to be used only for failover purposes (e.g. a 5G or satellite link), its prefix should not be advertised until the primary upstream fails, otherwise it will be used by hosts during normal operation, completely unaware of the failover-only nature of the connection.

3.2.2 TRANSPORT-LAYER APPROACHES

Another approach used in host-based multihoming is relying on the Transport Layer to handle the load balancing and/or failover of the different links.

Some Transport Layer protocols that support the use of multiple parallel connections over different Internet circuits at once are MPTCP (Multi-Path Transport Control Protocol) [41] and Multipath QUIC (Quick UDP Internet Connections) variants [42].

This setup brings significant advantages such as:

- **Resiliency**, as the failure of a subflow should not kill the connection
- **Bandwidth aggregation**, by using multiple upstreams at a time
- **Mobility**, as the host can switch between networks without breaking the connection

Relying on a Layer 4 protocol, however, requires support at both endpoints (client and server) for the multihoming to work, and therefore, it is not a substitute for IP-level multihoming, as a substantial number of services rely on traditional TCP or UDP connections. In addition, network administrators still lack control of the traffic flow.

3.2.3 IDENTIFIER/LOCATOR SEPARATION AT HOST LEVEL

In the IP protocol, the IP address traditionally serves two different purposes:

- **Identifier**: identifies the host participating in the communication
- **Locator**: identifies the host's position within the network

When host-based multihoming is introduced, each host may possess multiple IP addresses associated with different upstream providers. In this situation, the address selected as the source of an outgoing packet determines which upstream path the traffic will follow.

Site Multihoming by IPv6 Intermediation (SHIM6), defined in RFCs 5533-5535 [43], aims to address this issue by separating the identifier and locator roles within IP addresses.

SHIM6 introduces a software layer between the network and transport layers that allows hosts to maintain a stable identifier while dynamically switching between multiple locator addresses associated to the different upstream providers.

During the establishment of a communication, hosts exchange sets of available locator addresses, creating a context that allows packets to be redirected through alternative paths if a connection failure is detected, enabling transparent failover.

This scheme is architecturally elegant, as it allows multihoming without requiring Provider-Independent addressing, BGP routing, or Network Address Translation.

Despite its advantages, SHIM6's deployment in production environments is very limited. Its operation requires support at both endpoints and introduces additional complexity in the host's networking stack. Furthermore, dynamic changes in locator addresses may interact poorly with network middleboxes such as firewalls and intrusion detection systems.

As a result, although the separation of identifier and locator is a solid theoretical approach to multihoming, host-layer implementations such as SHIM6 have not seen a wide adoption in enterprise networks.

Other academic proposals have explored similar locator/identifier separation models at the architectural level. Examples include the Identifier-Locator Network Protocol (ILNP) [44] and the Locator/Identifier Separation Protocol (LISP) [45], which introduce addressing semantics or encapsulation mechanisms to decouple host identity from network location.

While these approaches offer a theoretically robust solution to multihoming and mobility, their deployment requires substantial changes to the Internet architecture or to network infrastructure. Consequently, their adoption has remained largely limited to research and experimental environments.

3.3 EDGE-BASED MULTIHOMING

On the one hand, BGP multihoming provides full control over routing decisions, but its operational and economic barriers make it unsuitable for smaller deployments. On the other hand, host-based multihoming proposes a conceptually sound approach for such environments, but its viability is limited by the lack of support for the required protocols.

Edge-based multihoming arises as a middle ground proposal between the previous approaches, by delegating the traffic path decision to the customer's edge router. This model is transparent to end hosts and works without the need for Provider-Independent addressing, while at the same time allowing for a certain level of traffic management.

3.3.1 SOURCE-BASED ROUTING AND POLICY-BASED ROUTING

When outgoing traffic from end hosts reach the company's edge router, different techniques can be applied to determine the upstream path packets will follow:

- **Equal Cost Multipath** (ECMP) [46]: traffic flows are evenly distributed across the multiple available circuits
- **Source-Based Routing** (SBR): the routing decision will be based on the source of the traffic (e.g. in a company, traffic from the sales department through ISP 1)
- **Policy-Based Routing** (PBR): routing decision will be based on policies that can be configured by the network administrator (e.g. HTTP traffic through ISP 1)

In all cases, failover policies can be configured in such a way that traffic will be able to go through another ISP in the event of a loss of service in one of the circuits.

Source-Based and Policy-Based Routing may rely on Virtual Routing and Forwarding (VRF) [47]. VRF is a network virtualization technology that enables multiple instances of a routing table to coexist within a single physical router, used for the different forwarding contexts depending on the traffic's source (SBR) and the configured policies (PBR).

3.3.2 PREFIX TRANSLATION MECHANISMS

In order to move traffic path decisions from the end hosts to the company router, multihoming needs to be performed in a transparent way for such hosts, meaning that a single IPv6 prefix should be announced to the Local Area Network.

The setup consists of announcing to the Local Area Network (LAN) a locally consistent prefix, and performing prefix translation to the addressing delegated by the different ISPs. A common practice is announcing non-globally routable prefixes, whose addresses are called ULAs (Unique Local Addresses) [48], conceptually similar to IPv4 private addresses.

The NPTv6 (Network Prefix Translation version 6) protocol, defined in RFC 6296 [49], allows to perform this translation, by mapping one IPv6 prefix to another. Its main differences with IPv4 Network Address Translation (NAT) are:

- **1:1-only operation**: while traditional NAT allows to share a single public IPv4 address across multiple hosts with private addressing (1:M), NPTv6 only works in 1:1 mode, meaning that a whole IPv6 prefix is translated into another prefix.
- **Statelessness**: the 1:M nature of IPv4 NAT requires the router performing the translation to keep state of the connections' addresses and ports. As NPTv6 only works in 1:1 mode, keeping state of such connections is not necessary.

In addition, NPTv6 also performs checksum-neutral prefix translation in order not to break connection-oriented transport layer protocols such as TCP.

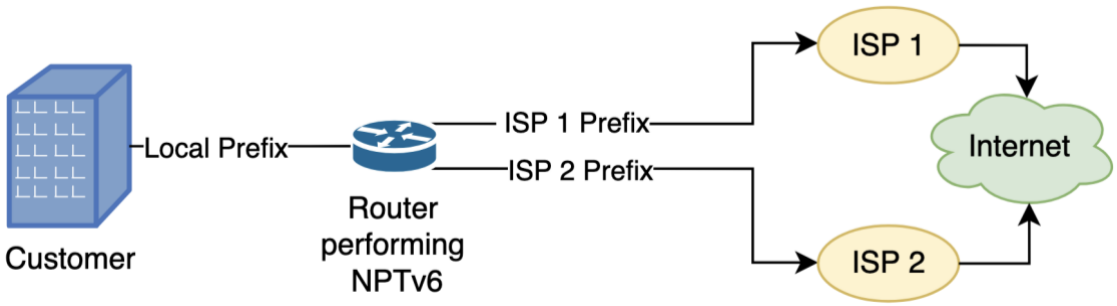


Figure 3.3.2.1.: NPTv6 – Source: self-made

3.3.3 OPERATIONAL CAPABILITIES & LIMITATIONS

Edge-based IPv6 multihoming brings numerous advantages from a customer's standpoint, such as allowing the implementation of outbound traffic engineering and failover.

Unlike BGP-based multihoming, it does not require Provider-Independent addressing, reducing the operational costs, and contrary to host-based multihoming, it does not require host support, allowing for a rapid and technically viable deployment.

Despite its evident advantages over BGP-based and host-based multihoming, some problems can be encountered when deploying edge-based multihoming with NPTv6.

First, contrary to IPv4 NAT, NPTv6 is stateless, meaning that when a traffic flow is created, no binding entry is stored in the router. The lack of flow persistence allows the load balancer to switch the outbound circuit of a traffic flow at any given moment, potentially breaking connection-oriented transport layer protocols such as TCP. This issue needs to be solved via explicit configuration with techniques such as policy routing, connection tracking or flow hashing.

Second, a common approach when setting up NPTv6 multihoming is using Unique Local Addresses internally. In such case, inbound connectivity requires explicit coordination between DNS records and the prefix translation rules configured at the edge router. In addition, ensuring consistent upstream path selection requires careful integration between prefix translation and routing policies, as hosts are unaware of the multihomed topology.

Finally, the lack of a de-facto approach to solve the aforementioned issues makes the implementation of NPTv6 a complex practice, relegated to scenarios where the use of IPv6 multihoming is mandatory and/or the customer possesses enough resources, both economic and technical, to successfully configure such a setup.

Therefore, the use of NPTv6 to perform IPv6 multihoming is viable on paper, but complex to execute in practice with the current state of the art.

3.4 STATE OF THE ART'S SUMMARY

3.4.1 COMPARATIVE SUMMARY TABLE

Approach	Mechanism	Requires PI/BGP	Requires Host Support	Traffic Engineering Capabilities	Deployment Complexity	Real-world adoption
Routing-based	BGP + PI addressing	Yes	No	Full	High	Limited to large-scale networks
Host-based	Multiple prefixes + RFC6724	No	Yes	Low	Medium	Limited
Transport-layer	MTCP / Multipath QUIC	No	Yes	Low	Medium	Limited
Locator/Identifier separation	SHIM6	No	Yes	Medium	High	Very Limited
Edge-based	NPTv6 + policy routing	No	No	Medium	Medium	Growing but fragmented

Table 3.4.1.1.: State of the Art's Comparison – Source: self-made

Table 3.4.1.1. summarizes the main characteristics of the approaches described in this chapter. Each solution provides a different trade-off between operational complexity, control over traffic flows and deployment requirements.

Routing-based multihoming offers the highest degree of control, but requires Provider-Independent addressing and BGP connectivity, which limits its applicability to large organizations. Host-based and transport-layer approaches remove the dependency on routing infrastructure but require support at end hosts, reducing their practical deployability in heterogeneous environments.

Edge-based approaches represent a compromise between these models by allowing routing decisions to be taken at the customer edge without requiring host modifications or global routing participation.

3.4.2 IDENTIFIED OPERATIONAL GAP

Although edge-based multihoming using mechanisms such as NPTv6 provides a technically viable alternative to BGP-based multihoming, its practical deployment remains complex.

While the underlying blocks (prefix translation, policy-based routing and source address selection) are already available within the TCP/IP protocol stack, they do not constitute a complete operational solution by themselves. Current implementations typically rely on ad-hoc configurations that couple these mechanisms in a platform-dependent manner, requiring significant networking expertise to design, deploy and maintain.

In addition, NPTv6 introduces several operational constraints that are not trivial to handle in practice, including the requirement for path symmetry, limitations in inbound reachability and the lack of native support for dynamic traffic steering. As a stateless mechanism, NPTv6 does not inherently provide flow awareness, which becomes a critical limitation when attempting to ensure session persistence during failover or to implement load balancing.

There is also a clear absence of a control plane capable of coordinating multihoming decisions. Existing approaches do not define how to monitor link health, how to react to failures, or how to enforce traffic engineering policies in a consistent and automated manner, so essential aspects such as failover behavior, path selection and load distribution are often implemented through external scripts or manual intervention, increasing system complexity.

This lack of integration contrasts with BGP-based solutions, where routing decisions, policy enforcement and failure recovery are inherently part of a unified framework. This gap becomes particularly evident in environments such as small and medium-sized networks or prosumer deployments where BGP is not feasible due to operational or economic constraints.

Therefore, despite the existence of the necessary protocol mechanisms, there is currently no widely adopted operational model that integrates them into a coherent, automated and reproducible framework. This absence of an integrated approach to IPv6 multihoming at the network edge constitutes the main gap identified in this work and directly motivates the design of the framework proposed in this thesis.

Chapter 4. WORK'S DEFINITION

This chapter introduces the definition and the development framework of NEPTUNE: Network Prefix Translation Unified Environment. Building upon the limitations identified in the state-of-the-art section, this work's definition serves as a rationale behind the design decisions and objectives that drive the project. In addition, it describes the methodology of the conception, implementation and validation of the proposed system.

4.1 JUSTIFICATION

This section is focused on describing the need for the work developed in this project, driven by the need for IPv6 multihoming in modern networks and the deficiencies of the current state of the art proposals for such purpose.

4.1.1 MOTIVATION FOR A NEW APPROACH

This project's motivation can be split into two main points: ***why is IPv6 multihoming necessary?*** and ***what do current proposals lack?***:

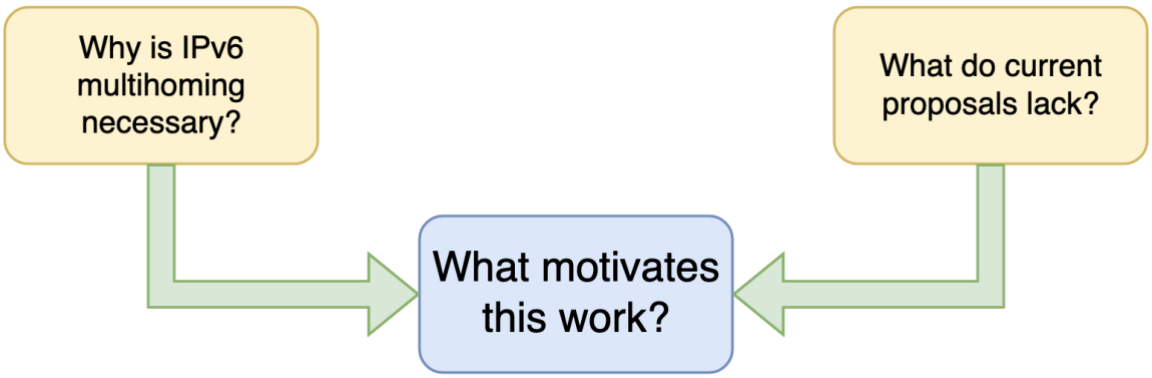


Figure 4.1.1.1.: Work's Motivation – Source: self-made

4.1.1.1 Why is IPv6 Multihoming Necessary

While the relative effectiveness of Network Address Translation, particularly in residential broadband plans, caused IPv6 adoption to remain modest during a long period of time, recent years have shown a consistent growth in the IPv6 adoption, reaching almost 50% as of today.

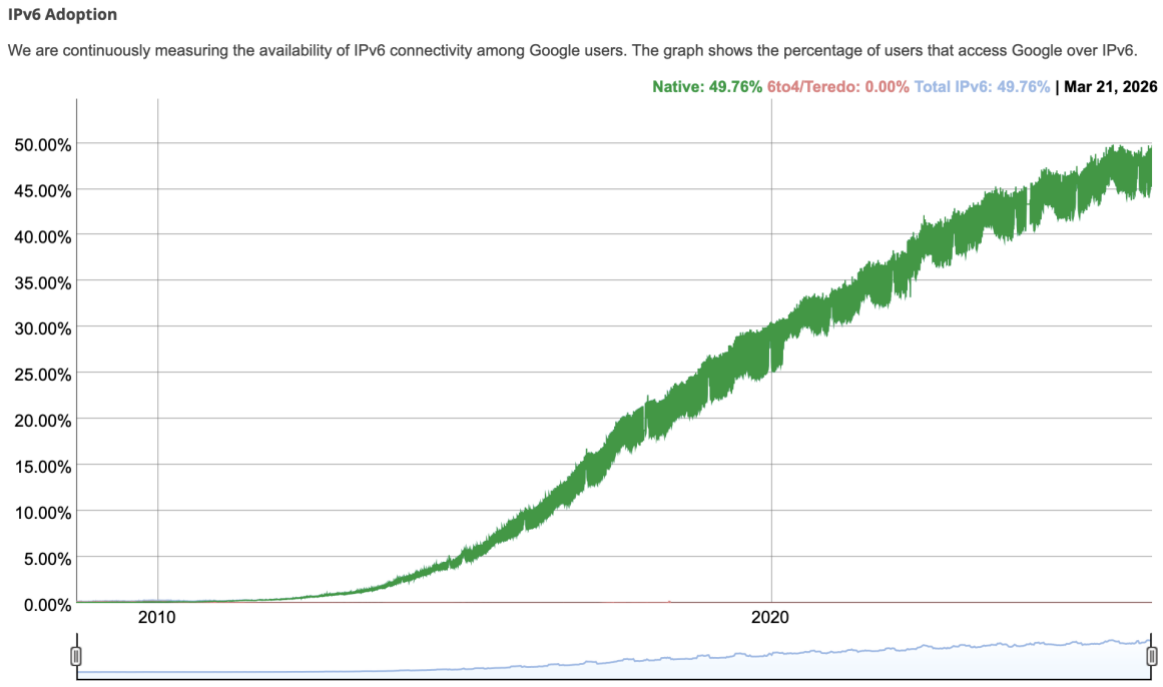


Figure 4.1.1.1.1.: Google's IPv6 adoption – Source: google.com

This transition has been performed in a way that IPv4 connectivity has been preserved for backwards compatibility purposes, however, a complete transition requires every scenario currently supported in IPv4 networks to be also supported by IPv6.

Multihoming is a widely used technique, not only in large companies but in any business network whose operations heavily rely on Internet connectivity, and therefore, poses itself as a vital environment to be supported in IPv6 networks in order to allow for a complete transition without relying on IPv4.

4.1.1.2 What do Current Proposals Lack?

As reviewed in Chapter 3, a wide range of approaches to IPv6 multihoming have been proposed, including routing-based, host-based, and edge-based solutions. However, none of these approaches fully satisfies the operational requirements of typical business networks.

Routing-based solutions, such as those relying on BGP and Provider-Independent (PI) addressing, offer the highest degree of control and flexibility. However, they introduce significant complexity, require cooperation from ISPs, and are generally unsuitable for small and medium-sized networks due to operational and economic constraints.

Host-based approaches, including multi-prefix advertisement and transport-layer mechanisms, delegate traffic path decisions to end hosts. While conceptually elegant, they suffer from limited adoption, lack of determinism, and reduced control from the network operator's perspective, making them impractical in real-world deployments.

Edge-based approaches, such as source-based routing or prefix translation techniques, provide a more feasible alternative for deployment at the network boundary. However, they often require manual configuration, lack dynamic adaptability, and do not provide a unified framework for failover and traffic engineering.

This situation leaves a clear gap between highly capable but impractical solutions and deployable but limited ones. In particular, there is a lack of a solution that is:

- Deployable in typical business networks
- Independent from the ISP
- Transparent to end hosts
- Capable of providing both failover and traffic engineering capabilities

Therefore, it can be concluded that IPv6 multihoming remains an open and relevant problem, where existing solutions fail to simultaneously achieve practicality, flexibility and both operational and cost effectiveness.

4.1.2 DESIGN PHILOSOPHY OF NEPTUNE

Based on the limitations of existing approaches, NEPTUNE is designed as a practical solution that can be easily deployed in both enterprise and residential networks. Instead of introducing new protocols, the goal is to make use of existing ones in a more coordinated and automated way. The design of NEPTUNE is driven by several key principles:

Edge-Based Control

All the decision-making logic is placed at the network edge. This allows the system to operate without requiring agreements with the ISP or the use of BGP, which significantly simplifies deployment while still giving control over how traffic is handled.

Transparency to End Hosts

One of the main goals is to avoid explicit support requirement end devices. From the point of view of the hosts, the network is not multihomed, which makes the solution easier to deploy and compatible with existing equipment.

Adaptability and Automation

Unlike many existing edge-based solutions, NEPTUNE is not intended to be static. The system monitors the state of the links and dynamically adjusts routing decisions. This makes it possible to implement failover and basic traffic engineering without manual configuration.

Provisioning-Agnostic Operation

NEPTUNE is designed to work in IPv6 environments, and its design considers that networks often use different address provisioning methods such as DS-Lite or 464XLAT. The goal is to remain flexible enough to work in these mixed scenarios. In addition, support for IPv4 multihoming will also be considered to provide a seamless configuration experience.

Overall, NEPTUNE's target is to provide a solution that balances practicality and functionality. It does not try to replace existing approaches, but rather to offer an alternative that works in scenarios where other solutions are either too complex or not applicable.

4.1.3 TARGET USE CASES

1. **Networks without BGP capabilities**, such as small to medium-sized businesses or advanced residential users, which might benefit from the use of multihoming, but cannot afford to use BGP due to the associated the economic and operational costs.

2. **Heterogeneous Access Environments**, where multiple ISPs provide connectivity using different provisioning methods, with different bandwidth, latencies and other connection parameters, abstracting a great part of the configuration from the network administrator, for example:

	Provider	ISP 1	ISP 2	ISP 3
Connection Characteristics	**Physical Link**	Optic Fiber	Optic Fiber	5G
	Bandwidth (download/upload)	8Gbps/ 8Gbps	2Gbps/ 2Gbps	1Gbps/ 150Mbps
	Average latency	18ms	4ms	35ms
	IP provisioning	Dual Stack	Dual Stack Lite	464XLAT
Traffic Engineering	**Upstream use**	Load Balancing	Load Balancing	Failover Only
	Balancing weight	0.8	0.2	0
	Protocol priority	FTP, SSH	VoIP, RTP	-

Table 4.1.3.1.: Heterogeneous Access Environment – Source: self-made

3. **System Integrators and Equipment Manufacturers**, willing to offer enterprise-grade gateways with built-in IPv6 multihoming capabilities via their own NEPTUNE implementation as a value proposition for potential customers, or introducing the functionality to existing equipment via software updates.

4.2 OBJECTIVES

This chapter is focused on defining the purpose of the work, understanding what this project aims to achieve and how the results obtained from the developed model will be measured and validated in order to verify whether the objectives have been met.

4.2.1 GENERAL OBJECTIVE

The objective of this work is developing an operative framework to perform edge-based IPv6 multihoming in a practical and easy-to-deploy way, without the need for BGP, Provider Independent (PI) addressing nor explicit support on the end hosts, while at the same time providing control over outbound traffic.

4.2.2 SPECIFIC OBJECTIVES

1. **Functional Objectives**
    - **a.** Monitoring the state of the links, allowing the system to make forwarding decisions based on such state at any given moment
    - **b.** Performing traffic engineering to outbound traffic via source-based routing and/or policy-based routing
    - **c.** Implementing automatic failover to avoid losing connectivity in the event of a link failing, and going back to the original state when such link is recovered
    - **d.** Applying weighted load balancing in order to aggregate the available capacity of multiple links
2. **Technical Objectives**
    - **a.** Developing a NEPTUNE controller that will act as the intermediary between the network administrator and the router's control plane
    - **b.** Creating a Linux-based virtual networking environment to test the controller and simulate different production-like environments
    - **c.** Setting up a telemetry module to gather data about the controller's behavior in order to analyze it and validate if the results are the expected ones

3. Validation Objectives

7. Design virtual scenarios that simulate:

i. Normal operative conditions with different types of traffic
ii. Circuit failure events
iii. Circuit recovery events

b. Measure:

i. Failover and recovery times
ii. Traffic continuity
iii. Added overhead and latency

c. Verify:

i. Route behavior

8. Prefix management

4. Design Objectives

a. Allow for ISP independence, avoiding the need to use Provider Independent (PI) addressing or BGP routing

b. Provide a host transparent solution, not requiring end-devices to support any additional protocol, therefore streamlining its deployment

c. Compatibility with different address provisioning methods, such as Dual Stack, Dual Stack Lite or 464XLAT

4.3 METHODOLOGY

This section will cover the approach used to develop the project, defining the features to implement, the constraints to meet and the process for carrying it out.

4.3.1 OVERALL APPROACH

This project is conceived as a process combining design, implementation and validation of a fully functional system. The goal is not only to propose a theoretical solution to the IPv6 multihoming matter, but also to show that such solution can be implemented.

After a conceptual analysis of the problem, the current state of the art solutions and their limitations in production environments, we define NEPTUNE's architecture based on the results of the previous analysis, and the requirements that stemmed from it.

Finally, the system is validated in a laboratory network, emulating a real production environment, and allowing to simulate events such as circuit failure and recovery while different types of traffic are flowing through the links, being able to measure and evaluate the behavior and performance of the system under such circumstances.

4.3.2 SYSTEM DESIGN METHODOLOGY

The design of the project will be carried out progressively, starting with the functional and technical requirements defined in previous sections, and developing an architecture that enables the implementation of the model in a real production environment.

First, the required functionalities and operational constraints are identified, that is, defining what the system is expected to do, under which conditions and what limitations need to be considered. This phase conditions future decisions taken during the development.

After that, an architectural model is proposed, defining the main components of the architecture, what their roles are, how they interact with each other, and what will the traffic management strategy used by the system be.

4.3.2.1 Requirements Definition

The proposed system is required to implement **IPv6 Multihoming** while offering:

- **Automatic failover and recovery**, minimizing downtime in the event of circuit failures and switching back when such circuits regain connectivity
- **Weighted load balancing** with flow stickiness, enhancing the total available bandwidth without breaking connection-oriented transport protocols such as TCP
- **Traffic Engineering capabilities**, allowing network administrators to set up custom routing policies for outbound traffic

Such functionalities need to be achieved within the following constraints:

- **No requirement for host support**, to enable for rapid and technically achievable mass deployments in production environments
- **No requirement for ISP agreements nor BGP**, making the product compatible with commercial broadband plans, lowering the total cost for the end user
- **Compatibility with IPv6 deployment methods** and transition mechanisms, such as Dual Stack, Dual Stack Lite and 464XLAT

4.3.2.2 Architectural Design

NEPTUNE will constitute both a deployment model and an element of such deployment, the NEPTUNE router, whose logic will be described in following sections. A basic setup will be integrated by 3 key components:

- **The NEPTUNE router**
- **At least two ISPs**, and their associated WAN interfaces
- **The Local Area Network(s)**, in which clients are located

The NEPTUNE router will monitor the WAN interfaces and perform traffic forwarding decisions based on the configured policies and the state of the links at any given moment.

4.3.2.3 Traffic Handling Strategy

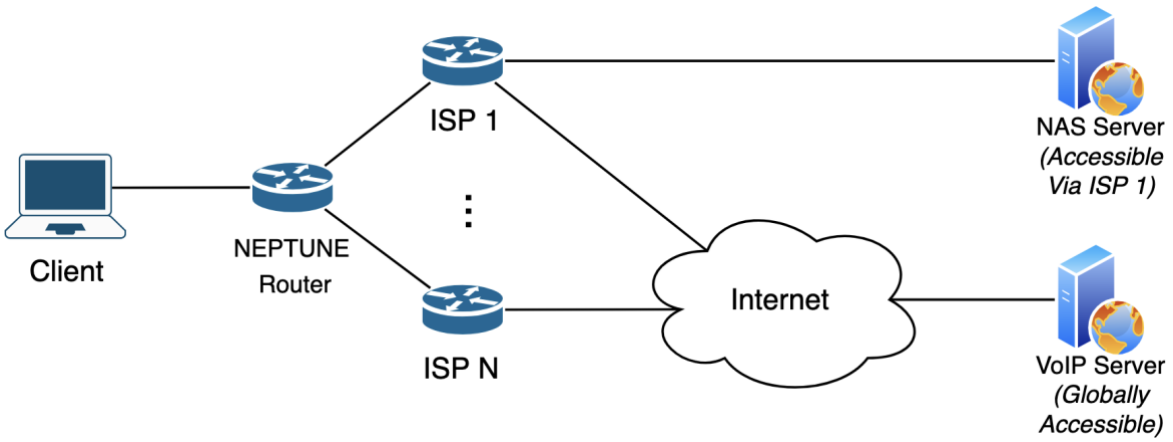


Figure 4.3.2.3.1.: Basic Neptune Setup – Source: self-made

Different situations might require network administrators to have the ability to configure routing policies with different levels of strictness, as shown in Figure 4.3.2.3.1, where traffic towards the VoIP server will preferably use the link from the ISP that offers lower latency, but can be forwarded through other links if the primary one fails.

However, the shown NAS server is a service strictly accessible through ISP 1, and therefore, trying to re-route traffic in the event of a circuit failure from ISP 1 would have no effect.

All traffic needs to be translated, as the system is targeted at being used with commercially available broadband plans, which make use of Provider Aggregatable (PA) addressing, therefore making it necessary to implement a mechanism such as NPTv6.

Traffic not affected by any of the routing policies will be forwarded through the link defined as primary, or split across various links if load balancing is used. If the later, a deterministic strategy such as 5-tuple hashing needs to be implemented for traffic flows not to switch circuits, potentially breaking connection-oriented transport protocols such as TCP, as the change in circuit would also imply a change in the source IP address.

4.3.3 IMPLEMENTATION METHODOLOGY

This section describes the approach used to translate the proposed model into a functional and validatable deployment inside a virtual network environment.

4.3.3.1 Technology Stack

The base system will use **Linux's kernel networking stack**, which natively includes IPv6 compatibility, policy routing and support for multiple routing tables. On top of it, **iproute2** enables the creation of such routing tables, policies, together with interfaces and addressing.

The different network elements (routers, clients, services) needed to simulate the setup will not require the use of different computers. **Linux's network namespaces** allow to virtualize multiple, independent networking stacks within the same virtual or physical Linux machine.

The data plane will be based on **iptables**, allowing to mark traffic flows, integrating with policy routing and supporting NPTv6. Such data plane will be deployed using bash scripts, which at the same time will be created and executed using a Python **generator** script.

The control plane will also be based on a Python script, monitoring the environment and making real-time decisions based on its configuration and the current state of the network.

Both python scripts will be fed by a **config.yaml file**, used for storing the generator setup and the controller policies and configuration, enabling modularity for both simulation and future upgrades and expansion.

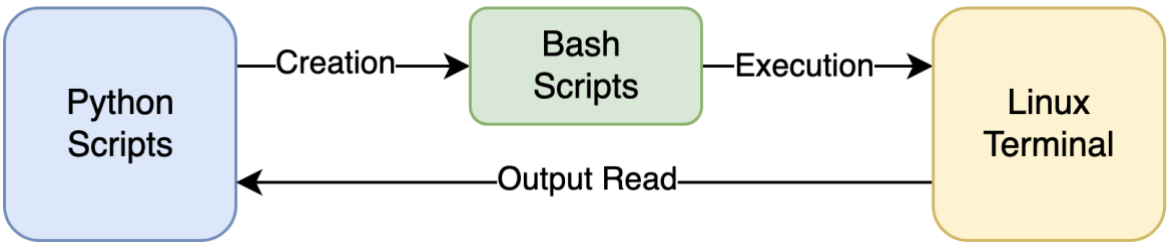


Figure 4.3.3.1.1.: Python & Linux Basic Interaction – Source: self-made

4.3.3.2 Laboratory Environment

Namespaces are the core technology behind containerization, like Docker and LXC, allowing to create isolated environments without creating independent virtual machines.

In our case, we will be using network namespaces, as the use will be limited to recreating network entities with independent TCP/IP stacks that simulate real hosts inside a network:

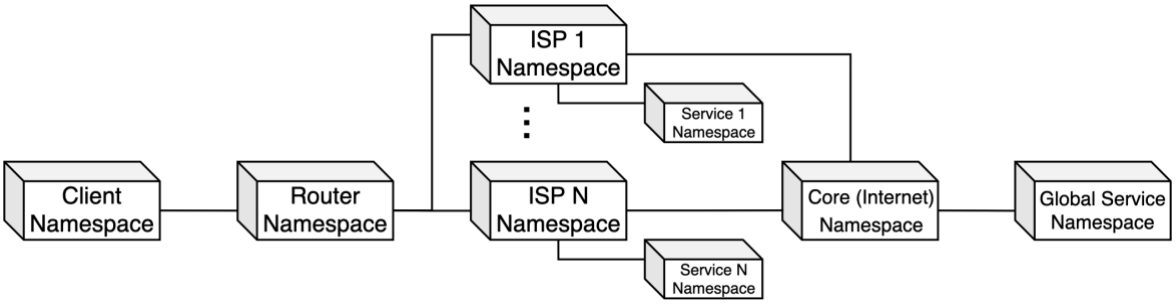


Figure 4.3.3.2.1.: Namespace Diagram Example – Source: self-made

Performance-wise aside, the data plane of any TCP/IP network host is practically identical in behavior, meaning that, changing the configuration at the control plane level, we can use make namespaces act as clients, routers, servers or any network entity required.

Thanks to the layered approach of the TCP/IP protocol stack, the network configuration will be agnostic to both data and physical layers, meaning that the design will be valid for all kinds of broadband access, despite it being based on radio, copper or fiber technologies.

In addition, most IPv6 transition mechanisms in use today, such as Dual Stack, DS Lite and 464XLAT, provide native IPv6 connectivity, meaning that adding an address provisioning layer would suffice to make the setup compatible with them. In the simulation environment, addressing will be statically assigned via the configuration file.

Therefore, the proposed simulation environment constitutes a robust model applicable to the vast majority of IPv6 deployments without requiring extensive adaptions.

4.3.3.3 Control Logic Implementation

On top of the controller and the generator, there will be an additional script, the **runner**, which enables the user to interact with both of them and execute different sets of scripts:

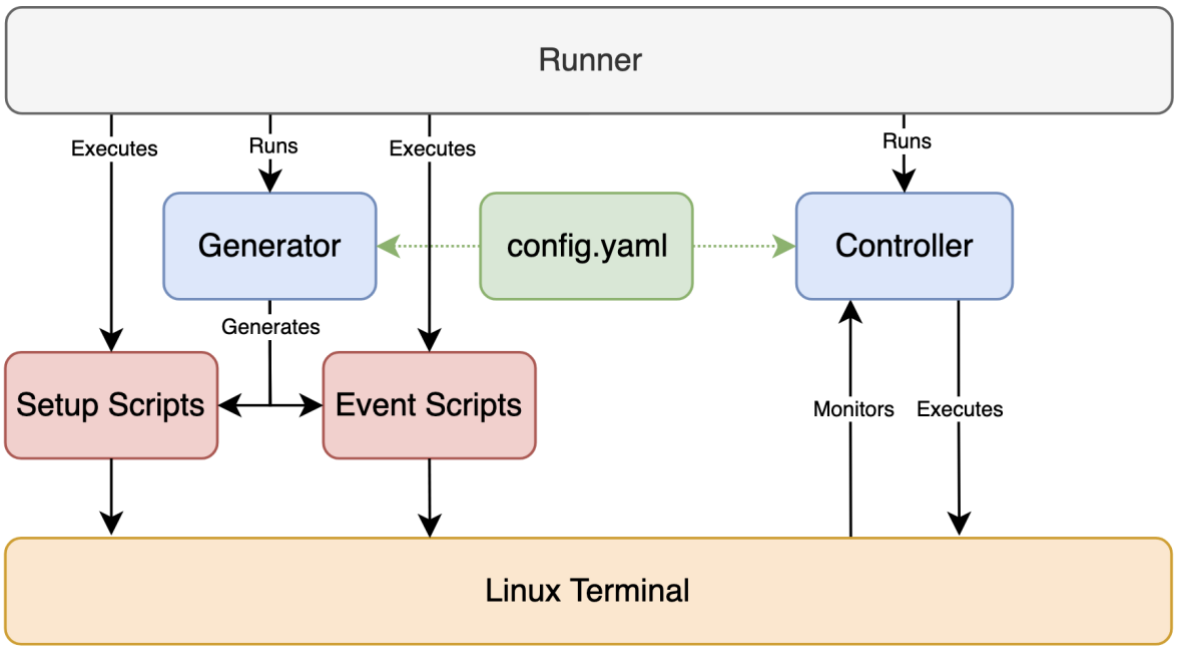


Figure 4.3.3.3.1.: Neptune's Control Logic – Source: self-made

The controller will monitor the health of the different circuits and the outgoing traffic and make real-time decisions based on the current state of the network, the configured policies and the characteristics of the traffic at any given moment.

Those decisions will be translated into bash commands and executed in the router's namespace terminal, effectively changing the behavior of its routing tables.

In addition to the scripts that enable the setup of the whole environment, the generator will also create scripts that simulate events, such as the failure or recovery of different circuits.

Those event scripts will be executable by the runner via user-inputted commands, enabling the verification of the behavior of the controller under different circumstances.

4.3.4 VALIDATION METHODOLOGY

Once the framework implementation was completed, a validation methodology was established to verify the behavior of the developed solution under various conditions.

The main objective of this phase is to verify that NEPTUNE can manage basic IPv6 multihoming scenarios using Provider-Aggregatable routing, maintaining traffic continuity, correctly applying routing policies, and automatically reacting to connectivity changes.

To this end, a set of tests was designed based on controlled scenarios of normal operation, link degradation, and connectivity recovery. During these tests, IPv6 traffic was generated between different nodes while the system's internal behavior was monitored.

Validation was performed using both traditional network analysis tools and observation mechanisms developed specifically for the framework, enabling system real-time analysis.

4.3.4.1 Test Scenarios

In order to evaluate various functional aspects of the system, several experimental scenarios were defined that are representative of typical situations in multihomed edge environments.

Scenario 1 — Normal Operation

The first validation scenario corresponds to normal operating conditions, where all ISPs defined in the experimental environment remain available.

During these tests, different types of IPv6 traffic were generated between the internal client and multiple remote services, allowing us to verify:

- The functioning of routing policies.
- Prefix-based route selection.
- Traffic balancing among active ISPs.
- Flow persistence once the initial assignment has been made.

This scenario allow us to analyze the system's behavior in the absence of topology changes.

Scenario 2 — ISP Outage

The second scenario was designed to evaluate the framework's behavior in the event of connectivity loss on one or more ISP links.

To do this, controlled failures were induced by dynamically deactivating virtual interfaces within the experimental environment. During these tests, continuous traffic was maintained between different nodes in the lab, allowing us to observe:

- Automatic failure detection by the controller.
- Dynamic reconfiguration of the pool of active ISPs.
- Reassignment of new communications.
- Automatic restoration of connectivity using alternative routes.

This scenario allowed us to validate the operation of the failover mechanisms implemented within the framework.

Scenario 3 — Connectivity Recovery

The third scenario focused on analyzing the system's behavior following the recovery of previously degraded links.

During these tests, ISPs affected by failure events were dynamically restored, allowing us to verify:

- Automatic detection of connectivity recovery.
- Reincorporation of links into the active pool.
- Redistribution of new flows.
- Progressive restoration of the system's original behavior.

This scenario demonstrated that NEPTUNE can manage bidirectional state transitions without the need to restart the environment or interrupt the system's overall operation.

Scenario 4 — Traffic Balancing

Additionally, specific tests were conducted to evaluate the behavior of the weighted balancing mechanism implemented by the controller.

To do this, multiple independent communications were generated to remote services not associated with strict policies, observing the distribution of flows among different active ISPs according to the weights defined in the system configuration.

These tests validated the behavior of the route selection algorithm and verified the consistency between the defined declarative policy and the actual distribution observed during execution.

4.3.4.2 Metrics and Evaluation

During the experimental phase, various metrics were analyzed to evaluate both the functional performance and the operational impact of the framework.

One of the key metrics examined was failover time, defined as the interval between the degradation of a link and the effective restoration of connectivity using alternative routes available within the system.

Likewise, the times associated with the detection and recovery of previously degraded links and their reincorporation into the active ISP pool were also analyzed.

Another metric considered was the packet loss observed during the transition processes. To this end, continuous ICMPv6 transmissions were used, which allowed for the direct observation of temporary connectivity interruptions during failover and recovery events.

In addition, the flow persistence behavior implemented by the controller was evaluated, verifying that active communications maintained consistency in route selection while their entries remained valid within the flow table.

Finally, the system's behavior was also analyzed from the perspective of routing and prefix management, verifying:

- Correct application of route selection policies.
- Dynamic ISP reassignment during topology changes.
- Consistent translation of IPv6 prefixes using NPTv6 mechanisms.

4.3.4.3 Measurements Tools

The experimental validation of the framework relied on various tools for traffic generation, monitoring, and observation of the system's internal state.

For IPv6 traffic generation, tools based on ICMPv6 were primarily used, particularly ping, which allowed for the generation of both one-off communications and continuous transmissions during connectivity transition events.

Likewise, route analysis and routing table tools such as traceroute and iproute2 were used, allowing us to inspect the behavior of the data plane and verify the routing decisions applied by the system.

The internal state of the framework was monitored using logs generated by the controller and the operational visualization layer developed specifically for this work. This interface allowed for real-time monitoring of:

- ISP status.
- Active pool of available links.
- Flow table.
- Failure and recovery events.
- Connectivity test results.

Finally, Linux system inspection mechanisms were also used to verify routing rules, traffic tags, and IPv6 NTP configurations dynamically applied to the experimental environment.

4.3.5 LIMITATIONS OF THE METHODOLOGY

Although the experimental environment developed allows us to validate the framework's functional behavior and demonstrate the technical feasibility of the proposal, the methodology used has certain limitations that must be considered.

The first limitation, and probably the most obvious one, is that all validation was performed in a virtualized environment based on Linux network namespaces and virtual links. Although this approach allows for the construction of reproducible and controlled scenarios, it does not fully represent the behavior of real networks, where additional factors such as latencies, congestion, hardware heterogeneity, or real-world traffic conditions come into play.

Furthermore, the number of ISPs used within the experimental environment was deliberately kept small to facilitate observation of the system's internal behavior. In real-world scenarios, operational complexity and the number of potential routes could increase significantly.

Another important limitation is the nature of the traffic used during testing. Validation relied primarily on ICMPv6 traffic and simple communications aimed at demonstrating routing behavior and basic connectivity continuity. No exhaustive testing was conducted on complex application protocols, high-volume traffic, or real production workloads.

Similarly, the controller implemented within NEPTUNE has an experimental and academic scope. Its primary objective is to demonstrate the operational viability of a policy-based IPv6 edge multihoming model, NPTv6, and dynamic route selection. Therefore, it does not incorporate features typical of complete industrial solutions, such as advanced high-availability mechanisms, distributed synchronization, or internal fault tolerance.

Finally, some time-based metrics obtained during testing depend directly on configurable system parameters, such as monitoring intervals and flow expiration times. For this reason, the observed values should be interpreted as results representative of the developed experimental environment and not as absolute limits that can be extrapolated to any real-world implementation.

4.4 PLANNING AND ECONOMIC ESTIMATION

Due to the primarily experimental and research-oriented nature of this work, the planning and cost estimates associated with the project take a more simplified approach than is typical in large-scale industrial or commercial developments. The main objective of NEPTUNE has not been the creation of a final production-oriented product, but rather the design, implementation, and validation of an experimental operational model for IPv6 edge multihoming scenarios.

In this context, most of the resources used during the project's development were related to research tasks, architectural design, software implementation, experimental validation, and the preparation of technical documentation. Similarly, much of the infrastructure used is based on open technologies and virtualized environments running on general-purpose hardware, minimizing the need for specific physical equipment or proprietary licenses.

The following sections present a rough estimate of the project timeline, the hours of work dedicated to the different development phases, and the main costs associated with the infrastructure and resources used during the project.

4.4.1 PROJECT PLANNING

The project was developed using an incremental and iterative methodology, allowing for the progressive evolution of both the system architecture and the validation mechanisms implemented. Due to the experimental nature of the work, much of the development focused on the practical exploration of different operational approaches for IPv6 multihoming scenarios, continuously combining research, implementation, and validation tasks.

In the first phase, a study was conducted on the state of the art and technologies related to IPv6 multihoming, policy routing, NPTv6, and high-availability mechanisms in Linux environments. This stage allowed us to identify the main existing operational limitations and define the general objectives of the proposed framework.

Subsequently, the general architecture of NEPTUNE was designed, defining the separation between the control plane, data plane, and observability layer. During this phase, the foundations of the experimental environment were also established, including the use of Linux network namespaces, virtual links, and automation based on YAML configurations.

Once the initial architecture was defined, the progressive implementation of the system's various components began. First, the mechanisms for automatic scenario generation and virtual lab deployment were developed. Next, the controller responsible for connectivity monitoring, dynamic route selection, flow persistence, and failover and recovery event management was implemented.

After completing the functional core of the framework, the lightweight observability and visualization layer used during the experimental validation phases was developed. This interface significantly simplified the monitoring of the system's internal behavior and facilitated the interpretation of the various dynamic scenarios evaluated during testing.

Finally, the last stages of the project focused on the experimental validation of the framework, the compilation of results, and the preparation of the technical documentation associated with the work.

4.4.2 TIME ESTIMATION

The project timeline was estimated by taking into account the various phases of research, design, implementation, and validation carried out throughout the project. Due to the iterative and experimental nature of NEPTUNE, several of these tasks proceeded in parallel, particularly during the controller development and experimental environment validation stages.

Table 4.4.2.1. shows an approximate estimate of the hours dedicated to each of the main phases of the project.

Project Phase	**Estimated Hours**
Investigation and state of the art study	80 h
Architectural design and model definition	50 h
Experimental environment development	60 h
Generator and automation implementation	45 h
NEPTUNE controller development	90 h
Validation and observability mechanisms implementation	40 h
Testing and experimental validation	55 h
Documentation and redaction	80 h
Estimated Total	**500 h**

Table 4.4.2.1.: Estimated time distribution during project development – Source: self-made

Most of the time spent during the project was devoted to the experimental development of the framework, including the implementation of the controller, the integration of dynamic routing and flow persistence mechanisms, as well as the validation of failover and connectivity recovery scenarios.

Likewise, a significant portion of the work was also dedicated to the conceptual analysis of the IPv6 multihoming problem and the design of a sufficiently modular architecture that would allow for the integration of heterogeneous mechanisms within a coherent operational environment.

4.4.3 ECONOMIC COST ESTIMATION

Due to the experimental and software-oriented nature of the project, the costs associated with the development of NEPTUNE have primarily related to the engineering time spent on the design, implementation, and validation of the framework, as well as the use of general-purpose hardware during experimental testing.

Unlike other networking projects focused on physical deployments or production infrastructures, the experimental environment developed for this work is based primarily on open technologies and virtualization on Linux, minimizing the need for specialized equipment, proprietary licenses, or dedicated hardware infrastructure.

Table 4.4.3.1. shows an estimate of the main costs associated with the project's development:

Concept	**Estimated Cost**
Partial computer amortization during development	300 €
Software and tools utilized	0 €
Engineering development time	7500 €
Total Estimated Cost	**7800 €**

Table 4.4.3.1.: Estimated economic cost of the project – Source: self-made

The cost associated with development time has been estimated based on an approximate rate of €15 per hour for research, implementation, experimental validation, and the preparation of technical documentation carried out during the project.

The partial cost of the computer amortization has been estimated dividing the total acquisition cost divided its expected lifespan, of six years, and multiplied by the time invested in the development, of around 9 months (full academic course):

$$Computer\ Amortization\ Cost = \frac{2400 \times 9}{6 \times 12} = 300\ €$$

It is also particularly important to note that the entire framework was developed using open-source tools and technologies, including Linux, Python and various native utilities from the Linux networking ecosystem. This has significantly reduced the infrastructure and licensing costs associated with the experimental development of the project.

Chapter 5. DEVELOPED MODEL

This chapter presents the design and implementation of NEPTUNE, detailing its underlying architecture, operational logic and experimental deployment; providing a comprehensive description of how the system is structured and how it behaves under different conditions.

5.1 CORE CONCEPTS AND TECHNOLOGICAL FOUNDATIONS

Before introducing the proposed model, it is necessary to establish the set of core concepts and technological elements on which NEPTUNE is built. This section revisits key aspects of router architecture, the Linux networking stack and namespace-based isolation.

5.1.1 ROUTER ARCHITECTURE

First off, it is important to define what a router is. A router is a network component integrated by multiple physical or logical interfaces associated with different network segments or routing domains [34], enabling the **forwarding of traffic between different networks**.

While externally being a single component, the logic inside a router can be modelled into three planes with differentiated functionality [35]:

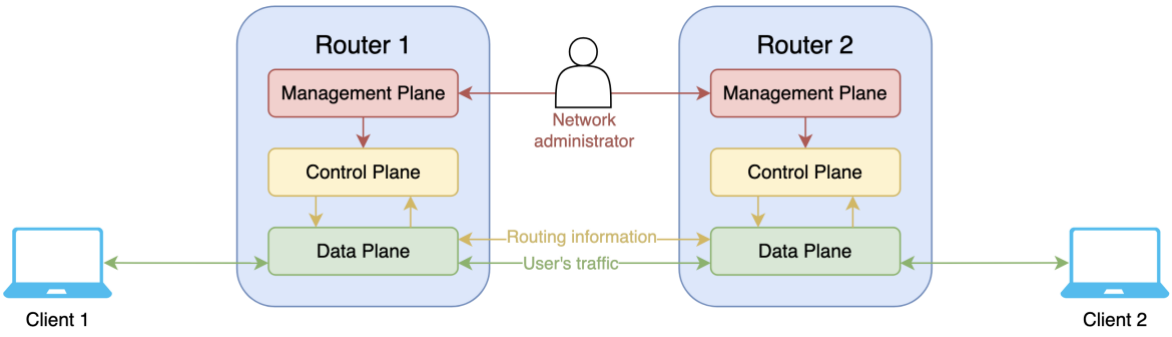


Figure 5.1.1.1.: Router Logical Planes – Source: self-made

- **Data plane**: performs the forwarding based on the control plane's decisions
- **Control plane**: makes the forwarding decisions the data plane follows
- **Management plane**: offers a user interface to configure the control plane

5.1.1.1 Data Plane

The data plane is responsible for forwarding packets according to rules installed by the control plane, executing operations such as header rewriting and, when required by the forwarding policies, packet encapsulation (e.g. in DS-Lite) or address translation (e.g. in 464XLAT).

Forwarding is performed according to the Forwarding Information Base (FIB) [34], a table that associates network prefixes with forwarding adjacencies, which contain the outbound interface and the Layer 2 next-hop information:

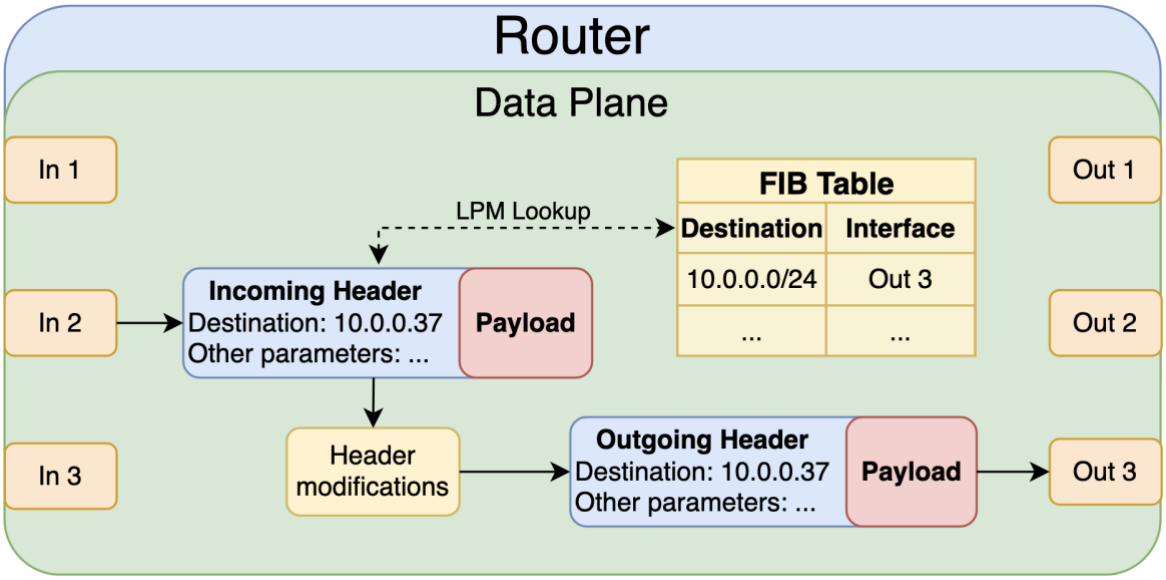


Figure 5.1.1.1.1.: Data Plane – Source: self-made

When a packet arrives at the router, the data plane performs a Longest Prefix Match lookup [36], in the example figure: “lookup (10.0.0.37) -> matches 10.0.0.0/24 -> Out 3”.

The data plane updates specific IP header fields such as TTL (or Hop Limit) [1][2] and, in IPv4, the header checksum, and rewrites the Layer 2 encapsulation parameters before forwarding the packet through the selected outbound interface.

5.1.1.2 Control Plane

While the data plane forwards packets, the control plane decides how such packets are forwarded. This process starts with route discovery. Routes can be learned in three ways:

- **Connected**: networks directly connected to the router's interfaces
- **Static**: configured manually by the network administrator
- **Dynamic**: learned from interior and exterior gateway protocols (IGPs & EGPs) [13]

All valid routes are stored in a routing table, also known as Routing Information Base (RIB) [34]. The main differences between the RIB and the FIB from the data plane are:

Routing Information Base (RIB)	Forwarding Information Base (FIB)
Includes all possible routes to a certain destination, together with their parameters	Only includes the best path(s), selected by the control plane
Entries point towards the IP address of the next hop	Entries point towards the Layer 2 address of the next hop and the outbound interface

Table 5.1.1.2.1.: RIB vs FIB – Source: self-made

Learnt routes are evaluated according to metrics (IGPs), attributes (BGP), and administrative policies, and the best path towards each destination is selected and installed in the FIB.

Traffic engineering allows forwarding decisions to be influenced by parameters such as packet source, protocol or configured policies. In such cases, packet classification is performed before the forwarding lookup in order to select the appropriate forwarding context.

When this classification and lookup process can be executed using hardware tables, forwarding remains in the *fast path* [37]. Otherwise, packets may be punted to the control plane for software processing, resulting in *slow-path* forwarding.

5.1.1.3 Management Plane

Management plane's main purpose is enabling network administrators to interact with the router by allowing to monitor its status and configure parameters, protocols and policies.

Via the management interface, the user can **monitor** parameters such as:

- Routing tables
- Interface status
- Protocol sessions
- Logs

It also allows to perform a wide range of **configurations**, such as:

- Routing protocols (BGP, OSPF, etc.)
- Static routes
- Enabling and disabling interfaces
- Setting or disabling routing policies

These parameters do not directly affect forwarding, but modify the behavior of the control plane, which in turn determines the forwarding state installed in the FIB.

The access to the management interface can be done via a Command Line Interface (**CLI**), a Graphical User Interface (**GUI**), with network management **APIs** like NETCONF, or by using the Simple Network Management Protocol (**SNMP**) [38].

Modern networks, particularly large-scale ones, use software-defined networking (**SDN**) architectures [39], where control-plane functions are logically centralized in a network controller, which programs the forwarding state of the devices directly. In such environments, the management plane interacts with the SDN controller, which assumes responsibility for routing and policy decisions across the network.

5.1.2 LINUX NETWORKING STACK

The Linux Networking stack constitutes the basis for this NEPTUNE implementation, as the kernel is responsible for both packet forwarding and routing policy enforcement. In this context, it is important to understand the implicit separation between the control and the data planes: while the decision logic is implemented in the user space (mainly using Python scripts), the forwarding is carried out entirely by the kernel.

Linux's routing mechanism is based on routing tables, being the ***main*** table the default one. However, the system allows the creation of additional tables, enabling the the implementation of complex routing policies [50]. In NEPTUNE, each outbound interface (ISP) is associated with an independent routing table, enabling precise control of flow paths.

The routing table selection is not only based on the destination address, but can also be conditioned using rules (***ip rule***), allowing to add additional decision criteria. In this project, the key criterium is using packet marking (***fwmark***), acting as an identifier associated to each packet, effectively decoupling the classification logic from the forwarding process [50][51].

The assignation of this marks is performed using the ***Netfilter*** subsystem, specifically through the mangle table from ***iptables*** [51]. In the PREROUTING hook, rules identifying concrete flows are installed, giving them a mark. From that moment, all packets that belong to such flow will be treated in a consistent way by the kernel, as the policy routing rules will send the marked traffic to its correspondent routing table.

In addition, Netfilter is used to implement the prefix translation via NPTv6, using the SNPT and DNPT targets [49]. This mechanism allows to statelessly map internal to external, a key aspect to enable multihoming without needing to modify end host's configuration.

A packet's processing flow consists on a well-defined sequence: the packet enters through an interface, goes through the PREROUTING hook where it can be marked, the routing decision is made based on the configured tables and policies, and finally is forwarded through the outbound interface, being able to apply additional transformations in POSTROUTING, as it is the case of NPTv6.

5.1.3 LINUX NAMESPACES

Linux network namespaces are the mechanism that allows to build NEPTUNE's experimental environment in an isolated, reproducible and completely controlled way [52]. A namespace can be seen as an independent TCP/IP stack instance, with its own interfaces, routing tables, forwarding configuration and policies.

This technology allows multiple logical nodes to coexist within the same physical machine, behaving as independent network entities. In the context of NEPTUNE, namespaces are used to emulate all the elements of a multihomed environment, including the client, the border gateway, the different ISPs, intermediate networks and remote servers.

Connectivity between these nodes is established with virtual interface pairs (***veth***), which act as point-to-point links [52]. Each side of the pair is assigned to a different namespace, simulating a physical link between devices. This approach allows to build different topologies and control aspects such as addressing, routing or link states.

From a functional standpoint, namespaces play a role purely associated to the data plane. All forwarding decisions, the routing behavior and the packet transformations are executed within the namespaces and managed directly by the kernel. NEPTUNE's control logic is located outside of this plane, interacting with the namespaces through tools such as ***iproute2*** or ***iptables***, without introducing any logic into the data plane.

One of the key advantages of this approach is its similarities with real environments. Despite namespaces being used here as a simulation tool, their network behavior is equivalent to real Linux systems, allowing to extrapolate the obtained results to real deployed environments with a high degree of accuracy.

In addition, namespaces add great flexibility when it comes to define and modify the environment. It is possible to create, configure or delete nodes dynamically, as well as alter the state of the different links in order to simulate circuit failures and recoveries. This capability is key for system validation and testing with different experiments.

5.2 SYSTEM DESIGN OVERVIEW

5.2.1 DESIGN REQUIREMENTS

Based on the analysis of the state of the art, and the identified operational gaps present in current solutions, the design requirements that guide the development of the proposed NEPTUNE model are defined. The goal is not only to build an IPv6 multihoming system that is functional, but also viable in real deployments.

First, the system needs to implement multihoming mechanisms that allow to maintain connectivity in the event of an upstream circuit failure. For that purpose, automatic failover capabilities are required, detecting and redirecting traffic towards alternative circuits with the least possible impact. The system also needs to be capable of detecting when connectivity has been recovered after an outage, recovering the initial forwarding state.

Second, the model must support load balancing, not only in terms of traffic distribution, but also ensuring flow-level consistency. This implies that a flow stickiness mechanism is required, so packets that belong to the same flow always traverse the same path, avoiding problems when using connection-oriented protocols, such as TCP. The balancing needs to be configurable using weights, allowing to take advantage of links with different capacities.

Another requirement is the support for Traffic Engineering, allowing to define routing policies for outbound traffic. These policies should offer enough flexibility to adapt to different scenarios, such as prioritizing traffic to certain destinations through certain links.

All these objectives need to be accomplished while meeting a set of constraints that condition the system's design. No host support nor modification should be required, allowing for an easy and rapid deployment, and it should not require the use of BGP nor any kind of ISP agreement, making it compatible with commercial broadband plans.

Finally, the model must be compatible with the most common IPv6 transition mechanisms, including Dual Stack, Dual Stack Lite and 464XLAT. This compatibility ensures that the solution can integrate and coexist with heterogeneous, existent infrastructure.

5.2.2 ENVIRONMENT MODEL

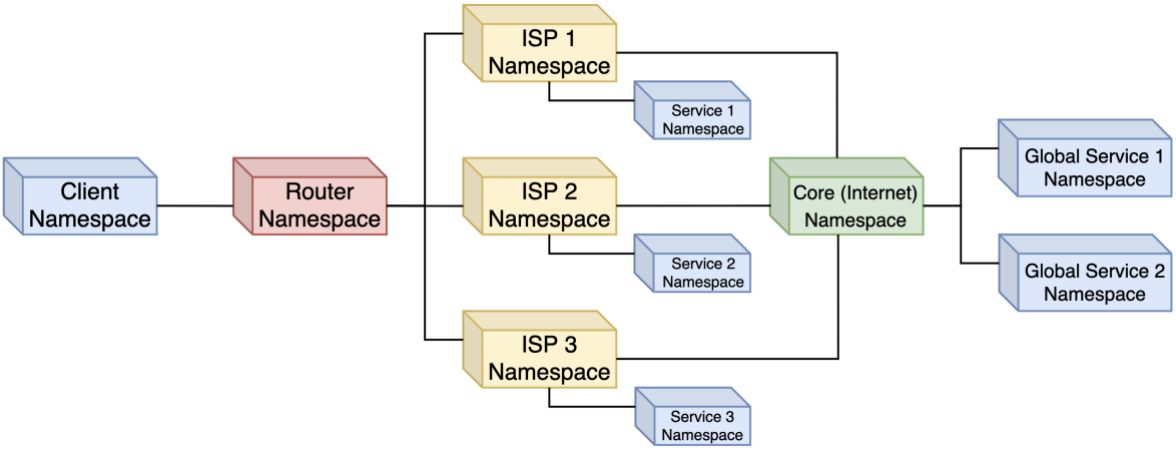


Figure 5.2.2.1.: Implemented Namespace Deployment – Source: self-made

For the final deployment, a total of 11 namespaces will be implemented:

- **A client**, from which requests will be sent to the available services, allowing to verify NEPTUNE's behavior under different circumstances
- **A router**, on top of which we will implement NEPTUNE's logic
- **Three ISPs**, in order to model the two main use cases for multihoming, load balancing and failover, being able to balance traffic between two of the ISPs, leaving the third one only for failover purposes
- **Three single-homed servers**, representing servers that can only be accessed through a single ISP, such as IPTV or VoIP services associated to a certain broadband plan
- **A core (Internet) namespace**, interconnecting ISPs and globally accessible servers
- **Two globally accessible servers**, representing services that can be accessed via any ISP with connection to the Internet

This model allows to represent in a controlled and reproducible way a realistic multihoming scenario in which single-homed and globally accessible services coexist. The namespaces separation not only makes the simulation easier but also allows to isolate the behavior of each component. This establishes a solid foundation to evaluate NEPTUNE's performance under different circumstances.

5.2.3 OPERATIONAL MODEL

NEPTUNE's operative model describes how traffic is processed in real time, from the reception of a packet until its routing through one of the available outbound interfaces. Contrary to static solutions, NEPTUNE adopts a reactive approach, in which routing decisions are made dynamically based on the configured policies and the network's state.

The processing flow starts when a packet enters the router from the local network. The monitoring system then captures the traffic and extracts the relevant information to identify the flow it belongs to, typically defined by a 5-tuple (source address, destination address, source port, destination port and protocol).

This information is then sent to the controller, which acts as a central coordination system. The controller does not directly make routing decisions, it keeps an updated vision of the current state of the links, and passes this forwarding context to a flow manager.

The flow manager is the component in charge of applying the decision logic. Depending on the configured policies, the available ISPs and the current state of the network, it selects the best suitable path for each new flow. Once the decision is made, a rule is installed in the packet filtering system (iptables), marking the corresponding traffic.

From then on, packet processing is completely handled by the kernel's data plane. The assigned mark allows the policy routing system to select the corresponding routing table, ensuring that all packets that belong to that flow follow the same path. This mechanism ensures routing consistency without keeping state within the kernel.

This approach decouples the control logic from the packet forwarding, allowing the implementation of complex routing policies in the user space while maintaining efficiency and robustness in the kernel routing.

5.3 *INTERNAL ARCHITECTURE AND EXECUTION MODEL*

5.3.1 COMPONENT INTERACTION OVERVIEW

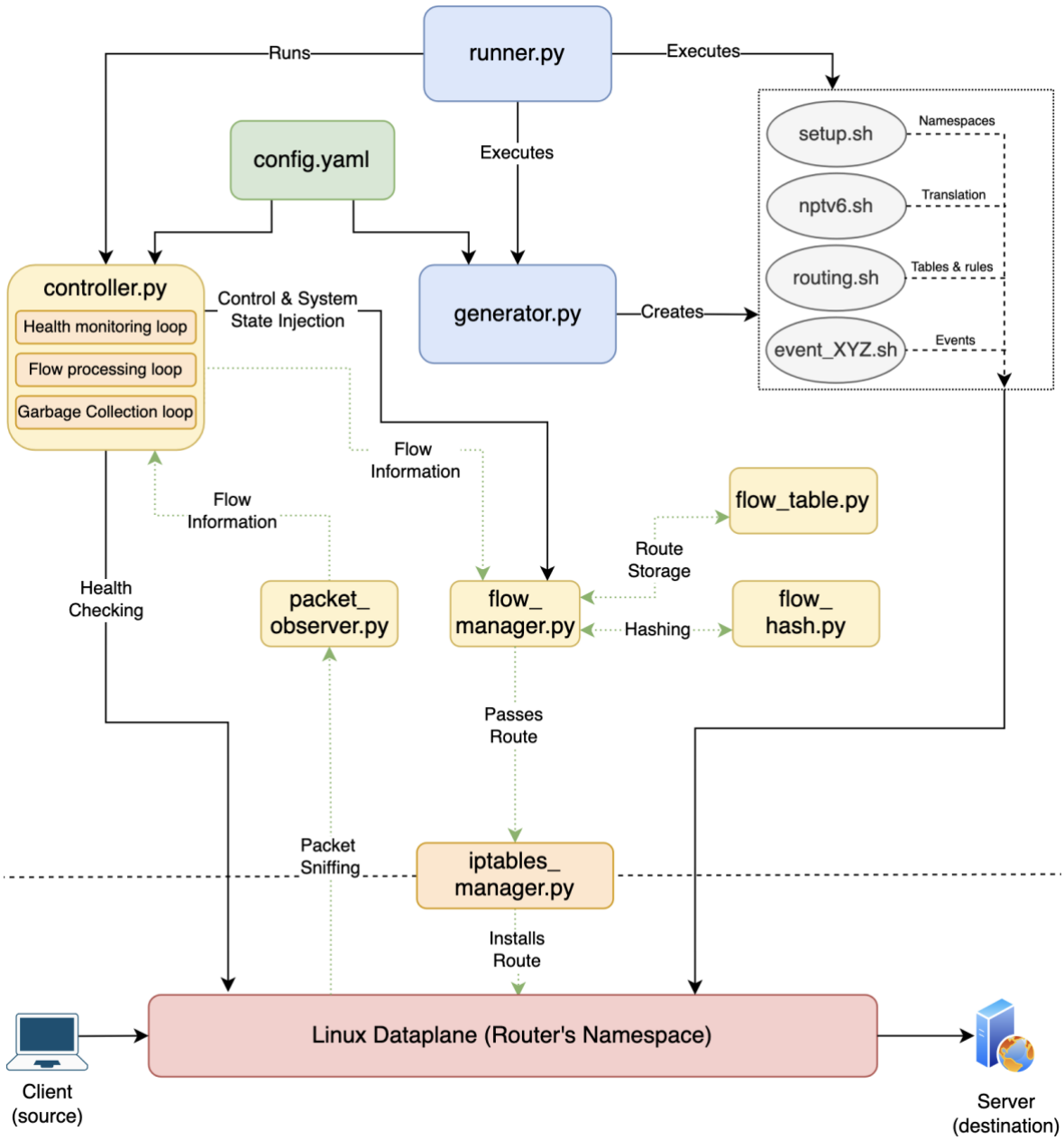


Figure 5.3.1.1.: NEPTUNE's internal design – Source: self-made

The NEPTUNE implementation consists on multiple software components that interact to cover two different phases: the experimental environment creation and the dynamic operation of the system during runtime.

The generator (**generator.py**) transforms the declarative configuration defined in the **config.yaml** file into runnable scripts, which create the laboratory, configuring routing, applying NPTv6 and generating failure and recovery events.

The runner (**runner.py**) on the other hand acts as an operational entry point, coordinating the execution of the scripts and allowing to deploy, reset or modify the environment in a controlled way.

When the environment is deployed, the control passes to the controller (**controller.py**), which is the core of the control plane. It loads the runtime configuration, monitors the state of the circuits and handles the processing of new flows. Such new flows are detected using an auxiliar process called the packet observer (**packet_observer.py**).

The routing decision is made by the flow manager (**flow_manager.py**). This component receives the flow key, checks the system's current state (injected by the controller), and applies the configured policies. In order to keep consistency for packets from the same flow, the flow manager uses **flow_table.py**, an auxiliar component that temporarily stores routes.

When the outbound ISP is selected, **iptables_manager.py** acts as an interface between the control and the data planes. This module installs the route in the namespace data plane. From then on, the forwarding is directly handled by the Linux kernel using policy routing rules, per-ISP routing tables and NPTv6 prefix translation.

That way, the architecture keeps a strict separation between the different planes of the system: the deployment layer, the control plane and the data plane. This division allows NEPTUNE to be modular, reproducible and extensible.

5.3.2 SOFTWARE COMPONENTS DESCRIPTION

NEPTUNE is integrated by a set of decoupled software components, each performing a specific function rather in the data or the control plane. This separation allows to structure the system behavior in a modular way, differentiating the environment creation, the traffic monitoring and the decision making.

We can distinguish three main functional blocks:

- **Environment** generation and deployment
- **Control plane** logic
- **Data plane** interface

Next, the main components that integrate each of these blocks are described:

Generator.py

This component acts as a transition point between the system specification and its creation and deployment. From the configuration file (config.yaml), this module automatically creates the environment generation scripts.

Its main role is translating abstract parameters (formalized in the config.yaml file) into executable commands for the Linux terminal. The generated scripts are:

- **setup.sh**: generation of namespaces, virtual interfaces and their addressing
- **routing.sh**: creation of routes and routing tables
- **nptv6.sh**: NPTv6 configuration
- **event_<XYZ>.sh**: event simulation scripts

That way, the environment's definition and deployment are completely decoupled, allowing to modify the simulation topology or the system's behavior without altering the code.

Runner.py

The runner module offers the user a unified interface to manage the lifecycle of the whole system. Its function is not implementing any network logic but coordinating the execution of the multiple additional components. It centralizes functions such as:

- Laboratory environment deployment
- Laboratory environment teardown
- Simulation event execution
- Controller start and stop

This component acts as an entry point to the system, easing its use and ensuring that the different stages (generation, deployment and execution) are performed in correct order.

Controller.py

The controller is the core of NEPTUNE's control plane. Its main function is keeping a global vision of the state of the system and coordinating the decision making.

This module handles:

- ISP health monitoring
- Determining which links are active
- Active ISPs management
- Flow and route lifecycle management
- Observation and decision modules coordination

The controller operates by using concurrent execution loops, among which are the health monitoring loop and the expired flow collection loop. In addition, it acts as an intermediary between the traffic detection and the decision engine, giving the second the information needed about the state of the system at any moment.

Packet_observer.py

The packet observer module acts as the system's sensor, enabling the detection of new flows.

This component captures the packets from the router's LAN interface and generates a key that uniquely identifies each flow, which can include two sets of parameters:

- **5-tuple** (source address & port, destination address & port and protocol) for IP traffic
- **3-tuple** (source address, destination address, ICMP flow identifier) for ICMP traffic

The extracted key is sent to the controller so the system can dynamically react when new flows appear, according to the configured policies and the state of the network.

Flow_manager.py

The flow manager is the traffic engineering core of the system and implements all the decision logic for path selection. From the information the controller provides, it determines:

- Whether a flow needs to be managed by the system
- Which policy to apply depending on the packet's destination
- Which ISP must be used as outbound

The decision process can follow different operation modes:

- **Strict selection**, where a destination can only be reached through a single ISP
- **Preferent selection**, prioritizing an ISP but allowing fallback
- **Balanced selection**, based on weighted distribution

This module is where the entire process of classification, selection, and decision-making is executed, and with its three different levels of selection strictness, it is a very flexible traffic management core that enables to validate NEPTUNE's use in relatively complex scenarios.

Flow_table.py

This component acts as a temporary system memory, storing previously made decisions so each time a new packet from an already processed flow reaches the router, the decision logic does not need to be applied again.

Its main goals are ensuring per-flow consistency while a stored decision is valid and offloading the system from performing the decision logic for each individual packet.

This module also manages inactive flow expiry after a configured expiration time, allowing to free up resources and adapt the system to topology or link state changes.

Flow_hash.py

The flow hash module implements the traffic distribution mechanism in balancing scenarios.

From the previously extracted flow key, this component generates a deterministic ISP selection, taking into account the health of the circuits at the moment and the configured circuit weights, achieving:

- Controlled load balancing
- Ensuring per-flow consistency
- Avoiding unpredictable behaviors

Its main goal is ensuring routing persistency, avoiding packets that belong to the same flow to be treated differently, even after a previously made decision expired, thanks to the implementation of per-flow hashes instead of using a pseudorandom decision algorithm. This is particularly important with connection-oriented protocols such as TCP, where a change in the source IP address can drop the connection.

While Flow_table.py ensures routing consistency while a flow path decision is active, Flow_hash.py guarantees that a routing decision for same flow always gives the same result, provided the topology and health conditions of the network are also the same.

Iptables_manager.py

The iptables manager acts as an interface between the control and the data plane.

Its role is translating the decisions made by the selection engine into concrete filtering and packet marking rules. Specifically, it carries out:

- Installing marking rules (fwmark) associated to the flows
- Deleting such rules after the expiry time

Those marks are later used by the policy routing system to forward traffic to the routing table corresponding to the selected ISP.

Together, these components allow to build a modular system in which packet detection, decision making and policy enforcement are clearly separated, simplifying both system analysis and expansion by allowing to isolate the different roles within the execution flow.

5.3.3 EXECUTION MODEL: LOOPS AND THREADS

Execution Model Overview

NEPTUNE is based on a concurrent execution model, in which different tasks are executed in parallel in order to give a continuous, reactive and non-blocking operation. Instead of using a single sequential flow (monolithic approach), the system uses multiple execution loops (threads), each of them having a separate role inside the control plane.

This distributed system approach allows the system to perform:

- Continuous link health monitoring
- Real time flow detection
- Immediate route policy enforcement
- Consistent behavior in the event of dynamic changes

Conceptually, the behavior of the system can be understood as the interaction of three main loops coordinated by the controller.

Health Monitoring Loop

This loop is periodically executed and evaluates the state of the defined ISPs by actively checking a destination associated to each link using ICMP probes, and builds an updated ISP health status vision. Based on this information, the system determines:

- Which ISPs are considered operative
- Which set of ISPs should integrate the active routing pool

When a state change is detected, such as the failure or recovery of a link, the system reacts immediately, updating the set of available outbound circuits.

A key aspect about this loop is that it invalidates previously learnt flows in the event of a change in the topology, such as a circuit failure or recovery, avoiding the use of a previously established criteria under a context that is no longer valid.

Garbage Collection Loop

The system keeps an active flow state table that allows to preserve routing coherence. However, this table must be dynamically managed in order to avoid undefined growth and the use of obsolete decisions.

The garbage collection loop:

- Identifies flows that have not been used for a certain period of time
- Deletes the corresponding entries from the stored flow table
- Removes the associated marking routes in the data plane

This mechanism allows the system to:

- Automatically free up resources
- Dynamically adapt to traffic pattern changes
- Avoid unnecessary state accumulation

In addition, it contributes to the reactive approach of the system, allowing new flows to be reclassified under the current environment conditions.

Packet Observer Loop

The third main component of NEPTUNE is the traffic monitoring process, implemented as an independent worker that captures packets in the router's LAN interface.

This process continuously analyzes traffic and detects new flows. Each time a relevant packet is detected, its flow key is extracted and sent to the controller.

Contrary to the previous loops, this process does not make any decision nor keep a complex state, it acts as a passive sensor that feeds the control plane with real time information.

The communication between this process and the controller is done through a standard output channel, allowing to completely decouple it from the rest of the system.

Main Controller Loop

The controller has a main execution loop, which acts as a coordination point:

- Receives the flow keys generated by the observer
- Processes each new flow through the decision engine
- Delegates the rule installation to the data plane

This loop is reactive by nature: it does not generate events by itself; it responds to them.

The flow processing follows a defined sequence:

1. Flow key reception
2. Flow relevance validation
3. Outbound ISP selection
4. Marking rule installation
5. State table flow registration

This way, the controller turns observed events into concrete actions.

Concurrence & Coordination Model

The combination of all loops constitutes a concurrent system in which different aspects are managed independently but in a coordinated way:

- The health monitor manages the global system state
- The observer detects data plane events
- The controller makes decisions
- The flow collector keeps temporal consistency

The synchronization between these elements is implicitly performed using shared structures (such as the flow table) and via the periodic health updates.

This design avoids unnecessary blockings and allows the system to scale.

Time-dependent Behavior Considerations

The concurrent approach of the system introduces certain features that need to be taken into account when evaluating the model.

There is a brief time frame between the appearance of a new flow and the installation of the rule in the data plane, an interval in which the first packet of the flow might follow the default route instead of the one selected by the decision logic.

However, once the marking rule has been installed, the flow behavior becomes deterministic, ensuring that the following packets traverse the same path.

This compromise between simplicity and reactiveness results acceptable in the context of the system, particularly taking into account that its main objective is validating the model.

Execution Model Summary

Overall, the system can be understood as an event-driven architecture, in which:

- Incoming packets generate events (packet observer)
- Events trigger decisions (controller + flow manager)
- Decisions materialize into rules (iptables + policy routing)
- State is kept, updated and cleaned dynamically (flow table + garbage collection loop)

This model allows the implementation of a dynamic, adaptable and coherent forwarding behavior, maintaining a clear separation between the different levels of the system.

5.3.4 FLOW AND ROUTE LIFECYCLE

Overview

NEPTUNE's behavior can be understood as the lifecycle of a traffic flow, from its creation within the local network to its forwarding through one of the available ISPs.

This cycle is not based on a static route configuration, but on a dynamic model in which each flow is detected, classified and treated independently. Throughout this process, both control and data planes play a major role, cooperating to ensure a coherent and adaptable forwarding.

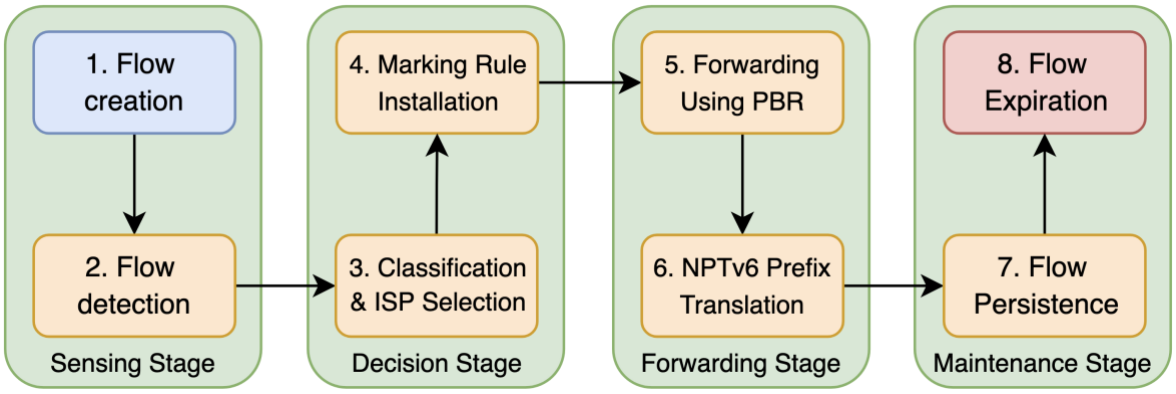


Figure 5.3.4.1.: Lifecycle of a Flow – Source: self-made

Next, this cycle, illustrated in Figure 5.3.4.1, is described step by step:

Step 1 – Flow Generation

The cycle starts when a host from the local network generates traffic towards an external destination. In the laboratory context, this is emulated by sending ICMPv6 packets from the client to the different available servers (i.e. pinging the servers).

At this point, the system has not yet made any specific decision regarding the flow's path, so the packet is treated according to the base configuration (i.e. using the default table).

Step 2 – Flow Detection

The packet is intercepted in the router's LAN interface by the packet observer module, which identifies the packet and extracts a key with information that uniquely identifies the flow.

The information used to identify the flow depends on the type of traffic:

- **5-tuple** (source address & port, destination address & port and protocol) for IP traffic
- **3-tuple** (source address, destination address, ICMP flow identifier) for ICMP traffic

The controller will be immediately notified about the new flow and its identifying key.

Step 3 – Classification and Decision

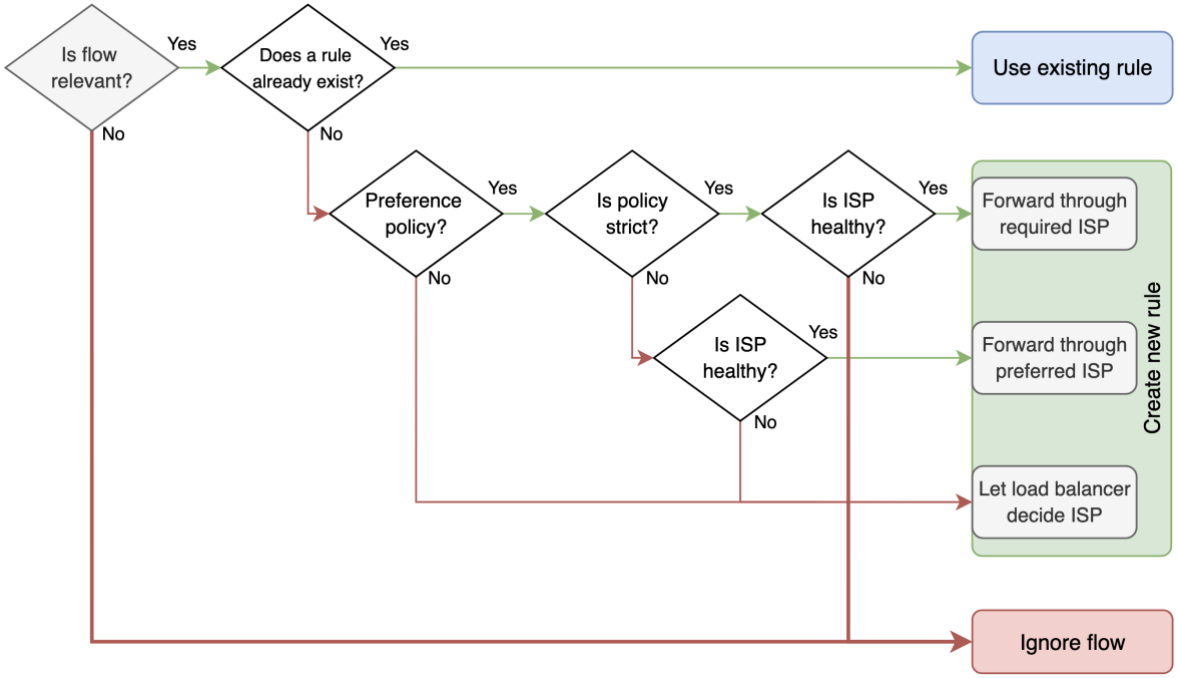


Figure 5.3.4.2.: NEPTUNE's Decision Logic – Source: self-made

Figure 5.3.4.2. illustrates the traffic classification and decision process. Once the controller receives the event, it delegates the decision to the flow manager, to which it also periodically injects health and system status.

With that information, the flow manager determines:

- Whether the flow is relevant to the system*
- Whether a specific policy needs to be applied
- The health status of the different ISPs
- The balancing configuration (when balancing is used)

Depending on those elements, the flow manager can decide which operation mode to use:

- Strict association to a certain ISP
- Preference through a certain ISP, but allowing fallback
- Let the load balancer decide (according to the configured weights)

This step constitutes the system's traffic engineering core.

**In this context a flow is defined as "relevant" if and only if its source address belongs to a LAN prefix and its destination address does not (i.e. outbound traffic)*

Step 4 – Routing Policy Installation

Once the decision is made, the system translates it into an action inside the data plane. A marking rule is installed, associating the flow with an identifier (fwmark) corresponding to the selected ISP.

That rule is applied in an early stage of the packet's processing by the kernel, allowing the following packets from the same flow to be identified and treated consistently.

The use of marks allows to decouple the decision logic from the forwarding mechanism, delegating the later to Linux's policy routing system.

Step 5 – Packet Forwarding

Once marked, traffic is processed by the kernel routing system. Depending on the value of the mark, a specific routing table associated to a certain ISP is used.

Inside the table, the packet is forwarded towards the next hop, going through the outbound interface associated with the selected ISP.

Step 6 – NPTv6 Prefix Translation

Before leaving the system, the packet must go through a network prefix translation process using NPTv6. This mechanism transforms the packet's source address from the internal network's prefix to the ISP's delegated prefix.

This translation enables:

- Decoupling internal and external addressing
- Seamlessly using multiple outbound prefixes
- Keeping independence from the ISPs' configuration

The reverse process is also applied to inbound traffic, ensuring bidirectional coherence.

Step 7 – Flow Persistence

Once the initial decision is made, the flow is registered in the system's flow table. This ensures that following packets that belong to the same flow are treated consistently, avoiding route changes that could affect the communication's stability.

Even after a rule has expired, due to a timeout, for example, the deterministic approach of the hash mechanism used for load balancing ensures that, under the same network health status, the same flow will go through the same ISP.

This mechanism becomes particularly relevant for connection-oriented protocols, such as TCP, where a change in the source IP address can drop the connection.

Step 8 – Rule Expiration

After some time, flows may become inactive. The system monitors for this situation via an expiration mechanism, deleting flows that have not been observed for a certain period.

When a flow expires, both its entry in the flow table and its associated marking rules are deleted, freeing up resources and allowing future communications to be treated as new flows.

Adaption to Network Changes

Parallel to the flows' lifecycle, the system continuously monitors the different ISPs' health status. When a change such as the failure or recovery of a circuit is detected, the set of available routes is updated.

In these cases, the system can invalidate the existent flows, forcing their reclassification under the new environment's conditions. This allows the system to adapt its behavior to dynamic topology changes without manual intervention.

Lifecycle Summary

Overall, the lifecycle of a flow inside NEPTUNE can be defined as a sequence of steps:

1. Traffic generation by a host inside the LAN
2. Flow detection by the packet observer
3. Flow classification & ISP selection by the flow manager
4. Installation of the marking rules through the iptables manager
5. Forwarding using policy routing in the kernel
6. Network prefix translation using NPTv6
7. Flow persistence within the flow table
8. Flow expiration

Meanwhile, the system remains aware of the health state of the available ISPs and the possible topology changes that can occur withing the network, being able to react to them by deleting old rules, allowing incoming traffic flows to be reclassified under current status.

This model allows to implement a dynamic, coherent and adaptable forwarding system, in which each flow is individually treated and processed, while at the same time maintaining a global vision about the state of the network.

5.4 EXPERIMENTAL ENVIRONMENT

The experimental environment constitutes the basis on top of which NEPTUNE's behavior is deployed and validated. Its objective is to reproduce, within a single Linux machine, a multihomed network composed by a client, a router, multiple ISPs, remote servers and a common core network. This approach allows to evaluate the system's behavior under controlled conditions, without relying on external infrastructure.

Contrary to purely abstract simulation, the environment uses actual Linux features, such as network namespaces, virtual interface pairs, forwarding tables, policy routing and NPTv6. Therefore, the evaluated behavior will be the one of a real Linux networking stack.

5.4.1 DEPLOYMENT AND TEARDOWN

The environment's deployment has been automated through declarative generation and orchestrated execution. The base configuration is defined in **config.yaml**, where the main elements are described: hosts, interfaces, prefixes, ISP roles and additional parameters.

From this configuration, **generator.py** creates the needed scripts to generate the laboratory. Among them are the namespace generation, addressing configuration, route installation, NPTv6 configuration and the failure and recovery events. This separation allows to modify the environment from the configuration without manually altering low-level commands.

The execution is centralized in **runner.py**, which acts as an entry point to the system. The "apply" command sequentially eliminates previous configurations, regenerates the scripts and executes them in order to create the environment and apply its configuration.

The same way, the "reset" command destroys the created namespaces and the associated virtual interfaces, effectively cleaning the system. This is particularly important in order to ensure the reproducibility of the experiments, avoiding that residual configurations affect later executions.

5.4.2 TOPOLOGY AND ADDRESSING

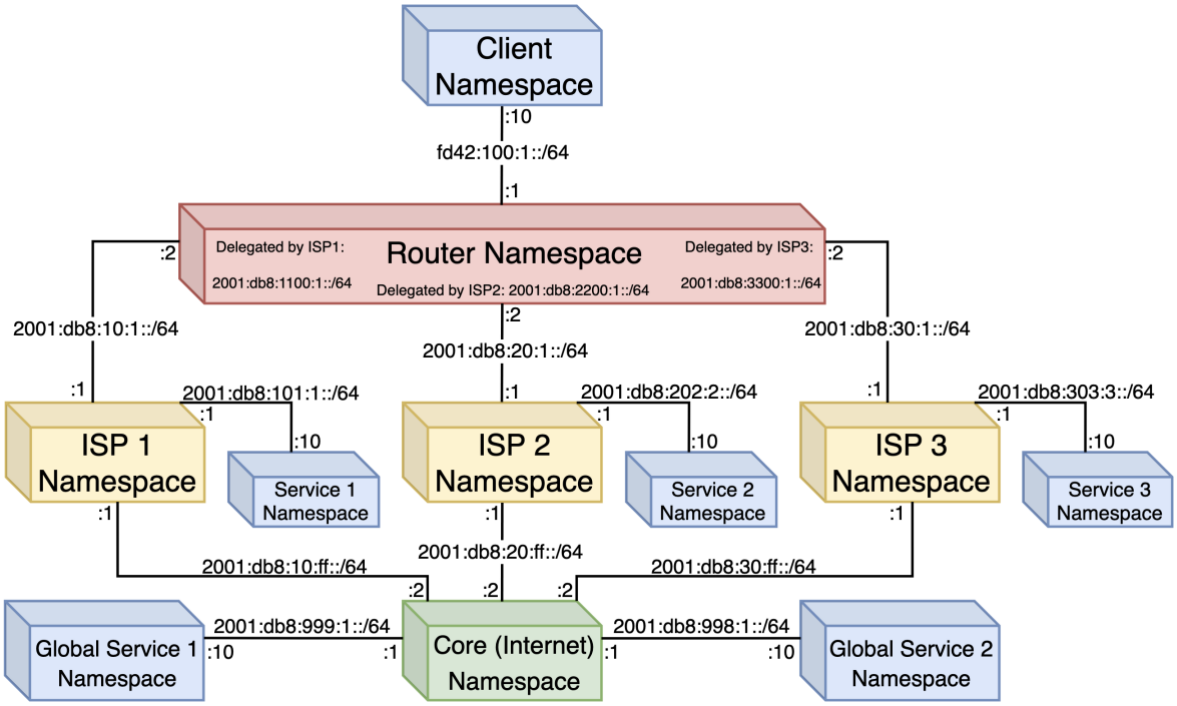


Figure 5.4.2.1.: Laboratory Environment Topology – Source: self-made

The implemented topology reproduces a multihomed network scenario in which a client accesses multiple external services through a NEPTUNE-controlled router, which has multiple WAN links, each of them connected to a different ISP.

The environment is organized around an internal LAN, where the client is placed. Each ISP has their own link towards the router, and can also provide access to specific remote services, which are single-homed, and therefore only accessible through a certain ISP.

In addition, the topology includes a core namespace, which represents a common intermediate network, equivalent to the Internet. Through such network, global services are reachable no matter which ISP the connection comes from.

Addressing is structured by separating the internal LAN prefix from the external ISP-delegated prefixes. This separation is key when using NPTv6, as we will be translating the internal prefix to the delegated one from the selected outbound ISP for each flow.

5.4.3 EVENT SYSTEM AND SCENARIO CONTROL

In order to evaluate the dynamic behavior of the system, the environment integrates an event triggering mechanism that allows to modify the state of the links during runtime. These events are automatically generated from the laboratory configuration and allow to simulate multiple ISP failures and recoveries.

Each event acts over the interfaces that connect the router with a certain provider. For example, a failure event disables the link both in the router's and the ISP's namespaces, while a recovery event enables those interfaces and restores the required routes.

This mechanism allows to reproduce typical scenarios in a multihomed network, such as the loss of a primary link, the switching to an alternative link or the later recovery of the preferred ISP. In addition, the integration with the NEPTUNE controller causes these changes to also affect the calculations for the active ISP pool, invalidating previously learnt flows that are no longer valid under the current state of the network.

That way, the event system constitutes a simple but effective tool for the validation of failover, recovery and dynamic adaption behaviors of the proposed model. Its main advantage is that tests are repeatable, controlled and always maintain the same physical base topology, altering only the connectivity conditions, i.e. the logical topology.

5.5 MONITORING AND VISUALIZATION LAYER

As development of the framework progressed, the need arose for mechanisms that would allow for a clearer and more accessible view of the system's internal behavior during the validation and experimentation phases.

The mechanisms implemented by NEPTUNE primarily operate on internal elements of the Linux network stack. Although all this behavior can be inspected using traditional terminal tools, direct observation of the system's operational state is complex and unintuitive, especially during dynamic failover and connectivity recovery scenarios.

For this reason, a lightweight monitoring and visualization layer was developed specifically for the framework's operational and experimental support. This layer allows for real-time observation of the system's internal state and facilitates the validation of NEPTUNE's dynamic behavior during the execution of the tests described later in the results chapter.

The objective of this component is not to implement a complete network management platform, but rather to provide a lightweight observability tool capable of clearly presenting the framework's main operational elements during the experimental environment.

5.5.1 OBJECTIVES

The primary motivation for developing this visualization layer stems from the distributed and dynamic nature of the mechanisms implemented by NEPTUNE.

The framework operates simultaneously on various components of the Linux system:

- Network namespaces.
- IPv6 routing.
- Policy routing.
- Dynamic traffic marking rules.
- NPTv6 prefix translation.
- Internal flow persistence tables.

During transition scenarios, such as connectivity failure or recovery events, multiple system elements change dynamically within short time intervals. Observing this behavior exclusively through a terminal and system logs hinders both experimental validation and the visual interpretation of the framework's internal operation.

Therefore, a lightweight operational visualization layer was designed specifically to:

- Monitor the system's operational status in real time.
- Facilitate experimental validation of the framework.
- Visualize the evolution of active routes and flows.
- Observe failover and recovery events.
- Simplify the interpretation of NEPTUNE's dynamic behavior.

Furthermore, this layer reduced the reliance on manual terminal tools during the validation and experimental demonstration phases, providing a clearer and more accessible representation of the system's behavior.

Figure 5.5.1.1. shows an example of the operational interface developed for the framework.

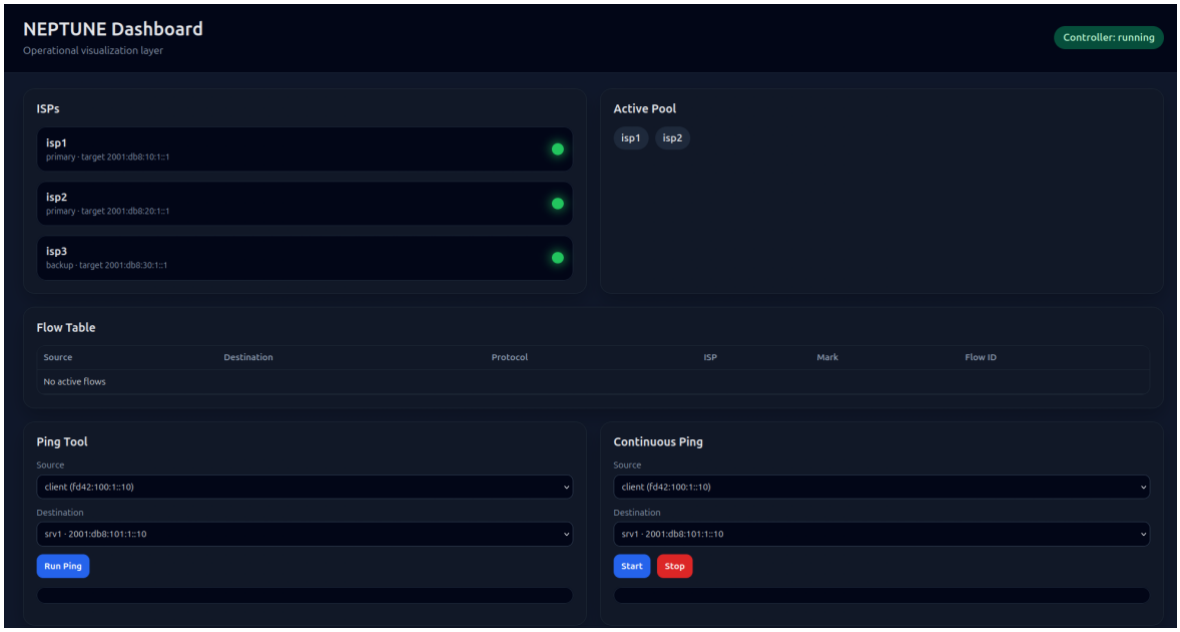


Figure 5.5.1.1.: NEPTUNE Dashboard – Source: self-made

5.5.2 ARCHITECTURE

The visualization layer architecture was designed using a decoupled approach from the framework's main control plane.

The NEPTUNE controller remains the sole component responsible for link monitoring, flow management, and routing decisions. The visualization layer does not directly intervene in the data plane nor does it participate in the internal route selection algorithms.

Instead, the controller periodically exports the system's internal state to a JSON file shared within the runtime environment. Subsequently, a lightweight dashboard server consumes this information and presents it via a web interface accessible during experimental testing.

The overall architecture can be summarized as follows:

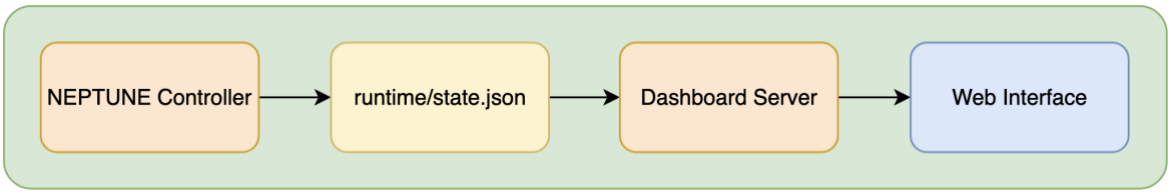


Figure 5.5.2.1.: NEPTUNE Visualization Architecture – Source: self-made

This separation allows for a clean, modular architecture, preventing the mixing of the controller's operational logic with visualization or graphical interface components.

Likewise, the dashboard does not directly modify the framework's internal behavior. Its primary function is to display information exported by the controller and to execute only predefined experimental actions, such as generating ICMPv6 traffic or simulating failure and recovery events using controlled scripts.

This approach simplifies the overall system design and reduces coupling between the framework's various components.

5.5.3 RUNTIME STATE EXPORT

To provide visibility into the system's internal behavior, the NEPTUNE controller implements a mechanism for periodically exporting its state using a JSON file shared within the runtime environment.

This file, named runtime/state.json, acts as a lightweight interface for exchanging information between the control plane and the visualization layer.

The main elements exported by the controller include:

- Operational status of the ISPs.
- Active pool of available links.
- Table of learned flows.
- Failure and recovery events.
- Temporal information on the system's internal state.

The exported structure allows for a simple and decoupled representation of the framework's dynamic behavior during the various validation phases.

Figure 5.5.3.1. shows a simplified snippet of the state exported by the controller.

```
{
  "isps": {
    "isp1": {
      "status": "up",
      "role": "primary"
    }
  },
  "active_pool": ["isp1", "isp2"],
  "flows": [],
  "events": []
}
```

Figure 5.5.3.1.: Simplified Exported State – Source: self-made

Using a JSON format also facilitates future system extensibility and simplifies integration with external observability or analytics tools.

5.5.4 VISUALIZATION FEATURES

The developed interface provides various monitoring and operational support mechanisms specifically tailored to the framework's experimental environment.

Key features include:

ISP Status Indicators

The interface displays the operational status of each ISP through real-time visual indicators, allowing for quick detection of connectivity degradation and recovery events.

It also displays additional information associated with each provider, including operational role, prefixes, and basic configuration parameters.

Dynamic flow table

The dashboard incorporates a representation of the active flow table managed by the controller, displaying:

- Source and destination addresses.
- Selected ISP.
- Applied routing marks.
- Internal flow identifiers.

This functionality allows for direct observation of the behavior of the persistence and dynamic route selection mechanism implemented by NEPTUNE.

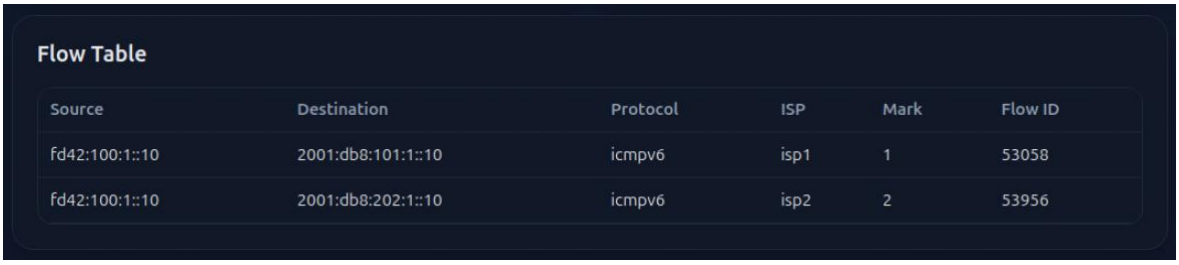


Figure 5.5.4.1.: NEPTUNE Flow Table – Source: self-made

Event Log

The interface also incorporates a system for visualizing operational events exported by the controller, including:

- Fault detection.
- Link recovery.
- Changes in the active pool.
- Internal system events.

This facilitates the temporal monitoring of the framework's dynamic behavior during experimental tests.

Traffic generation tools

To simplify validation tasks, the interface incorporates basic mechanisms for generating ICMPv6 traffic between different nodes. Implemented functionalities include:

- Execution of connectivity tests.
- Generation of continuous traffic.
- Execution of test suites for load balancing validation.

These tools were primarily used during the experimental phases described later in the validation chapter.

Event simulation controls

Finally, the interface allows for the execution of predefined link failure and recovery events through integration with the experimental scripts developed for the laboratory.

This allows for the dynamic reproduction of connectivity transition scenarios without the need for direct manual interaction with the terminal.

It is important to note that this interface should not be interpreted as a full-fledged network management application, but rather as a lightweight tool for operational support and experimental validation developed specifically for this work.

5.5.5 INTEGRATION WITH EXPERIMENTAL VALIDATION

The monitoring and visualization layer developed for NEPTUNE was used extensively during the project's experimental phase, particularly in the validation tests described later in the results analysis chapter.

The interface allowed for real-time observation of dynamic system behaviors, including:

- State changes in ISPs.
- Reconfiguration of the active pool.
- Traffic redistribution.
- Route persistence.
- Failover and recovery events.
- Dynamic prefix translation.

Furthermore, the integrated traffic generation tools facilitated the controlled execution of multiple experimental scenarios without relying exclusively on terminal interaction.

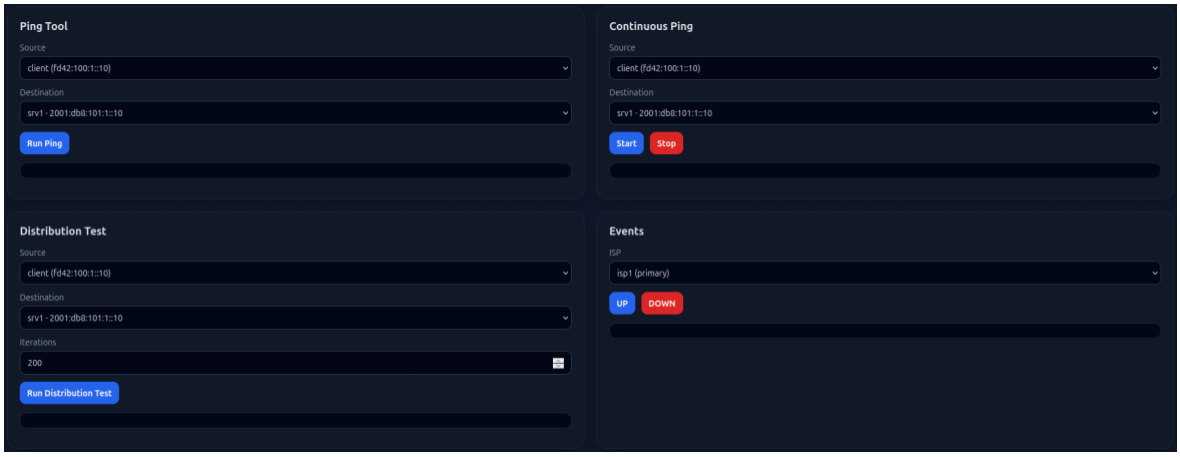


Figure 5.5.5.1.: Dashboard Traffic Generation Tools – Source: self-made

The integration of this observability layer significantly improved the framework's analysis and validation capabilities, providing a clearer and more accessible representation of NEPTUNE's internal operational behavior during the various phases of experimentation.

Chapter 6. RESULTS ANALYSIS & VALIDATION

This chapter presents the experimental validation of the NEPTUNE framework in the virtual environment developed throughout this work. The main objective of this phase is to verify the system's operational behavior in various IPv6 multihoming scenarios, evaluating both its performance under normal conditions and its ability to respond to failure events.

To this end, various experimental scenarios were designed to validate the objectives defined in the project's scope chapter, including traffic continuity, failover times, routing behavior, load balancing, and prefix management via NPTv6.

The results obtained are based on experimental metrics, controlled traffic generation, and observation tools developed specifically for this project, allowing for real-time visualization of link status, routing decisions, and the evolution of active flows within the framework.

6.1 VALIDATION METHODOLOGY

Once the experimental environment and the control plane described in previous chapters were implemented, the next step is validating the operative behavior of the developed system. For that purpose, a validation methodology is proposed in order to verify that the NEPTUNE framework is capable of managing IPv6 multihoming scenarios coherently, maintaining traffic continuity and applying the defined routing policies.

The validation was carried out over the virtual laboratory developed specifically for this work, based on Linux network namespaces, virtual links with veth interfaces and multiple simulated ISP domains. On top of this infrastructure, the different system components were deployed, including the data plane, NPTv6 translation mechanisms, routing policies and the NEPTUNE controller, which monitors and dynamically selects the outbound paths.

In order to evaluate the system's behavior under different operative conditions, multiple experimental scenarios that reproduce common situations in multihomed environments were defined. These scenarios include regular operative conditions, link failure events and connectivity recovery processes.

During the testing, different observation and traffic generation tools were used. Notable among these are the use of ICMPv6 via ping to generate continuous traffic, event scripts developed to simulate circuit failures and recoveries, and the operational visualization layer integrated into the framework, which enables real-time monitoring of ISP status, active flow tables, the set of available links, and events detected by the controller.

The validation methodology was structured around three main groups of objectives previously defined in the project scope chapter:

- Validation of operational scenarios under different network conditions.
- Measurement of parameters related to traffic continuity, recovery times, and the additional overhead introduced by the system.
- Verification of routing behavior and IPv6 prefix management using NPTv6.

Based on these objectives, the tests conducted allowed for the analysis of various aspects of NEPTUNE's behavior, including:

- Policy-based route selection.
- Weighted traffic balancing across multiple ISPs.
- Flow persistence and reassignment.
- Fault detection and automatic reconfiguration of the data plane.
- Dynamic recovery of previously degraded links.
- Consistent translation of IPv6 prefixes in multihomed scenarios.

Finally, the obtained results were saved as system snapshots, metric tables and direct observations from the internal state exported by the controller, allowing to have both a functional and an operative vision of the framework's behavior during the validation.

6.2 NORMAL OPERATION VALIDATION

Once the overall functionality of the experimental environment had been validated, we proceeded to analyze the framework's behavior under normal operating conditions. The primary objective of this phase is to verify that NEPTUNE is capable of correctly applying the defined routing policies, distributing traffic among multiple ISPs, and maintaining consistency in the assignment of active flows.

To this end, various tests were conducted using ICMPv6 traffic between the internal client and multiple remote services defined within the virtual laboratory. During these tests, all ISPs remained operational and available, allowing for the observation of the system's stable behavior and the control plane's operation under normal conditions.

The validations performed in this section focused primarily on three fundamental aspects:

- Strict route selection using static policies.
- Weighted traffic balancing across multiple available links.
- Flow persistence once the initial assignment has been made.

System behavior was observed using both traffic generation tools and the operational visualization interface developed for the framework, allowing real-time monitoring of ISP selection, active flow tables, and the internal state exported by the controller.

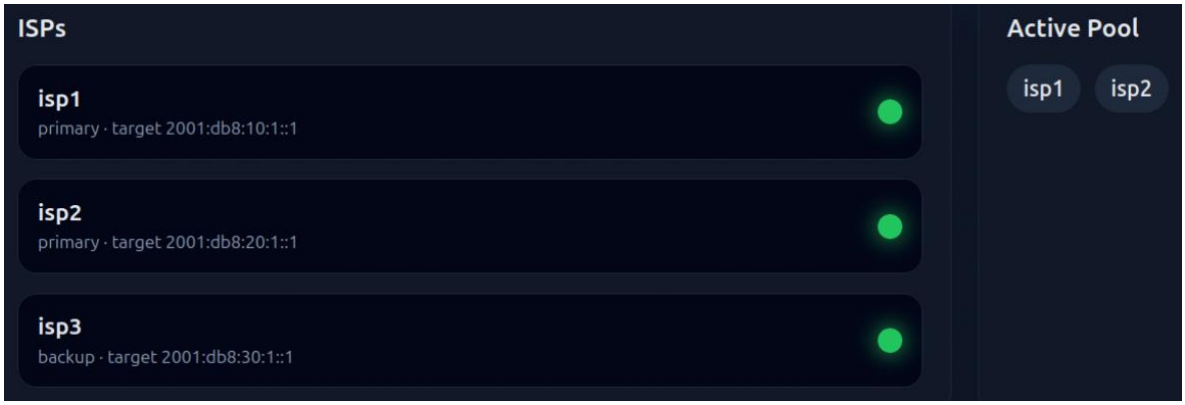


Figure 6.2.1.: Dashboard during normal operation – Source: self-made

6.2.1 STRICT ROUTE SELECTION

The first validation performed involved verifying the operation of the strict routing policies defined within the NEPTUNE configuration. These policies allow specific destination prefixes to be associated with a particular ISP, forcing traffic destined for those destinations to use exclusively the specified link as long as it remains available.

To this end, various remote services were configured, each individually associated with an ISP in the experimental environment. Each of these services uses a different IPv6 prefix, to which a strict policy was applied within the controller's YAML configuration.

During testing, ICMPv6 traffic was generated from the internal client to each of the defined remote services. The controller processed the newly detected flows and applied the corresponding policies, selecting the ISP previously associated with each prefix.

Examination of the flow table exported by the controller confirmed that the ISP selection matched the policies defined in the system configuration. Furthermore, the rules dynamically installed in the data plane correctly reflected the routing labels associated with each provider.

The results obtained show that the framework is capable of correctly applying deterministic route selection policies, ensuring that specific destinations consistently use specific links as long as they remain operational.

Flow Table

Source	Destination	Protocol	ISP	Mark	Flow ID
fd42:100:1::10	2001:db8:101:1::10	icmpv6	isp1	1	19078
fd42:100:1::10	2001:db8:202:1::10	icmpv6	isp2	2	19863
fd42:100:1::10	2001:db8:303:1::10	icmpv6	isp3	3	20640

Figure 6.2.1.1.: Strict Route Selection of Single-homed Services – Source: self-made

6.2.2 WEIGHTED LOAD BALANCING

Once the strict policies had been validated, we proceeded to evaluate the weighted load-balancing implemented by NEPTUNE for flows that do not have explicit routing rules.

In this mode of operation, the controller uses a selection algorithm based on flow hashing and weights associated with each ISP, allowing traffic to be distributed across multiple links in proportion to the capacity or priority configured for each provider.

To validate this behavior, multiple independent flows were generated toward global services not associated with strict policies. Each new flow was processed by the controller, which selected an exit ISP by considering the set of available links and the defined weights.

During the tests conducted, the primary ISPs were configured with differentiated weights, allowing for the observation of a non-uniform distribution of traffic according to those values. The results obtained showed a trend consistent with the defined balancing policy, assigning a greater number of flows to the ISP with the highest relative weight.

The monitoring interface developed allowed for real-time visualization of the distribution of active flows among providers, as well as the ISP selected for each new communication.

In this case, we configured 70% / 30% balancing weights, and we observed an effective 72,5% / 27,5% balancing.

This behavior confirms that NEPTUNE can implement basic traffic engineering mechanisms in multihomed IPv6 scenarios, distributing load across multiple links without the need for BGP.

Figure 6.2.2.1.: Balancing Distribution Test – Source: self-made

6.2.3 FLOW PERSISTENCE

In addition to the initial route selection, one of the fundamental aspects of the system is maintaining consistency in path assignment once a communication has been established.

In multihomed environments, the dynamic reassignment of routes during an active communication can cause continuity issues, especially in connection-oriented protocols. For this reason, NEPTUNE incorporates a flow persistence mechanism based on internal tracking tables and dynamic rules implemented on the data plane.

To validate this behavior, continuous communications were generated between the client and various remote services, observing the evolution of active flows within the table exported by the controller. Once an ISP was assigned to a specific flow, subsequent transmissions belonging to that communication continued to use the same path as long as the corresponding entry remained active.

During testing, the operation of the expiration and garbage collection mechanism implemented by the controller was also verified. After a configurable period of inactivity, old entries were automatically removed from the flow table, allowing new communications to be processed and dynamically redistributed.

The results obtained show that the framework maintains consistency in route selection for active communications, avoiding arbitrary ISP changes and providing stable behavior compatible with persistent traffic scenarios.

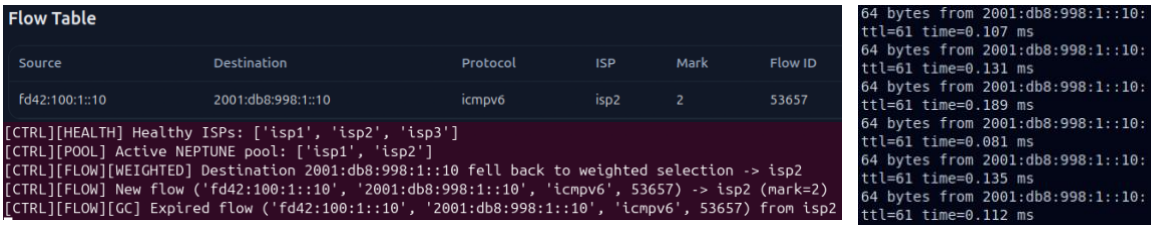


Figure 6.2.3.1.: Lifecycle of a Flow inside NEPTUNE – Source: self-made

6.3 CIRCUIT FAILURE VALIDATION

Once the system's behavior under normal operating conditions had been validated, we proceeded to analyze the framework's response to connectivity failure events. The main objective of this phase was to verify NEPTUNE's ability to detect degraded performance in available links, dynamically reconfigure the set of active ISPs, and automatically maintain traffic continuity.

To this end, various failure scenarios were designed in which specific ISP links were manually disabled using event scripts developed for the experimental environment. During these tests, continuous IPv6 traffic was maintained between the client and various remote services, allowing for real-time observation of the controller's reaction and the behavior of the data plane during the failover process.

Test monitoring was performed using the operational visualization layer integrated into the framework, as well as through direct observation of the flow table and the internal events exported by the controller.

Figure 6.3.1. shows an example of the system's initial state before a failure event occurred, with the primary ISPs operational and the active pool functioning under normal conditions.

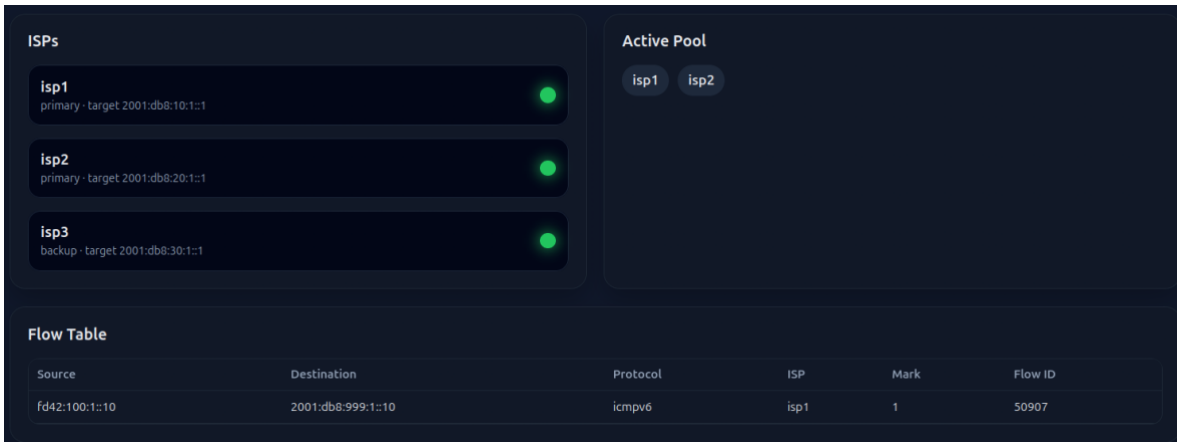


Figure 6.3.1.: System Before Circuit Failure – Source: self-made

6.3.1 FAILURE DETECTION

The first validation performed in this phase consisted of verifying the controller's ability to automatically detect a loss of connectivity on one of the available ISP links.

To do this, continuous traffic was generated from the internal client to various remote services while one of the main links was dynamically disabled using the lab's event scripts. The NEPTUNE controller periodically monitors the status of the links using active connectivity checks to the targets defined in the system configuration.

Once the failure occurred, the controller detected the loss of response associated with the affected ISP and automatically updated the system's internal status, marking that provider as unavailable. This transition could be observed both in the internal logs and in the monitoring interface developed for the framework.

Figure 6.3.1.1. shows when one of the ISPs goes into the "down" state after the failure event.

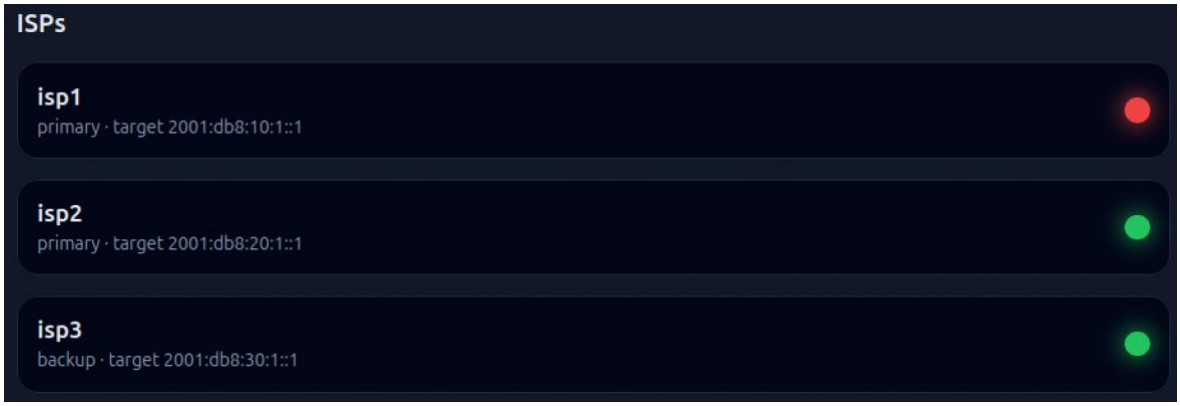


Figure 6.3.1.1.: Dashboard State after ISP1 failure – Source: self-made

During the tests conducted, the observed detection time depended primarily on the monitoring interval configured in the controller, remaining within margins compatible with basic failover scenarios in multihomed edge environments.

Furthermore, it was verified that failure detection did not require additional intervention or system reboot, allowing the framework to react automatically to state changes.

6.3.2 POOL RECONFIGURATION

After detecting a link failure, the next validation objective was to verify the system's ability to dynamically reconfigure the set of available ISPs for load balancing and routing flows.

Once the failure was identified, the controller automatically removed the degraded ISP from the active pool used for route selection. As part of this process, previously learned flows associated with the previous topology state were also cleared, preventing new communications from continuing to use paths that were no longer valid.

The reconfiguration of the pool could be observed in real time via the operational interface developed for the framework, where the affected ISP went into a down state and ceased to be part of the set of active links used by the system.

Figure 6.3.2.1. shows the dashboard status after the automatic removal of the degraded ISP.

Figure 6.3.2.1.: Active Pool and Flows after ISP1 Failure – Source: self-made

Subsequently, new communications generated by the client began to use exclusively the links that remained operational, allowing traffic to continue flowing without the need for external intervention.

In scenarios where only one of the primary ISPs was affected, the system continued to operate using the remaining available links. Likewise, the behavior of the backup mechanism was also validated, confirming that the ISPs defined as secondary could be automatically incorporated into the active pool when all primary links became unavailable.

Figure 6.3.2.2.: Failure of all Primary ISPs – Source: self-made

In both cases, we can observe that not only the degraded ISPs get removed from the active pool, but also the active flow is removed from the flow table, where a new flow pointing to an ISP from the new active pool is installed. The process can be observed in terminal:

```
[CTRL][HEALTH] isp1=down, isp2=up, isp3=up
[CTRL][HEALTH] Healthy ISPs: ['isp2', 'isp3']
[CTRL][POOL] Active NEPTUNE pool: ['isp2']
[CTRL][FLOW] Flushing 1 learned flow entries due to topology change
[CTRL][FLOW][WEIGHTED] Destination 2001:db8:999:1::10 fell back to weighted selection -> isp2
[CTRL][FLOW] New flow ('fd42:100:1::10', '2001:db8:999:1::10', 'icmpv6', 23354) -> isp2 (mark=2)
```

In addition, the controller follows the primary/backup hierarchy, in a way that it will not include backup-marked ISPs (ISP3 in this case) in the active pool unless all primary-marked ISPs (ISP1 and ISP2 in this case) have fallen.

The following table summarizes some of the failure scenarios evaluated during this validation phase:

Scenario	Degraded ISP	Initial Pool	Resulting Pool
F1	ISP1	ISP1+ISP2	ISP2
F2	ISP2	ISP1+ISP2	ISP1
F3	ISP1+ISP2	ISP1+ISP2	ISP3

Table 6.3.2.1.: Failure Scenarios Tested – Source: self-made

6.3.3 TRAFFIC RESTORATION

The final aspect analyzed during the failure tests involved evaluating the system's ability to maintain traffic continuity following the reconfiguration of the data plane.

To this end, continuous ICMPv6 communications were generated between the client and various remote services while failure events were induced on the ISP links. During these tests, traffic behavior was observed before, during, and after the failover process.

After the failure was detected and the active pool was updated, new communications automatically began using available alternative routes within the system. Depending on the exact moment of the topology change and the state of previously established flows, some tests showed temporary packet loss during the transition process. However, connectivity was automatically restored without manual intervention.

Figure 6.3.3.1. shows an example of continuous traffic during a failover event, where a brief interruption can be observed, followed by the automatic restoration via an alternative ISP.

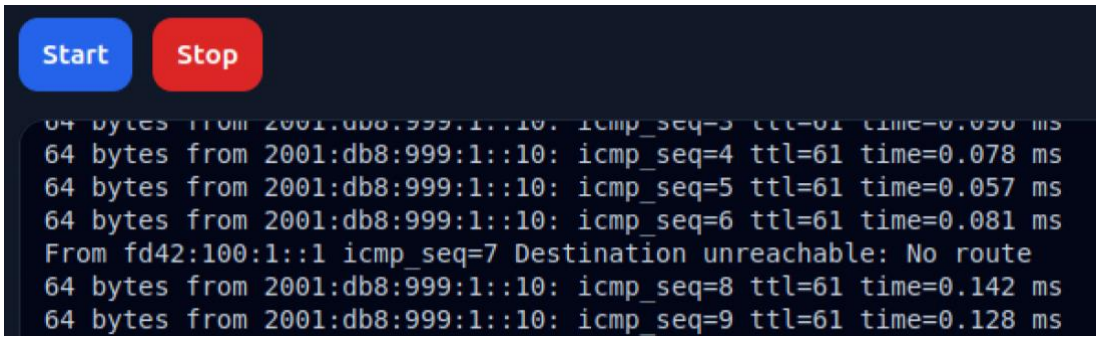


Figure 6.3.3.1.: Brief Interruption after an ISP Failure – Source: self-made

During the tests conducted, it was also verified that the new communications generated after the failure were correctly reassigned using only the ISPs available in the pool. The flow table exported by the controller consistently reflected this transition, showing how the new flows were dynamically associated with valid routes following the degradation of the original link.

The results show that NEPTUNE is capable of automatically reacting to connectivity failures, reconfiguring the data plane, and restoring traffic using alternative paths.

6.4 CIRCUIT RECOVERY VALIDATION

After analyzing the system's behavior in the event of failures, the next validation phase focused on evaluating NEPTUNE's ability to detect the restoration of connectivity and dynamically reintegrate previously degraded links into the active data plane.

In multihomed environments, it is not only important to react correctly to connectivity outages but also to consistently restore the use of recovered links without compromising the stability of existing traffic. For this reason, various tests were conducted to verify the controller's behavior during ISP circuit recovery processes.

We started with previously degraded scenarios where one or more ISPs had been removed from the active pool following failure events. Subsequently, the links were manually restored using recovery scripts developed for the experimental environment, allowing us to observe the complete process of detection, reintegration, and redistribution of flows.

These tests were monitored using continuous ICMPv6 traffic, flow table observation, and the operational visualization interface integrated into the framework.

Figure 6.4.1. shows an example of the system's initial state before initiating the recovery process for a degraded ISP:

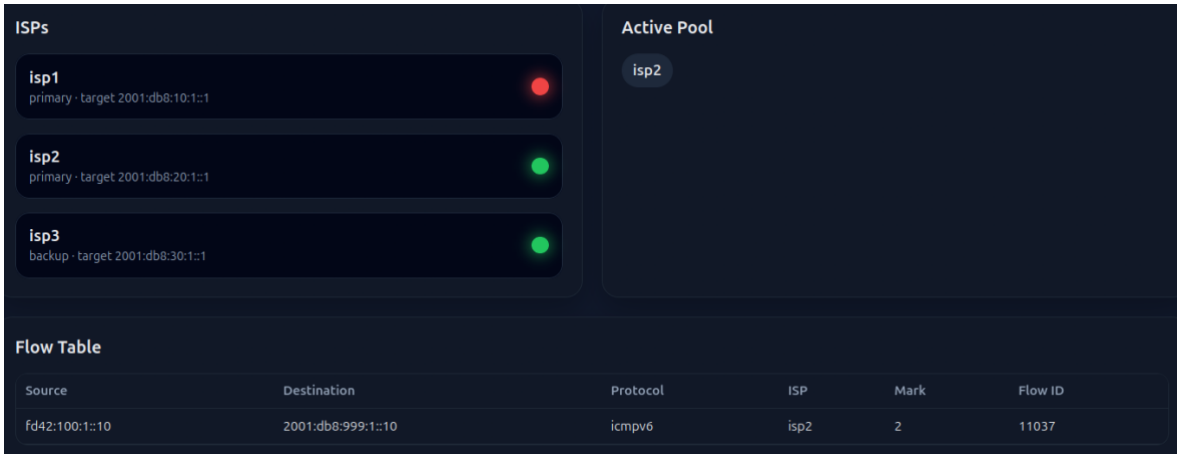


Figure 6.4.1.: Dashboard State before ISP Recovery – Source: self-made

6.4.1 RECOVERY DETECTION

The first validation performed in this phase consisted of verifying the controller's ability to automatically detect the restoration of connectivity for an ISP that had previously failed.

To do this, after inducing a failure on one of the main links, the circuit was restored using the recovery mechanisms defined in the experimental environment. The controller continued to perform the periodic connectivity checks configured for each ISP, allowing it to automatically detect the reappearance of the previously degraded link.

After the response from the monitoring targets was restored, the controller updated the system's internal state, marking the corresponding ISP as operational again. This transition could be observed both in the events exported by the controller and in the monitoring interface developed for the framework.

Figure 6.4.1.1. shows the moment when the recovered ISP reappears as available:

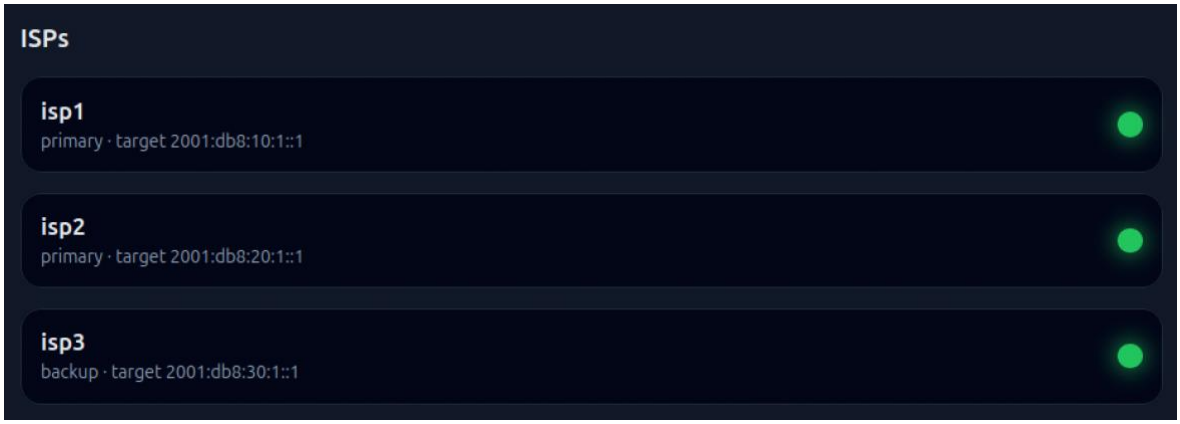


Figure 6.4.1.1.: Dashboard after ISP Recovery – Source: self-made

During the tests conducted, the observed detection time was primarily determined by the monitoring interval configured in the controller, remaining within values consistent with those previously observed during failure events.

Furthermore, it was verified that the recovery process did not require manually reconfiguring the routing environment, allowing the system to automatically recover the available link.

6.4.2 POOL REINTEGRATION

Once the link recovery was detected, the next validation objective was to verify the system's ability to dynamically reintegrate the recovered ISP into the set of active links.

After updating the ISP's internal status, the controller added that provider back to the selection pool used for load balancing and routing new traffic flows. This transition could be observed from the interface, where the recovered link once again became part of the set of ISPs available to the system. Figure 6.4.2.1. shows the dashboard after the reintegration:

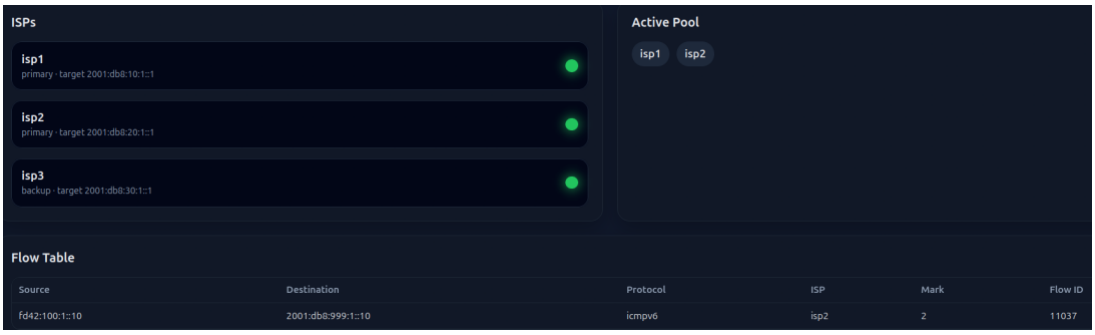


Figure 6.4.2.1.: Dashboard after Pool Reintegration – Source: self-made

During the tests conducted, it was verified that the link recovery did not cause global system reboots or unnecessary disruptions to already established communications. Instead, the controller maintained the stability of the data plane while dynamically updating the set of routes available for new communications.

Likewise, it was verified that ISPs defined as backups automatically left the active pool when previously degraded primary links became available again, restoring the system's original behavior. Table 6.2 summarizes some of the recovery scenarios evaluated during this phase.

Scenario	Recovered ISP	Previous Pool	Resulting Pool
R1	ISP1	ISP2	ISP1+ISP2
R2	ISP2	ISP1	ISP1+ISP2
R3	ISP1+ISP2	ISP3	ISP1+ISP2

Table 6.4.2.1.: Recovery Scenarios Tested – Source: self-made

6.4.3 FLOW REDISTRIBUTION AFTER RECOVERY

The final aspect analyzed during this phase involved evaluating the system's behavior once connectivity was restored and the recovered links were reincorporated into the active pool.

To this end, new communications were generated following the recovery of the previously degraded ISPs, observing the route selection performed by the controller and the resulting distribution of the new active flows.

The tests conducted showed that the new communications began to use the recovered links again in accordance with the policies and weights configured in the system. The distribution of new flows gradually returned to the original behavior defined prior to the failure. Figure 6.4.3.1. shows an example of flow redistribution after the recovery of a primary ISP:

Flow Table

Source	Destination	Protocol	ISP	Mark	Flow ID
fd42:100:1::10	2001:db8:999:1::10	icmpv6	isp2	2	11037
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp1	1	33264
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp1	1	33272
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp1	1	33280
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp1	1	33287
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp2	2	33296
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp2	2	33301

Figure 6.4.3.1.: Load Balancing after ISP Recovery – Source: self-made

It was also verified that communications previously established before recovery maintained consistency in the assigned route as long as their entries remained active within the flow table. In this way, NEPTUNE avoids arbitrary modifications to already established traffic, applying changes only to new communications or expired flows.

During testing, it was also observed that, following the natural expiration of old entries, new flows could be redistributed using the full set of available load balanced ISPs.

The results obtained show that NEPTUNE is capable of managing connectivity recovery processes automatically and consistently, reincorporating previously degraded links into the data plane without compromising the overall stability of the multihomed environment.

6.5 PERFORMANCE MEASUREMENTS

In addition to validating the framework's functional behavior under different operational scenarios, it is also necessary to analyze the system's impact from a temporal and operational perspective. For this reason, during the experimental phase, various measurements were taken to evaluate the controller's response time, traffic continuity during transition events, and the potential overhead introduced by the mechanisms implemented in the control plane.

The tests described in this section were conducted using continuous ICMPv6 traffic between different nodes in the experimental environment, combining the generation of controlled events, monitoring of the internal state exported by the controller, and direct observation of traffic behavior during network transitions.

6.5.1 FAILOVER AND RECOVERY TIME

The first set of measures focused on evaluating the times associated with fault detection and connectivity recovery within the framework.

To do this, continuous ICMPv6 traffic was generated between the client and various remote services while controlled ISP link failure and restoration events were triggered. In order to get a precise measurement of downtime, pings with a cadence of 1000 packets per second (pps) were generated, allowing to measure recovery time with a precision of one millisecond.

```
64 bytes from 2001:db8:999:1::10: icmp_seq=584 ttl=61 time=0.022 ms
64 bytes from 2001:db8:999:1::10: icmp_seq=585 ttl=61 time=0.022 ms
From fd42:100:1::1 icmp_seq=586 Destination unreachable: No route
From fd42:100:1::1 icmp_seq=587 Destination unreachable: No route
From fd42:100:1::1 icmp_seq=595 Destination unreachable: No route
64 bytes from 2001:db8:999:1::10: icmp_seq=605 ttl=61 time=0.136 ms
64 bytes from 2001:db8:999:1::10: icmp_seq=606 ttl=61 time=0.030 ms
64 bytes from 2001:db8:999:1::10: icmp_seq=607 ttl=61 time=0.021 ms
```

Figure 6.5.1.1.: Path Switching after Circuit Failure – Source: self-made

We simulated a failure in ISP1 while a communication was occurring between the client and a remote service through such ISP. As shown in figure 6.5.1.1., there are only 3 packets lost after the ISP went down, meaning that recovery time is only about 3 milliseconds.

Additionally, tests were conducted to recover previously degraded links, verifying that the system was capable of automatically reincorporating restored ISPs into the available routes.

```
64 bytes from 2001:db8:999:1::10: icmp_seq=3921 ttl=61 time=0.027 ms
64 bytes from 2001:db8:999:1::10: icmp_seq=3922 ttl=61 time=0.026 ms
From fd42:100:1::1 icmp_seq=3923 Destination unreachable: No route
From fd42:100:1::1 icmp_seq=3924 Destination unreachable: No route
From fd42:100:1::1 icmp_seq=3925 Destination unreachable: No route
From fd42:100:1::1 icmp_seq=3926 Destination unreachable: No route
From fd42:100:1::1 icmp_seq=3927 Destination unreachable: No route
From fd42:100:1::1 icmp_seq=3928 Destination unreachable: No route
64 bytes from 2001:db8:999:1::10: icmp_seq=3930 ttl=61 time=0.077 ms
64 bytes from 2001:db8:999:1::10: icmp_seq=3931 ttl=61 time=0.017 ms
```

Figure 6.5.1.2.: Path Switching after Circuit Recovery – Source: self-made

As we can observe, after the recovery of ISP1, the switching loss is 6 packets, which is equivalent to a switching time of 6 milliseconds at the 1 packet per millisecond testing rate.

The observed asymmetry between failover and recovery times is consistent with the operational behavior of the framework. Failure events immediately invalidate active paths, whereas recovery events require reintegration of the restored ISP into the active pool and subsequent redistribution of new flows.

The results show that the framework is capable of automatically responding to topology changes within timeframes compatible with basic edge multihoming scenarios, maintaining operational continuity of traffic even during transition processes.

6.5.2 TRAFFIC CONTINUITY

Another key aspect analyzed during the experimental validation was the system's ability to maintain traffic continuity during failure events and connectivity recovery.

To evaluate this behavior, continuous ICMPv6 transmissions were used between different nodes in the laboratory while dynamically altering the status of available ISPs. These tests allowed for direct observation of traffic behavior before, during, and after failover processes.

Under normal operating conditions, communications remained stable using the ISP initially assigned to each flow. After a failure event occurred, the controller automatically detected the link degradation and rerouted traffic to available alternative paths from the active pool.

During the tests, it was observed that traffic loss was typically limited to the time interval between failure detection and the effective reconfiguration of the data plane. Once this process was complete, connectivity was automatically restored using another available ISP.

Source	Destination	Protocol	ISP	Mark	Flow ID
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp1	1	34711

Source	Destination	Protocol	ISP	Mark	Flow ID
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp2	2	34711

Source	Destination	Protocol	ISP	Mark	Flow ID
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp1	1	34711

Figure 6.5.2.1.: Traffic Continuity after Failure and Recovery – Source: self-made

Furthermore, it was verified that new communications generated after the event correctly utilized the system's updated links, while flow persistence mechanisms prevented arbitrary changes to previously established traffic, and reconducted traffic traversing an alternative circuit back to the original path once such path was recovered, as shown in figure 6.5.2.1.

The results obtained show that NEPTUNE is capable of maintaining operational continuity of traffic even during scenarios of partial connectivity degradation, minimizing the disruption perceived by active communications.

6.5.3 ADDED OVERHEAD AND LATENCY

The latest set of measures focused on analyzing the potential impact of the framework on latency and overall traffic behavior within the experimental environment.

To this end, various tests were conducted comparing ICMPv6 transmissions in scenarios with and without active intervention by the NEPTUNE controller, monitoring basic metrics such as Round Trip Time (RTT), packet loss, and communication stability.

Since the framework operates primarily through initial route selection and dynamic application of routing rules, the observed impact on traffic under steady-state conditions was minimal. Once a flow is established within the data plane, subsequent forwarding is performed directly using native Linux kernel mechanisms, avoiding continuous processing within the controller.

The measurements showed that the system introduces only a small overhead associated with the initial control plane processing. For a strict routing policy destination, in which the process consists of installing the route, this translates in approximately 0.3ms added latency.

After a second consecutive battery of pings, the initial processing of the first packet is reduced, due to the fact that a route towards such destination is cached in the flow table.

```
root@lucas-NEPTUNE:/home/lucas/NEPTUNE# sudo ip netns exec client ping -6 -c 3 2001:db8:101:1::10
PING 2001:db8:101:1::10 (2001:db8:101:1::10) 56 data bytes
64 bytes from 2001:db8:101:1::10: icmp_seq=1 ttl=62 time=0.299 ms
64 bytes from 2001:db8:101:1::10: icmp_seq=2 ttl=62 time=0.098 ms
64 bytes from 2001:db8:101:1::10: icmp_seq=3 ttl=62 time=0.116 ms

--- 2001:db8:101:1::10 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2027ms
rtt min/avg/max/mdev = 0.098/0.171/0.299/0.090 ms
root@lucas-NEPTUNE:/home/lucas/NEPTUNE# sudo ip netns exec client ping -6 -c 3 2001:db8:101:1::10
PING 2001:db8:101:1::10 (2001:db8:101:1::10) 56 data bytes
64 bytes from 2001:db8:101:1::10: icmp_seq=1 ttl=62 time=0.091 ms
64 bytes from 2001:db8:101:1::10: icmp_seq=2 ttl=62 time=0.109 ms
64 bytes from 2001:db8:101:1::10: icmp_seq=3 ttl=62 time=0.074 ms

--- 2001:db8:101:1::10 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2076ms
rtt min/avg/max/mdev = 0.074/0.091/0.109/0.014 ms
```

Figure 6.5.3.1.: Measured RTT Towards a Single-Homed Service – Source: self-made

The processing time becomes higher when the destination is a service without any strict policy routing applied. In such cases, the traffic falls to the balancing pool, and a flow hash needs to be calculated to decide which ISP the traffic will traverse. This added additional processing causes the loss of the first packet, after which communication is established.

Again, after a second consecutive battery of pings, the initial processing of the first packet is reduced, due to the fact that a route towards such destination is cached in the flow table.

```
root@lucas-NEPTUNE:/home/lucas/NEPTUNE# sudo ip netns exec client ping -6 -c 3 2001:db8:999:1::10
PING 2001:db8:999:1::10 (2001:db8:999:1::10) 56 data bytes
From fd42:100:1::1 icmp_seq=1 Destination unreachable: No route
64 bytes from 2001:db8:999:1::10: icmp_seq=2 ttl=61 time=0.205 ms
64 bytes from 2001:db8:999:1::10: icmp_seq=3 ttl=61 time=0.083 ms

--- 2001:db8:999:1::10 ping statistics ---
3 packets transmitted, 2 received, +1 errors, 33.3333% packet loss, time 2047ms
rtt min/avg/max/mdev = 0.083/0.144/0.205/0.061 ms
root@lucas-NEPTUNE:/home/lucas/NEPTUNE# sudo ip netns exec client ping -6 -c 3 2001:db8:999:1::10
PING 2001:db8:999:1::10 (2001:db8:999:1::10) 56 data bytes
64 bytes from 2001:db8:999:1::10: icmp_seq=1 ttl=61 time=0.095 ms
64 bytes from 2001:db8:999:1::10: icmp_seq=2 ttl=61 time=0.092 ms
64 bytes from 2001:db8:999:1::10: icmp_seq=3 ttl=61 time=0.091 ms

--- 2001:db8:999:1::10 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2074ms
rtt min/avg/max/mdev = 0.091/0.092/0.095/0.001 ms
```

Figure 6.5.3.2.: Measured RTT Towards a Load-Balanced Service – Source: self-made

In addition, the RTTs with and without the controller active were compared:

	Minimum RTT	Average RTT	Maximum RTT
Base Linux Routing	0.092	0.479	3.069
Active NEPTUNE	0.061	0.119	0.350

Table 6.5.3.1.: Measured RTTs with and without NEPTUNE – Source: self-made

Overall, the results indicate that the framework has a minimal impact on overall traffic performance within the experimental environment, in fact, the measured RTTs with NEPTUNE on are lower, but remain within the noise error of a laboratory network.

This allows for the incorporation of dynamic multihoming and failover mechanisms without causing significant degradation under normal operating conditions.

6.6 ROUTE AND PREFIX VALIDATION

In addition to analyzing the system's dynamic behavior in the event of failures and recoveries, it is also necessary to verify that the routing decisions and prefix management mechanisms implemented by NEPTUNE function consistently within the environment.

This validation phase focused primarily on verifying two fundamental aspects:

- The behavior of the route selection policies applied by the controller.
- The correct operation of IPv6 prefix translation mechanisms via NPTv6.

To this end, various connectivity tests and data plane observations were performed using ICMPv6 traffic, flow tables exported by the controller, and direct monitoring of the active rules installed on the Linux environment.

6.6.1 ROUTE BEHAVIOR VERIFICATION

The first set of validations focused on verifying that the routing decisions made by the framework matched the policies defined in the system configuration.

To this end, communications were generated to various remote services with different route selection modes, including strict policies and weighted scenarios. Subsequently, the entries generated within the flow table exported by the controller were analyzed.

For destinations associated with strict policies, traffic was consistently routed through the ISP previously defined in the YAML file. During the tests conducted, no deviations from the expected policies were observed as long as the corresponding links remained operational.

On the other hand, in the weighted load-balancing scenarios, new communications were dynamically distributed among the active ISPs according to the weights configured for each provider. This behavior could be verified both through direct observation of the flow tables and via the validation tools integrated into the operational interface developed for the framework. Figure 6.6.1.1. shows an example of route distribution observed during testing:

Source	Destination	Protocol	ISP	Mark	Flow ID
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp1	1	6440
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp2	2	6661
fd42:100:1::10	2001:db8:998:1::10	icmpv6	isp1	1	6895
fd42:100:1::10	2001:db8:101:1::10	icmpv6	isp1	1	7373
fd42:100:1::10	2001:db8:202:1::10	icmpv6	isp2	2	7694
fd42:100:1::10	2001:db8:303:1::10	icmpv6	isp3	3	7970

Figure 6.6.1.1.: Route Distribution Towards Multiple Destinations – Source: self-made

Table 6.6.1.1. summarizes some of the route validation scenarios performed:

Destination	Policy	Expected ISP	Observed ISP	Result
Srv1	strict	ISP1	ISP1	Correct
Srv2	strict	ISP2	ISP2	Correct
Srv3	strict	ISP3	ISP3	Correct
Srv-global-bal	Balanced	ISP1/ISP2	ISP1/ISP2	Correct

Table 6.6.1.1.: Observed Route Behavior – Source: self-made

Likewise, the system's behavior was also verified during dynamic topology changes, observing how the controller automatically removed degraded ISPs from the set of available routes and reassigned new communications using only valid links.

Destination	ISPs' State	Expected Route	Observed Route	Result
Srv1	ISP1 Down	No Connection	No Connection	Correct
Srv2	ISP2 Down	No Connection	No Connection	Correct
Srv3	ISP3 Down	No Connection	No Connection	Correct
Srv-global-bal	ISP1&2 Down	Via ISP3	Via ISP3	Correct

Table 6.6.1.2.: Observed Route Behavior During Circuit Failure – Source: self-made

The results show that the framework correctly applies the path selection policies defined within the system, ensuring consistency between the declarative configuration and the actual behavior observed in the data plane.

6.6.2 PREFIX MANAGEMENT VERIFICATION

The second set of validation tests focused on verifying the operation of the IPv6 prefix translation and management mechanisms implemented via NPTv6 within the framework.

As described in previous chapters, NEPTUNE uses prefix translation to enable the use of Provider-Aggregatable addressing in multihomed scenarios without resorting to traditional BGP-based mechanisms. To do this, the system dynamically applies NPTv6 rules associated with the various ISPs available within the experimental environment.

During the tests conducted, it was verified that traffic from the internal network was correctly translated using the external prefix corresponding to the ISP selected. Likewise, it was verified that the translations remained consistent throughout the lifetime of each flow.

```
17:25:32.296958 IP6 fd42:100:1::10 > 2001:db8:998:1::10: ICMP6, echo request, id 51869, seq 4, length 64
17:25:32.297068 IP6 2001:db8:998:1::10 > fd42:100:1::10: ICMP6, echo reply, id 51869, seq 4, length 64
```

Figure 6.6.2.1.: Packet Captured before Translation – Source: self-made

```
17:25:32.297030 IP6 2001:db8:1100:1:bf89::10 > 2001:db8:998:1::10: ICMP6, echo request, id 51869, seq 4, length 64
17:25:32.297058 IP6 2001:db8:998:1::10 > 2001:db8:1100:1:bf89::10: ICMP6, echo reply, id 51869, seq 4, length 64
```

Figure 6.6.2.2.: Packet Captured after Translation – Source: self-made

Direct observation of the rules installed on the Linux system confirmed that the associations between internal and external prefixes matched the configuration defined for each provider, as shown in figures 6.6.2.1. and 6.6.2.2., and in the following summary table:

ISP	Internal Prefix	External Prefix	Observed Translation
ISP1	fd42:100:1::/64	2001:db8:1100:1::/64	Correct
ISP2	fd42:100:1::/64	2001:db8:2200:1::/64	Correct
ISP3	fd42:100:1::/64	2001:db8:3300:1::/64	Correct

Table 6.6.2.1.: NPTv6 Prefix Translation – Source: self-made

The results show that NEPTUNE is capable of correctly managing the dynamic translation of IPv6 prefixes in multihomed edge scenarios, ensuring consistency between routing decisions and the translation mechanisms applied to the traffic.

6.7 VALIDATION SUMMARY

Throughout this chapter, we have conducted experimental validation of the NEPTUNE framework using the virtual environment developed specifically for this work. The tests performed allowed us to evaluate both the system's functional behavior and its ability to respond to dynamic topology changes in multihomed IPv6 scenarios.

The results show that the framework is capable of correctly managing multiple ISP links using route selection, traffic balancing, and automatic connectivity recovery mechanisms without resorting to traditional inter-domain routing protocols such as BGP.

During normal operation tests, the functioning of the implemented routing policies was verified, with consistent behavior observed in both strict route selection scenarios and situations involving weighted load balancing among multiple active ISPs. Furthermore, the system demonstrated the ability to maintain flow persistence and communication stability once the initial path assignment was completed.

Validations related to failure and recovery events verified the controller's ability to automatically detect state changes on available links, dynamically reconfigure the active ISP pool, and restore traffic using alternative routes within the multihomed environment.

From a performance standpoint, the measurements showed that the framework's impact on traffic under steady-state conditions is minimal, as continuous packet processing remains primarily delegated to the data plane implemented on the Linux kernel. The controller intervenes only during the initial detection of new flows, topology changes, and periodic monitoring operations.

Likewise, route and prefix validation tests confirmed that the system maintains consistency between the policies defined in the declarative configuration and the behavior observed in the data plane, including the correct application of NPTv6 mechanisms on the various prefixes associated with each ISP.

Table 6.7.1. summarizes the final status of the validation objectives previously defined in the project definition chapter, specifically in subsection 4.2.2:

Validation Objective	Result
Behavior under normal operative conditions	**Validated**
Behavior during circuit failure events	**Validated**
Behavior after circuit recovery events	**Validated**
Failover and recovery time measurement	**Validated**
Traffic continuity verification	**Validated**
Overhead and latency measurement	**Validated**
Correct route behavior verification	**Validated**
Correct prefix management verification	**Validated**

Table 6.7.1.: Validation Objectives Summary – Source: self-made

Overall, the results obtained lead to the conclusion that NEPTUNE provides a functional operational model for IPv6 edge multihoming scenarios based on Provider-Aggregatable addressing, integrating mechanisms for dynamic route selection, automatic failover, and prefix management within a unified and experimentally validated architecture.

Chapter 7. CONCLUSIONS & FUTURE WORKS

This paper addresses the issue of IPv6 multihoming in edge environments using Provider-Aggregatable addressing, exploring existing operational limitations and proposing an alternative model based on native mechanisms of the Linux network stack.

The development of NEPTUNE has enabled the integration of various components—such as policy routing, flow persistence, dynamic route selection, connectivity monitoring, and prefix translation via NPTv6 within a unified and experimentally validated architecture. All of this has been achieved without resorting to traditional inter-domain routing protocols like BGP, while maintaining a modular approach tailored to small and medium scale multihomed edge scenarios.

In addition to the conceptual proposal, the work has included the design and implementation of a complete experimental environment capable of reproducing different operational scenarios, including traffic balancing, connectivity failure events, and dynamic link recovery. The validation performed has made it possible to verify the framework's functional behavior and analyze its viability from an operational standpoint.

This chapter summarizes the main conclusions drawn during the project's development, evaluates the degree to which the initially set objectives were met, and presents possible avenues for future evolution of the framework and the proposed model.

7.1 GENERAL CONCLUSIONS

This work has demonstrated the feasibility of an operational model for IPv6 edge multihoming based on Provider-Aggregatable addressing and built on native mechanisms of the TCP/IP protocol stack. Throughout the project, a framework capable of managing multiple ISP links using policy routing, dynamic route selection, flow persistence, and prefix translation via NPTv6 was designed, implemented, and experimentally validated.

One of the key observations made during the project's development is that many of the mechanisms required to implement IPv6 multihoming already exist within the Linux ecosystem and the TCP/IP stack itself. However, these mechanisms typically appear in isolation, distributed across different tools and specific configurations, making it difficult to integrate them into a coherent and reusable operational model. In this context, NEPTUNE has enabled the exploration of a unified approach targeted at medium scale environments.

The implementation has demonstrated that it is possible to build functional mechanisms for failover, traffic balancing, and communication persistence without resorting to traditional BGP-based solutions, using instead techniques such as policy routing and IPv6 prefix translation. Likewise, experimental validation has confirmed that the system is capable of automatically reacting to dynamic topology changes, maintaining operational continuity of traffic and adapting the behavior of the data plane based on the status of the available links.

Another relevant aspect of the work has been the separation between the control plane, data plane, and observability layer. This modular architecture has allowed for a relatively decoupled design, facilitating both experimental validation and the future evolution of the framework. The incorporation of a lightweight operational visualization layer has also significantly contributed to improving the observation and understanding of the system's internal behavior during the tests conducted.

From an experimental standpoint, the developed environment has enabled the reproduction of scenarios representative of normal operation, failure, and connectivity recovery within a fully virtualized laboratory based on Linux network namespaces. Although this environment does not represent all the complexities present in real production networks, it has allowed for the validation of the framework's functional behavior and the verification of the proposed model's consistency under different operational conditions.

Overall, the results obtained show that NEPTUNE constitutes a functional and technically viable approach for IPv6 edge multihoming scenarios based on Provider-Aggregatable addressing, integrating dynamic routing mechanisms, flow persistence, and prefix management within a unified experimental architecture.

7.2 ACHIEVED OBJECTIVES

The objectives set at the start of the project have been successfully achieved through the design, implementation, and experimental validation of the NEPTUNE framework. This work has not only developed a conceptual proposal for IPv6 edge multihoming scenarios but has also allowed for the implementation of that proposal within a functional environment.

First, we have successfully designed a modular architecture capable of integrating various routing and connectivity management mechanisms within a fully virtualized Linux environment. The developed framework allows for the combination of policy routing, flow persistence, dynamic route selection, and IPv6 prefix translation via NPTv6.

Likewise, the objective of implementing dynamic load balancing and failover mechanisms using multiple ISP links within multihomed scenarios has been achieved. During the tests conducted, the system has proven capable of automatically detecting connectivity degradation events and adapting the behavior of the data plane using only the links available.

Another key objective addressed during this work was the validation of flow persistence and routing consistency mechanisms. The implementation of dynamic tables has enabled the maintenance of stability in active communications, preventing arbitrary path changes.

The feasibility of using NPTv6 as a prefix management mechanism within multihomed scenarios based on Provider-Aggregatable addressing has also been validated. The tests conducted have verified that prefix translations are applied consistently based on the ISP selected for each flow, maintaining consistency between control and data planes.

A laboratory environment capable of reproducing normal operation, failure, and connectivity recovery scenarios has been developed in Linux, enabling the validation of the framework's functional behavior under different operational conditions. In addition, an observability and monitoring layer has been incorporated to facilitate the interpretation of the system behavior.

Overall, the results obtained show that the initially set objectives have been successfully achieved, allowing for the validation of the technical feasibility of the proposed model.

7.3 MAIN CONTRIBUTIONS

Beyond the experimental implementation developed during the project, one of the main contributions of this work has been the definition of a unified operational model for IPv6 edge multihoming scenarios based on Provider-Aggregatable addressing.

As discussed in the state-of-the-art review, many of the mechanisms needed to implement this type of environment already exist within the Linux ecosystem and the TCP/IP stack itself. However, these capabilities are often scattered across different tools, manual configurations, and solutions that are highly dependent on each specific deployment. In this context, NEPTUNE proposes an integrated approach that combines dynamic routing, flow persistence, connectivity monitoring, and IPv6 prefix translation in a unified architecture.

Another significant contribution of this work has been the experimental validation of IPv6 multihoming mechanisms without resorting to classic inter-domain routing protocols such as BGP. The developed framework demonstrates that it is possible to implement functional load balancing and failover mechanisms using only native capabilities of the Linux stack, relying on techniques such as policy routing and NPTv6 to manage multiple providers.

Likewise, the project also provides a practical approach to communication persistence within dynamic multihomed scenarios. The incorporation of flow tracking tables and steering mechanisms allows for maintaining stability in active communications and reducing arbitrary route changes during transition events, providing more consistent behavior from an operational standpoint.

From an architectural standpoint, another significant contribution of this work has been the explicit separation between the control plane, data plane, and observability layer. This division has enabled the development of a relatively decoupled environment where the controller manages dynamic routing decisions while the actual forwarding continues to be performed directly by the Linux kernel using native routing mechanisms and policy rules.

The incorporation of a lightweight operational visualization layer also constitutes a significant contribution within the project's experimental context. Although its primary objective was not to develop a complete management platform, this layer has significantly facilitated the interpretation and validation of the system's internal behavior during testing, improving the observability of elements traditionally difficult to analyze via terminal alone.

Furthermore, the work also provides a fully reproducible experimental environment based on Linux network namespaces and automatic configuration generation via YAML. This allows for the partial decoupling of the logical definition of scenarios from the specific laboratory implementation, facilitating both test automation and the future evolution of the framework.

Overall, the contributions made are not limited solely to the specific implementation developed during the project, but also extend to the definition and validation of an alternative operational approach for IPv6 edge multihoming scenarios, exploring mechanisms that have traditionally received less attention compared to models based exclusively on BGP.

7.4 LIMITATIONS

Despite the results obtained and the functional validation carried out during the project's development, this work has several limitations stemming both from the initially defined scope and from the experimental nature of the framework itself.

The most obvious limitation is that all validation was performed within a virtualized environment based on Linux network namespaces. Although this approach has allowed us to build a flexible, reproducible, and sufficiently realistic lab to validate the system's functional behavior, there remains a significant difference compared to real production environments. Aspects such as physical latencies, network congestion, heterogeneous hardware, real WAN links, or the behavior of commercial operators were not considered in the tests conducted.

Likewise, the number of ISPs and scenarios evaluated has been deliberately limited in order to maintain a manageable and comprehensible experimental environment. Although the framework has been designed following a potentially extensible modular architecture, the validations performed have been limited to relatively small configurations with a small number of links and routing policies.

Another significant limitation is that much of the testing focused primarily on ICMPv6 traffic and relatively controlled scenarios. Although this allowed for the proper validation of the framework's behavior from a functional standpoint, no comprehensive analyses were conducted on real application protocols, long-duration persistent TCP traffic, jitter-sensitive services, or sustained high-load scenarios.

Similarly, the controller implemented during the project has a deliberately limited scope and is geared toward experimental validation. Although it incorporates functional mechanisms for monitoring, flow persistence, and dynamic route reconfiguration, it was not designed as a distributed platform or as a system ready for large-scale production deployments. Aspects such as controller fault tolerance, distributed synchronization, advanced authentication, or integration with external orchestration systems fall outside the scope of the work performed.

It is also important to note that the proposed solution does not aim to completely replace traditional BGP-based architectures within large networks operated at carrier scale. The proposed approach is primarily geared toward small- and medium-scale multihomed edge scenarios where the operational and administrative complexity of BGP may prove excessive compared to lighter alternatives based on policy routing and prefix translation.

On the other hand, although the framework implements basic flow balancing and persistence mechanisms, no advanced dynamic optimization algorithms based on link quality, congestion, latency, or SLA metrics have been developed. Routing decisions made during testing are based primarily on connectivity availability and statically configured weights.

Finally, the observability and visualization layer developed during the project serves a primarily experimental purpose and supports validation. Although it has proven useful for interpreting the system's internal behavior during the tests conducted, it does not constitute a complete monitoring platform nor does it intend to compete with professional network management and telemetry solutions.

Despite these limitations, the work carried out has successfully validated the technical feasibility of the proposed model and established a functional foundation upon which future extensions and more advanced lines of research could be developed.

7.5 FUTURE WORK

The work carried out during this project establishes a functional foundation upon which multiple future lines of development could be explored, both from an experimental standpoint and from a perspective more oriented toward real-world production environments.

One of the main lines of development would involve expanding the validation of the framework using real application traffic and more complex protocols than those used during the current experimental phase. Although the tests conducted with ICMPv6 have successfully validated the system's functional behavior, it would be particularly interesting to analyze NEPTUNE's performance with persistent TCP communications, HTTP/3 traffic, VoIP services, or applications sensitive to jitter and latency variations.

Likewise, another possible future line of work would be the incorporation of dynamic link quality metrics into the route selection mechanisms. Currently, routing decisions are based primarily on connectivity availability and statically configured weights. However, the framework could evolve toward more advanced models capable of considering parameters such as latency, packet loss, congestion, or bandwidth utilization to make dynamic traffic engineering decisions.

The integration of the proposed model with hybrid architectures could also be explored, where traditional BGP-based mechanisms coexist alongside edge multihoming techniques based on policy routing and NPTv6. This would allow for the analysis of potential interoperability scenarios and the study of the role that solutions such as NEPTUNE could play within more complex and heterogeneous networks.

From an architectural standpoint, the controller developed during the project could evolve toward distributed and fault-tolerant models. Currently, the framework uses a centralized architecture primarily geared toward experimental validation. As future work, high-availability mechanisms, synchronization among multiple controller instances, and advanced telemetry and observability systems could be investigated.

Another particularly interesting line of research would be validating the framework on real hardware and network environments. Although the virtual lab developed has allowed for the successful validation of the system's functional behavior, the use of real physical links and commercial connectivity providers would enable the analysis of phenomena not present within the current experimental environment, such as real-world variations in WAN latency, congestion, partial link degradation, or the specific behavior of different operators.

Likewise, the YAML-based scenario generation model also offers significant potential for evolution. In future versions, more advanced orchestration and automation mechanisms could be developed, capable of generating more complex dynamic topologies, integrating automatic validations, distributed deployments, and advanced experimental analysis tools.

On the other hand, the observability and visualization layer developed during the project could be significantly expanded by incorporating historical metrics, time-series graphs, statistical traffic analysis, and advanced real-time monitoring mechanisms. This would allow the current lightweight operational visualization layer to be transformed into a more comprehensive operational support platform.

Finally, another possible future line of research would be to explore the applicability of the mechanisms implemented within environments related to SDN and network automation. The separation between the control plane and the data plane present in the NEPTUNE architecture opens the door to possible future integrations with external controllers, orchestration systems, or programmable network management platforms.

Taken together, all these potential avenues for evolution demonstrate that the work carried out during this project primarily serves as an experimental starting point for exploring alternative IPv6 edge multihoming models, leaving open multiple possibilities for future expansion and evolution.

Chapter 8. BIBLIOGRAPHY

[1] J. Postel. "Internet Protocol." Internet Engineering Task Force (IETF), RFC 791, September 1981. https://datatracker.ietf.org/doc/html/rfc791

[2] S. Deering; R. Hinden. "Internet Protocol, Version 6 (IPv6) Specification." Internet Engineering Task Force (IETF), RFC 8200, July 2017. https://datatracker.ietf.org/doc/html/rfc8200

[3] P. Srisuresh; K. Egevang. "Traditional IP Network Address Translator (Traditional NAT)." Internet Engineering Task Force (IETF), RFC 3022, January 2001. https://datatracker.ietf.org/doc/html/rfc3022

[4] M. Ford; P. Srisuresh; D. Kegel. "IP Address Sharing (Carrier-Grade NAT)." Internet Engineering Task Force (IETF), RFC 6888, April 2013. https://datatracker.ietf.org/doc/html/rfc6888

[5] R. Hinden; S. Deering. "IP Version 6 Addressing Architecture." Internet Engineering Task Force (IETF), RFC 4291, February 2006. https://datatracker.ietf.org/doc/html/rfc4291

[6] V. Fuller; T. Li. "Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan." Internet Engineering Task Force (IETF), RFC 4632, August 2006. https://datatracker.ietf.org/doc/html/rfc4632

[7] M. Bagnulo; A. García-Martínez; I. van Beijnum. "The IPv6 Multihoming Problem." IEEE Communications Magazine, Vol. 43, No. 6, June 2005. https://doi.org/10.1109/MCOM.2005.1452827

[8] Kurose, J. F., Ross, K. W. Computer Networking: A Top-Down Approach. 7th ed. Boston: Pearson, 2017. —

[9] Postel, J. Internet Control Message Protocol. RFC 792. Internet Engineering Task Force (IETF), September 1981. Available: https://www.rfc-editor.org/rfc/rfc792

—

[10] Kent, S., Seo, K. Security Architecture for the Internet Protocol. RFC 4301. Internet Engineering Task Force (IETF), December 2005. Available: https://www.rfc-editor.org/rfc/rfc4301 —

[11] Peterson, L. L., Davie, B. S. Computer Networks: A Systems Approach. 5th ed. Burlington, MA: Morgan Kaufmann, 2011. —

[12] Hawkinson, J., Bates, T. Guidelines for Creation, Selection, and Registration of an Autonomous System (AS). RFC 1930. Internet Engineering Task Force (IETF), March 1996. Available: https://www.rfc-editor.org/rfc/rfc1930 —

[13] Rekhter, Y., Li, T., Hares, S. A Border Gateway Protocol 4 (BGP-4). RFC 4271. Internet Engineering Task Force (IETF), January 2006. Available: https://www.rfc-editor.org/rfc/rfc4271 —

[14] Nordmark, E., Gilligan, R. Basic Transition Mechanisms for IPv6 Hosts and Routers. RFC 4213. Internet Engineering Task Force (IETF), October 2005. Available: https://www.rfc-editor.org/rfc/rfc4213 —

[15] Durand, A., Droms, R., Woodyatt, J., Lee, Y. Dual-Stack Lite Broadband Deployments Following IPv4 Exhaustion. RFC 6333. Internet Engineering Task Force (IETF), August 2011. Available: https://www.rfc-editor.org/rfc/rfc6333 —

[16] Olive, X. Overview of IPv6 Transition Mechanisms. RFC 9386. Internet Engineering Task Force (IETF), May 2023. Available: https://www.rfc-editor.org/rfc/rfc9386

[17] Bagnulo, M., Matthews, P., van Beijnum, I. Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers. RFC 6146. Internet Engineering Task Force (IETF), April 2011. Available: https://www.rfc-editor.org/rfc/rfc6146 —

[18] Mawatari, M., Kawashima, M., Byrne, C. 464XLAT: Combination of Stateful and Stateless Translation. RFC 6877. Internet Engineering Task Force (IETF), April 2013. Available: https://www.rfc-editor.org/rfc/rfc6877

[19] Moy, J. OSPF Version 2. RFC 2328. Internet Engineering Task Force (IETF), April 1998. Available: https://www.rfc-editor.org/rfc/rfc2328

[20] Huston, G. BGP Routing Table Analysis Reports. APNIC Labs, ongoing publication. Available: https://labs.apnic.net

[21] Norton, W. B. The Art of Peering: The Peering Playbook. DrPeering Press, 2014. Available: https://drpeering.net

[22] Krishnamurthy, B., Wills, C. E. On the Use and Performance of Content Distribution Networks. Proceedings of the ACM SIGCOMM Internet Measurement Workshop, 2000. Available: https://doi.org/10.1145/347090.347137

[23] Narten, T., Huston, G., Roberts, L. IPv6 Address Assignment to End Sites. RFC 6177. Internet Engineering Task Force (IETF), March 2011. Available: https://www.rfc-editor.org/rfc/rfc6177

[24] Carpenter, B., Jiang, S. Enterprise IPv6 Deployment Guidelines. RFC 7381. Internet Engineering Task Force (IETF), October 2014. Available: https://www.rfc-editor.org/rfc/rfc7381

[25] Abley, J., Lindqvist, K. IPv4 Multihoming Practices and Limitations. RFC 4116. Internet Engineering Task Force (IETF), July 2005. Available: https://www.rfc-editor.org/rfc/rfc4116

[26] Awduche, D., Chiu, A., Elwalid, A., Widjaja, I., Xiao, X. Overview and Principles of Internet Traffic Engineering. RFC 3272. Internet Engineering Task Force (IETF), May 2002. Available: https://www.rfc-editor.org/rfc/rfc3272

[27] Mockapetris, P. Domain Names – Concepts and Facilities. RFC 1034. Internet Engineering Task Force (IETF), November 1987. Available: https://www.rfc-editor.org/rfc/rfc1034

[28] Mockapetris, P. Domain Names – Implementation and Specification. RFC 1035. Internet Engineering Task Force (IETF), November 1987. Available: https://www.rfc-editor.org/rfc/rfc1035

[29] Thomson, S., Huitema, C., Ksinant, V., Souissi, M. DNS Extensions to Support IP Version 6. RFC 3596. Internet Engineering Task Force (IETF), October 2003. Available: https://www.rfc-editor.org/rfc/rfc3596

[30] Bagnulo, M., Sullivan, A., Matthews, P., van Beijnum, I. DNS64: DNS Extensions for Network Address Translation from IPv6 Clients to IPv4 Servers. RFC 6147. Internet Engineering Task Force (IETF), April 2011. Available: https://www.rfc-editor.org/rfc/rfc6147

[31] Droms, R. Dynamic Host Configuration Protocol. RFC 2131. Internet Engineering Task Force (IETF), March 1997. Available: https://www.rfc-editor.org/rfc/rfc2131

[32] Thomson, S., Narten, T., Jinmei, T. IPv6 Stateless Address Autoconfiguration. RFC 4862. Internet Engineering Task Force (IETF), September 2007. Available: https://www.rfc-editor.org/rfc/rfc4862

[33] Mrugalski, T., Siodelski, M., Volz, B., et al. Dynamic Host Configuration Protocol for IPv6 (DHCPv6). RFC 8415. Internet Engineering Task Force (IETF), November 2018. Available: https://www.rfc-editor.org/rfc/rfc8415

[34] Tanenbaum, A. S., Wetherall, D. J. Computer Networks. 5th ed. Pearson, 2011.

[35] Kreutz, D., Ramos, F. M. V., Veríssimo, P. E., Rothenberg, C. E., Azodolmolky, S., Uhlig, S. Software-Defined Networking: A Comprehensive Survey. Proceedings of the IEEE, vol. 103, no. 1, pp. 14–76, 2015. Available: https://doi.org/10.1109/JPROC.2014.2371999

[36] Gupta, P., McKeown, N. Algorithms for Packet Classification. IEEE Network, vol. 15, no. 2, pp. 24–32, 2001. Available: https://doi.org/10.1109/65.912717

[37] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J. OpenFlow: Enabling Innovation in Campus Networks. ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69–74, April 2008. Available: https://doi.org/10.1145/1355734.1355746

[38] Harrington, D., Presuhn, R., Wijnen, B. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. RFC 3411. Internet Engineering Task Force (IETF), December 2002. Available: https://www.rfc-editor.org/rfc/rfc3411

[39] Open Networking Foundation (ONF) Software-Defined Networking: The New Norm for Networks. ONF White Paper, April 2012. Available: https://opennetworking.org

[40] Thaler, D., Draves, R., Matsumoto, A., Chown, T. Default Address Selection for Internet Protocol Version 6 (IPv6). RFC 6724. Internet Engineering Task Force (IETF), September 2012. https://www.rfc-editor.org/rfc/rfc6724

[41] Ford, A., Raiciu, C., Handley, M., Bonaventure, O. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 8684. Internet Engineering Task Force (IETF), March 2020. https://www.rfc-editor.org/rfc/rfc8684

[42] Iyengar, J., Thomson, M. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. Internet Engineering Task Force (IETF), May 2021. https://www.rfc-editor.org/rfc/rfc9000

[43] Nordmark, E., Bagnulo, M. Shim6: Level 3 Multihoming Shim Protocol for IPv6. RFC 5533. Internet Engineering Task Force (IETF), June 2009. https://www.rfc-editor.org/rfc/rfc5533

[44] Atkinson, R. Identifier-Locator Network Protocol (ILNP) Architectural Description. RFC 6740. Internet Engineering Task Force (IETF), November 2012. https://www.rfc-editor.org/rfc/rfc6740

[45] Farinacci, D., Fuller, V., Meyer, D., Lewis, D. Locator/Identifier Separation Protocol (LISP). RFC 6830. Internet Engineering Task Force (IETF), January 2013. https://www.rfc-editor.org/rfc/rfc6830

[46] Hopps, C. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992. Internet Engineering Task Force (IETF), November 2000. https://www.rfc-editor.org/rfc/rfc2992

[47] Rosen, E., Rekhter, Y. BGP/MPLS IP Virtual Private Networks (VPNs). RFC 4364. Internet Engineering Task Force (IETF), February 2006. https://www.rfc-editor.org/rfc/rfc4364

[48] Hinden, R., Haberman, B. Unique Local IPv6 Unicast Addresses. RFC 4193. Internet Engineering Task Force (IETF), October 2005. https://www.rfc-editor.org/rfc/rfc4193

[49] Bagnulo, M., Matthews, P., van Beijnum, I. IPv6-to-IPv6 Network Prefix Translation. RFC 6296. Internet Engineering Task Force (IETF), June 2011. https://www.rfc-editor.org/rfc/rfc6296

[50] Wright, G., Stevens, W. R. TCP/IP Illustrated, Volume 2: The Implementation. Addison-Wesley, 1995.

[51] Welte, H. The Netfilter.org Project. Available: https://www.netfilter.org

[52] Kerrisk, M. Linux Namespaces. The Linux Programming Interface. No Starch Press, 2010. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html

ANNEX I: ALIGNMENT OF THE PROJECT WITH THE SDGs

The NEPTUNE project is linked to several of the Sustainable Development Goals (SDGs) defined in the United Nations 2030 Agenda, particularly those related to the development of resilient digital infrastructure, technological innovation, and improved connectivity.

In particular, the work carried out is primarily aligned with:

SDG 9 — Industry, Innovation, and Infrastructure

This goal promotes the development of resilient infrastructure, technological innovation, and the modernization of communications and connectivity systems.

The NEPTUNE project contributes to this goal by designing and validating an experimental model aimed at improving the resilience and flexibility of multihomed IPv6 networks in edge environments.

The ability to maintain connectivity using multiple access providers, as well as the incorporation of automatic failover and dynamic recovery mechanisms, facilitates the construction of more robust and fault-tolerant network infrastructures.

Furthermore, the work explores alternative IPv6 multihoming mechanisms based on open technologies and native capabilities of the Linux ecosystem, fostering research and experimentation on more accessible and modular network architectures.

Additionally, the project is partially related to:

SDG 8 — Decent Work and Economic Growth

Reliable and resilient digital infrastructures are a fundamental element for economic development and for the operation of numerous current digital services. Although the project is primarily technical and experimental in nature, technologies related to high availability, service continuity, and connectivity resilience indirectly contribute to improving the stability of digital systems used by companies and organizations.

Finally, the work is also aligned with:

SDG 4 — Quality Education

The experimental environment developed during the project was built using open and reproducible technologies based on Linux network namespaces, automation via YAML, and standard tools from the networking ecosystem. This facilitates its potential future reuse for teaching, research, or academic experimentation in the field of IPv6 networking, multihoming, and policy routing.

Overall, although NEPTUNE is primarily a technical and experimental project focused on IPv6 network architectures, the capabilities explored during its development are directly related to objectives linked to innovation, digital infrastructure resilience, and sustainable technological development.

ANNEX II: DEVELOPED CODE

This annex contains the code developed throughout the project, whose working principles were described along Chapter 5, and whose output results were discussed along Chapter 6.

The code main folder is structured as follows:

```
NEPTUNE
├── config
│   └── config.yaml
├── dashboard
│   ├── static
│   │   ├── app.js
│   │   └── style.css
│   └── templates
│       └── index.html
└── src
    ├── controller.py
    ├── dashboard_server.py
    ├── flow
    │   ├── flow_hash.py
    │   ├── flow_manager.py
    │   ├── flow_table.py
    │   ├── iptables_manager.py
    │   └── packet_observer.py
    ├── generator.py
    └── runner.py
```

Source Python Files

controller.py

```python
from __future__ import annotations

from pathlib import Path
import argparse
import json
import subprocess
import threading
import yaml

from flow.flow_manager import FlowManager
from flow import iptables_manager

from runtime_state import write_state
from datetime import datetime

BASE_DIR = Path(__file__).resolve().parent.parent
CONFIG_FILE = BASE_DIR / "config" / "config.yaml"
PACKET_OBSERVER = BASE_DIR / "src" / "flow" / "packet_observer.py"

CHECK_NAMESPACE = "client"
ROUTER_NAMESPACE = "router"


def load_config() -> dict:
    if not CONFIG_FILE.exists():
        raise FileNotFoundError(f"Config file not found: {CONFIG_FILE}")

    with CONFIG_FILE.open("r", encoding="utf-8") as f:
        data = yaml.safe_load(f)

    if not isinstance(data, dict):
        raise ValueError("Invalid YAML: root element must be a mapping")

    return data


def build_runtime_config(cfg: dict) -> dict:
    isps = cfg["isps"]
    neptune = cfg["neptune"]

    isp_order = [isp["name"] for isp in isps]

    health_targets = {
        isp["name"]: isp["neptune"]["health_target"]
        for isp in isps
    }

    mark_by_isp = {
        isp["name"]: isp["neptune"]["mark"]
        for isp in isps
    }

    isp_weights = {
        isp["name"]: isp["neptune"]["weight"]
        for isp in isps
```

```
    }

    primary_pool = [
        isp["name"]
        for isp in isps
        if isp["neptune"]["role"] == "primary"
    ]

    backup_pool = [
        isp["name"]
        for isp in isps
        if isp["neptune"]["role"] == "backup"
    ]

    excluded_destination_prefixes =
list(neptune["excluded_destination_prefixes"])
    policies = list(neptune["policies"])

    return {
        "isp_order": isp_order,
        "health_targets": health_targets,
        "mark_by_isp": mark_by_isp,
        "isp_weights": isp_weights,
        "primary_pool": primary_pool,
        "backup_pool": backup_pool,
        "lan_prefixes": [cfg["lab"]["lan_prefix"]],
        "excluded_destination_prefixes": excluded_destination_prefixes,
        "policies": policies,
        "check_interval": neptune["check_interval"],
        "health_log_interval": neptune["health_log_interval"],
        "flow_gc_interval": neptune["flow_gc_interval"],
        "flow_timeout_seconds": neptune["flow_timeout_seconds"],
        "health_target_info": health_targets,
    }


CFG = load_config()
RUNTIME = build_runtime_config(CFG)

ISP_ORDER = RUNTIME["isp_order"]
PRIMARY_POOL = RUNTIME["primary_pool"]
BACKUP_POOL = RUNTIME["backup_pool"]
ISP_NEXT_HOPS = RUNTIME["health_targets"]

HEALTH_LOG_INTERVAL = RUNTIME["health_log_interval"]
FLOW_GC_INTERVAL = RUNTIME["flow_gc_interval"]
FLOW_TIMEOUT_SECONDS = RUNTIME["flow_timeout_seconds"]
CHECK_INTERVAL = RUNTIME["check_interval"]


def run(cmd: list[str], check: bool = True) -> subprocess.CompletedProcess:
    return subprocess.run(cmd, check=check, capture_output=True, text=True)


def is_isp_usable(isp_name: str) -> bool:
    target = ISP_NEXT_HOPS[isp_name]
    cmd = [
        "sudo", "ip", "netns", "exec", ROUTER_NAMESPACE,
        "ping", "-6", "-c", "1", "-W", "1", target,
    ]
    result = run(cmd, check=False)
```

```
    return result.returncode == 0


def get_health_snapshot() -> dict[str, bool]:
    return {isp: is_isp_usable(isp) for isp in ISP_ORDER}


def healthy_isps_from_snapshot(health: dict[str, bool]) -> list[str]:
    return [isp for isp in ISP_ORDER if health.get(isp, False)]


def balanced_pool_from_snapshot(health: dict[str, bool]) -> list[str]:
    healthy_primaries = [isp for isp in PRIMARY_POOL if health.get(isp, False)]
    if healthy_primaries:
        return healthy_primaries

    healthy_backups = [isp for isp in BACKUP_POOL if health.get(isp, False)]
    if healthy_backups:
        return healthy_backups

    return []


def get_healthy_isps() -> list[str]:
    return healthy_isps_from_snapshot(get_health_snapshot())


def get_active_data_plane_isps() -> list[str]:
    return balanced_pool_from_snapshot(get_health_snapshot())


def health_monitor_loop(flow_manager: FlowManager, stop_event: threading.Event,
runtime_state: dict) -> None:
    last_snapshot: tuple[tuple[str, bool], ...] | None = None
    last_active_pool: list[str] | None = None

    while not stop_event.is_set():
        try:
            health = get_health_snapshot()
            healthy_isps = healthy_isps_from_snapshot(health)
            active_pool = balanced_pool_from_snapshot(health)

            for isp, ok in health.items():
                runtime_state["isps"][isp]["status"] = "up" if ok else "down"

            runtime_state["active_pool"] = active_pool
            runtime_state["flows"] = flow_manager.export_flows()

            runtime_state["events"].append({
                "time": datetime.now().isoformat(timespec="seconds"),
                "type": "health",
                "message": f"Healthy ISPs: {healthy_isps if healthy_isps else
'none'}",
            })

            runtime_state["events"] = runtime_state["events"][-50:]

            write_state(runtime_state)

            snapshot = tuple((isp, health[isp]) for isp in ISP_ORDER)
```

```
            if snapshot != last_snapshot:
                health_str = ", ".join(
                    f"{isp}={'up' if ok else 'down'}" for isp, ok in snapshot
                )
                print(f"[CTRL][HEALTH] {health_str}")
                print(f"[CTRL][HEALTH] Healthy ISPs: {healthy_isps if
healthy_isps else 'none'}")
                last_snapshot = snapshot

            if last_active_pool != active_pool:
                print(f"[CTRL][POOL] Active NEPTUNE pool: {active_pool if
active_pool else 'none'}")

                if last_active_pool is not None:
                    flow_manager.flush_all_flows()

                last_active_pool = list(active_pool)

        except Exception as exc:
            print(f"[CTRL][HEALTH][ERROR] {exc}")

        stop_event.wait(CHECK_INTERVAL)

def flow_gc_loop(flow_manager: FlowManager, stop_event: threading.Event) -> None:
    while not stop_event.is_set():
        try:
            flow_manager.gc_expired_flows()
        except Exception as exc:
            print(f"[CTRL][FLOW][GC][ERROR] {exc}")
        stop_event.wait(FLOW_GC_INTERVAL)


def start_observer_worker() -> subprocess.Popen:
    lan_if = CFG["router"]["lan_if"]
    cmd = [
        "sudo", "ip", "netns", "exec", ROUTER_NAMESPACE,
        "python3",
        str(PACKET_OBSERVER),
        "--iface", lan_if,
    ]
    return subprocess.Popen(
        cmd,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        text=True,
        bufsize=1,
    )


def neptune_controller_loop() -> None:
    print("[CTRL] NEPTUNE controller started")
    print(f"[CTRL] Health targets: {RUNTIME['health_target_info']}")
    print(f"[CTRL] Primary balancing pool: {PRIMARY_POOL}")
    print(f"[CTRL] Backup pool: {BACKUP_POOL}")

    stop_event = threading.Event()

    iptables_manager.configure(
        router_ns=ROUTER_NAMESPACE,
        lan_if=CFG["router"]["lan_if"],
    )
```

```
flow_manager = FlowManager(
    healthy_isps_provider=get_healthy_isps,
    balanced_isps_provider=get_active_data_plane_isps,
    timeout_seconds=FLOW_TIMEOUT_SECONDS,
    lan_prefixes=RUNTIME["lan_prefixes"],
    excluded_destination_prefixes=RUNTIME["excluded_destination_prefixes"],
    isp_weights=RUNTIME["isp_weights"],
    mark_by_isp=RUNTIME["mark_by_isp"],
    policies=RUNTIME["policies"],
)

runtime_state = {
    "controller": {
        "status": "running",
        "mode": "neptune",
    },
    "isps": {
        isp: {
            "status": "unknown",
            "role": "primary" if isp in PRIMARY_POOL else "backup",
            "health_target": ISP_NEXT_HOPS[isp],
        }
        for isp in ISP_ORDER
    },
    "active_pool": [],
    "flows": [],
    "events": [],
}

gc_thread = threading.Thread(
    target=flow_gc_loop,
    args=(flow_manager, stop_event),
    daemon=True,
)
gc_thread.start()

health_thread = threading.Thread(
    target=health_monitor_loop,
    args=(flow_manager, stop_event, runtime_state),
    daemon=True,
)
health_thread.start()

worker = start_observer_worker()

if worker.stderr:
    def _stderr_reader() -> None:
        for line in worker.stderr:
            print(f"[OBSERVER][ERR] {line.strip()}", flush=True)

    threading.Thread(
        target=_stderr_reader,
        daemon=True,
    ).start()

try:
    assert worker.stdout is not None

    for line in worker.stdout:
        line = line.strip()
```

```
            if not line:
                continue

            try:
                payload = json.loads(line)
                flow_key = (
                    payload["src_ip"],
                    payload["dst_ip"],
                    payload["proto"],
                    int(payload["flow_id"]),
                )
                flow_manager.handle_flow_key(flow_key)
            except Exception as exc:
                print(f"[CTRL][FLOW][OBSERVER][ERROR] Could not parse line
'{line}': {exc}")

    except KeyboardInterrupt:
        print("\n[CTRL] Controller stopped by user")
    finally:
        stop_event.set()
        worker.terminate()
        worker.wait()


def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(description="NEPTUNE controller")
    parser.add_argument(
        "--mode",
        choices=["neptune", "flowlb", "failover"],
        default="neptune",
        help="Controller mode (flowlb/failover kept as compatibility aliases)",
    )
    return parser.parse_args()


if __name__ == "__main__":
    args = parse_args()

    if args.mode in {"neptune", "flowlb", "failover"}:
        neptune_controller_loop()
```

dashboard_server.py

```
from __future__ import annotations

from pathlib import Path
import json
import subprocess

import yaml
from flask import Flask, jsonify, request, render_template

import threading

BASE_DIR = Path(__file__).resolve().parent.parent
CONFIG_FILE = BASE_DIR / "config" / "config.yaml"
STATE_FILE = BASE_DIR / "runtime" / "state.json"
EVENTS_DIR = BASE_DIR / "scripts" / "generated" / "events"

app = Flask(
    __name__,
    template_folder=str(BASE_DIR / "dashboard" / "templates"),
    static_folder=str(BASE_DIR / "dashboard" / "static"),
)

continuous_ping_process = None
continuous_ping_lines = []

def load_yaml() -> dict:
    with CONFIG_FILE.open("r", encoding="utf-8") as f:
        return yaml.safe_load(f)


def load_state() -> dict:
    if not STATE_FILE.exists():
        return {
            "controller": {"status": "not_running"},
            "isps": {},
            "active_pool": [],
            "flows": [],
            "events": [],
        }

    with STATE_FILE.open("r", encoding="utf-8") as f:
        return json.load(f)


def build_config_summary(cfg: dict) -> dict:
    destinations = []

    for isp in cfg["isps"]:
        remote = isp.get("remote")
        if remote:
            destinations.append({
                "name": remote["name"],
                "ip": remote["remote_ip"].split("/")[0],
                "type": "remote",
                "isp": isp["name"],
            })

        destinations.append({
```

```
            "name": f"{isp['name']}-health",
            "ip": isp["neptune"]["health_target"],
            "type": "health-target",
            "isp": isp["name"],
        })

    for srv in cfg.get("global_services", []):
        destinations.append({
            "name": srv["name"],
            "ip": srv["srv_ip"].split("/")[0],
            "type": "global-service",
            "isp": None,
        })

    sources = [
        {
            "name": cfg["client"]["name"],
            "namespace": cfg["client"]["name"],
            "ip": cfg["client"]["ip"].split("/")[0],
        },
        {
            "name": cfg["router"]["name"],
            "namespace": cfg["router"]["name"],
            "ip": cfg["router"]["lan_ip"].split("/")[0],
        },
    ]

    isps = [
        {
            "name": isp["name"],
            "role": isp["neptune"]["role"],
            "weight": isp["neptune"]["weight"],
            "mark": isp["neptune"]["mark"],
            "health_target": isp["neptune"]["health_target"],
            "delegated_prefix": isp["delegated_prefix"],
            "wan_prefix": isp["wan_prefix"],
        }
        for isp in cfg["isps"]
    ]

    return {
        "lab": cfg["lab"],
        "sources": sources,
        "destinations": destinations,
        "isps": isps,
    }

@app.get("/")
def index():
    return render_template("index.html")

@app.get("/api/state")
def api_state():
    return jsonify(load_state())


@app.get("/api/config")
def api_config():
    cfg = load_yaml()
    return jsonify(build_config_summary(cfg))
```

```
@app.post("/api/ping")
def api_ping():
    data = request.get_json(force=True)

    source_ns = data["source"]
    destination_ip = data["destination_ip"]
    count = int(data.get("count", 4))

    if count < 1 or count > 20:
        return jsonify({"ok": False, "error": "count must be between 1 and 20"}),
400

    cmd = [
        "sudo", "ip", "netns", "exec", source_ns,
        "ping", "-6", "-c", str(count), destination_ip,
    ]

    result = subprocess.run(
        cmd,
        capture_output=True,
        text=True,
        check=False,
    )

    return jsonify({
        "ok": result.returncode == 0,
        "cmd": " ".join(cmd),
        "stdout": result.stdout,
        "stderr": result.stderr,
        "returncode": result.returncode,
    })


@app.post("/api/event")
def api_event():
    data = request.get_json(force=True)

    isp = data["isp"]
    action = data["action"]

    if action not in {"up", "down"}:
        return jsonify({"ok": False, "error": "action must be up or down"}), 400

    script = EVENTS_DIR / f"{isp}_{action}.sh"

    if not script.exists():
        return jsonify({"ok": False, "error": f"script not found: {script}"}),
404

    result = subprocess.run(
        ["bash", str(script)],
        capture_output=True,
        text=True,
        check=False,
    )

    return jsonify({
        "ok": result.returncode == 0,
        "script": str(script),
        "stdout": result.stdout,
```

```
        "stderr": result.stderr,
        "returncode": result.returncode,
    })

def _read_process_output(process: subprocess.Popen) -> None:
    global continuous_ping_lines

    if process.stdout is None:
        return

    for line in process.stdout:
        continuous_ping_lines.append(line.rstrip())
        continuous_ping_lines = continuous_ping_lines[-200:]


@app.post("/api/ping/continuous/start")
def api_continuous_ping_start():
    global continuous_ping_process, continuous_ping_lines

    if continuous_ping_process is not None and continuous_ping_process.poll() is
None:
        return jsonify({"ok": False, "error": "continuous ping already
running"}), 409

    data = request.get_json(force=True)

    source_ns = data["source"]
    destination_ip = data["destination_ip"]
    interval = float(data.get("interval", 1.0))

    if interval < 0.2:
        return jsonify({"ok": False, "error": "interval must be >= 0.2
seconds"}), 400

    continuous_ping_lines = []

    cmd = [
        "sudo", "ip", "netns", "exec", source_ns,
        "ping", "-6", "-i", str(interval), destination_ip,
    ]

    continuous_ping_process = subprocess.Popen(
        cmd,
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        text=True,
        bufsize=1,
    )

    threading.Thread(
        target=_read_process_output,
        args=(continuous_ping_process,),
        daemon=True,
    ).start()

    return jsonify({
        "ok": True,
        "cmd": " ".join(cmd),
    })
```

```
@app.post("/api/ping/continuous/stop")
def api_continuous_ping_stop():
    global continuous_ping_process

    if continuous_ping_process is None or continuous_ping_process.poll() is not
None:
        return jsonify({"ok": True, "message": "no continuous ping running"})

    continuous_ping_process.terminate()

    try:
        continuous_ping_process.wait(timeout=2)
    except subprocess.TimeoutExpired:
        continuous_ping_process.kill()

    return jsonify({"ok": True})


@app.get("/api/ping/continuous/status")
def api_continuous_ping_status():
    running = (
        continuous_ping_process is not None
        and continuous_ping_process.poll() is None
    )

    return jsonify({
        "running": running,
        "lines": continuous_ping_lines[-80:],
    })

@app.post("/api/tests/distribution")
def api_distribution_test():
    data = request.get_json(force=True)

    source_ns = data["source"]
    destination_ip = data["destination_ip"]
    iterations = int(data.get("iterations", 20))

    if iterations < 1 or iterations > 200:
        return jsonify({"ok": False, "error": "iterations must be between 1 and
200"}), 400

    results = []

    for i in range(iterations):
        cmd = [
            "sudo", "ip", "netns", "exec", source_ns,
            "ping", "-6", "-c", "1", destination_ip,
        ]

        result = subprocess.run(
            cmd,
            capture_output=True,
            text=True,
            check=False,
        )

        state = load_state()
        flows = state.get("flows", [])

        last_flow = flows[-1] if flows else None
```

```
        results.append({
            "iteration": i + 1,
            "ok": result.returncode == 0,
            "selected_isp": last_flow.get("selected_isp") if last_flow else None,
            "flow_id": last_flow.get("flow_id") if last_flow else None,
        })

    distribution = {}

    for item in results:
        isp = item["selected_isp"] or "unknown"
        distribution[isp] = distribution.get(isp, 0) + 1

    return jsonify({
        "ok": True,
        "iterations": iterations,
        "distribution": distribution,
        "results": results,
    })

if __name__ == "__main__":
    app.run(host="127.0.0.1", port=8080, debug=True)
```

flow_hash.py

```
import hashlib
from typing import Tuple

FlowKey = Tuple[str, str, str, int]


def serialize_flow_key(flow_key: FlowKey) -> bytes:
    src_ip, dst_ip, proto, icmp_id = flow_key
    return f"{src_ip}|{dst_ip}|{proto}|{icmp_id}".encode("utf-8")


def compute_bucket(flow_key: FlowKey, num_buckets: int) -> int:
    if num_buckets <= 0:
        raise ValueError("num_buckets must be > 0")

    data = serialize_flow_key(flow_key)
    digest = hashlib.sha256(data).digest()
    value = int.from_bytes(digest[:8], byteorder="big", signed=False)
    return value % num_buckets


def select_isp(flow_key: FlowKey, available_isps: list[str]) -> str:
    if not available_isps:
        raise ValueError("No available ISPs")

    if len(available_isps) == 1:
        return available_isps[0]

    bucket = compute_bucket(flow_key, len(available_isps))
    return available_isps[bucket]


def build_weighted_isp_pool(
    available_isps: list[str],
    isp_weights: dict[str, int],
) -> list[str]:
    """
          weights = {"isp1": 7, "isp2": 3}
      -> ["isp1", "isp1", ..., "isp2", "isp2", ...]
    """
    pool: list[str] = []

    for isp in available_isps:
        weight = isp_weights.get(isp, 0)
        if weight > 0:
            pool.extend([isp] * weight)

    return pool


def select_weighted_isp(
    flow_key: FlowKey,
    available_isps: list[str],
    isp_weights: dict[str, int],
) -> str:
    pool = build_weighted_isp_pool(available_isps, isp_weights)

    if not pool:
```

```
        raise ValueError("No weighted ISP pool could be built")

    if len(pool) == 1:
        return pool[0]

    bucket = compute_bucket(flow_key, len(pool))
    return pool[bucket]
```

flow_manager.py

```
from __future__ import annotations

from typing import Callable, Tuple, Any
import ipaddress

from flow.flow_hash import select_weighted_isp
from flow.flow_table import FlowTable, FlowEntry
from flow.iptables_manager import (
    install_icmpv6_mark_rule,
    delete_icmpv6_mark_rule,
)

FlowKey = Tuple[str, str, str, int]


class FlowManager:
    def __init__(
        self,
        healthy_isps_provider: Callable[[], list[str]],
        balanced_isps_provider: Callable[[], list[str]],
        timeout_seconds: int,
        lan_prefixes: list[str],
        excluded_destination_prefixes: list[str],
        isp_weights: dict[str, int],
        mark_by_isp: dict[str, int],
        policies: list[dict[str, Any]] | None = None,
    ) -> None:
        self.healthy_isps_provider = healthy_isps_provider
        self.balanced_isps_provider = balanced_isps_provider
        self.timeout_seconds = timeout_seconds
        self.flow_table = FlowTable()
        self.isp_weights = isp_weights
        self.mark_by_isp = mark_by_isp

        self.lan_networks = [ipaddress.IPv6Network(p) for p in lan_prefixes]
        self.excluded_destination_networks = [
            ipaddress.IPv6Network(p) for p in excluded_destination_prefixes
        ]

        self.policies: list[dict[str, Any]] = []
        if policies:
            for policy in policies:
                parsed = dict(policy)
                parsed["network"] = ipaddress.IPv6Network(policy["prefix"])
                self.policies.append(parsed)

    def handle_flow_key(self, flow_key: FlowKey) -> None:
        src_ip, dst_ip, _proto, _flow_id = flow_key

        if not self._is_steerable_flow(src_ip, dst_ip):
            return

        existing = self.flow_table.get(flow_key)
        if existing is not None:
            self.flow_table.touch(flow_key)
            return

        try:
```

```
            selected_isp = self._select_isp_for_flow(flow_key)
        except Exception as exc:
            print(f"[CTRL][FLOW][ERROR] Could not select ISP for {flow_key}:
{exc}")
            return

        if selected_isp is None:
            print(f"[CTRL][FLOW] No valid ISP for flow {flow_key}")
            return

        mark = self.mark_by_isp.get(selected_isp)
        if mark is None:
            print(f"[CTRL][FLOW][ERROR] No mark defined for ISP {selected_isp}")
            return

        try:
            install_icmpv6_mark_rule(flow_key, mark)
        except Exception as exc:
            print(f"[CTRL][FLOW][ERROR] Could not install rule for {flow_key}:
{exc}")
            return

        self.flow_table.upsert(flow_key, selected_isp, mark)
        print(f"[CTRL][FLOW] New flow {flow_key} -> {selected_isp}
(mark={mark})")

    def gc_expired_flows(self) -> None:
        expired = self.flow_table.expired(self.timeout_seconds)
        if not expired:
            return

        for entry in expired:
            self._expire_entry(entry)

    def flush_all_flows(self) -> None:
        entries = self.flow_table.all()
        if not entries:
            return

        print(f"[CTRL][FLOW] Flushing {len(entries)} learned flow entries due to
topology change")

        for entry in entries:
            self._expire_entry(entry, silent=True)

    def _expire_entry(self, entry: FlowEntry, silent: bool = False) -> None:
        try:
            delete_icmpv6_mark_rule(entry.flow_key, entry.mark)
        except Exception as exc:
            print(
                f"[CTRL][FLOW][GC][WARN] Could not delete iptables rule "
                f"for {entry.flow_key}: {exc}"
            )
        finally:
            self.flow_table.delete(entry.flow_key)

        if not silent:
            print(f"[CTRL][FLOW][GC] Expired flow {entry.flow_key} from
{entry.selected_isp}")

    def _select_isp_for_flow(self, flow_key: FlowKey) -> str | None:
```

```
        _src_ip, dst_ip, _proto, _flow_id = flow_key

        healthy_isps = self._get_healthy_isps()
        balanced_isps = self._get_balanced_isps()

        matched_policy = self._match_policy(dst_ip)
        if matched_policy is not None:
            mode = matched_policy["mode"]

            if mode == "strict":
                return self._select_strict(dst_ip, matched_policy, healthy_isps)

            if mode == "preferred":
                return self._select_preferred(dst_ip, matched_policy,
healthy_isps, balanced_isps)

            if mode == "balanced":
                return self._select_balanced(dst_ip, flow_key, balanced_isps,
policy_label=True)

            raise ValueError(f"Unknown policy mode: {mode}")

        return self._select_balanced(dst_ip, flow_key, balanced_isps,
policy_label=False)

    def _select_strict(
        self,
        dst_ip: str,
        policy: dict[str, Any],
        healthy_isps: list[str],
    ) -> str | None:
        isp = policy["isp"]

        if isp in healthy_isps:
            print(f"[CTRL][FLOW][STRICT] Destination {dst_ip} pinned to {isp}")
            return isp

        print(f"[CTRL][FLOW][STRICT] Destination {dst_ip} requires {isp}, but it
is not healthy")
        return None

    def _select_preferred(
        self,
        dst_ip: str,
        policy: dict[str, Any],
        healthy_isps: list[str],
        balanced_isps: list[str],
    ) -> str:
        preferred_isp = policy["preferred_isp"]

        if preferred_isp in healthy_isps:
            print(f"[CTRL][FLOW][PREFERRED] Destination {dst_ip} prefers
{preferred_isp}")
            return preferred_isp

        fallback_pool = [isp for isp in balanced_isps if isp != preferred_isp]
        if not fallback_pool:
            raise ValueError(f"No fallback ISP available for preferred policy on
{dst_ip}")

        selected = fallback_pool[0]
```

```
        print(
            f"[CTRL][FLOW][PREFERRED] Destination {dst_ip} preferred
{preferred_isp} unavailable, "
            f"falling back to {selected}"
        )
        return selected

    def _select_balanced(
        self,
        dst_ip: str,
        flow_key: FlowKey,
        balanced_isps: list[str],
        policy_label: bool,
    ) -> str:
        if not balanced_isps:
            raise ValueError(f"No balanced ISP available for destination
{dst_ip}")

        selected = select_weighted_isp(
            flow_key=flow_key,
            available_isps=balanced_isps,
            isp_weights=self.isp_weights,
        )

        if policy_label:
            print(f"[CTRL][FLOW][BALANCED] Destination {dst_ip} selected
{selected}")
        else:
            print(f"[CTRL][FLOW][WEIGHTED] Destination {dst_ip} fell back to
weighted selection -> {selected}")

        return selected

    def _match_policy(self, dst_ip: str) -> dict[str, Any] | None:
        addr = ipaddress.IPv6Address(dst_ip)

        for policy in self.policies:
            if addr in policy["network"]:
                return policy

        return None

    def _is_steerable_flow(self, src_ip: str, dst_ip: str) -> bool:
        return self._is_lan_source(src_ip) and not
self._is_excluded_destination(dst_ip)

    def _is_lan_source(self, ip: str) -> bool:
        addr = ipaddress.IPv6Address(ip)
        return any(addr in net for net in self.lan_networks)

    def _is_excluded_destination(self, ip: str) -> bool:
        addr = ipaddress.IPv6Address(ip)
        return any(addr in net for net in self.excluded_destination_networks)

    def _get_healthy_isps(self) -> list[str]:
        return [
            isp
            for isp in self.healthy_isps_provider()
            if isp in self.mark_by_isp and self.isp_weights.get(isp, 0) > 0
        ]
```

```
def export_flows(self) -> list[dict[str, object]]:
    flows = []

    for entry in self.flow_table.all():
        src_ip, dst_ip, proto, flow_id = entry.flow_key

        flows.append({
            "src_ip": src_ip,
            "dst_ip": dst_ip,
            "proto": proto,
            "flow_id": flow_id,
            "selected_isp": entry.selected_isp,
            "mark": entry.mark,
            "created_at": entry.created_at,
            "last_seen": entry.last_seen,
        })

    return flows

def _get_balanced_isps(self) -> list[str]:
    return [
        isp
        for isp in self.balanced_isps_provider()
        if isp in self.mark_by_isp and self.isp_weights.get(isp, 0) > 0
    ]
```

flow_table.py

```
from __future__ import annotations

import time
from dataclasses import dataclass
from typing import Dict, Tuple

FlowKey = Tuple[str, str, str, int]

@dataclass
class FlowEntry:
    flow_key: FlowKey
    selected_isp: str
    mark: int
    created_at: float
    last_seen: float

class FlowTable:
    def __init__(self) -> None:
        self._flows: Dict[FlowKey, FlowEntry] = {}

    def get(self, flow_key: FlowKey) -> FlowEntry | None:
        return self._flows.get(flow_key)

    def upsert(self, flow_key: FlowKey, selected_isp: str, mark: int) ->
FlowEntry:
        now = time.time()
        entry = self._flows.get(flow_key)
        if entry is None:
            entry = FlowEntry(
                flow_key=flow_key,
                selected_isp=selected_isp,
                mark=mark,
                created_at=now,
                last_seen=now,
            )
            self._flows[flow_key] = entry
        else:
            entry.last_seen = now
        return entry

    def touch(self, flow_key: FlowKey) -> None:
        entry = self._flows.get(flow_key)
        if entry is not None:
            entry.last_seen = time.time()

    def expired(self, timeout_seconds: int) -> list[FlowEntry]:
        now = time.time()
        return [
            entry
            for entry in self._flows.values()
            if now - entry.last_seen > timeout_seconds
        ]

    def delete(self, flow_key: FlowKey) -> None:
        self._flows.pop(flow_key, None)

    def all(self) -> list[FlowEntry]:
        return list(self._flows.values())
```

iptables_manager.py

```
from __future__ import annotations

import subprocess
from typing import Tuple

FlowKey = Tuple[str, str, str, int]

ROUTER_NS = "router"
LAN_IF = None


def configure(router_ns: str, lan_if: str) -> None:
    global ROUTER_NS, LAN_IF
    ROUTER_NS = router_ns
    LAN_IF = lan_if


def _run(cmd: list[str]) -> None:
    subprocess.run(cmd, check=True)


def _base_cmd() -> list[str]:
    return ["sudo", "ip", "netns", "exec", ROUTER_NS, "ip6tables", "-t",
"mangle"]


def _require_configured() -> str:
    if LAN_IF is None:
        raise RuntimeError("iptables_manager not configured: LAN_IF is unset")
    return LAN_IF


def install_icmpv6_mark_rule(flow_key: FlowKey, mark: int) -> None:
    src_ip, dst_ip, proto, _flow_id = flow_key

    if proto != "icmpv6":
        raise ValueError(f"Unsupported proto for now: {proto}")

    lan_if = _require_configured()

    cmd = _base_cmd() + [
        "-I", "PREROUTING", "1",
        "-i", lan_if,
        "-s", src_ip,
        "-d", dst_ip,
        "-p", "ipv6-icmp",
        "-m", "icmp6", "--icmpv6-type", "echo-request",
        "-j", "MARK", "--set-mark", str(mark),
    ]
    _run(cmd)


def delete_icmpv6_mark_rule(flow_key: FlowKey, mark: int) -> None:
    src_ip, dst_ip, proto, _flow_id = flow_key

    if proto != "icmpv6":
        raise ValueError(f"Unsupported proto for now: {proto}")
```

```
lan_if = _require_configured()

cmd = _base_cmd() + [
    "-D", "PREROUTING",
    "-i", lan_if,
    "-s", src_ip,
    "-d", dst_ip,
    "-p", "ipv6-icmp",
    "-m", "icmp6", "--icmpv6-type", "echo-request",
    "-j", "MARK", "--set-mark", str(mark),
]
_run(cmd)
```

packet_observer.py

```
from __future__ import annotations

import argparse
import json
from typing import Callable, Tuple

from scapy.all import sniff, IPv6, ICMPv6EchoRequest  # type: ignore

FlowKey = Tuple[str, str, str, int]

def extract_flow_key(pkt) -> FlowKey | None:
    if IPv6 not in pkt:
        return None
    if ICMPv6EchoRequest not in pkt:
        return None

    ipv6 = pkt[IPv6]
    icmp = pkt[ICMPv6EchoRequest]

    return (
        str(ipv6.src),
        str(ipv6.dst),
        "icmpv6",
        int(icmp.id),
    )

def sniff_icmpv6(iface: str, packet_callback: Callable[[FlowKey], None]) -> None:
    def _handler(pkt) -> None:
        flow_key = extract_flow_key(pkt)
        if flow_key is not None:
            packet_callback(flow_key)

    sniff(iface=iface, prn=_handler, store=False)

def emit_flow_key_json(flow_key: FlowKey) -> None:
    payload = {
        "src_ip": flow_key[0],
        "dst_ip": flow_key[1],
        "proto": flow_key[2],
        "flow_id": flow_key[3],
    }
    print(json.dumps(payload), flush=True)

def worker_main(iface: str) -> None:
    sniff_icmpv6(iface=iface, packet_callback=emit_flow_key_json)

def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(description="NEPTUNE packet observer
worker")
    parser.add_argument("--iface", required=True, help="Interface to sniff")
    return parser.parse_args()


if __name__ == "__main__":
    args = parse_args()
    worker_main(args.iface)
```

generator.py

```
from pathlib import Path
import sys
import yaml

BASE_DIR = Path(__file__).resolve().parent.parent
CONFIG_FILE = BASE_DIR / "config" / "config.yaml"
OUTPUT_DIR = BASE_DIR / "scripts" / "generated"
OUTPUT_FILE = OUTPUT_DIR / "setup.sh"
ROUTING_FILE = OUTPUT_DIR / "routing.sh"
EVENTS_DIR = OUTPUT_DIR / "events"
NPTV6_FILE = OUTPUT_DIR / "nptv6.sh"


def load_config() -> dict:
    if not CONFIG_FILE.exists():
        raise FileNotFoundError(f"Config file not found: {CONFIG_FILE}")

    with CONFIG_FILE.open("r", encoding="utf-8") as f:
        data = yaml.safe_load(f)

    if not isinstance(data, dict):
        raise ValueError("Invalid YAML: root element must be a mapping")

    return data


def check_ifname(name: str, field: str) -> None:
    if not isinstance(name, str):
        raise ValueError(f"{field} must be a string")

    if len(name) > 15:
        raise ValueError(
            f"Interface name too long in {field}: '{name}' ({len(name)} chars,
max 15)"
        )

    if " " in name:
        raise ValueError(f"Interface name cannot contain spaces in {field}:
'{name}'")


def require_fields(obj: dict, fields: list[str], context: str) -> None:
    for key in fields:
        if key not in obj:
            raise ValueError(f"Missing field in {context}: {key}")


def validate_neptune_policy(policy: dict, context: str, isp_names: set[str]) ->
None:
    require_fields(policy, ["prefix", "mode"], context)

    mode = policy["mode"]
    if mode not in {"strict", "preferred", "balanced"}:
        raise ValueError(
            f"Invalid policy mode in {context}: {mode}. "
            f"Allowed values: strict, preferred, balanced"
        )
```

```
    if mode == "strict":
        require_fields(policy, ["isp"], context)
        if policy["isp"] not in isp_names:
            raise ValueError(
                f"Unknown ISP in {context}.isp: {policy['isp']}"
            )

    if mode == "preferred":
        require_fields(policy, ["preferred_isp"], context)
        if policy["preferred_isp"] not in isp_names:
            raise ValueError(
                f"Unknown ISP in {context}.preferred_isp:
{policy['preferred_isp']}"
            )


def validate_config(cfg: dict) -> None:
    require_fields(
        cfg,
        ["lab", "router", "client", "isps", "core", "global_services",
"neptune"],
        "root",
    )

    if not isinstance(cfg["isps"], list) or len(cfg["isps"]) == 0:
        raise ValueError("'isps' must be a non-empty list")

    require_fields(cfg["lab"], ["name", "lan_prefix"], "lab")
    require_fields(cfg["router"], ["name", "lan_ip", "lan_if"], "router")
    require_fields(cfg["client"], ["name", "ip", "ifname", "gateway"], "client")
    require_fields(cfg["core"], ["name"], "core")

    if not isinstance(cfg["global_services"], list) or
len(cfg["global_services"]) == 0:
        raise ValueError("'global_services' must be a non-empty list")

    global_service_names: set[str] = set()

    for idx, service in enumerate(cfg["global_services"], start=1):
        sctx = f"global_services[{idx}]"

        require_fields(
            service,
            ["name", "prefix", "core_if", "srv_if", "core_ip", "srv_ip"],
            sctx,
        )

        service_name = service["name"]
        if service_name in global_service_names:
            raise ValueError(f"Duplicated global service name in {sctx}:
{service_name}")
        global_service_names.add(service_name)

        check_ifname(service["core_if"], f"{sctx}.core_if")
        check_ifname(service["srv_if"], f"{sctx}.srv_if")

    check_ifname(cfg["router"]["lan_if"], "router.lan_if")
    check_ifname(cfg["client"]["ifname"], "client.ifname")

    isp_names: set[str] = set()
    marks_seen: set[int] = set()
```

```
    priorities_seen: set[int] = set()

    for idx, isp in enumerate(cfg["isps"], start=1):
        ctx = f"isps[{idx}]"
        require_fields(
            isp,
            [
                "name",
                "router_if",
                "isp_if",
                "wan_prefix",
                "router_wan_ip",
                "isp_ip",
                "delegated_prefix",
                "priority",
                "neptune",
                "core",
            ],
            ctx,
        )

        name = isp["name"]
        if name in isp_names:
            raise ValueError(f"Duplicated ISP name in {ctx}: {name}")
        isp_names.add(name)

        check_ifname(isp["router_if"], f"{ctx}.router_if")
        check_ifname(isp["isp_if"], f"{ctx}.isp_if")

        if not isinstance(isp["priority"], int):
            raise ValueError(f"{ctx}.priority must be an integer")
        if isp["priority"] in priorities_seen:
            raise ValueError(f"Duplicated ISP priority in {ctx}:
{isp['priority']}")
        priorities_seen.add(isp["priority"])

        require_fields(
            isp["neptune"],
            ["mark", "role", "weight", "health_target"],
            f"{ctx}.neptune",
        )

        mark = isp["neptune"]["mark"]
        role = isp["neptune"]["role"]
        weight = isp["neptune"]["weight"]

        if not isinstance(mark, int) or mark <= 0:
            raise ValueError(f"{ctx}.neptune.mark must be a positive integer")
        if mark in marks_seen:
            raise ValueError(f"Duplicated ISP mark in {ctx}: {mark}")
        marks_seen.add(mark)

        if role not in {"primary", "backup", "excluded"}:
            raise ValueError(
                f"Invalid {ctx}.neptune.role: {role}. "
                f"Allowed values: primary, backup, excluded"
            )

        if not isinstance(weight, int) or weight < 0:
            raise ValueError(f"{ctx}.neptune.weight must be an integer >= 0")
```

```
        cctx = f"{ctx}.core"
        require_fields(
            isp["core"],
            ["isp_if", "core_if", "isp_ip", "core_ip"],
            cctx,
        )
        check_ifname(isp["core"]["isp_if"], f"{cctx}.isp_if")
        check_ifname(isp["core"]["core_if"], f"{cctx}.core_if")

        if "remote" in isp:
            remote = isp["remote"]
            rctx = f"{ctx}.remote"
            require_fields(
                remote,
                ["name", "isp_if", "remote_if", "prefix", "isp_ip", "remote_ip"],
                rctx,
            )
            check_ifname(remote["isp_if"], f"{rctx}.isp_if")
            check_ifname(remote["remote_if"], f"{rctx}.remote_if")

    neptune = cfg["neptune"]
    require_fields(
        neptune,
        [
            "check_interval",
            "health_log_interval",
            "flow_gc_interval",
            "flow_timeout_seconds",
            "excluded_destination_prefixes",
            "policies",
        ],
        "neptune",
    )

    if not isinstance(neptune["excluded_destination_prefixes"], list):
        raise ValueError("neptune.excluded_destination_prefixes must be a list")

    if not isinstance(neptune["policies"], list):
        raise ValueError("neptune.policies must be a list")

    for idx, policy in enumerate(neptune["policies"], start=1):
        if not isinstance(policy, dict):
            raise ValueError(f"neptune.policies[{idx}] must be a mapping")
        validate_neptune_policy(policy, f"neptune.policies[{idx}]", isp_names)


def generate_routing_script(cfg: dict) -> str:
    router = cfg["router"]
    client = cfg["client"]
    isps = cfg["isps"]
    core = cfg["core"]
    global_services = cfg["global_services"]

    router_name = router["name"]
    client_name = client["name"]
    core_name = core["name"]
    lan_prefix = cfg["lab"]["lan_prefix"]

    lines = [
        "#!/usr/bin/env bash",
        "set -euo pipefail",
```

```
        "",
        "# Auto-generated by generator.py",
        "",
        "echo '[+] Applying routing configuration...'",
        "",
        "echo '[+] Enabling IPv6 forwarding on router...'",
        f"sudo ip netns exec {router_name} sysctl -w
net.ipv6.conf.all.forwarding=1 >/dev/null",
        "",
        "echo '[+] Enabling IPv6 forwarding on core...'",
        f"sudo ip netns exec {core_name} sysctl -w net.ipv6.conf.all.forwarding=1
>/dev/null",
        "",
        "echo '[+] Preserving IPv6 addresses on WAN link down events...'",
    ]

    for isp in isps:
        lines.append(
            f"sudo ip netns exec {router_name} sysctl -w
net.ipv6.conf.{isp['router_if']}.keep_addr_on_down=1 >/dev/null"
        )
        lines.append(
            f"sudo ip netns exec {isp['name']} sysctl -w
net.ipv6.conf.{isp['isp_if']}.keep_addr_on_down=1 >/dev/null"
        )

    lines += [
        "",
        "echo '[+] Adding client default route...'",
        f"sudo ip netns exec {client_name} ip -6 route replace default via
{client['gateway']}",
    ]

    for isp in isps:
        lines += [
            "",
            f"echo '[+] Enabling IPv6 forwarding on {isp['name']}...'",
            f"sudo ip netns exec {isp['name']} sysctl -w
net.ipv6.conf.all.forwarding=1 >/dev/null",
        ]

    for isp in isps:
        if "remote" in isp:
            remote = isp["remote"]
            remote_prefix = remote["prefix"]
            isp_wan_ip = isp["isp_ip"].split("/")[0]
            router_wan_ip = isp["router_wan_ip"].split("/")[0]
            remote_isp_ip = remote["isp_ip"].split("/")[0]
            delegated_prefix = isp["delegated_prefix"]

            lines += [
                "",
                f"echo '[+] Adding base routes for {remote['name']} via
{isp['name']}...'",
                f"sudo ip netns exec {router_name} ip -6 route replace
{remote_prefix} via {isp_wan_ip}",
                f"sudo ip netns exec {isp['name']} ip -6 route replace
{lan_prefix} via {router_wan_ip}",
                f"sudo ip netns exec {isp['name']} ip -6 route replace
{delegated_prefix} via {router_wan_ip}",
```

```
                f"sudo ip netns exec {remote['name']} ip -6 route replace default
via {remote_isp_ip}",
            ]

    for service in global_services:
        lines += [
            "",
            f"echo '[+] Adding default route on {service['name']} towards
core...'",
            f"sudo ip netns exec {service['name']} ip -6 route replace default
via {service['core_ip'].split('/')[0]}",
        ]

    for isp in isps:
        c = isp["core"]
        isp_name = isp["name"]
        core_side_ip = c["core_ip"].split("/")[0]
        isp_side_ip = c["isp_ip"].split("/")[0]
        delegated_prefix = isp["delegated_prefix"]
        core_if = c["core_if"]

        lines += [
            "",
            f"echo '[+] Adding routes between {isp_name}, core and srv-
global...'",
            f"sudo ip netns exec {core_name} ip -6 route replace
{delegated_prefix} via {isp_side_ip} dev {core_if}",
        ]

        lines += [
            "",
            f"echo '[+] Adding routes between {isp_name}, core and global
services...'",
        ]

        for service in global_services:
            lines.append(
                f"sudo ip netns exec {isp_name} ip -6 route replace
{service['prefix']} via {core_side_ip}"
            )

        lines.append(
            f"sudo ip netns exec {core_name} ip -6 route replace
{delegated_prefix} via {isp_side_ip} dev {core_if}"
        )

    lines += [
        "",
        "echo '[+] Cleaning previous backend routing tables...'",
    ]

    for isp in isps:
        table = isp["priority"]
        lines.append(
            f"sudo ip netns exec {router_name} ip -6 route flush table {table}
2>/dev/null || true"
        )

    lines += [
        "",
        "echo '[+] Building per-ISP backend routing tables...'",
```

```
    ]

    for isp in isps:
        table = isp["priority"]
        isp_wan_ip = isp["isp_ip"].split("/")[0]
        wan_prefix = isp["wan_prefix"]
        wan_if = isp["router_if"]

        lines += [
            f"echo '    - Table {table} via {isp['name']}'",
            f"sudo ip netns exec {router_name} ip -6 route replace {wan_prefix}
dev {wan_if} table {table}",
            f"sudo ip netns exec {router_name} ip -6 route replace default via
{isp_wan_ip} dev {wan_if} table {table}",
        ]

    for service in global_services:
        lines.append(
            f"sudo ip netns exec {router_name} ip -6 route replace
{service['prefix']} via {isp_wan_ip} dev {wan_if} table {table}"
        )

    lines += [
        "",
        "echo '[+] Rebuilding IPv6 policy rules for fwmark steering...'",
    ]

    for idx, isp in enumerate(isps, start=100):
        lines.append(
            f"sudo ip netns exec {router_name} ip -6 rule del priority {idx}
2>/dev/null || true"
        )

    for idx, isp in enumerate(isps, start=100):
        mark = isp["neptune"]["mark"]
        table = isp["priority"]
        lines.append(
            f"sudo ip netns exec {router_name} ip -6 rule add fwmark 0x{mark:x}
lookup {table} priority {idx}"
        )

    lines += [
        "",
        "echo '[+] Router rules:'",
        f"sudo ip netns exec {router_name} ip -6 rule",
        "",
        "echo '[+] Main table:'",
        f"sudo ip netns exec {router_name} ip -6 route show table main",
    ]

    for isp in isps:
        table = isp["priority"]
        lines += [
            "",
            f"echo '[+] Table {table}:'",
            f"sudo ip netns exec {router_name} ip -6 route show table {table} ||
true",
        ]

    for service in global_services:
        lines += [
```

```
            "",
            f"echo '[+] {service['name']} routes:'",
            f"sudo ip netns exec {service['name']} ip -6 route",
        ]

    lines += [
        "",
        "echo '[+] Core routes:'",
        f"sudo ip netns exec {core_name} ip -6 route",
        "",
        "echo '[+] Routing configuration complete.'",
    ]

    return "\n".join(lines) + "\n"


def generate_nptv6_script(cfg: dict) -> str:
    router_name = cfg["router"]["name"]
    lan_if = cfg["router"]["lan_if"]

    lines = [
        "#!/usr/bin/env bash",
        "set -euo pipefail",
        "",
        "ACTION=\"${1:-}\"",
        "",
        "if [[ -z \"$ACTION\" ]]; then",
        "  echo 'Usage: nptv6.sh clear | apply <wan_if> <inside_prefix>
<outside_prefix>'",
        "  exit 1",
        "fi",
        "",
        "case \"$ACTION\" in",
        "  clear)",
        "    echo '[+] Clearing NPTv6 rules (mangle table)...'",
        f"    sudo ip netns exec {router_name} ip6tables -t mangle -F",
        "    ;;",
        "",
        "  apply)",
        "    WAN_IF=\"${2:-}\"",
        "    INSIDE_PREFIX=\"${3:-}\"",
        "    OUTSIDE_PREFIX=\"${4:-}\"",
        "",
        "    if [[ -z \"$WAN_IF\" || -z \"$INSIDE_PREFIX\" || -z
\"$OUTSIDE_PREFIX\" ]]; then",
        "      echo 'Usage: nptv6.sh apply <wan_if> <inside_prefix>
<outside_prefix>'",
        "      exit 1",
        "    fi",
        "",
        "    echo \"[+] Applying NPTv6 on ${WAN_IF}: ${INSIDE_PREFIX} ->
${OUTSIDE_PREFIX}\"",
        "",
        "    # Never touch local LAN traffic",
        f"    sudo ip netns exec {router_name} ip6tables -t mangle -A POSTROUTING
-d \"$INSIDE_PREFIX\" -j RETURN",
        f"    sudo ip netns exec {router_name} ip6tables -t mangle -A POSTROUTING
-o {lan_if} -j RETURN",
        "",
        "    # DNPT: inbound traffic from WAN back to inside prefix",
```

```
        f"    sudo ip netns exec {router_name} ip6tables -t mangle -A PREROUTING \\",
        "      -i \"$WAN_IF\" -d \"$OUTSIDE_PREFIX\" \\",
        "      -j DNPT --src-pfx \"$OUTSIDE_PREFIX\" --dst-pfx \"$INSIDE_PREFIX\"",
        "",
        "    # SNPT: outbound traffic from inside prefix to WAN",
        f"    sudo ip netns exec {router_name} ip6tables -t mangle -A POSTROUTING \\",
        "      -s \"$INSIDE_PREFIX\" ! -d \"$INSIDE_PREFIX\" -o \"$WAN_IF\" \\",
        "      -j SNPT --src-pfx \"$INSIDE_PREFIX\" --dst-pfx \"$OUTSIDE_PREFIX\"",
        "    ;;",
        "",
        "  *)",
        "    echo 'Usage: nptv6.sh clear | apply <wan_if> <inside_prefix> <outside_prefix>'",
        "    exit 1",
        "    ;;",
        "esac",
        "",
    ]

    return "\n".join(lines) + "\n"


def generate_setup_script(cfg: dict) -> str:
    router = cfg["router"]
    client = cfg["client"]
    isps = cfg["isps"]
    core = cfg["core"]
    global_services = cfg["global_services"]

    router_name = router["name"]
    client_name = client["name"]
    core_name = core["name"]

    remote_names = [isp["remote"]["name"] for isp in isps if "remote" in isp]

    lines = [
        "#!/usr/bin/env bash",
        "set -euo pipefail",
        "",
        "# Auto-generated by generator.py",
        "",
        "echo '[+] Creating namespaces...'",
        f"sudo ip netns add {client_name}",
        f"sudo ip netns add {router_name}",
        f"sudo ip netns add {core_name}",
    ]

    for service in global_services:
        lines.append(f"sudo ip netns add {service['name']}")

    for isp in isps:
        lines.append(f"sudo ip netns add {isp['name']}")

    for remote_name in remote_names:
        lines.append(f"sudo ip netns add {remote_name}")

    lines += [
```

```
        "",
        "echo '[+] Creating veth pairs...'",
        f"sudo ip link add {client['ifname']} type veth peer name
{router['lan_if']}",
    ]

    for isp in isps:
        lines.append(
            f"sudo ip link add {isp['router_if']} type veth peer name
{isp['isp_if']}"
        )

    for isp in isps:
        if "remote" in isp:
            remote = isp["remote"]
            lines.append(
                f"sudo ip link add {remote['isp_if']} type veth peer name
{remote['remote_if']}"
            )

    for isp in isps:
        c = isp["core"]
        lines.append(
            f"sudo ip link add {c['isp_if']} type veth peer name {c['core_if']}"
        )

    for service in global_services:
        lines.append(
            f"sudo ip link add {service['core_if']} type veth peer name
{service['srv_if']}"
        )

    lines += [
        "",
        "echo '[+] Moving interfaces to namespaces...'",
        f"sudo ip link set {client['ifname']} netns {client_name}",
        f"sudo ip link set {router['lan_if']} netns {router_name}",
    ]

    for isp in isps:
        lines.append(f"sudo ip link set {isp['router_if']} netns {router_name}")
        lines.append(f"sudo ip link set {isp['isp_if']} netns {isp['name']}")

    for isp in isps:
        if "remote" in isp:
            remote = isp["remote"]
            lines.append(f"sudo ip link set {remote['isp_if']} netns
{isp['name']}")
            lines.append(f"sudo ip link set {remote['remote_if']} netns
{remote['name']}")

    for isp in isps:
        c = isp["core"]
        lines.append(f"sudo ip link set {c['isp_if']} netns {isp['name']}")
        lines.append(f"sudo ip link set {c['core_if']} netns {core_name}")

    for service in global_services:
        lines.append(f"sudo ip link set {service['core_if']} netns {core_name}")
        lines.append(f"sudo ip link set {service['srv_if']} netns
{service['name']}")
```

```
    lines += [
        "",
        "echo '[+] Bringing loopback interfaces up...'",
        f"sudo ip netns exec {client_name} ip link set lo up",
        f"sudo ip netns exec {router_name} ip link set lo up",
        f"sudo ip netns exec {core_name} ip link set lo up",
    ]

    for service in global_services:
        lines.append(f"sudo ip netns exec {service['name']} ip link set lo up")

    for isp in isps:
        lines.append(f"sudo ip netns exec {isp['name']} ip link set lo up")

    for remote_name in remote_names:
        lines.append(f"sudo ip netns exec {remote_name} ip link set lo up")

    lines += [
        "",
        "echo '[+] Assigning IPv6 addresses...'",
        f"sudo ip netns exec {client_name} ip -6 addr add {client['ip']} dev
{client['ifname']}",
        f"sudo ip netns exec {router_name} ip -6 addr add {router['lan_ip']} dev
{router['lan_if']}",
    ]

    for isp in isps:
        lines.append(
            f"sudo ip netns exec {router_name} ip -6 addr add
{isp['router_wan_ip']} dev {isp['router_if']}"
        )
        lines.append(
            f"sudo ip netns exec {isp['name']} ip -6 addr add {isp['isp_ip']} dev
{isp['isp_if']}"
        )

    for isp in isps:
        if "remote" in isp:
            remote = isp["remote"]
            lines.append(
                f"sudo ip netns exec {isp['name']} ip -6 addr add
{remote['isp_ip']} dev {remote['isp_if']}"
            )
            lines.append(
                f"sudo ip netns exec {remote['name']} ip -6 addr add
{remote['remote_ip']} dev {remote['remote_if']}"
            )

    for isp in isps:
        c = isp["core"]
        lines.append(
            f"sudo ip netns exec {isp['name']} ip -6 addr add {c['isp_ip']} dev
{c['isp_if']}"
        )
        lines.append(
            f"sudo ip netns exec {core_name} ip -6 addr add {c['core_ip']} dev
{c['core_if']}"
        )

    for service in global_services:
        lines.append(
```

```
            f"sudo ip netns exec {core_name} ip -6 addr add {service['core_ip']}
dev {service['core_if']}"
        )
        lines.append(
            f"sudo ip netns exec {service['name']} ip -6 addr add
{service['srv_ip']} dev {service['srv_if']}"
        )

    lines += [
        "",
        "echo '[+] Bringing interfaces up...'",
        f"sudo ip netns exec {client_name} ip link set {client['ifname']} up",
        f"sudo ip netns exec {router_name} ip link set {router['lan_if']} up",
    ]

    for service in global_services:
        lines.append(f"sudo ip netns exec {core_name} ip link set
{service['core_if']} up")
        lines.append(f"sudo ip netns exec {service['name']} ip link set
{service['srv_if']} up")

    for isp in isps:
        lines.append(
            f"sudo ip netns exec {router_name} ip link set {isp['router_if']} up"
        )
        lines.append(
            f"sudo ip netns exec {isp['name']} ip link set {isp['isp_if']} up"
        )

    for isp in isps:
        if "remote" in isp:
            remote = isp["remote"]
            lines.append(
                f"sudo ip netns exec {isp['name']} ip link set {remote['isp_if']}
up"
            )
            lines.append(
                f"sudo ip netns exec {remote['name']} ip link set
{remote['remote_if']} up"
            )

    for isp in isps:
        c = isp["core"]
        lines.append(
            f"sudo ip netns exec {isp['name']} ip link set {c['isp_if']} up"
        )
        lines.append(
            f"sudo ip netns exec {core_name} ip link set {c['core_if']} up"
        )

    lines += [
        "",
        "sleep 1",
        "",
        "echo '[+] Current namespaces:'",
        "ip netns list",
        "",
        "echo '[+] Router addresses:'",
        f"sudo ip netns exec {router_name} ip -6 addr",
        "",
        "echo '[+] Core addresses:'",
```

```
        f"sudo ip netns exec {core_name} ip -6 addr",
        "",
        "echo '[+] Setup complete.'",
    ]

    return "\n".join(lines) + "\n"


def generate_event_scripts(cfg: dict) -> dict[str, str]:
    router_name = cfg["router"]["name"]
    lan_prefix = cfg["lab"]["lan_prefix"]
    events: dict[str, str] = {}

    for isp in cfg["isps"]:
        isp_name = isp["name"]
        router_if = isp["router_if"]
        isp_if = isp["isp_if"]
        router_wan_ip = isp["router_wan_ip"].split("/")[0]
        isp_wan_ip = isp["isp_ip"].split("/")[0]
        delegated_prefix = isp["delegated_prefix"]

        # Backend routing table used by fwmark-based policy routing
        table = isp["priority"]
        wan_prefix = isp["wan_prefix"]

        down_name = f"{isp_name}_down.sh"
        up_name = f"{isp_name}_up.sh"

        down_lines = [
            "#!/usr/bin/env bash",
            "set -euo pipefail",
            "",
            f"echo '[+] Bringing down link {router_if}<->{isp_if}
({router_name}<->{isp_name})...'",
            f"sudo ip netns exec {router_name} ip link set {router_if} down",
            f"sudo ip netns exec {isp_name} ip link set {isp_if} down",
            f"sudo ip netns exec {router_name} ip link show {router_if}",
            f"sudo ip netns exec {isp_name} ip link show {isp_if}",
            "",
        ]

        up_lines = [
            "#!/usr/bin/env bash",
            "set -euo pipefail",
            "",
            f"echo '[+] Bringing up link {router_if}<->{isp_if} ({router_name}<-
>{isp_name})...'",
            f"sudo ip netns exec {router_name} ip link set {router_if} up",
            f"sudo ip netns exec {isp_name} ip link set {isp_if} up",
            "sleep 2",
            f"sudo ip netns exec {router_name} ip link show {router_if}",
            f"sudo ip netns exec {isp_name} ip link show {isp_if}",
            f"sudo ip netns exec {router_name} ip -6 addr show dev {router_if}",
            f"sudo ip netns exec {isp_name} ip -6 addr show dev {isp_if}",
            f"echo '[+] Restoring delegated prefix route for {isp_name}...'",
            f"sudo ip netns exec {isp_name} ip -6 route replace
{delegated_prefix} via {router_wan_ip}",
            f"sudo ip netns exec {isp_name} ip -6 route show {delegated_prefix}",
            "",
            f"echo '[+] Rebuilding backend routing table {table} for
{isp_name}...'",
```

```
            f"sudo ip netns exec {router_name} ip -6 route flush table {table}
2>/dev/null || true",
            f"sudo ip netns exec {router_name} ip -6 route replace {wan_prefix}
dev {router_if} table {table}",
            f"sudo ip netns exec {router_name} ip -6 route replace default via
{isp_wan_ip} dev {router_if} table {table}",
            f"sudo ip netns exec {router_name} ip -6 route show table {table}",
            "",
        ]

        for service in cfg["global_services"]:
            up_lines.append(
                f"sudo ip netns exec {router_name} ip -6 route replace
{service['prefix']} via {isp_wan_ip} dev {router_if} table {table}"
            )

        if "remote" in isp:
            remote = isp["remote"]
            remote_name = remote["name"]
            remote_prefix = remote["prefix"]
            remote_isp_ip = remote["isp_ip"].split("/")[0]

            up_lines += [
                f"echo '[+] Restoring routes for {isp_name} and
{remote_name}...'",
                f"sudo ip netns exec {router_name} ip -6 route replace
{remote_prefix} via {isp_wan_ip}",
                f"sudo ip netns exec {isp_name} ip -6 route replace {lan_prefix}
via {router_wan_ip}",
                f"sudo ip netns exec {remote_name} ip -6 route replace default
via {remote_isp_ip}",
                "",
                f"sudo ip netns exec {router_name} ip -6 route show
{remote_prefix}",
                f"sudo ip netns exec {isp_name} ip -6 route show {lan_prefix}",
                f"sudo ip netns exec {remote_name} ip -6 route show default",
                "",
            ]

        events[down_name] = "\n".join(down_lines)
        events[up_name] = "\n".join(up_lines)

    return events


def main() -> int:
    try:
        cfg = load_config()
        validate_config(cfg)

        OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
        EVENTS_DIR.mkdir(parents=True, exist_ok=True)

        setup_content = generate_setup_script(cfg)
        routing_content = generate_routing_script(cfg)
        event_scripts = generate_event_scripts(cfg)
        nptv6_content = generate_nptv6_script(cfg)

        with OUTPUT_FILE.open("w", encoding="utf-8") as f:
            f.write(setup_content)
        OUTPUT_FILE.chmod(0o755)
```

```
        with ROUTING_FILE.open("w", encoding="utf-8") as f:
            f.write(routing_content)
        ROUTING_FILE.chmod(0o755)

        with NPTV6_FILE.open("w", encoding="utf-8") as f:
            f.write(nptv6_content)
        NPTV6_FILE.chmod(0o755)

        for filename, content in event_scripts.items():
            path = EVENTS_DIR / filename
            with path.open("w", encoding="utf-8") as f:
                f.write(content)
            path.chmod(0o755)

        print("[OK] Configuration loaded successfully")
        print(f"[OK] Generated: {OUTPUT_FILE}")
        print(f"[OK] Generated: {ROUTING_FILE}")
        print(f"[OK] Generated: {NPTV6_FILE}")
        print(f"[OK] Generated events in: {EVENTS_DIR}")

        print("\nSummary:")
        print(f"  Lab:    {cfg['lab']['name']}")
        print(f"  Router: {cfg['router']['name']}")
        print(f"  Client: {cfg['client']['name']}")
        print(f"  ISPs:   {', '.join(isp['name'] for isp in cfg['isps'])}")

        return 0

    except Exception as e:
        print(f"[ERROR] {e}", file=sys.stderr)
        return 1


if __name__ == "__main__":
    raise SystemExit(main())
```

runner.py

```
from __future__ import annotations

from pathlib import Path
import subprocess
import sys
import time
import yaml


BASE_DIR = Path(__file__).resolve().parent.parent
SRC_DIR = BASE_DIR / "src"
SCRIPTS_DIR = BASE_DIR / "scripts"
GENERATED_DIR = SCRIPTS_DIR / "generated"
MANUAL_DIR = SCRIPTS_DIR / "manual"
RUNTIME_DIR = BASE_DIR / "runtime"

GENERATOR = SRC_DIR / "generator.py"
CONTROLLER = SRC_DIR / "controller.py"
DASHBOARD = SRC_DIR / "dashboard_server.py"

TEARDOWN = MANUAL_DIR / "teardown.sh"
SETUP = GENERATED_DIR / "setup.sh"
ROUTING = GENERATED_DIR / "routing.sh"
NPTV6 = GENERATED_DIR / "nptv6.sh"
EVENTS_DIR = GENERATED_DIR / "events"

CONFIG_FILE = BASE_DIR / "config" / "config.yaml"
CONTROLLER_PID_FILE = RUNTIME_DIR / "controller.pid"
CONTROLLER_LOG_FILE = RUNTIME_DIR / "controller.log"
DASHBOARD_PID_FILE = RUNTIME_DIR / "dashboard.pid"
DASHBOARD_LOG_FILE = RUNTIME_DIR / "dashboard.log"

def run(cmd: list[str]) -> None:
    print(f"[RUN] {' '.join(cmd)}")
    subprocess.run(cmd, check=True)


def load_config() -> dict:
    with CONFIG_FILE.open("r", encoding="utf-8") as f:
        data = yaml.safe_load(f)

    if not isinstance(data, dict):
        raise ValueError("Invalid YAML root")

    return data


def apply_base_nptv6(cfg: dict) -> None:
    lan_prefix = cfg["lab"]["lan_prefix"]
    isps = cfg["isps"]

    run(["bash", str(NPTV6), "clear"])

    for isp in isps:
        wan_if = isp["router_if"]
        outside_prefix = isp["delegated_prefix"]
        run(["bash", str(NPTV6), "apply", wan_if, lan_prefix, outside_prefix])
```

```
def cmd_apply() -> None:
    run(["sudo", str(TEARDOWN)])
    run(["python3", str(GENERATOR)])
    run(["bash", str(SETUP)])
    run(["bash", str(ROUTING)])

    cfg = load_config()
    apply_base_nptv6(cfg)

    print("[OK] NEPTUNE lab applied successfully")


def cmd_reset() -> None:
    cmd_stop_dashboard(ignore_missing=True)
    cmd_stop_controller(ignore_missing=True)
    run(["sudo", str(TEARDOWN)])


def cmd_event(event_name: str) -> None:
    event_script = EVENTS_DIR / f"{event_name}.sh"

    if not event_script.exists():
        raise FileNotFoundError(f"Event not found: {event_script}")

    run(["bash", str(event_script)])


def controller_pid() -> int | None:
    if not CONTROLLER_PID_FILE.exists():
        return None

    try:
        return int(CONTROLLER_PID_FILE.read_text(encoding="utf-8").strip())
    except Exception:
        return None

def dashboard_pid() -> int | None:
    if not DASHBOARD_PID_FILE.exists():
        return None

    try:
        return int(DASHBOARD_PID_FILE.read_text(encoding="utf-8").strip())
    except Exception:
        return None

def is_pid_alive(pid: int) -> bool:
    result = subprocess.run(
        ["kill", "-0", str(pid)],
        check=False,
        capture_output=True,
        text=True,
    )
    return result.returncode == 0


def cmd_start_controller() -> None:
    existing_pid = controller_pid()
    if existing_pid is not None and is_pid_alive(existing_pid):
        print(f"[OK] Controller already running with PID {existing_pid}")
        return
```

```
    RUNTIME_DIR.mkdir(parents=True, exist_ok=True)

    log_f = CONTROLLER_LOG_FILE.open("a", encoding="utf-8")
    process = subprocess.Popen(
        ["python3", "-u", str(CONTROLLER), "--mode", "neptune"],
        stdout=log_f,
        stderr=subprocess.STDOUT,
        text=True,
    )

    CONTROLLER_PID_FILE.write_text(str(process.pid), encoding="utf-8")
    time.sleep(0.5)

    if is_pid_alive(process.pid):
        print(f"[OK] Controller started with PID {process.pid}")
        print(f"[OK] Log file: {CONTROLLER_LOG_FILE}")
    else:
        raise RuntimeError("Controller failed to start")

def cmd_start_dashboard() -> None:
    existing_pid = dashboard_pid()
    if existing_pid is not None and is_pid_alive(existing_pid):
        print(f"[OK] Dashboard already running with PID {existing_pid}")
        return

    RUNTIME_DIR.mkdir(parents=True, exist_ok=True)

    log_f = DASHBOARD_LOG_FILE.open("a", encoding="utf-8")
    process = subprocess.Popen(
        ["python3", "-u", str(DASHBOARD)],
        stdout=log_f,
        stderr=subprocess.STDOUT,
        text=True,
    )

    DASHBOARD_PID_FILE.write_text(str(process.pid), encoding="utf-8")
    time.sleep(0.5)

    if is_pid_alive(process.pid):
        print(f"[OK] Dashboard started with PID {process.pid}")
        print(f"[OK] URL: http://127.0.0.1:8080")
        print(f"[OK] Log file: {DASHBOARD_LOG_FILE}")
    else:
        raise RuntimeError("Dashboard failed to start")

def cmd_stop_controller(ignore_missing: bool = False) -> None:
    pid = controller_pid()
    if pid is None:
        if ignore_missing:
            return
        print("[OK] Controller is not running")
        return

    if not is_pid_alive(pid):
        CONTROLLER_PID_FILE.unlink(missing_ok=True)
        if ignore_missing:
            return
        print("[OK] Controller was not alive; cleaned stale PID file")
        return
```

```
    print(f"[RUN] Stopping controller PID {pid}")
    subprocess.run(["kill", str(pid)], check=False)

    for _ in range(20):
        if not is_pid_alive(pid):
            CONTROLLER_PID_FILE.unlink(missing_ok=True)
            print("[OK] Controller stopped")
            return
        time.sleep(0.2)

    subprocess.run(["kill", "-9", str(pid)], check=False)
    CONTROLLER_PID_FILE.unlink(missing_ok=True)
    print("[OK] Controller force-stopped")

def cmd_stop_dashboard(ignore_missing: bool = False) -> None:
    pid = dashboard_pid()
    if pid is None:
        if ignore_missing:
            return
        print("[OK] Dashboard is not running")
        return

    if not is_pid_alive(pid):
        DASHBOARD_PID_FILE.unlink(missing_ok=True)
        if ignore_missing:
            return
        print("[OK] Dashboard was not alive; cleaned stale PID file")
        return

    print(f"[RUN] Stopping dashboard PID {pid}")
    subprocess.run(["kill", str(pid)], check=False)

    for _ in range(20):
        if not is_pid_alive(pid):
            DASHBOARD_PID_FILE.unlink(missing_ok=True)
            print("[OK] Dashboard stopped")
            return
        time.sleep(0.2)

    subprocess.run(["kill", "-9", str(pid)], check=False)
    DASHBOARD_PID_FILE.unlink(missing_ok=True)
    print("[OK] Dashboard force-stopped")

def cmd_status() -> None:
    pid = controller_pid()
    if pid is None:
        print("[STATUS] Controller: stopped")
    elif is_pid_alive(pid):
        print(f"[STATUS] Controller: running (PID {pid})")
        print(f"[STATUS] Controller log: {CONTROLLER_LOG_FILE}")
    else:
        print(f"[STATUS] Controller: stale PID file ({pid})")

    dash_pid = dashboard_pid()
    if dash_pid is None:
        print("[STATUS] Dashboard: stopped")
    elif is_pid_alive(dash_pid):
        print(f"[STATUS] Dashboard: running (PID {dash_pid})")
        print("[STATUS] Dashboard URL: http://127.0.0.1:8080")
        print(f"[STATUS] Dashboard log: {DASHBOARD_LOG_FILE}")
    else:
```

```
        print(f"[STATUS] Dashboard: stale PID file ({dash_pid})")

def print_usage() -> None:
    print("Usage:")
    print("  python3 src/runner.py apply")
    print("  python3 src/runner.py reset")
    print("  python3 src/runner.py event <event_name>")
    print("  python3 src/runner.py start-controller")
    print("  python3 src/runner.py stop-controller")
    print("  python3 src/runner.py start-dashboard")
    print("  python3 src/runner.py stop-dashboard")
    print("  python3 src/runner.py status")

def main() -> int:
    if len(sys.argv) < 2:
        print_usage()
        return 1

    action = sys.argv[1]

    try:
        if action == "apply":
            cmd_apply()

        elif action == "reset":
            cmd_reset()

        elif action == "event":
            if len(sys.argv) < 3:
                raise ValueError("Missing event name")
            cmd_event(sys.argv[2])

        elif action == "start-controller":
            cmd_start_controller()

        elif action == "stop-controller":
            cmd_stop_controller()

        elif action == "start-dashboard":
            cmd_start_dashboard()

        elif action == "stop-dashboard":
            cmd_stop_dashboard()

        elif action == "status":
            cmd_status()

        else:
            raise ValueError(f"Unknown action: {action}")

        return 0

    except Exception as e:
        print(f"[ERROR] {e}", file=sys.stderr)
        return 1


if __name__ == "__main__":
    raise SystemExit(main())
```

Configuration

config.yaml

```yaml
lab:
  name: neptune-lab
  lan_prefix: fd42:100:1::/64

router:
  name: router
  lan_if: veth-rlan
  lan_ip: fd42:100:1::1/64

client:
  name: client
  ifname: veth-cl
  ip: fd42:100:1::10/64
  gateway: fd42:100:1::1

core:
  name: core

global_services:
  - name: srv-global-pref
    prefix: 2001:db8:999:1::/64
    core_if: veth-core-srvp
    srv_if: veth-srvp
    core_ip: 2001:db8:999:1::1/64
    srv_ip: 2001:db8:999:1::10/64

  - name: srv-global-bal
    prefix: 2001:db8:998:1::/64
    core_if: veth-core-srvb
    srv_if: veth-srvb
    core_ip: 2001:db8:998:1::1/64
    srv_ip: 2001:db8:998:1::10/64

isps:
  - name: isp1
    router_if: veth-rw1
    isp_if: veth-i1
    wan_prefix: 2001:db8:10:1::/64
    router_wan_ip: 2001:db8:10:1::2/64
    isp_ip: 2001:db8:10:1::1/64
    delegated_prefix: 2001:db8:1100:1::/64
    priority: 100

    neptune:
      mark: 1
      role: primary
      weight: 7
      health_target: 2001:db8:10:1::1

    remote:
      name: srv1
      isp_if: veth-i1-s1
      remote_if: veth-s1
      prefix: 2001:db8:101:1::/64
      isp_ip: 2001:db8:101:1::1/64
```

```
    remote_ip: 2001:db8:101:1::10/64

  core:
    isp_if: veth-i1-core
    core_if: veth-core-i1
    isp_ip: 2001:db8:10:ff::1/64
    core_ip: 2001:db8:10:ff::2/64

- name: isp2
  router_if: veth-rw2
  isp_if: veth-i2
  wan_prefix: 2001:db8:20:1::/64
  router_wan_ip: 2001:db8:20:1::2/64
  isp_ip: 2001:db8:20:1::1/64
  delegated_prefix: 2001:db8:2200:1::/64
  priority: 200

  neptune:
    mark: 2
    role: primary
    weight: 3
    health_target: 2001:db8:20:1::1

  remote:
    name: srv2
    isp_if: veth-i2-s2
    remote_if: veth-s2
    prefix: 2001:db8:202:1::/64
    isp_ip: 2001:db8:202:1::1/64
    remote_ip: 2001:db8:202:1::10/64

  core:
    isp_if: veth-i2-core
    core_if: veth-core-i2
    isp_ip: 2001:db8:20:ff::1/64
    core_ip: 2001:db8:20:ff::2/64

- name: isp3
  router_if: veth-rw3
  isp_if: veth-i3
  wan_prefix: 2001:db8:30:1::/64
  router_wan_ip: 2001:db8:30:1::2/64
  isp_ip: 2001:db8:30:1::1/64
  delegated_prefix: 2001:db8:3300:1::/64
  priority: 300

  neptune:
    mark: 3
    role: backup
    weight: 1
    health_target: 2001:db8:30:1::1

  remote:
    name: srv3
    isp_if: veth-i3-s3
    remote_if: veth-s3
    prefix: 2001:db8:303:1::/64
    isp_ip: 2001:db8:303:1::1/64
    remote_ip: 2001:db8:303:1::10/64

  core:
```

```
      isp_if: veth-i3-core
      core_if: veth-core-i3
      isp_ip: 2001:db8:30:ff::1/64
      core_ip: 2001:db8:30:ff::2/64

neptune:
  check_interval: 0.1
  health_log_interval: 0.5
  flow_gc_interval: 5
  flow_timeout_seconds: 30

  excluded_destination_prefixes:
    - fd42:100:1::/64
    - fe80::/10
    - ff00::/8

  policies:
    - prefix: 2001:db8:101:1::/64
      mode: strict
      isp: isp1

    - prefix: 2001:db8:202:1::/64
      mode: strict
      isp: isp2

    - prefix: 2001:db8:303:1::/64
      mode: strict
      isp: isp3
```

Dashboard Front End

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>NEPTUNE Dashboard</title>
  <link rel="stylesheet" href="/static/style.css" />
</head>
<body>
  <header>
    <div>
      <h1>NEPTUNE Dashboard</h1>
      <p>Operational visualization layer</p>
    </div>
    <div id="controller-status" class="status-pill">Loading...</div>
  </header>

  <main>
    <section class="card">
      <h2>ISPs</h2>
      <div id="isp-grid" class="isp-grid"></div>
    </section>

    <section class="card">
      <h2>Active Pool</h2>
      <div id="active-pool" class="pool"></div>
    </section>

    <section class="card wide">
      <h2>Flow Table</h2>
      <div class="table-scroll">
        <table>
          <thead>
            <tr>
              <th>Source</th>
              <th>Destination</th>
              <th>Protocol</th>
              <th>ISP</th>
              <th>Mark</th>
              <th>Flow ID</th>
            </tr>
          </thead>
          <tbody id="flow-table"></tbody>
        </table>
      </div>
    </section>

    <section class="card">
      <h2>Ping Tool</h2>
      <div class="form-row">
        <label>Source</label>
        <select id="ping-source"></select>
      </div>
      <div class="form-row">
        <label>Destination</label>
        <select id="ping-destination"></select>
```

```
      </div>
      <button id="ping-btn">Run Ping</button>
      <pre id="ping-output"></pre>
    </section>

    <section class="card">
      <h2>Continuous Ping</h2>

      <div class="form-row">
        <label>Source</label>
        <select id="continuous-source"></select>
      </div>

      <div class="form-row">
        <label>Destination</label>
        <select id="continuous-destination"></select>
      </div>

      <div class="button-row">
        <button id="continuous-start">Start</button>
        <button id="continuous-stop" class="danger">Stop</button>
      </div>

      <pre id="continuous-output"></pre>
    </section>

    <section class="card">
      <h2>Distribution Test</h2>

      <div class="form-row">
        <label>Source</label>
        <select id="distribution-source"></select>
      </div>

      <div class="form-row">
        <label>Destination</label>
        <select id="distribution-destination"></select>
      </div>

      <div class="form-row">
        <label>Iterations</label>
        <input id="distribution-iterations" type="number" value="30" min="1"
max="200" />
      </div>

      <button id="distribution-run">Run Distribution Test</button>

      <pre id="distribution-output"></pre>
    </section>

    <section class="card">
      <h2>Events</h2>
      <div class="form-row">
        <label>ISP</label>
        <select id="event-isp"></select>
      </div>
      <div class="button-row">
        <button id="event-up">UP</button>
        <button id="event-down" class="danger">DOWN</button>
      </div>
      <pre id="event-output"></pre>
```

```
    </section>

    <section class="card wide">
      <h2>Event Log</h2>
      <div id="event-log" class="event-log"></div>
    </section>
  </main>

  <footer>
    Last update: <span id="last-update">-</span>
  </footer>

  <script src="/static/app.js"></script>
</body>
</html>
```

style.css

```
* {
  box-sizing: border-box;
}

body {
  margin: 0;
  font-family: system-ui, -apple-system, BlinkMacSystemFont, "Segoe UI", sans-
serif;
  background: #0f172a;
  color: #e5e7eb;
}

header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 24px 32px;
  border-bottom: 1px solid #1e293b;
  background: #020617;
}

h1, h2, p {
  margin: 0;
}

h1 {
  font-size: 28px;
}

header p {
  margin-top: 6px;
  color: #94a3b8;
}

main {
  display: grid;
  grid-template-columns: repeat(2, minmax(0, 1fr));
  gap: 20px;
  padding: 24px 32px;
}

.card {
  background: #111827;
  border: 1px solid #1f2937;
  border-radius: 18px;
  padding: 20px;
  box-shadow: 0 12px 30px rgba(0, 0, 0, 0.25);
}

.card h2 {
  margin-bottom: 16px;
  font-size: 18px;
}

.wide {
  grid-column: span 2;
}
```

```
.status-pill {
  padding: 8px 14px;
  border-radius: 999px;
  background: #334155;
  font-weight: 600;
}

.status-running {
  background: #064e3b;
  color: #bbf7d0;
}

.status-stopped {
  background: #7f1d1d;
  color: #fecaca;
}

.isp-grid {
  display: grid;
  gap: 14px;
}

.isp-card {
  display: flex;
  justify-content: space-between;
  align-items: center;
  background: #020617;
  border: 1px solid #1e293b;
  padding: 14px;
  border-radius: 14px;
}

.isp-name {
  font-weight: 700;
  font-size: 17px;
}

.isp-meta {
  margin-top: 4px;
  color: #94a3b8;
  font-size: 13px;
}

.light {
  width: 18px;
  height: 18px;
  border-radius: 50%;
}

.up {
  background: #22c55e;
  box-shadow: 0 0 16px #22c55e;
}

.down {
  background: #ef4444;
  box-shadow: 0 0 16px #ef4444;
}

.unknown {
  background: #64748b;
```

```
}

.pool {
  display: flex;
  gap: 10px;
  flex-wrap: wrap;
}

.pool span {
  background: #1e293b;
  padding: 8px 12px;
  border-radius: 999px;
  font-weight: 600;
}

table {
  width: 100%;
  border-collapse: collapse;
  overflow: hidden;
}

th, td {
  padding: 10px;
  border-bottom: 1px solid #1f2937;
  text-align: left;
  font-size: 14px;
}

th {
  color: #94a3b8;
  font-weight: 600;
}

.form-row {
  display: flex;
  flex-direction: column;
  gap: 6px;
  margin-bottom: 12px;
}

label {
  color: #94a3b8;
  font-size: 14px;
}

select, button, input {
  border: none;
  border-radius: 10px;
  padding: 10px 12px;
  font-size: 14px;
}

select, input {
  background: #020617;
  color: #e5e7eb;
  border: 1px solid #334155;
}

button {
  background: #2563eb;
  color: white;
```

```
  font-weight: 700;
  cursor: pointer;
}

button:hover {
  opacity: 0.9;
}

.danger {
  background: #dc2626;
}

.button-row {
  display: flex;
  gap: 10px;
}

pre {
  margin-top: 14px;
  max-height: 220px;
  overflow: auto;
  background: #020617;
  border: 1px solid #1e293b;
  border-radius: 12px;
  padding: 12px;
  color: #cbd5e1;
  font-size: 13px;
  white-space: pre-wrap;
}

.event-log {
  display: flex;
  flex-direction: column;
  gap: 8px;
  max-height: 260px;
  overflow: auto;
}

.event-item {
  display: grid;
  grid-template-columns: 180px 100px 1fr;
  gap: 10px;
  padding: 10px;
  background: #020617;
  border: 1px solid #1e293b;
  border-radius: 10px;
  font-size: 14px;
}

.event-time {
  color: #94a3b8;
}

.event-type {
  font-weight: 700;
  color: #93c5fd;
}

footer {
  padding: 16px 32px;
  color: #94a3b8;
```

```
  border-top: 1px solid #1e293b;
}

.table-scroll {
  max-height: 320px;
  overflow: auto;
  border: 1px solid #1f2937;
  border-radius: 12px;
}

.table-scroll table {
  min-width: 900px;
}

.table-scroll thead th {
  position: sticky;
  top: 0;
  background: #111827;
  z-index: 1;
}
```

app.js

```
let config = null;

function populateExtraSelectors() {
  const pairs = [
    ["continuous-source", "continuous-destination"],
    ["distribution-source", "distribution-destination"],
  ];

  pairs.forEach(([sourceId, destinationId]) => {
    const sourceSelect = document.getElementById(sourceId);
    const destinationSelect = document.getElementById(destinationId);

    sourceSelect.innerHTML = "";
    destinationSelect.innerHTML = "";

    config.sources.forEach((source) => {
      const option = document.createElement("option");
      option.value = source.namespace;
      option.textContent = `${source.name} (${source.ip})`;
      sourceSelect.appendChild(option);
    });

    config.destinations.forEach((destination) => {
      const option = document.createElement("option");
      option.value = destination.ip;
      option.textContent = `${destination.name} · ${destination.ip}`;
      destinationSelect.appendChild(option);
    });
  });
}

async function startContinuousPing() {
  const source = document.getElementById("continuous-source").value;
  const destinationIp = document.getElementById("continuous-destination").value;
  const output = document.getElementById("continuous-output");

  const result = await fetchJson("/api/ping/continuous/start", {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify({
      source: source,
      destination_ip: destinationIp,
      interval: 1.0,
    }),
  });

  output.textContent = result.ok ? `$ ${result.cmd}\n` : result.error;
}

async function stopContinuousPing() {
  await fetchJson("/api/ping/continuous/stop", {
    method: "POST",
  });
}

async function refreshContinuousPing() {
  const output = document.getElementById("continuous-output");
  const result = await fetchJson("/api/ping/continuous/status");
```

```
  if (result.lines && result.lines.length > 0) {
    output.textContent = result.lines.join("\n");
    output.scrollTop = output.scrollHeight;
  }
}

async function runDistributionTest() {
  const source = document.getElementById("distribution-source").value;
  const destinationIp = document.getElementById("distribution-
destination").value;
  const iterations = Number(document.getElementById("distribution-
iterations").value);
  const output = document.getElementById("distribution-output");

  output.textContent = "Running distribution test...";

  const result = await fetchJson("/api/tests/distribution", {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify({
      source: source,
      destination_ip: destinationIp,
      iterations: iterations,
    }),
  });

  let text = "Distribution:\n";

  Object.entries(result.distribution || {}).forEach(([isp, count]) => {
    const percentage = ((count / result.iterations) * 100).toFixed(1);
    text += `${isp}: ${count}/${result.iterations} (${percentage}%)\n`;
  });

  text += "\nRuns:\n";

  result.results.forEach((item) => {
    text += `#${item.iteration}: ${item.ok ? "OK" : "FAIL"} → ${item.selected_isp
|| "unknown"} flow_id=${item.flow_id || "-"}\n`;
  });

  output.textContent = text;

  await refreshState();
}

async function fetchJson(url, options = {}) {
  const response = await fetch(url, options);
  return await response.json();
}

function renderController(state) {
  const pill = document.getElementById("controller-status");
  const status = state.controller?.status || "unknown";

  pill.textContent = `Controller: ${status}`;
  pill.className = "status-pill";

  if (status === "running") {
    pill.classList.add("status-running");
  } else {
```

```
    pill.classList.add("status-stopped");
  }
}

function renderIsps(state) {
  const grid = document.getElementById("isp-grid");
  grid.innerHTML = "";

  const isps = state.isps || {};

  Object.entries(isps).forEach(([name, isp]) => {
    const card = document.createElement("div");
    card.className = "isp-card";

    const left = document.createElement("div");

    const title = document.createElement("div");
    title.className = "isp-name";
    title.textContent = name;

    const meta = document.createElement("div");
    meta.className = "isp-meta";
    meta.textContent = `${isp.role || "-"} · target ${isp.health_target || "-"}`;

    left.appendChild(title);
    left.appendChild(meta);

    const light = document.createElement("div");
    light.className = `light ${isp.status || "unknown"}`;

    card.appendChild(left);
    card.appendChild(light);
    grid.appendChild(card);
  });
}

function renderActivePool(state) {
  const pool = document.getElementById("active-pool");
  pool.innerHTML = "";

  const activePool = state.active_pool || [];

  if (activePool.length === 0) {
    pool.textContent = "No active ISP available";
    return;
  }

  activePool.forEach((isp) => {
    const span = document.createElement("span");
    span.textContent = isp;
    pool.appendChild(span);
  });
}

function renderFlows(state) {
  const tbody = document.getElementById("flow-table");
  tbody.innerHTML = "";

  const flows = state.flows || [];

  if (flows.length === 0) {
```

```
    const row = document.createElement("tr");
    row.innerHTML = `<td colspan="6">No active flows</td>`;
    tbody.appendChild(row);
    return;
  }

  flows.forEach((flow) => {
    const row = document.createElement("tr");
    row.innerHTML = `
      <td>${flow.src_ip}</td>
      <td>${flow.dst_ip}</td>
      <td>${flow.proto}</td>
      <td>${flow.selected_isp}</td>
      <td>${flow.mark}</td>
      <td>${flow.flow_id}</td>
    `;
    tbody.appendChild(row);
  });
}

function renderEvents(state) {
  const log = document.getElementById("event-log");
  log.innerHTML = "";

  const events = (state.events || []).slice().reverse();

  if (events.length === 0) {
    log.textContent = "No events yet";
    return;
  }

  events.forEach((event) => {
    const item = document.createElement("div");
    item.className = "event-item";

    item.innerHTML = `
      <div class="event-time">${event.time || "-"}</div>
      <div class="event-type">${event.type || "-"}</div>
      <div>${event.message || ""}</div>
    `;

    log.appendChild(item);
  });
}

function renderLastUpdate(state) {
  document.getElementById("last-update").textContent = state.updated_at || "-";
}

async function refreshState() {
  const state = await fetchJson("/api/state");

  renderController(state);
  renderIsps(state);
  renderActivePool(state);
  renderFlows(state);
  renderEvents(state);
  renderLastUpdate(state);
}

function populateSelectors() {
```

```
  const sourceSelect = document.getElementById("ping-source");
  const destinationSelect = document.getElementById("ping-destination");
  const eventIspSelect = document.getElementById("event-isp");

  sourceSelect.innerHTML = "";
  destinationSelect.innerHTML = "";
  eventIspSelect.innerHTML = "";

  config.sources.forEach((source) => {
    const option = document.createElement("option");
    option.value = source.namespace;
    option.textContent = `${source.name} (${source.ip})`;
    sourceSelect.appendChild(option);
  });

  config.destinations.forEach((destination) => {
    const option = document.createElement("option");
    option.value = destination.ip;
    option.textContent = `${destination.name} · ${destination.ip}`;
    destinationSelect.appendChild(option);
  });

  config.isps.forEach((isp) => {
    const option = document.createElement("option");
    option.value = isp.name;
    option.textContent = `${isp.name} (${isp.role})`;
    eventIspSelect.appendChild(option);
  });
}

async function runPing() {
  const source = document.getElementById("ping-source").value;
  const destinationIp = document.getElementById("ping-destination").value;
  const output = document.getElementById("ping-output");

  output.textContent = "Running ping...";

  const result = await fetchJson("/api/ping", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      source: source,
      destination_ip: destinationIp,
      count: 4,
    }),
  });

  output.textContent =
    `$ ${result.cmd}\n\n` +
    (result.stdout || "") +
    (result.stderr ? `\nSTDERR:\n${result.stderr}` : "");

  await refreshState();
}

async function runEvent(action) {
  const isp = document.getElementById("event-isp").value;
  const output = document.getElementById("event-output");
```

```
  output.textContent = `Running ${isp} ${action}...`;

  const result = await fetchJson("/api/event", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      isp: isp,
      action: action,
    }),
  });

  output.textContent =
    `$ bash ${result.script}\n\n` +
    (result.stdout || "") +
    (result.stderr ? `\nSTDERR:\n${result.stderr}` : "");

  setTimeout(refreshState, 1500);
}

async function init() {
  config = await fetchJson("/api/config");
  populateSelectors();
  populateExtraSelectors();

  document.getElementById("ping-btn").addEventListener("click", runPing);
  document.getElementById("event-up").addEventListener("click", () =>
runEvent("up"));
  document.getElementById("event-down").addEventListener("click", () =>
runEvent("down"));

  document.getElementById("continuous-start").addEventListener("click",
startContinuousPing);
  document.getElementById("continuous-stop").addEventListener("click",
stopContinuousPing);
  document.getElementById("distribution-run").addEventListener("click",
runDistributionTest);

  await refreshState();
  setInterval(refreshState, 500);
  setInterval(refreshContinuousPing, 500);
}

init();
```