Eberhard Karls Universität Tübingen
Faculty of Mathematics and Natural Sciences
Department of Computer Science
Chair of Communication Networks

Master Thesis Computer Science

# Implementation and Evaluation of a Routing Architecture for LEO-Satellite Networks with Inter-Satellite Links

Oliver Richter

01.12.2025

**Reviewers**

Prof. Dr. Michael Menth
Department of Computer Science
Chair of Communication Networks
Eberhard Karls Universität Tübingen

Prof. Dr. Oliver Bringmann
Department of Computer Science
Chair of Embedded Systems
Eberhard Karls Universität Tübingen

**Supervisor**

Moritz Flüchter M.Sc.

**Oliver Richter:**
*Implementation and Evaluation of a Routing Architecture for LEO-Satellite Networks with Inter-Satellite Links*
Master Thesis Computer Science
Eberhard Karls Universität Tübingen
Time period: 01.07.2025 - 01.12.2025

## Erklärung

Laut Beschlüssen der Prüfungsausschüsse Bioinformatik, Informatik, Informatik Lehramt, Kognitionswissenschaft, Machine Learning, Medieninformatik und Medizininformatik der Universität Tübingen vom 05.02.2025. Gültig für Abschlussarbeiten (B.Sc./M.Sc./B.Ed./ M.Ed.) in den zugehörigen Fächern. Bei Studienarbeiten und Hausarbeiten bitte nach Maßgabe des/der jeweiligen Prüfers/Prüferin.

## 1. Allgemeine Erklärungen

Hiermit erkläre ich:

- Ich habe die vorgelegte Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.
- Ich habe alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet.
- Die Arbeit war weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens.
- Falls ich ein elektronisches Exemplar und eines oder mehrere gedruckte und gebundene Exemplare eingereicht habe (z.B., weil der/die Prüfer/in(nen) dies wünschen): Das elektronisch eingereichte Exemplar stimmt exakt mit dem bzw. den von mir eingereichten gedruckten und gebundenen Exemplar(en) überein.

## 2. Erklärung bezüglich Veröffentlichungen

Eine Veröffentlichung ist häufig ein Qualitätsmerkmal (z.B. bei Veröffentlichung in Fachzeitschrift, Konferenz, Preprint, etc.). Sie muss aber korrekt angegeben werden. Bitte kreuzen Sie die für Ihre Arbeit zutreffende Variante an:

☒ Die Arbeit wurde bisher weder vollständig noch in Teilen veröffentlich

☐ Die Arbeit wurde in Teilen oder vollständig schon veröffentlicht. Hierfür findet sich im Anhang eine vollständige Tabelle mit bibliographischen Angaben

## 3. Nutzung von Methoden der künstlichen Intelligenz (KI)

Die Nutzung von KI kann sinnvoll sein. Sie muss aber korrekt angegeben werden und kann die Schwerpunkte bei der Bewertung der Arbeit beeinflussen. Bitte kreuzen Sie alle für Ihre Arbeit zutreffenden Varianten an und beachten Sie, dass die Varianten 3.4 - 3.6 eine vorherige Absprache mit dem/der Betreuer/in voraussetzen:

☒ 3.1. Keine Nutzung: Ich habe zur Erstellung meiner Arbeit keine KI benutzt.

☐ 3.2. Korrektur Rechtschreibung & Grammatik: Ich habe KI für Korrekturen der Rechtschreibung und Grammatik genutzt, ohne dass es dabei zu inhaltlich relevanter Textgeneration oder Übersetzungen kam. Das heißt, ich habe von mir verfasste Texte in derselben Sprache korrigieren lassen. Es handelt sich um rein sprachliche Korrekturen, sodass die von mir ursprünglich intendierte Bedeutung nicht wesentlich verändert oder erweitert wurde. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

☐ 3.3. Unterstützung bei der Softwareentwicklung: Ich habe KI als Unterstützung beim Schreiben von Code in der Softwareentwicklung genutzt. Es handelt sich hierbei lediglich um Unterstützung und nicht um die automatische Generierung von größeren Programm-Teilen. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

☐ 3.4. Übersetzung: Ich habe nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in KI zur Übersetzung von mir in einer anderen Sprache geschriebenen Texte genutzt. Jede derartige Übersetzung ist im laufenden Text gekennzeichnet und der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller übersetzten Textstellen und der verwendeten Programme mit Versionsnummer.

☐ 3.5. Code-Generierung: Ich habe nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in KI zur Erzeugung von Code in der Softwareentwicklung genutzt. Der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller derartigen Nutzungen, der verwendeten Programme mit Versionsnummer und der verwendeten Prompts.

☐ 3.6. Text-Generierung: Ich habe nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in KI zur Erzeugung von Text in meiner Arbeit genutzt. Jede derartige Verwendung von KI ist im laufenden Text gekennzeichnet und der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller derartigen Nutzungen, der verwendeten Programme mit Versionsnummer und der verwendeten Prompts.

Falls ich in irgendeiner Form KI genutzt haben (siehe oben), dann erkläre ich:
Mir ist bewusst, dass ich die Verantwortung trage, falls es durch die Verwendung von KI zu fehlerhaften Inhalten, zu Verstößen gegen das Datenschutzrecht, Urheberrecht oder zu wissenschaftlichem Fehlverhalten (z.B. Plagiaten) kommt.

## 4. Abschluss und Unterschrift(en)

Mir ist bekannt, dass ein Verstoß gegen diese Erklärung prüfungsrechtliche Konsequenzen haben und insbesondere dazu führen kann, dass die Prüfungsleistung mit „nicht ausreichend“ bzw. die Studienleistung mit „nicht bestanden“ bewertet wird und bei mehrfachem oder schwerwiegendem Täuschungsversuch eine Exmatrikulation erfolgen bzw. ein Verfahren zur Entziehung eines eventuell verliehenen akademischen Titels eingeleitet werden kann.

| Oliver, Richter | Balingen, 30.11.2025 | O.Richter |
|---|---|---|
| Vorname, Nachname Student/in | Ort, Datum | Unterschrift |

Die Punkte 3.4 - 3.6 erfordern eine Zustimmung des/r Betreuer/in. Sollten Sie einen dieser Punkte angekreuzt haben, dann sollte der/die Betreuer/in bitte hier unterschreiben:
Ich habe der oben genannten Nutzung von KI zur Erstellung der Arbeit zugestimmt.

| | | |
|---|---|---|
| Vorname, Nachname Betreuer/in | Ort, Datum | Unterschrift |

# Contents

Low Earth orbit (LEO) satellite networks introduce rapidly changing topologies and strict resource constraints, making routing a central challenge. This thesis investigates Int2Sat, a routing architecture for the integration of LEO satellite networks with inter-satellite links into the Internet. This thesis focuses on the concrete realization and evaluation of Int2Sat in an emulated network environment.

The main contribution of this work is the complete implementation of Int2Sat's forwarding and control components. The data plane is realized through P4-based packet processors for satellites and gateways. These processors use an SR-MPLS encoding tailored to the constraints of LEO satellite payloads. The control plane is implemented as a modular Python framework that computes routing state, manages mobility events, and programs the data plane via the P4Runtime. Additionally, this thesis introduces the non-terrestrial network emulator (NTNE), a scalable and extensible platform capable of emulating dynamic LEO constellations, custom routing logic, and large-scale programmable networks.

Using realistic constellation scenarios, the thesis validates the functional correctness of the Int2Sat implementation and demonstrates stable end-to-end connectivity under fast topology changes, predictable routing-state updates, and efficient control plane behavior. The NTNE itself is shown to support multi-satellite scenarios of varying scales, confirming its applicability as a research tool for future work on routing, resilience, and traffic engineering in non-terrestrial networks.

# 1 Introduction

The deployment of broadband Internet access in remote or sparsely populated regions through terrestrial networks is often economically and technically impractical. In many areas, extending terrestrial infrastructure is infeasible due to challenging topographies such as mountain ranges, valleys, or deserts. To address these limitations, satellite networks have emerged as an essential complement to terrestrial infrastructures, bridging connectivity gaps in rural and hard-to-reach regions. Standardization efforts, such as those by the 3rd Generation Partnership Project (3GPP), highlight the potential of non-terrestrial networks (NTNs) to enhance 5G mobile broadband by extending coverage, improving resiliency, and offloading terrestrial traffic [1], [2]. The Broadband Commission for Sustainable Development also emphasizes the role of geostationary satellites in achieving global coverage and bridging the digital divide [3].

Recent large-scale satellite network deployments, including SpaceX's Starlink, Amazon's Kuiper, and Eutelsat OneWeb, demonstrate the feasibility of NTNs for delivering broadband Internet on a global scale. These systems leverage large-scale low Earth orbit (LEO) constellations equipped with inter-satellite links (ISLs), forming dynamic mesh networks in space. However, the rapid orbital movement of LEO satellites (with orbital periods of roughly 100 minutes at altitudes up to 1,000 km [4]) results in a highly dynamic network topology. These frequent but predictable link changes require a specialized routing architecture.

To address these challenges, the Int2Sat routing architecture introduces a novel approach for integrating the Internet and NTNs, which employ LEO satellite networks with ISLs. It applies the software-defined networking (SDN) principle to enable programmable control of the NTN data and control plane, thereby separating control and forwarding functionalities. The data plane employs a segment routing paradigm to steer packets through the NTN based on lists of forwarding instructions, also called segments. Satellites and gateways (GWs) are equipped with packet processors capable of encapsulating unlabeled packets with segment routes and forwarding them between user terminals (UTs), that represent end-user devices in the NTN, and the GWs. Additionally, GWs implement Int2Sat's control plane and are responsible for advertising IP prefixes associated with UT addresses to connected autonomous systems via BGP, thereby integrating UTs into the global Internet [5].

## 1.1 Thesis Goal

The objective of this thesis is to implement the Int2Sat routing architecture in a testbed, evaluate its feasibility and demonstrate its functional correctness. More specifically, this work focuses on implementing the satellite and gateway packet

processors and the control plane logic of the gateways. The packet processors are implemented using the programming protocol-independent packet processor (P4) language. For the control plane logic, the controller presented in [6] serves as foundation, providing the routing information which are translated by the GWs into configurations for the packet processors.

For the functional evaluation of the Int2Sat routing architecture an emulated testbed is required as illustrated in Figure 1.1. This non-terrestrial network emulator (NTNE) must be capable of representing the core NTN nodes required for the routing architecture evaluation. It must allow the integration of the developed P4 packet processors for the Int2Sat data plane and the developed Int2Sat control plane logic. Furthermore, the emulator should allow the assessment of the routing architecture's capability to handle NTN dynamics by emulating handovers for links between satellites and ground nodes.

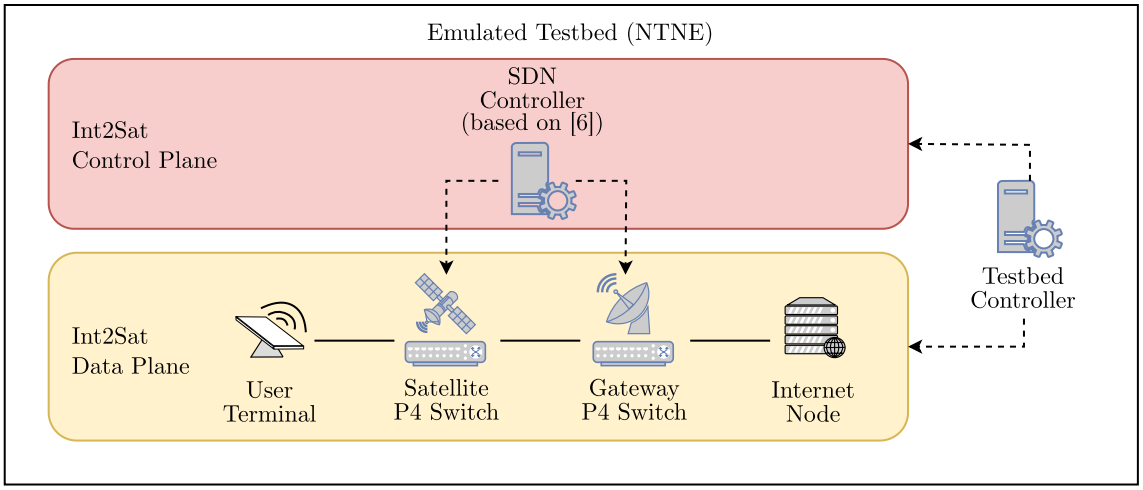



Figure 1.1: Concept for the emulated testbed (NTNE).

## 1.2 Thesis Structure

Chapter 2 provides the technical background necessary for understanding the subsequent chapters. Chapter 3 reviews related work, focusing on existing network and satellite network emulators, and analyzes their suitability for the NTNE. The results of this analysis form the basis for the design of the new, generic, and extensible NTNE described in Chapter 4. Chapter 5 presents a detailed overview of the Int2Sat routing architecture, followed by Chapter 6, which describes its implementation within the NTNE. The evaluation in Chapter 7 demonstrates the practical applicability of the implemented routing architecture using different NTN scenarios. Chapter 8 discusses potential future work, focusing on extensions of the non-terrestrial network emulator and open implementations for the Int2Sat architecture. Finally, Chapter 9 summarizes the findings of this thesis and provides an outlook on future research directions.

# 2 Technical Background

This chapter provides the foundational concepts and technologies required for understanding the design and implementation of the Int2Sat routing architecture and its realization within the NTNE. First, the chapter outlines the fundamental properties of non-terrestrial networks. It then introduces segment routing and software-defined networking, which form the conceptual basis for Int2Sat and are used throughout this thesis. Building on this, the chapter discusses the P4 language and relevant execution environments, that enable the implementation of programmable packet processors on satellites and gateways.

## 2.1 Non-Terrestrial Networks

Non-terrestrial networks (NTNs) extend communication infrastructures beyond the terrestrial domain by incorporating satellites in space at altitudes above 180 km and airborne platforms, such as drones or balloons, in the lower Earth atmosphere at an altitude between 8 and 50 km. Satellites are differentiated by three main orbital operating heights: geostationary orbit (GEO), medium Earth orbit (MEO), and low Earth orbit (LEO). GEO satellites operate at a specific altitude of 35 786 km above Earth's equator, that exactly matches Earth's rotation. This enables them to remain in a fixed position relative to a point on Earth's surface, rendering GEO satellites ideal for telecommunications applications, such as radio and television. However, the minimal propagation delay of approximately 120 ms at the speed of light renders GEO satellites unsuitable for broadband Internet support [7], [8]. Instead, projects like the 3GPP rely on MEO satellites, which are situated below GEO satellites, and LEO satellites, which typically operate at altitudes between 500 and 2000 km. LEO satellites are preferred over MEO satellites as they provide lower latency and a better link budget. However, LEO satellites provide less Earth surface coverage, because of their low altitude. This requires multiple interconnected LEO satellites to provide global broadband connectivity [1].

To better understand the routing architecture introduced later, the following sections introduce the core node and link types of a NTN.

### 2.1.1 Nodes

A non-terrestrial network consists of several types of nodes, each fulfilling a distinct role within the network [9], [10]:

- **User terminals (UTs)** act as the access endpoints of the network. They represent end-user devices, that are either deployed on fixed or mobile platforms, or relay devices that serve multiple end-user devices. A UT connects

to the satellite constellation via radio interfaces, leveraging special antennas to connect to and track orbiting satellites.

- **Gateways (GWs)** interface the satellite constellation with the terrestrial Internet. They serve as ingress and egress points for terrestrial traffic and host enough computational resources and antennas to connect to multiple satellites at once. Gateways are also referred to as ground stations in 6G NTNs.
- **Satellites (SATs)** form the space segment of the NTN. Multiple satellites are interconnected as a mesh of nodes, forming a satellite constellation. Each satellite acts as a forwarding node, relaying packets between UTs, GWs, and other satellites. Constellations, especially LEO constellations, create continuously changing network topologies as they orbit the Earth at high velocities.

### 2.1.2 Links

Communication in an NTN is realized through a set of heterogeneous link types that differ in physical properties and operational behavior [8]–[10]:

- **User-terminal-satellite links (USLs)** connect UTs to their serving satellites. In this context, they are also referred to as service links. Service links are subject to rapid changes due to UT mobility, satellite movement, or environmental conditions.
- **Gateway-satellite links (GSLs)** provide connectivity between satellites and gateways. These links form the attachment points between the space segment and the terrestrial Internet segments. They are also referred to as feeder links.
- **Inter-satellite links (ISLs)** are optical or traditional radio links between satellites. They can be established either between satellites in the same orbit or between satellites in different orbits. Multiple ISLs create the spaceborne mesh network that enables multi-hop forwarding through the constellation. Their availability is dictated by satellite geometry, pointing constraints, and orbital motion.
- **Terrestrial links (TLs)** interconnect gateways with terrestrial Internet infrastructure and ensure end-to-end connectivity to Internet nodes.

## 2.2 Segment Routing

This section summarizes the key concepts and mechanisms introduced in the main IETF specifications for segment routing (SR), focusing on the architectural foundations (RFC 8402 [11]), the MPLS data plane instantiation (RFC 8660 [12]), the IPv6 data plane instantiation (RFC 8754 [13]), and the more recent compressed SRv6 encoding (RFC 9800 [14]).

RFC 8402 specifies the general architecture of segment routing, which enables source routing by encoding a topologically or service-defined sequence of instructions, called segments, into packets. Segments are represented by segment identifiers (SIDs) and may be local or global within an SR domain. SR supports multiple control plane models (distributed, centralized, hybrid) and allows per-flow state to be maintained only at the ingress of the SR domain. It defines two data plane instantiations: SR-MPLS and SRv6 [11].

RFC 8660 describes the instantiation of SR on the MPLS dataplane by mapping each SID to a single MPLS label. An ordered list of segments is encoded as a label stack, where the active segment is always the top label. Completion of a segment is signaled by popping the corresponding label. SR-MPLS relies on existing MPLS forwarding behavior without requiring changes to the MPLS dataplane. The RFC defines how Prefix-SIDs, Node-SIDs, and Adjacency-SIDs are signaled and used within MPLS networks [12].

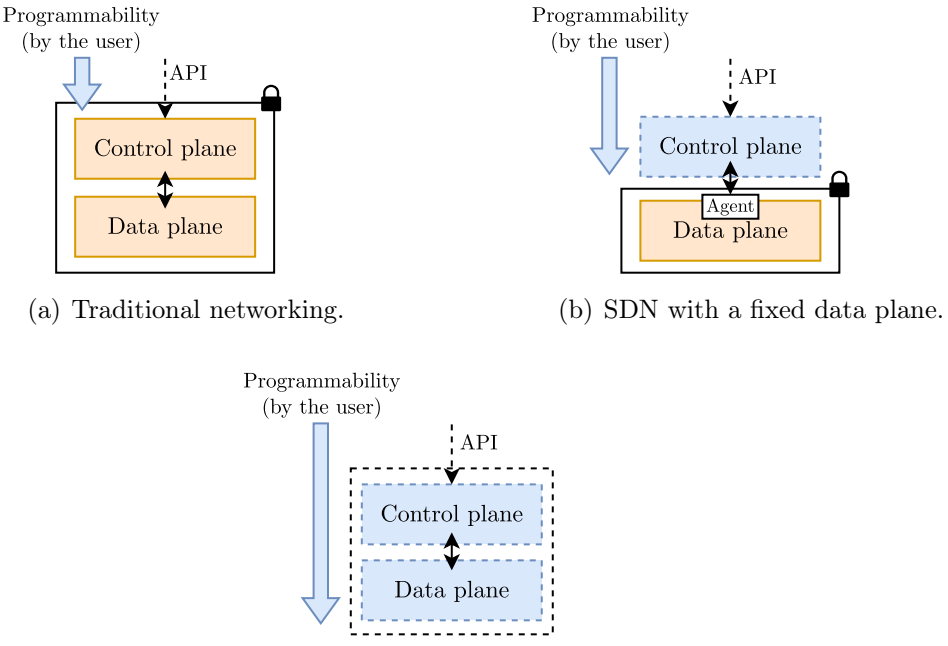
RFC 8754 introduces the segment routing header (SRH), a new IPv6 routing header used to encode an ordered list of IPv6 SIDs. The active SID is carried in the packet's IPv6 destination address, while the remaining SIDs are stored in the SRH. Processing of an SRv6 SID involves updating the destination address with the next SID and decrementing the segments left pointer. The RFC specifies SRH structure, processing rules, TLV options, and interactions with IPv6 extension headers [13].

RFC 9800 extends RFC 8754 by defining mechanisms for compressing SRv6 SID lists using compressed segment identifiers (CSIDs). The document introduces two new SRv6 endpoint flavors: NEXT-CSID and REPLACE-CSID. These allow multiple SIDs to be packed into a single 128-bit IPv6 address container. This reduces the size of SRv6 encapsulations, particularly beneficial for long SID lists used in traffic engineering. The RFC updates SRH processing rules to accept either standard 128-bit IPv6 SIDs or CSID containers and defines rules for SID allocation and multi-domain compression [14].

## 2.3 Software-Defined Networking

Traditional networks tightly couple the control plane and the data plane: network devices make forwarding decisions locally, based on distributed routing protocols, and expose little to no programmability beyond fixed-function forwarding. This classical model is depicted in Figure 2.1(a), where each device independently embeds routing logic, forwarding state, and vendor-specific configuration interfaces. Such vertically integrated designs hinder innovation, complicate network-wide coordination, and limit the ability to introduce new forwarding behaviors [15].

Software-defined networking (SDN) breaks with this paradigm by decoupling the control and data planes through a standardized, programmable interface as illustrated in Figure 2.1(b). The SDN model introduces an external controller that maintains a logically centralized view of the network, computes forwarding decisions, and installs these decisions on the data plane devices through a southbound API. This architectural separation provides three key advantages: (i) simplified device functionality, (ii) controller-driven adaptation to network dynamics, and (iii)




(a) Traditional networking.

(b) SDN with a fixed data plane.

(c) Software-defined networking with full network programmability

Figure 2.1: Networking concepts according to Hauser et. al. [15]

a unified programmatic interface for configuring heterogeneous forwarding elements [16].

A full network programmability, as shown in Figure 2.1(c), extends SDN by not only enabling the control plane but also the data plane to be redefined through software. This programmability spans multiple abstraction layers: from high-level intent-based configuration down to protocol-independent packet processing within forwarding hardware [15].

This development is exemplified by the P4 language, which allows operators to implement custom packet-processing pipelines that are independent of specific protocols or hardware vendors. Instead of being bound to predefined match-action tables, developers can describe how packets are parsed, processed, and forwarded, enabling full flexibility in implementing routing architectures, new protocols, or application-specific processing [15].

## 2.4 Programming Protocol-Independant Packet Processors (P4)

Protocol-independent data plane programmability enables network devices to support customized packet-processing behavior. Programming protocol-independent packet processor (P4) is the predominant language in this domain, allowing developers to specify how packets are parsed, matched, and acted upon without relying

on predefined protocol semantics. P4 programs define the full processing logic of a packet processor: its parser, match-action pipeline, tables, and control flow, while remaining independent of the underlying hardware platform.

A P4-based system is built around three key abstractions: the P4 program, the P4 architecture, and the P4 target. The P4 program describes protocol-independent packet-processing behavior. The P4 architecture specifies the abstract switch model (such as the widely used v1model), including available parser states, table structures, and pipeline blocks. The P4 target is the concrete hardware or software system, such as programmable ASICs or the behavioral model version 2 (BMv2) software switch, that executes compiled P4 code. The relationship between these components is illustrated in Figure 2.2, which shows how a P4 program is compiled against a specific architecture and deployed on a target [15], [17].

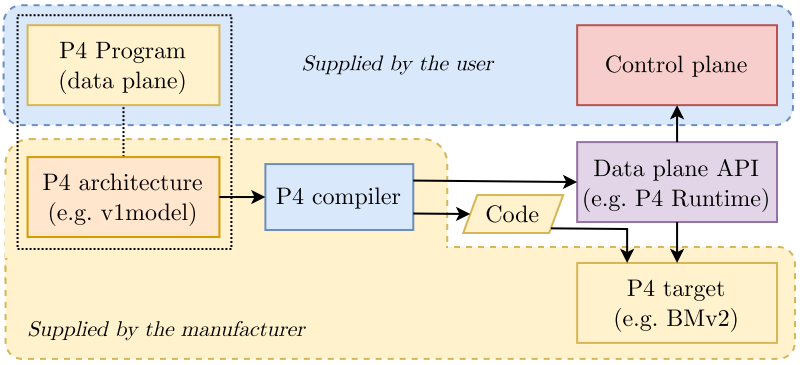



Figure 2.2: P4 development and deployment process according to [15], [17].

Compilation produces both device-specific code and a data plane API[1], such as P4Runtime. The control plane installs table entries, configures actions, and manages runtime behavior, while the data plane strictly executes programmer-defined logic at line rate. This separation allows flexible, fine-grained control plane programmability without modifying the forwarding pipeline itself. In contrast to traditional network devices, P4 targets do not embed protocol-specific assumptions; instead, packet formats, parsing logic, and processing behavior are fully defined by the P4 program. This enables operators to introduce new protocols, enforce custom routing architectures, or implement telemetry and security mechanisms without hardware redesign [15].

P4 has evolved through two major language specifications: $P4_{14}$ and $P4_{16}$[2]. The original $P4_{14}$ specification introduced the core concepts of protocol-independent parsing and match-action processing, but it was tightly bound to a fixed switch model and lacked strong type safety. The newer $P4_{16}$ specification significantly restructures the language by providing a more modular and expressive design, improved type checking, and clearer architectural abstraction. Whereas $P4_{14}$ implicitly encoded architectural constraints into the language itself, $P4_{16}$ decouples the language

[1] In this thesis, the term data plane API is used interchangeably with the term control plane API.
[2] At the time of this thesis $P4_{16}$ v1.2.5 is the most recent version of the P4 language specification [17].

from the architecture, allowing architectures, such as v1model, to be defined independently. This shift enables portability across targets and cleaner integration with control plane APIs such as P4Runtime. As a result, $P4_{16}$ has become the dominant specification for modern P4 development, while $P4_{14}$ is maintained primarily for legacy support [15], [17].

### 2.4.1 P4 Architecture: v1model

The v1model architecture is the reference $P4_{16}$ device model designed to preserve compatibility with the original $P4_{14}$ programming model while enabling the structured and strongly typed abstractions of $P4_{16}$. The v1model architecture provides a logical pipeline abstraction that closely mirrors the behavior of the BMv2 software switch. It exposes a fixed set of programmable blocks i.e., parser, ingress, egress, and deparser, together with intrinsic metadata and built-in externs, allowing P4 programs to be executed on any target implementing the same model [15], [18], [19].

Figure 2.3 shows the block-level structure of v1model, highlighting its packet-processing pipeline.

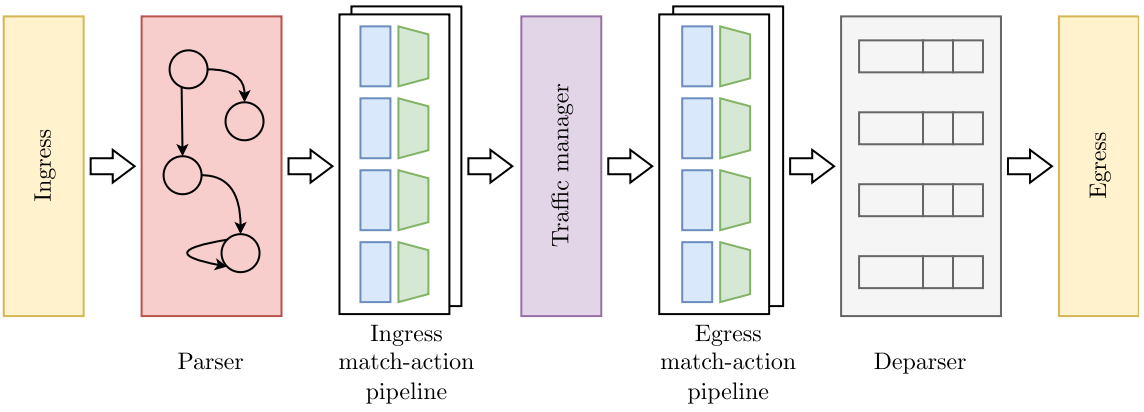



Figure 2.3: P4 v1model architecture according to Hauser et. al. [15].

The v1model architecture consists of the following elements [15], [18], [19]:

- **Parser:** Extracts packet header fields and populates metadata. Its behavior is defined entirely by the P4 program as a state machine that transitions between parsing states based on header values.
- **Ingress Match-Action Pipeline:** Implements the primary packet-processing logic. P4 tables in this stage can perform header modifications, encapsulation or decapsulation, selection of outgoing ports, or preparation for further processing. The ingress pipeline has access to intrinsic metadata such as the packet's ingress port, length, and parser outputs.
- **Traffic Manager:** A fixed-function component responsible for queueing, scheduling, and buffer management. While not programmable in P4, it interacts with the programmable stages through intrinsic metadata fields such as the egress port.

- **Egress Match-Action Pipeline:** Provides additional programmable processing after queueing, enabling actions such as header updates, mirroring, cloning, or specialized egress-side transformations.
- **Deparser:** Serializes the modified packet headers and payload back into a byte stream before transmission.

Externs in v1model provide additional functionality that cannot be expressed directly within P4 language constructs. These include counters, meters, registers, checksum units, hash functions, and clone/mirror engines. Externs operate through well-defined APIs and may maintain persistent state across packets, making them central for implementing measurement, stateful processing, or auxiliary protocol logic [15], [18].

Within the landscape of P4 architectures, v1model occupies a transitional role between $P4_{14}$ and more modern architectures such as the portable switch architecture (PSA). While PSA generalizes programmability for a broader range of hardware targets, v1model remains the dominant model for research, prototyping, and teaching because of its simplicity, compatibility with BMv2, and widespread tooling support. Its rigid pipeline structure and intrinsic metadata conventions make it a natural architecture for software-based experimentation and controlled evaluations of novel routing or forwarding concepts.

### 2.4.2 P4 Target: Behavioral Model Version 2

The platforms capable of executing P4 programs are the so called P4 targets. These targets could either be software-based switches, FPGA back ends, ASICs, and NPUs. Among these, software targets play a central role for prototyping and experimentation due to their accessibility, flexibility, and ease of integration into research environments. The most prominent and widely used software target is the behavioral model version 2 (BMv2), which constitutes the reference implementation for P4 [15].

BMv2 is a modular software switch designed to execute P4 programs independently of hardware constraints. In contrast to its predecessor p4c-behavioral, BMv2 uses a static, program-independent C++ code base. P4 programs written in $P4_{16}$ or $P4_{14}$ are compiled into a JSON representation that BMv2 loads at runtime, enabling rapid deployment and iterative development. This architecture separates compiler output from switch implementation, simplifying extensibility: new externs, custom packet processing logic, or debugging features may be added by modifying BMv2's C++ source code [15], [19].

BMv2 consists of multiple backend variants. The most feature-rich backend is `simple_switch`, which supports the $P4_{16}$ v1model architecture and provides a program-independent thrift API for control plane interaction. The extended variant `simple_switch_grpc`, additionally implements the P4Runtime API via gRPC, enabling interoperable, controller-driven configuration. Another backend, `psa_switch`, targets the PSA architecture, offering a higher-fidelity representation of modern hardware switches. For didactic purposes, reduced versions such as `simple_router`

and `l2_switch` are provided, though they do not support full $P4_{16}$ functionality [15], [19].

Performance-wise, while BMv2 is not intended for production deployment, measurements show throughput up to approximately 1 Gbit/s for standard IPv4 forwarding pipelines. This makes BMv2 suitable for functional validation, debugging of complex packet-processing pipelines, and educational use. Its active community-driven development ensures evolving support for new P4 features and architectures [19].

### 2.4.3 P4 Data Plane API: P4Runtime

P4 data plane APIs provide the interface through which an external control plane configures, monitors, and coordinates the behavior of P4-programmable forwarding devices. These APIs expose essential runtime mechanisms such as table entry management, counter and meter access, packet cloning or mirroring control, and event reporting. They decouple the control plane from target-specific internals, thereby enabling external controllers to operate on heterogeneous P4 targets in a uniform manner. Modern SDN controllers rely on these APIs to enforce network-wide policies, update forwarding state, and integrate P4 targets into larger control architectures [15].

A central role among these APIs is played by P4Runtime[3]. It is a platform-agnostic, gRPC-based interface defined jointly by the P4 language consortium and the open networking foundation. P4Runtime operates independently of the specific switch architecture or target type, allowing the same controller logic to manage software switches, FPGA-based systems, and ASIC-based devices. Its API is generated automatically from the P4 program during compilation, ensuring that the control plane and data plane remain tightly aligned. The interface supports all essential runtime operations, including the insertion, modification, and deletion of match-action table entries, reading and writing counters, meters, and registers, and managing packet replication, cloning, and digest messages. The message and object model of P4Runtime is defined in a set of protobuf specifications that precisely encode table schemas, action formats, and metadata descriptions [15], [20].

Through its use of gRPC bidirectional streams, P4Runtime enables asynchronous event delivery, allowing the data plane to notify the controller about events such as digests, errors, or exceptional packet behaviors. This facilitates responsive controller logic and supports advanced SDN applications including telemetry, dynamic routing, and runtime debugging [15].

[3] At the time of this thesis v1.4.1 is the most recent version of the P4Runtime specification [20].

# 3 Related Work

This chapter provides a structured analysis of existing network emulators from related work and assesses their suitability for realizing the NTNE. First, the approach for the network emulator analysis is described. Second, existing network emulators from related work are analyzed based on the approach. Last, the analysis results are summarized and the motivation for the development of the NTNE is given.

## 3.1 Network Emulator Analysis Approach

The network emulator analysis is split into two parts. The first part covers classical network emulators, while the second part explores specialized satellite network emulators. The analysis evaluates the emulators based on their scalability, extensibility and integration of custom routing architectures. In order to assess the scalability and extensibility of each emulator, a common model for all network emulators is defined, as shown in Figure 3.1. From that model different, comparable categories for the network emulator analysis can be inferred: (1) hosting (2) network node emulation (3) network link emulation (4) network orchestration (5) source code availability.

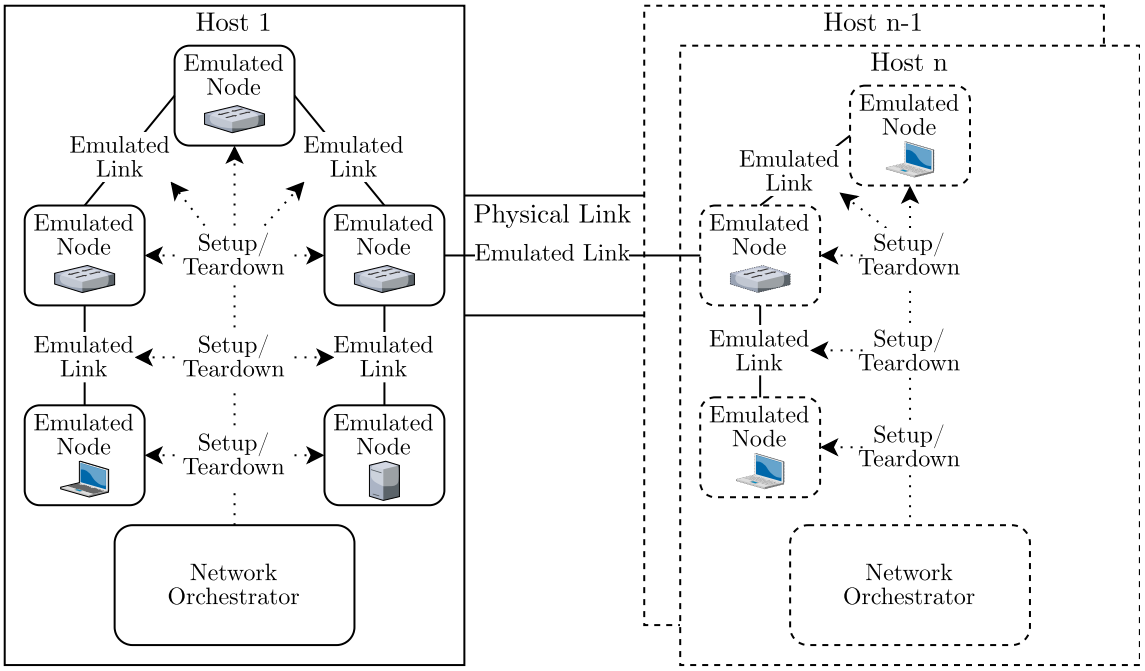



Figure 3.1: Common model for the analyzed network emulators.

The hosting category considers the possibility to deploy the emulator on one or more host machines. With the network node and link emulation categories, the

technologies used for emulating network nodes and network links are examined. Emulated network nodes in a classical network emulator are switches, clients and servers. These can also be mapped to the nodes of a NTN, such that satellites and gateways are represented as switches, user terminals as clients and internet nodes as servers. The setup and teardown of the emulated network nodes and links is performed by a network orchestrator. The capabilities of this orchestrator are analysed in the network orchestration category. The last category considers the source code availability, i.e. if the source code is open-sourced, proprietary or unpublished, as this a major factor for the extensibility of the network emulators.

## 3.2 Classical Network Emulators

This section analyzes different classical network emulators based on the defined approach. The selected list of network emulators focuses on prominent and recent network emulators that represent different emulation approaches. A complete survey of existing emulator solutions is out of scope for this work. Included in this list are Mininet [21], Distrinet [22], CORE [23], VITO [24] and the hybrid testbed proposed by Windisch et. al. [25].

### 3.2.1 Mininet

The Mininet emulator introduced in [21], is a network emulator developed as a testbed for the evaluation of software-defined networks. Mininet is designed to efficently use the resources of a single host machine. To that end, Mininet leverages the control group and network namespace features of the Linux kernel, as well as standard shell processes to emulate network end nodes, like clients and servers. These features allow sharing the same kernel among all end nodes, while also isolating the mandatory network resources [26], [27]. Network switches are emulated using software switches, that are running as processes in the root namespace. Hereby, Mininet natively supports OpenFlow switches and Open vSwitches. For the emulation of network links, Mininet leverages Linux virtual Ethernet devices, which can tunnel traffic between the different network namespaces at the data link layer. The transmission of packets over this link is seamless, meaning packets transmitted from one network namespace are immediately received in the other network namespace [28]. For the network orchestration a command-line interface and a Python-based API are provided. The Mininet source code is published as open source on GitHub under the BSD license[1].

Mininet offers scalability on a single host machine, through its lightweight virtualization approach, essentially allowing it to be scalable with the resources available on the host machine. However it cannot be natively scaled onto multiple hosts. The open-sourced Python API for Mininet's network orchestrator allows for extensibility and already incorporates functions to setup and teardown network nodes and links. These functions can be used to emulate the required NTN dynamics. The implementation of custom routing protocols is limited to the protocols supported

[1] https://github.com/mininet/mininet Accessed: 06.11.2025

by the used software switches. However, Mininet can be extended to support other availalbe software switches through its flexible architecture.

### 3.2.2 Distrinet

The Distrinet emulator [22] is developed as a distributed alternative for Mininet, allowing hosting on multiple machines. To that end, Linux containers are leveraged for the emulation of network nodes, which in turn are based on Linux namespaces and Linux control groups [29]. For the emulation of network links between the distributed hosts, Distrinet builds VXLAN tunnels over the physical links between the hosts. These tunnels encapsulate the original layer 2 packets in an UDP datagram [30]. The Distrinet network orchestrator API is compatible with the Mininet API. It leverages Ansible to maintain the Linux containers on each machine from a centralized master machine. Furthermore, the network orchestrator is responsible to setup SSH connections from the master machine to all other hosts, which are used for accessing containers on remote hosts. The source code for the network emulator is provided as open source on GitHub under the MIT license[2].

The VXLAN tunnels allow enough scaling of emulated point-to-point links between physically-separated network nodes. However, using UDP datagrams for the encapsulation introduces a certain protocol overhead. Furthermore, the traffic on these physical links is prone to additional delay and jitter, as the links are shared by multiple flows. The extensibility of Distrinet is equal to that of Mininet, as Distrinet is API-compatible and also open source.

### 3.2.3 Common Open Research Emulator

The common open research emulator (CORE) by Ahrenholz et. al [23], is another mutiple-host network emulator like Distrinet. It is written for the FreeBSD operating system. The network node emulation is based on FreeBSD's network stack virtualization together with the FreeBSD jail mechanism, forming so called virtual images. These virtual images are managed by the network orchestrator using a FreeBSD netgraph. Such a netgraph contains kernel objects as nodes and hooks into the nodes for interconnecting them. A pair of hooks creates an edge and allows building any type of network [31]. Links between virtual images are emulated using special netgraph pipe nodes, that can impose certain bandwidth, delay and packet loss requirements on the transmitted packets [32]. In order to build tunnels between network nodes on separated hosts, CORE introduces a C daemon called Span, which is running as a user space program on FreeBSD, Linux and Windows. The network orchestrator can be accessed via the CORE API and a graphical user interface (GUI). The CORE source code is published on GitHub under the BSD license[3].

Considering the lightweight kernel virtualization technologies used by CORE, the scalability can be compared to that of Distrinet. The portability of Span onto machines with UNIX and Windows operating systems enhances its scalability. However,

[2]https://github.com/Giuseppe1992/Distrinet Accessed: 06.11.2025
[3]https://github.com/coreemu/core Accessed: 06.11.2025

by running Span in the user space, additional processing overhead is introduced.

### 3.2.4 Hybrid Testbed

The hybrid testbed by Windisch et. al. [25] relies on the same virtualization technologies as Distrinet i.e., Linux containers and VXLANs, to create a multi-host network emulator. The main difference to Distrinet is the proposed orchestration approach, which splits the network definition from the network creation. This is done using an intermediate representation (IR) format, that contains the topology and networking information of the whole network and is computed as part of the network definition. The IR format is distributed to the hosts as a file, enabling offline network creations, or directly via SSH. The source code for the network emulator is published on GitHub under the GPL-3 license[4].

Looking at the virtualization technologies, the hybrid testbed provides equal scalability and extensibility as Distrinet and CORE. However, the IR format introduces an abstraction layer, that allows incorporating hosts with any operating system. This is currently not considered in the implementation, but could be an extension on the open sourced network emulator.

### 3.2.5 Virtual Testbed Orchestration

In contrast to Mininet, Distrinet and CORE, the virtual testbed orchestration (VITO) emulator proposed in [24], is a single-host emulator using virtual machines (VMs) for the emulation of network nodes. It leverages QEMU for the processor hardware emulation and the KVM hypervisor for privileged accesses by the VMs. Network links are emulated using the Linux 802.1d Ethernet bridging module. For the network orchestration libvirt is leveraged. The source code for VITO is unpublished.

The QEMU hardware emulation and the KVM hypervisor introduce a certain amount of overhead, especially considering the deployment of large-scale satellite networks, where each satellite requires its own emulated hardware. Another deficit of VITO, in terms of its extensibility, is the unpublished source code, which requires the NTNE to be implemented from scratch. Still, VITO's approach of using virtual machines, allows running any program that cannot be run on the host's kernel, contributing to the approach's extensibility.

## 3.3 Satellite Network Emulators

While classical emulators such as Mininet and Distrinet provide a solid foundation for network experimentations, they lack built-in mechanisms to model orbital dynamics, link variability, and handover behavior inherent to non-terrestrial networks. Therefore, additional satellite-specific emulators are considered.

[4]https://github.com/FriwiDev/new_testbed Accessed: 06.11.2025

### 3.3.1 LeoEM

The LeoEM satellite network emulator described in [33] is used for evaluating TCP implementations on LEO satellite constellations. It is built upon Mininet, using Mininet end nodes to emulate user terminals, Open vSwitches for satellites and gateways, and virtual Ethernet devices for the NTN links. Mininet leverages a data pipeline approach for the network orchestration and especially for the emulation of NTN dynamics. The pipeline starts by sampling the locations of each satellite in a shell. Then these locations are used, first to compute handover events between UTs, GWs and their respective access satellites (AcSs) and second to compute the shortest routing path between two UTs. These steps are repeated for each sample of the emulation time. Afterwards the precomputed data is replayed using the Mininet emulator. The source code of LeoEM is available on GitHub under the MIT license[5].

The data pipeline approach allows precomputing all the necessary information of the emulated scenario, offloading the emulation from large computations. This is sufficient for small scenarios with only two UTs, but does not scale with larger scenarios. Furthermore LeoEM is specialized for the evaluation of TCP, making Open vSwitches perfectly suitable. However they only support data plane programming via OpenFlow, but cannot be used to implement packet processors for a custom routing architecture like Int2Sat [34].

### 3.3.2 StarryNet

The StarryNet satellite network emulator proposed in [35], is a multi-host emulator, leveraging a container-based emulation approach, comparable to Distrinet. In contrast to the Linux containers used in Distrinet, StarryNet relies on Docker containers. Satellites and gateways are emulated as Open vSwitches inside the Docker containers. The emulation of network links is accomplished using Docker virtual bridges and Docker VLANs. The centralized network orchestrator uses the Docker engine command line interface (CLI) together with the Docker Swarm mode to manage and access containers on the distributed hosts. For the emulation of NTN dynamics, the orchestrator uses a pregenerated file similarly to the approach of LeoEM. This file contains events for link handovers, link failures, and routing updates, which are sequentially processed and distributed to each node or link by the centralized orchestrator. The new routing information for a routing update event are distributed by the network orchestrator using BIRD configuration files [36]. The source code for StarryNet is published on GitHub under the MIT license[6].

The scalability of StarryNet is comparable to that of Distrinet, as both use a container-based emulation approach with multi-host deployment capabilities. However, StarryNet does not scale with larger scenarios, as it uses a pregenerated event file, similarly to LeoEM. Furthermore, the sequential processing of the event file by the centralized network orchestrator introduces a synchronization problem in the network, as routing updates and changes in the network topology are not applicable in parallel. Also similar to LeoEM, StarryNet relies on Open vSwitches and

[5]https://github.com/XuyangCaoUCSD/LeoEM/tree/main Accessed: 06.11.2025
[6]https://github.com/SpaceNetLab/StarryNet Accessed: 06.11.2025

the BIRD routing daemon. These technologies support IP routing architectures, but must be replaced with other technologies to allow implementing custom routing architectures [34], [36].

### 3.3.3 OpenSN

OpenSN builds upon the network emulation approach of StarryNet, but introduces some extensions to the network orchestration. As described in [37] OpenSN relies on the Docker engine software development kit (SDK), instead of the Docker engine CLI. Second, a distributed container runtime manager on each host is introduced, responsible for creating containers and executing commands inside the containers. In addition, OpenSN also adds a virtual link manager on each host, that manages the Linux virtual links and Linux network namespaces, replacing the Docker network manager used by StarryNet. At last OpenSN, also introduces a replacement for StarryNet's event file, by setting up a distributed database for the storage of key-value pairs on all hosts using etcd. These extensions and the remaining source code are published on GitHub without any license[7].

Switching from the Docker engine CLI to the Docker engine SDK improves the performance of all Docker engine API accesses as no shell processes must be spawned anymore. The deployment of the container runtime manager, virtual link manager and key-value database on each host, allow a distributed network orchestration, that tackles the synchronization problem in LeoEM and StarryNet. However, the synchronization overhead of the key-value database via the etcd backend must be taken into account [38].

### 3.3.4 Proto2Testbed

The Proto2Testbed is an emulator developed for the evaluation of security protocols over satellite networks. It uses virtual machines for the emulation of network nodes. Similar to VITO, the virtual machines are running on a single host using QEMU and KVM. The network link emulation leverages virtio to create paravirtualized network interfaces and Linux bridges to interconnect these interfaces. The Proto2Testbed network orchestrator is running on the host and accesses the virtual machines via a dedicated management network and custom protocol [39]. The Proto2Testbed source code is published on GitHub under the GNU GPL-3.0 license[8].

The VM-based emulation approach of the Proto2Testbed is suitable for the evaluation of security protocols, as some programs e.g., WireGuard, must be installed as kernel modules on a dedicated kernel. However, the virtualization approach introduces a significant overhead through the processor hardware emulation, which is not negligible for the emulation of large-scale satellite networks. Instead Ottens et. al. propose to use a link emulation approach that pregenerates satellite link traces using a simulator and then replaying these traces in the emulator using netem.

[7] https://github.com/OpenSN-Library/OpenSN-Library Accessed: 06.11.2025
[8] https://github.com/martin-ottens/proto2testbed/tree/main Accessed: 07.11.2025

## 3.4 Summary

This chapter reviewed a range of existing emulation platforms relevant for the development of the NTNE. Table 3.1 summarizes the evaluated emulators and highlights their used emulation technologies, which contribute to their scalability and extensibility. Classical network emulators such as Mininet and Distrinet provide efficient and lightweight virtualization environments for networking research. Their use of container- kernel-based virtualization enables rapid topology deployment and reproducibility, but their static architecture requires additional extensions to represent the temporal and spatial variations of NTNs.

| Emulator | Virtualization | Multi-Host | Orchestration | License | NTN Dynamics | Custom Routing |
|---|---|---|---|---|---|---|
| **Classical Network Emulators** | | | | | | |
| Mininet [21] | Kernel (Linux) | ✗ | Python API / CLI | BSD | ✗ | Limited (OpenFlow) |
| Distrinet [22] | Containers (LXC) | ✓ | Python API / CLI / Ansible | MIT | ✗ | Limited (OpenFlow) |
| CORE [23] | Containers (FreeBSD) | ✓ | C API / GUI | BSD | ✗ | Limited |
| Hybrid Testbed [25] | Containers (LXC) | ✓ | Python API / Pipeline | GPLv3 | ✗ | Limited |
| VITO [24] | VMs (QEMU/KVM) | ✗ | Unknown API / libvirt | Unpublished | ✗ | Limited |
| **Satellite Network Emulators** | | | | | | |
| LeoEM [33] | Kernel (Mininet) | ✗ | Python API / Pipeline | MIT | Limited | Limited |
| StarryNet [35] | Containers (Docker) | ✓ | Python API / Docker CLI (Swarm) | MIT | Limited | Limited |
| OpenSN [37] | Containers (Docker) | ✓ | GO API / Docker SDK / etcd | None | Limited | Limited |
| Proto2Testbed [39] | VMs (QEMU/KVM) | ✗ | Python API / Custom protocol | GPLv3 | Limited | Limited |

Table 3.1: Comparison of classical and satellite network emulators from related work.

Satellite-oriented emulators, including LeoEM, StarryNet, and OpenSN, have extended the classical emulation paradigm to address orbital motion and inter-satellite communication. However, these systems typically rely on pregenerated mobility traces or static event files rather than real-time orbital propagation. While they offer valuable frameworks for simulating specific scenarios, they remain limited in flexibility and runtime adaptability. Moreover, few of these platforms natively support integration with programmable packet processors or the fine-grained control necessary for custom routing architectures like Int2Sat.

The comparative analysis reveals a clear research gap: no existing emulator provides an extensible framework that combines real-time topology dynamics, scalability, and programmable data plane integration. The new NTNE introduced in this

thesis is designed to fill this gap by combining the scalability of kernel-based virtualization with a hybrid orchestration and seamless integration of custom packet processors. This approach enables functional evaluations of advanced satellite routing architectures under time-varying network conditions.

# 4 Non-Terrestrial Network Emulator (NTNE)

The evaluation of custom routing architectures, such as Int2Sat, requires a scalable and extensible emulator that allows for full network programmability. Chapter 3 showed that the analyzed network emulators from related work do not fulfill these requirements out of the box. This chapter describes the non-terrestrial network emulator, which is desgined to fill this gap among the available network emulators. The following sections first cover the requirements for the NTNE and then show its architecture and implementation.

## 4.1 Requirements

This section defines the requirements for the NTNE:

1. **Programmability**

   The NTNE must allow for full network programmability. Therefore, it must support the programmability of both the control and data plane planes of the emulated network. Control plane programmability is achieved by following the SDN paradigm. Software-defined networking splits the networking functionality of a switch into a fixed-function data plane and a programmable control plane. Adhering to the SDN paradigm enables users of the NTNE to implement any custom control plane algorithm.

   While following the SDN paradigm allows users to program a switch's control plane, it traditionally implies a fixed-function data plane. To provide full network programmability, the data planes of switches in the NTNE must also be programmable.

2. **Scalability**

   Currently active or planned LEO satellite constellations are designed to consist of numerous satellites. For example, as of February 2025, SpaceX's Starlink already operates a satellite constellation of over 6750 satellites [40], and Amazon's Kuiper is planned to initially include 3232 satellites [41]. These large-scale satellite networks require the NTNE to provide means for scaling to different scenarios and also adapting to future deployments.

3. **Extensibility**

   The NTNE must be designed as a modular and extensible framework so that new functionality can be added with minimal changes to the core. This includes well-defined interfaces and stable public APIs.

These requirements build the foundation for the following proposed architecture of the NTNE.

## 4.2 Architecture

The high-level architecture of the NTNE is shown in Figure 4.1. The NTNE is built upon the Mininet framework. This section provides the reasoning behind the selection of Mininet for the network emulation and its applicability in realising the NTNE requirements.

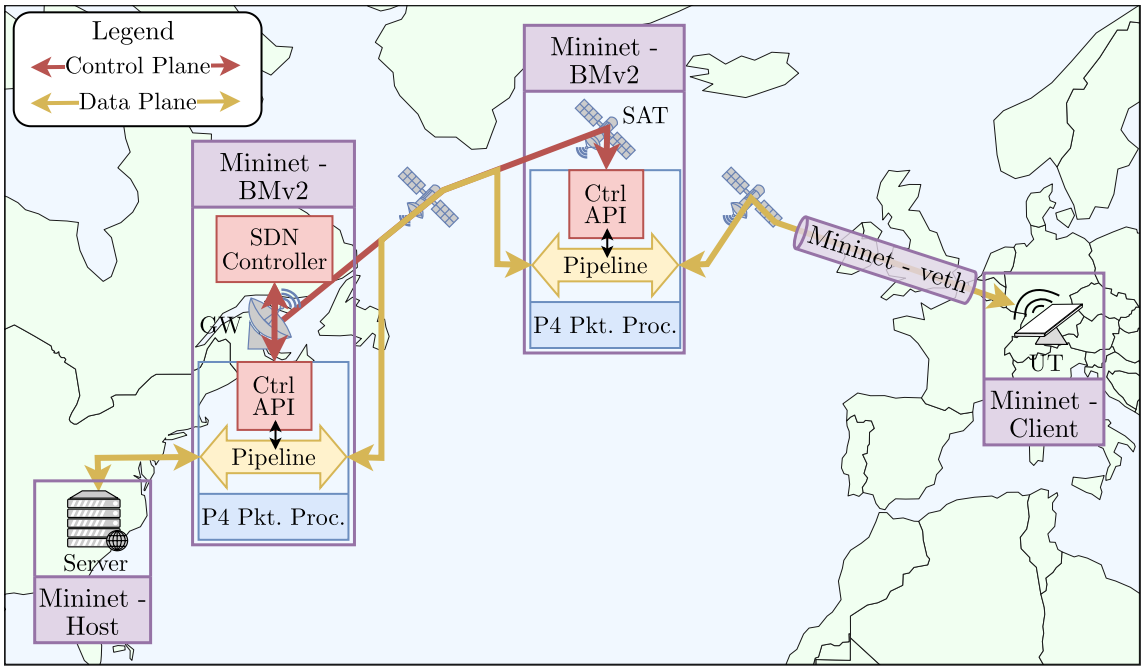



Figure 4.1: High-level architecture of the NTNE.

Mininet uses Linux network namespaces and Linux control groups to virtualise network end nodes, i.e. clients and hosts. These Linux kernel features isolate the networking system resources for each node and limit the resource usage of the processes running in each node [26], [27]. The NTNE leverages Mininet clients to emulate user terminals and Mininet hosts to emulate internet nodes, i.e. servers.

Mininet can also emulate switches. To that end, it supports software switches such as the Open vSwitch, as well as all OpenFlow-capable switches. These switches run as processes in the Linux root namespace and can be used to implement the software-defined networking paradigm. However, they do not fulfil the requirement for programmable data planes, as they only support a limited variety of fixed data plane functionalities [15], [34]. This issue is resolved by extending the Mininet framework to support the BMv2 software switch, that can run P4-based packet processors and thus provides full data plane programmability. The Mininet BMv2 software switches are used by the NTNE to emulate the switches of SATs and GWs.

Each P4 packet processor provides a control plane API. This allows an SDN controller to program the forwarding pipeline of the packet processor. The NTNE assigns the role of the SDN controller to the GWs in the NTN. Control plane algorithms for the NTN can be implemented on one or more GWs, allowing the evaluation of centralised or distributed algorithms.

The network links in the NTN are emulated by a Mininet link using Linux virtual Ethernet (veth) devices. These devices can be assigned to any network namespace and are always created as interconnected pairs, forming a bidirectional link between the namespaces [28].

As described above, Mininet relies on Linux kernel modules for its network emulation. This makes it a lightweight emulation framework with vertical scalability on a single host. Furthermore, Mininet is API-compatible with Distrinet, an alternative network emulator that supports multiple-host deployment [22]. Therefore, the NTNE could also be adapted to provide horizontal scalability.

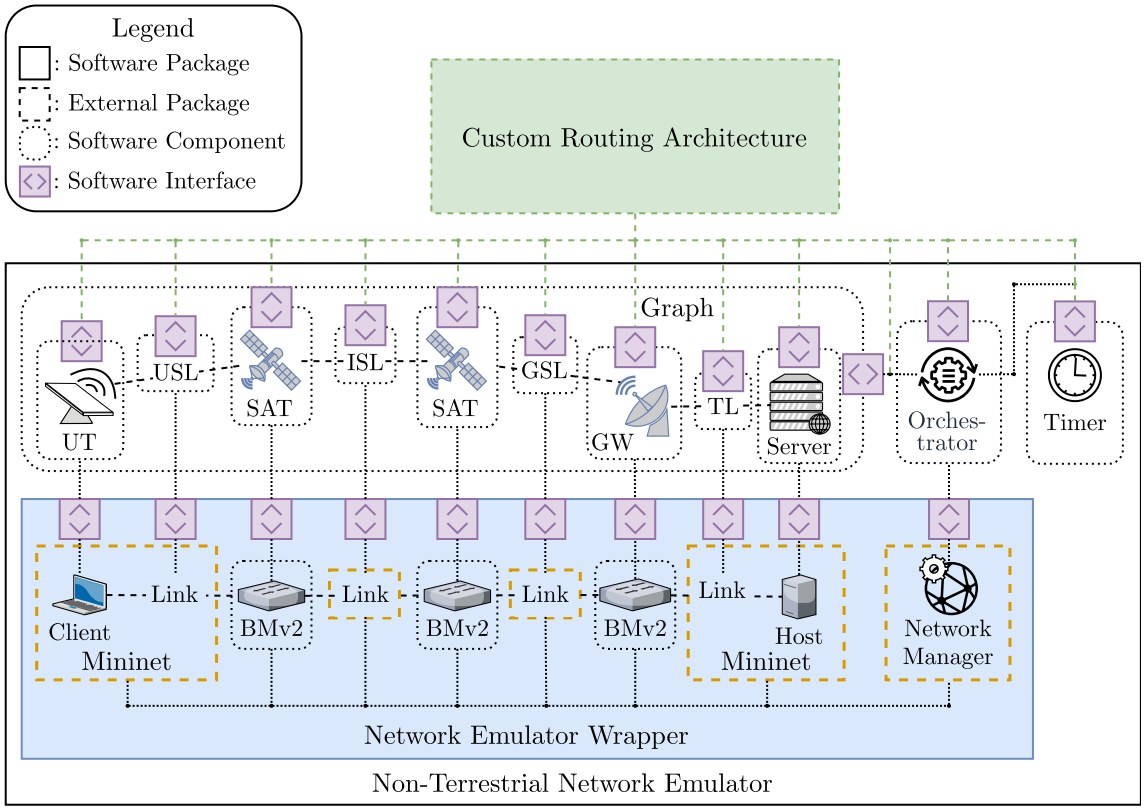



Figure 4.2: System architecture of the NTNE.

The extensibility of the NTNE is given by its modular design and abstract extension points. The system architecture in Figure 4.2 shows the software packages, software components and software interfaces that comprise this design. The NTNE software is split into three software packages: the external Mininet package, the network emulator wrapper (NEW) package and the non-terrestrial network emulator (NTNE) package. The Mininet package contains the Python source code of the Mininet framework, while the NTNE package contains the core software components

of the NTNE. These include components for the NTN nodes (i.e. GWs, SATs, UTs and servers), NTN links (i.e. ISLs, USLs, GSLs and TLs), a graph, a timer and an orchestrator. The NEW package serves as an abstraction layer between the Mininet and NTNE packages. It contains software components that wrap the Mininet components in a standardised network emulation API. The NEW package also includes the extensions for the Mininet framework, such as the BMv2 software switch.

The following sections describe the key features of each component of the NEW and NTNE software packages. The description starts with the NEW package, as it serves as the basis for the NTNE package.

## 4.3 Implementation: Network Emulator Wrapper Package

The network emulator wrapper package serves as an intermediate abstraction layer between the lower-level, classical network emulation layer and the non-terrestrial network emulation layer. It wraps the classical network emulator i.e., Mininet, into a standardised network emulation API and implements required extensions to the emulator. The network emulation API and the BMv2 extension implemented for the Mininet emulator are described in the following.

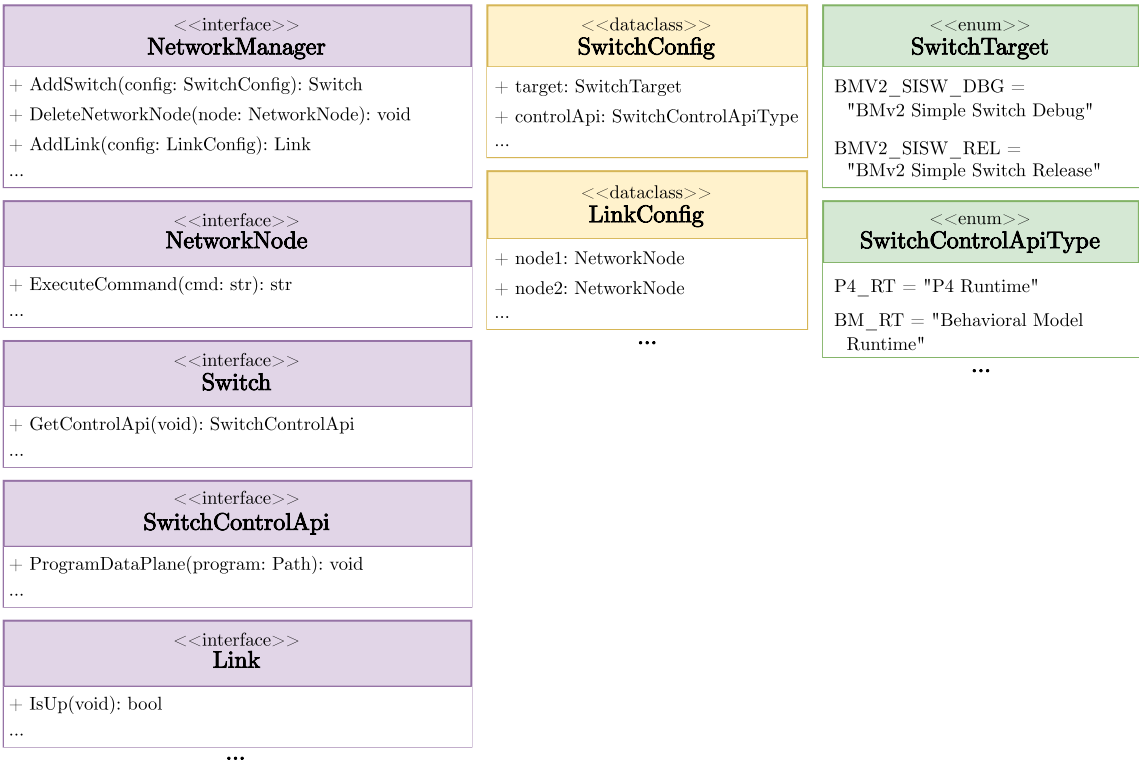



Figure 4.3: Excerpt of the software components i.e., software interfaces (purple), data classes (yellow) and enumerations (green) of the network emulation API.

### 4.3.1 Network Emulation API

The network emulation API consists of multiple software interfaces, data classes and enumerations as shown in Figure 4.3. These separate the classical network emulation i.e., the Mininet emulation, from the non-terrestrial network emulation.

The software interfaces of the network emulation API define the operations supported by each network emulation component. For instance, the NetworkNode and Link interfaces offer common methods for all emulated network nodes and links. These interfaces are further extended by the methods of other interfaces, forming node-specific interfaces, such as the Switch interface.

The configuration options required by certain interface methods are provided using the data classes of the network emulation API. The API's enumerations are used to define specific values for these configuration options.

These software components decouple the NTNE package from the underlying network emulation implementation. This enables the emulator to be replaced or extended (e.g. with Distrinet or different BMv2 modes) without requiring any changes to the NTNE logic. Furthermore, this design simplifies unit testing by separating the software components of the NEW package and enabling lightweight mock implementations.

### 4.3.2 Behavioral Model Version 2

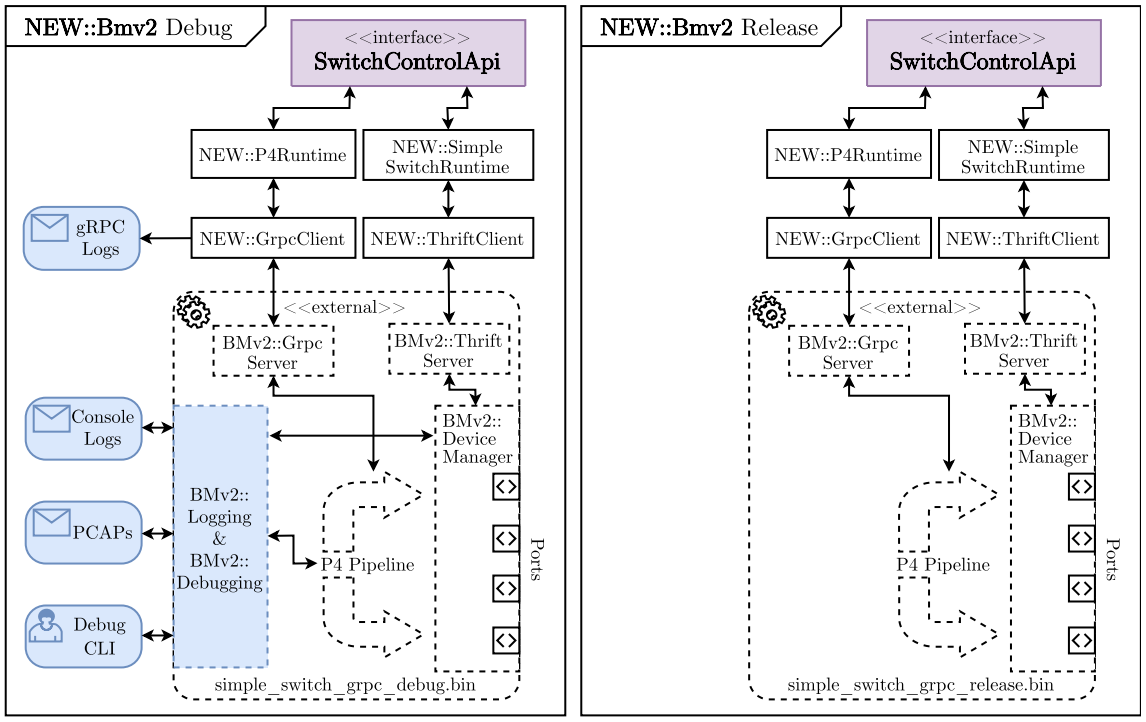



Figure 4.4: Implementation of the BMv2 software switch in the NEW package as debug variant (left) and as release variant (right). The differences are highlighted in blue.

The behavioral model version 2 (BMv2) is a software switch, implemented in C++, designed for developing P4 packet processors. It can run arbitrary P4 programs without recompiling it. Therefore, the P4 program is compiled into a JSON format and then loaded into the BMv2, while it is running [19].

The BMv2 software switch is implemented as multiple targets: simple_switch, simple_switch_grpc, psa_switch, simple_router and l2_switch. These targets implement different P4 architectures, each of which provides a different set of features. For instance, the simple_switch(_grpc) targets implement the v1model architecture, while the psa_switch target implements the portable switch architecture [19]. Currently, the v1model architecture of the simple_switch_grpc target implements the largest feature range of both the $P4_{14}$ and the $P4_{16}$ language specifications [15]. Furthermore, the simple_switch_grpc target natively supports multiple runtimes for controlling the switch's internal components. This is why the NEW package focuses on this target, but allows extension to the others.

The NEW package provides two variants of the BMv2 simple_switch_grpc target, as shown in Figure 4.4: a debug variant and a release variant. The debug variant is used for development purposes. To this end, it provides multiple logging mechanisms and packet capture (PCAP) file recording on all network ports, as well as a debugging command-line interface. The release variant is built with certain compilation flags to exclude these logging and debugging mechanisms from the binary, thus optimising its performance. This makes the release variant useful for evaluating large-scale networks.

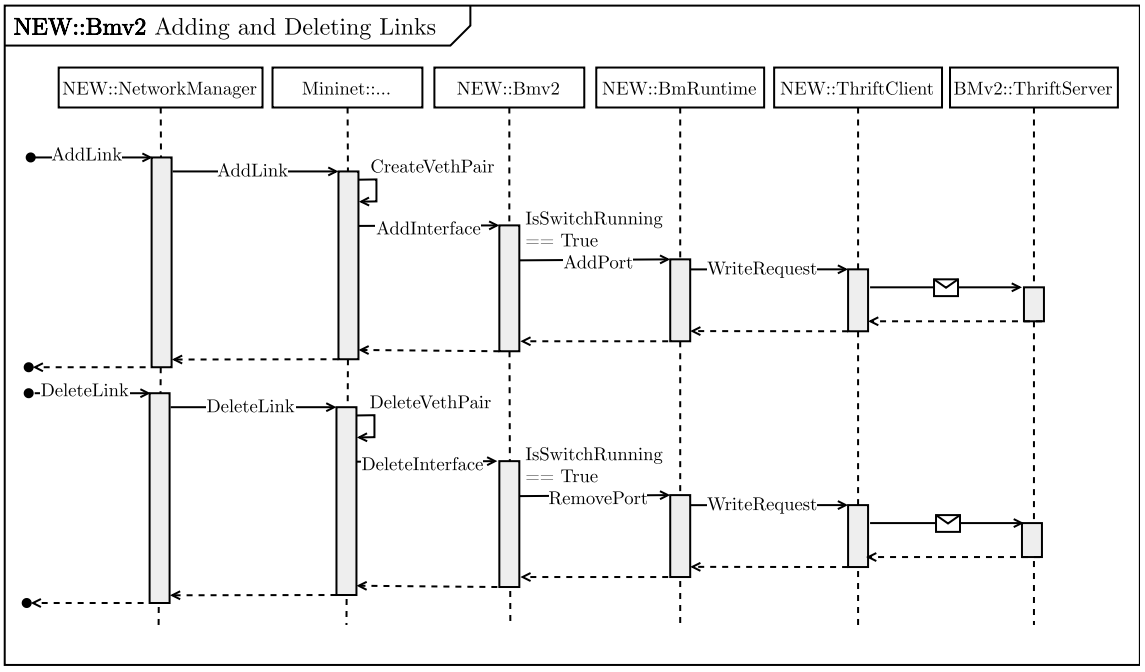



Figure 4.5: Sequence diagram for adding and removing a link from the BMv2 software switch in the NEW package.

The simple_switch_grpc target integrates two distinct control plane runtimes: P4Runtime and SimpleSwitchRuntime. The NEW package employs P4Runtime

as the primary API for configuring the pipeline of the P4 packet processor, as it represents one of the most widely adopted and actively maintained control interfaces [15]. Within the BMv2 simple_switch_grpc target, P4Runtime is provided through a gRPC server (cf. Figure 4.4). To interact with this server, the NEW package implements a corresponding gRPC client along with a software layer responsible for constructing P4Runtime requests. These components are exposed to the NTNE via the SwitchControlApi interface of the network emulation API, thereby enabling the NTNE to configure and control the pipelines of the P4 packet processors.

The SimpleSwitchRuntime of the simple_switch_grpc target is implemented as a Thrift RPC server. The NEW package contains the necessary software components for this runtime and also exposes it to the NTNE package via the SwitchControlApi interface. However, these components are primarily utilized by the BMv2 extension within the NEW package itself. For instance, the SimpleSwitchRuntime is used to access the device manager in the BMv2 target, which is not possible with the P4Runtime. The BMv2 device manager enables control over the veth devices and their assigned switch ports specifically. Figure 4.5 shows a specific use case for the device manager. Here, the BMv2 extension uses the device manager to add and delete links from the software switch while it is running. This allows the emulation of link dynamics as required by NTN topologies.

## 4.4 Implementation: Non-Terrestrial Network Emulator Package

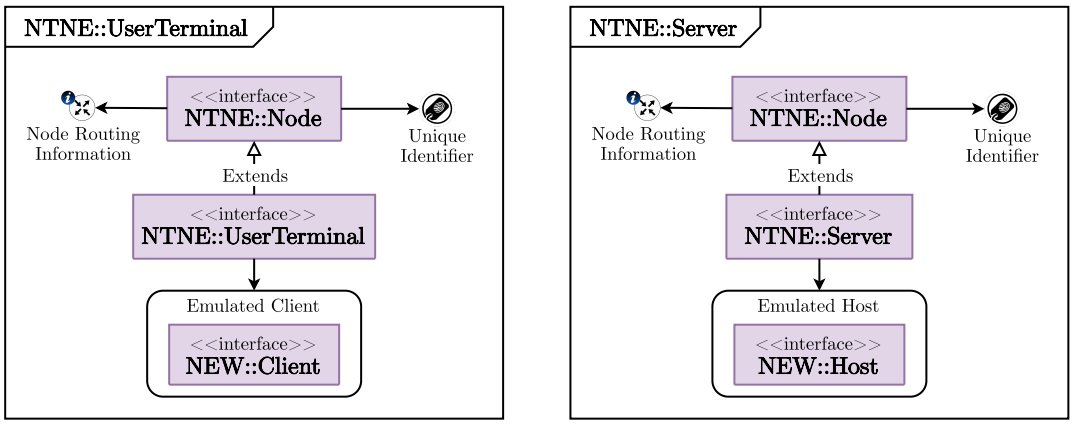
The non-terrestrial network emulator package contains the core software components for the NTN emulation, including components for the NTN nodes, NTN links, a graph, a timer and an orchestrator. The implementation and key features for all components are described in the following.

### 4.4.1 Nodes

A node in the NTNE can either be a user terminal, server, gateway or satellite. All NTN nodes in the NTNE package implement the Node interface as shown in Figure 4.6 and Figure 4.7.

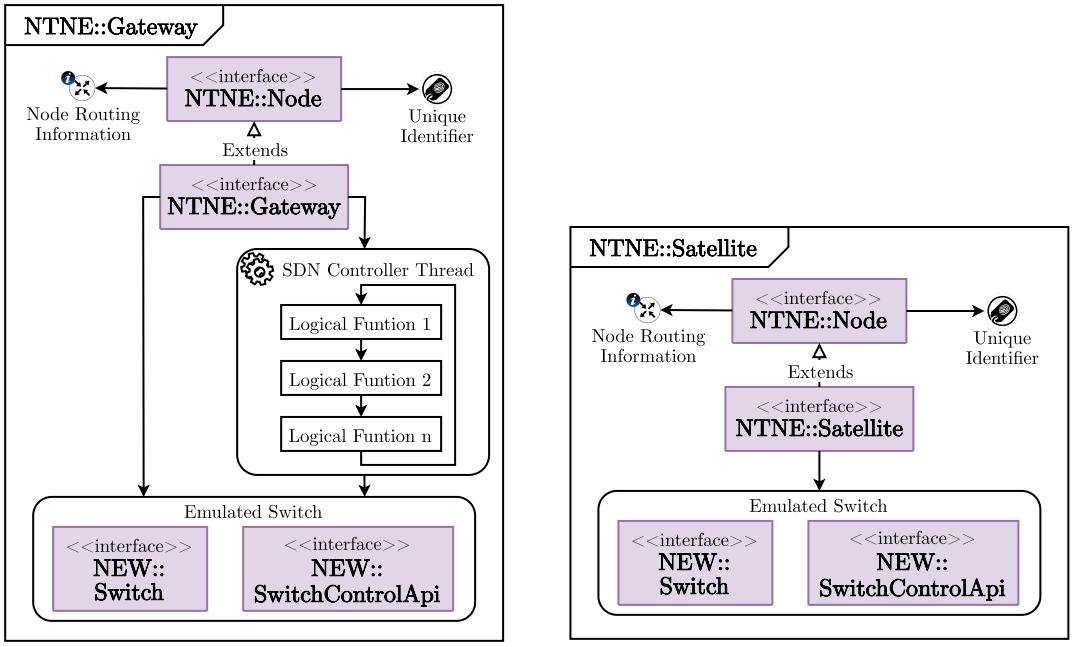
The Node interface exposes a unique identifier and a routing information object for every node. The unique identifier is used to derive the Python object hash, that allows nodes to be used as keys in dictionaries and other hashed containers. The routing information object is intended to carry state used by the custom routing architecture (e.g., neighbor tables, link metrics, next-hop information) and can be updated at runtime.

Each concrete node type encapsulates an emulated network node. The UserTerminal and Server implementations are thin wrappers around Mininet hosts and clients, exposing their specific configurations (e.g., IP and MAC addresses). The Gateway and Satellite implementations wrap BMv2 switch instances. Additionally, Gateway instances host the control plane components of the SDN controller used to program the P4 pipelines of the BMv2 switches. To that end, a concurrent thread is spawned




(a) User Terminal (b) Server

Figure 4.6: Implementation of user terminal and server nodes in the NTNE package.




(a) Gateway (b) Satellite

Figure 4.7: Implementation of gateway and satellite nodes in the NTNE package.

that executes one or more logical functions. These functions can be registered by the custom routing architecture at runtime and are executed sequentially in the order they were registered. Satellite instances provide the programmable data plane and expose the control plane API used by the gateways.

The uniform node abstraction decouples higher-layer routing logic from the underlying network emulation, enabling easy testing, replacement or extension of node behavior without changes to the orchestration or routing components.

### 4.4.2 Links

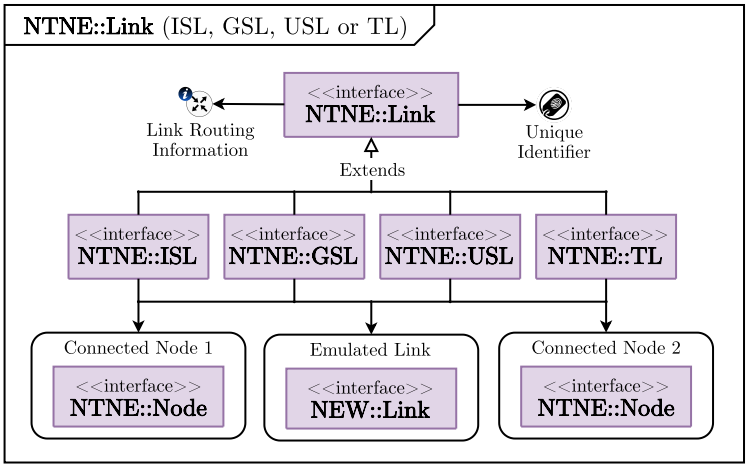



Figure 4.8: Implementation of the NTN links in the NTNE package.

The NTNE provides software components for inter-satellite links, gateway-satellite links, user-terminal-satellite links and terrestrial links. Figure 4.8 shows the implementation structure for all NTN links in the NTNE. Similarly to the NTN nodes, each link implements a common Link interface.

The Link interface exposes a unique identifier for the link and allows assigning an object, that is intended to carry link-specific routing information by the custom routing architecture. This object can be used for example, for link metrics or adjacency segment identifiers in a segment routing architecture.

Each NTN link encapsulates a link from the network emulation layer and provides access to its properties, such as the link state or its connected veth devices. Furthermore, all NTN links can be used to access its connected NTN nodes.

### 4.4.3 Graph

The graph component maintains the NTN topology of the NTNE as an undirected NetworkX graph (cf. Figure 4.9). Nodes in the graph correspond to NTN node objects and edges correspond to NTN link objects. Multiple parallel edges are supported to model redundant physical paths or separate logical channels.

The Graph interface provides various topology queries, such as retrieving individual NTN nodes by their unique identifiers or obtaining the neighbors of a given node. In addition, it offers atomic update operations for adding and removing nodes and links, ensuring consistency during topology changes. Finally, the interface includes functions for computing paths through the graph, enabling routing algorithms to operate directly on the maintained NTN topology.

The graph component is designed to carry spare NTN nodes. These are nodes that are not part of the current active topology, but are pre-provisioned and available to be atomically inserted into the graph. This can be used to emulate network dynamics, for example, for failed network nodes that are reactivated. Spare nodes

can carry pre-configured properties, such as interface assignments, initial P4 pipeline images and routing state, to enable fast activation.

All operations to the NetworkX graph are performed under a semaphore lock ensuring atomicity and a consistent runtime view for routing algorithms and the orchestrator.

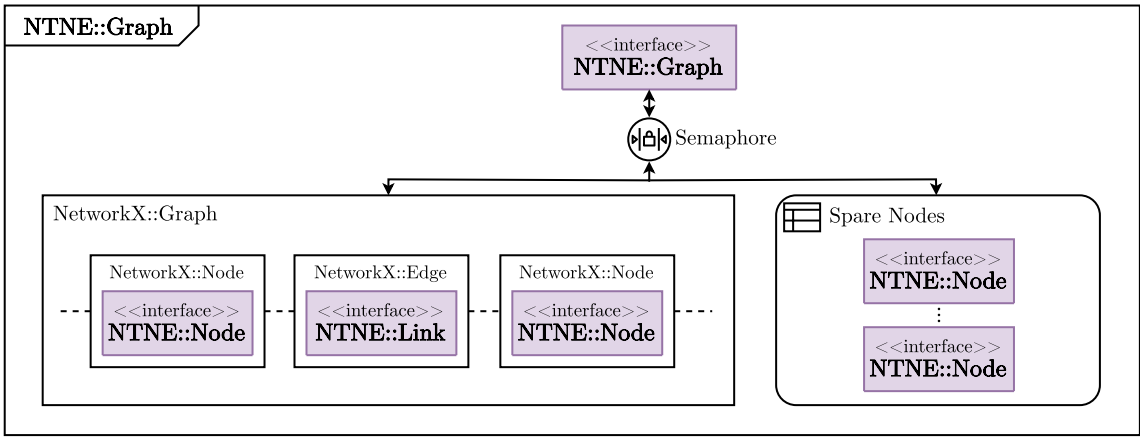



Figure 4.9: Implementation of the NTN graph in the NTNE package.

### 4.4.4 Timer

The timer implements the emulation clock and is intended as a deterministic event scheduler. It tracks the elapsed time in the NTN since a predefined epoch time. The timer allows replaying NTN events for example, link handovers or routing updates, in real-time with nanosecond precision. It is leveraged for the synchronisation of internal events in the NTNE components and external events from the custom routing architecture. To that end, routing functions and also the orchestrator workflows use the timer to schedule future actions and to drive time-dependent topology changes.

### 4.4.5 Orchestrator

Figures 4.10 and 4.11 show the internal architecture and dynamic behavior of the NTNE orchestrator. The orchestrator acts as the central coordination component of the emulator, responsible for synchronizing topology changes, managing time progression, and ensuring that the emulated NTN evolves according to the defined scenario.

#### 4.4.5.1 Orchestration Workflow

Figure 4.10 shows the operational workflow implemented by the NTNE orchestrator. It functions as a state-machine-like controller that continuously monitors the emulation clock and applies scheduled modifications to the emulated topology. The orchestrator interacts with three main software components of the NTNE, through their respective interfaces:

- **NTNE::Timer**: Retrieve the current emulation timestamp.
- **NTNE::Graph**: Maintain the emulated NTN topology as a dynamic graph.
- **NEW::NetworkManager**: Apply node and link changes to the underlying emulation platform.

The orchestration loop is executed in a dedicated thread, ensuring the required concurrency of the orchestration workflow. It follows these steps:

1. **Get emulation time:** The orchestrator queries the NTNE timer for the current emulation time.
2. **Activate next scene?:** It checks whether the activation time of the upcoming scene has been reached. If so, it loads the corresponding scene description.
3. **Apply scene changes:** The orchestrator instructs the NetworkManager to apply the listed modifications i.e., adding or removing nodes and links, and updates the topology graph accordingly.
4. **Last scene active?:** If the active scene is the final entry in the scenario, the orchestrator remains in a steady state until the emulation terminates. Otherwise, it advances to the next scheduled scene.

This loop runs throughout the emulation, ensuring that time-based events, such as mobility-induced link changes, handovers, or ISL reconnections, are consistently applied. The orchestrator thus provides the core mechanism by which the emulator captures the dynamic behavior of LEO satellite networks.

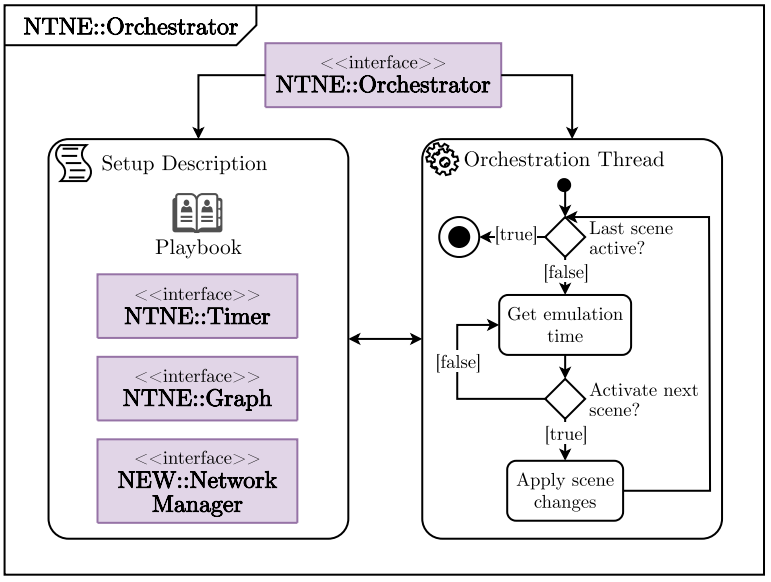



Figure 4.10: Implementation of the NTN orchestrator in the NTNE package.

#### 4.4.5.2 Scene-Based Playbook Structure

Figure 4.11 shows how emulation scenarios are defined using a sequence of scenes, each representing the state of the NTN at a specific moment in time. The playbook is implemented as a single-linked list, where each scene points to the next scheduled scene. Every scene includes a change description that specifies:

- **Added nodes**, e.g., satellites entering visibility,
- **Removed nodes**, e.g., satellites leaving coverage,
- **Added links**, e.g., ISLs established or ground-to-satellite connections formed,
- **Removed links**, e.g., broken ISLs or handover-induced disconnections,
- **Activation time**, when the changes become effective.

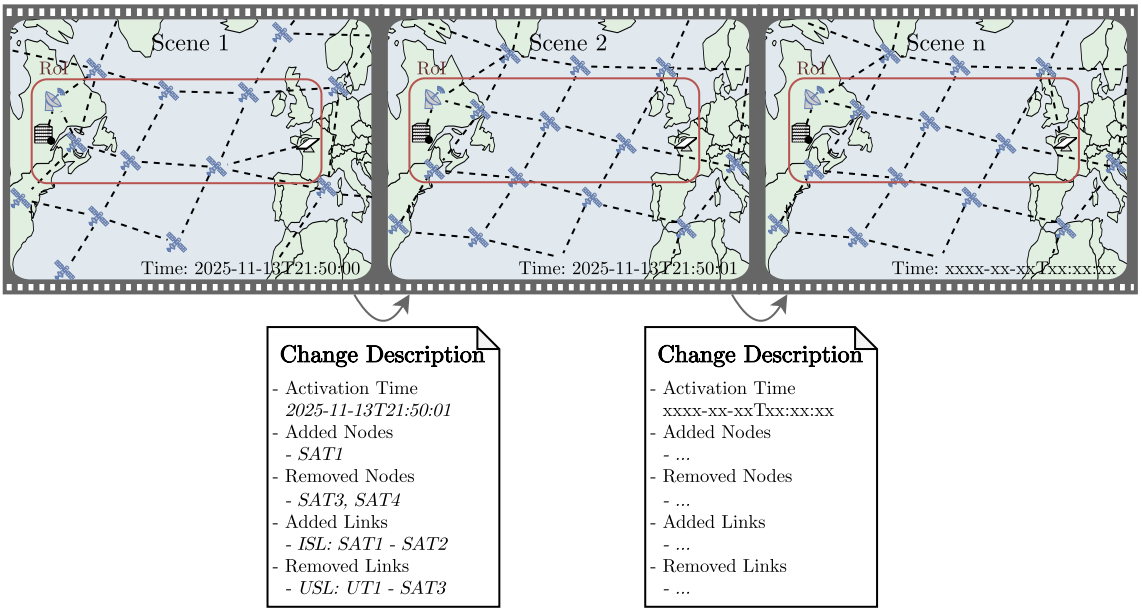



Figure 4.11: Abstract implementation of the NTN playbook used by the NTN orchestrator in the NTNE package.

This scene-based representation decouples scenario definition from runtime execution. Dynamic behavior such as satellite motion, topology transitions, and coverage changes can be precomputed or externally generated and then encoded as a structured playbook. During emulation, the orchestrator interprets the playbook and applies each scene's changes exactly at its scheduled activation time.

In addition to its scene-based design, the playbook was extended with further mechanisms to ensure scalability in large or long-running emulation scenarios. To improve scalability for scenarios with a large number of scenes, the playbook implementation was extended to support multi-threading environments and enhanced through the introduction of the region of interest (RoI) concept.

By protecting accesses to the playbook's single-linked scene list with a semaphore, scene generation can be delegated to a separate thread that runs concurrently to the orchestrator. This decoupling enables new scenes to be produced while the orchestrator processes previously generated ones, significantly reducing bottlenecks and allowing the emulator to handle complex and large-scale NTN scenarios more efficiently.

In addition to enabling parallelisation, the RoI mechanism provides another scalability advantage by restricting scene generation to a defined geographical region of interest. Instead of producing constellation data and NTN state information for the entire global satellite system, the RoI limits computation and storage to satellites, ground nodes, and links that are relevant for the selected region. This significantly reduces the number of nodes and events that must be processed, lowers memory consumption, and avoids generating scenes for satellites that never interact with the emulated portion of the network. As a result, the emulator can handle larger constellations and more complex scenarios while maintaining efficient scene generation and real-time applicability.

## 4.5 Summary

This chapter presented the non-terrestrial network emulator (NTNE), a new emulation framework developed to support the evaluation of programmable routing architectures in dynamic LEO satellite networks. Since existing emulators do not provide the required combination of SDN support, data plane programmability, scalability, and extensibility, the NTNE was designed to fill this gap.

The chapter outlined the key requirements for such an emulator and introduced the overall architecture built on top of Mininet and extended through the NEW package. This design enables the NTNE to emulate programmable satellites and gateways using BMv2 switches while maintaining a modular structure that is easy to extend.

Furthermore, the chapter described the main components of the NTNE, including its unified node and link abstractions, the centralized graph representation of the topology, the deterministic emulation timer, and the orchestrator. The orchestrator, together with its scene-based playbook, provides a flexible mechanism to model the dynamic behavior of LEO constellations. Extensions such as multi-process scene generation and the region of interest concept enhance scalability and allow the emulator to handle large and complex scenarios efficiently.

Overall, the NTNE offers a flexible and scalable foundation for experimenting with and evaluating routing architectures in non-terrestrial networks.

# 5 Int2Sat Routing Architecture

Although low Earth orbit satellite constellations with inter-satellite links offer global coverage and resilient connectivity, their highly dynamic topologies and strict resource constraints pose significant routing challenges. The Int2Sat routing architecture, proposed by Menth and Flüchter [5], addresses these challenges and provides a framework for the seamless integration of LEO satellite networks with the terrestrial Internet.

This chapter summarizes Int2Sat's internals, including its system model, design principles, and architectural routing mechanisms. These foundations serve as the basis for the subsequent implementation on the NTNE and the evaluation of its behavior under representative NTN scenarios.

## 5.1 System Model

Figure 5.1 shows the system model of the Int2Sat routing architecture. Int2Sat considers non-terrestrial networks to consist of a LEO satellite constellation composed of numerous satellites that continuously orbit the Earth. These satellites provide connectivity to ground-based UTs and GWs through radio links. The satellite that maintains an active radio link to a UT or GW is referred to as the access satellite (AcS) for the ground node.

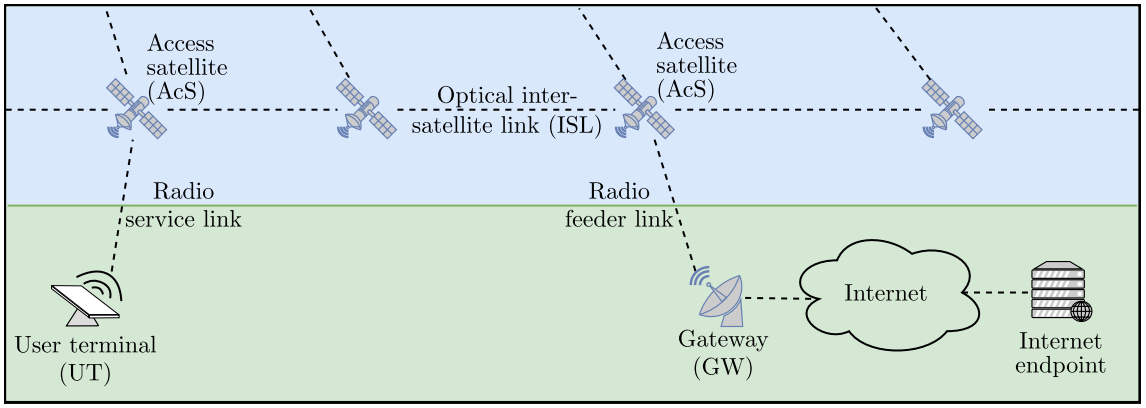



Figure 5.1: System model of the Int2Sat routing architecture.

Links between UTs and their AcSs are called service links, and links between GWs and their AcSs are called feeder links. Gateways serve as the interface between the NTN and the terrestrial Internet, enabling the exchange of user traffic across both domains. Satellites are interconnected via optical inter-satellite links, forming a spaceborne mesh network. Both satellites and gateways are equipped with packet

processors that forward packets across their attached links. This establishes a unified routing and forwarding infrastructure across the entire NTN.

## 5.2 Design Principles

The design of Int2Sat is guided by three core principles. These are described in the following.

First, UTs are entirely excluded from Int2Sat's routing domain. UTs operate as conventional IP nodes without maintaining routing state or participating in constellation-level forwarding decisions. This ensures vendor independence, simplifies UT hardware, and avoids exposing routing logic to untrusted devices.

Second, the architecture is designed to minimize the complexity of satellite packet processors. Satellites are limited to lightweight forwarding and minimal IP routing for directly attached UTs. They do not perform routing decisions, compute paths, or update forwarding states. This design choice prevents excessive overhead from frequent topology changes and addresses the satellites' limited processing, memory, and energy capabilities.

Third, Int2Sat shifts all routing intelligence from the constellation to the gateways by adopting the software-defined networking (SDN) paradigm. Instead of relying on distributed routing onboard satellites, the control plane is implemented in the NTN's GWs, which possess significantly greater computational resources and stable, ground-based connectivity. The gateways maintain a global, time-aware view of the constellation. They compute end-to-end paths, manage topology changes, and proactively update the forwarding state across the network.

Together, these principles establish a robust and scalable foundation for integrating Internet routing with LEO constellations while respecting their unique operational constraints.

## 5.3 Basic Internet Integration

Int2Sat is designed to facilitate a seamless and robust integration of NTNs with the terrestrial Internet. The interface between the NTN and the Internet is realized exclusively through the NTN's gateways, which implement all mandatory data plane and control plane functionalities. These functionalities are detailed in the following.

In the data plane, the GWs ensure the correct forwarding of traffic originating within the NTN and destined for the Internet, as well as traffic originating in the Internet and addressed to nodes served by the NTN. Int2Sat adopts the segment routing architecture standardized by the IETF in RFC 8402 [11]. Segment identifiers (SIDs) are used to address either nodes or links in the NTN. An ordered list of segments encodes a complete source route across the constellation. Packets received from a UT or Internet node are encapsulated with a source route at their respective NTN ingress i.e., the UT's AcS or the GW connected to the Internet. The packets are then steered through the network via the SIDs of the inserted route.

The computation of segment lists and their SIDs is a central element of the GWs control plane responsibilities. These computations reflect the current topology of

the constellation and its anticipated evolution. Int2Sat distributes control plane functions across three distinct logical GW roles: the home gateway (HGW), the default gateway (DGW), and the control gateway (CGW). Each role is shown in Figure 5.2 and is further described below.

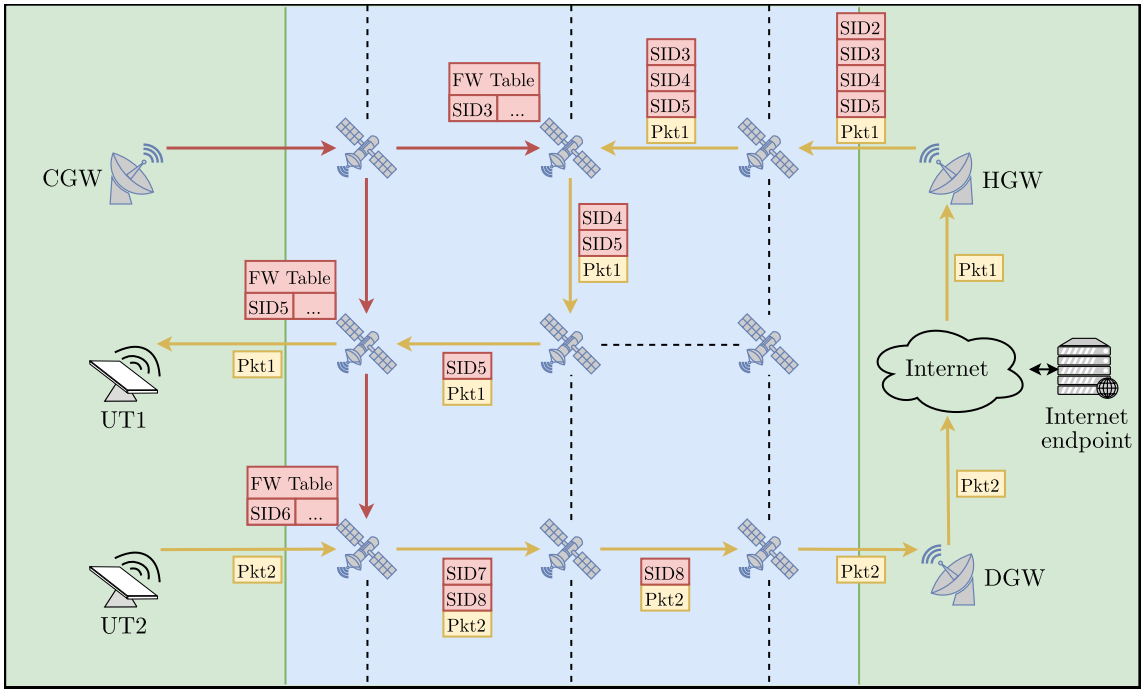



Figure 5.2: Basic routing architecture of Int2Sat for integrating NTNs leveraging LEO satellite constellations into the Internet. The figure highlights Int2Sat's SR data plane and the distribution of the control plane logic among the HGW, DGW, and CGW roles.

The HGW is responsible for forwarding Internet-originated traffic toward UTs. To ensure global reachability, each HGW advertises an IP prefix to the Internet via BGP. Each UT is associated with a specific HGW and obtains an IP address from its prefix, thereby enabling identification within the Internet. When an HGW receives a packet from the Internet destined for an associated UT, it determines the appropriate segment list for the UT by consulting its SR encapsulation table. After encapsulating the packet with the segment list, it is forwarded through the constellation along the prescribed source route.

Conversely, packets originating from UTs and destined for the Internet are encapsulated at the serving AcS of the UT and forwarded toward a designated DGW. The DGW is responsible for configuring and maintaining the source routes from all assigned AcSs to itself. Similar to the HGW, the DGW maintains Internet connectivity and forwards received Internet-bound traffic via the route obtained from standard Internet routing mechanisms.

Finally, the CGW is responsible for configuring all SIDs employed in the SR forwarding tables throughout the NTN. It ensures that the SID assignments required for source routes established by the HGW and DGW are correctly installed and

continuously updated on all satellites in accordance with the evolving constellation topology.

## 5.4 Traffic Engineering

Although the basic Int2Sat Internet integration design ensures correct forwarding under dynamic LEO topologies, the resulting paths are not always optimal in terms of latency or load distribution. Traffic may traverse suboptimal gateway locations or follow longer-than-necessary satellite paths, especially when the HGW or DGW is geographically distant from the source or destination of the traffic. To address these inefficiencies, Int2Sat introduces optional traffic engineering (TE) mechanisms.

Gateways maintain a global, time-dependent view of the constellation. This enables them to compute optimized segment routing paths tailored to specific performance objectives. Several TE strategies are supported:

- **Latency-aware path selection:** HGWs or DGWs may choose alternative ingress or egress gateways, or compute satellite-only detours, to reduce end-to-end delay for Internet- or UT-bound traffic.
- **Constellation load distribution:** Segment routing enables the installation of multiple disjoint or partially overlapping satellite paths. This allows load balancing across ISLs, mitigating congestion and avoiding hotspots near frequently used satellites.
- **Feeder link load distribution:** When radio interfaces of gateways become overloaded, traffic may be temporarily rerouted through alternate gateways to improve service continuity or prevent link saturation.

All TE mechanisms rely on the same segment routing approach as described in the previous section and require no additional satellite functionality. This makes Int2Sat well-suited for experimentation with routing optimizations in LEO constellations, where predictable topology changes and centralized control allow for proactive, fine-grained traffic steering.

## 5.5 Protection Switching

In addition to supporting basic forwarding and traffic engineering, Int2Sat aims to provide resilience against unexpected link or node failures within the constellation. Although LEO satellite networks exhibit largely predictable dynamics, unplanned disruptions, such as temporary ISL or satellite outages, can still occur. Therefore, the architecture incorporates protection switching concepts.

The fundamental design principle is to maintain lightweight protection logic on satellites while placing the computation of backup paths in the gateways. Satellites only store a small set of precomputed backup next hops for their segment routing entries. This enables immediate local deviation when a link or adjacent node

fails. Gateways, informed by predicted link availability and potential failure patterns, proactively compute these detours and install them alongside primary paths. This approach avoids complex local failure detection or recomputation on satellites, keeping onboard processing requirements minimal.

## 5.6 Summary

Int2Sat provides a structured and scalable approach for integrating the Internet with non-terrestrial networks leveraging LEO satellite constellations. The architecture aligns naturally with the resource constraints and dynamic behaviour of these non-terrestrial networks by decoupling control and data plane functions, assigning all routing intelligence to gateways, and limiting satellites to lightweight segment-routing operations. Segment routing serves as an efficient forwarding mechanism, enabling explicit path control, predictable updates under orbital motion, and extensibility toward advanced features, such as traffic engineering and protection switching.

# 6 Routing Architecture Implementation

The previous chapter outlined the conceptual design of the Int2Sat routing architecture and its role within non-terrestrial networks. This chapter presents its concrete realisation on the non-terrestrial network emulator introduced in Chapter 4. It translates the architectural principles into an operational control and data plane suitable for evaluation under dynamic LEO conditions.

The following sections first describe the overall software architecture, including the interaction between Int2Sat, and the NTNE. Subsequently, the selection process of a suitable encoding for Int2Sat's segment routing approach is presented. Finally, the implementation of the Int2Sat software package is detailed, covering the processing of routing information and the configuration of the P4-based packet processors used by satellites and gateways.

## 6.1 Software Architecture

This section presents the software architecture for the implementation of the Int2Sat routing architecture on the NTNE presented in Chapter 4. The complete system architecture for the implementation is shown in Figure 6.1.

The system architecture consists of five packages. The Int2Sat package contains the main software components for the routing architecture logic. It also comprises the simulation and satellites (SandSat) package, which provides constellation information, as well as the packet processors package. The latter contains the P4-based packet processors for the satellite and gateway switches in the NTN. The system architecture also shows the underlying non-terrestrial network emulator package and its network emulator wrapper package described in Chapter 4. Together, these packages enable a full-stack emulation of dynamic LEO satellite networks running the Int2Sat routing logic. The specific functionality of each package is described below.

At the top of the system architecture, the Int2Sat package implements the control plane functions for the NTN. Its core is formed by three logic functions, that are executed on the NTNE gateway nodes: the HGW, CGW, and DGW functions. These functions operate on both static and dynamic routing information. Static routing information includes MAC addresses, IP addresses, and SIDs, which form the addresses for the nodes and links in the NTN. Dynamic routing information comprises link-state changes, topology updates, and route recalculations. As shown in the system architecture, updates are distributed among the different gateway functions (HGW, CGW and DGW), reflecting their responsibilities in the hierarchical Int2Sat control plane design.

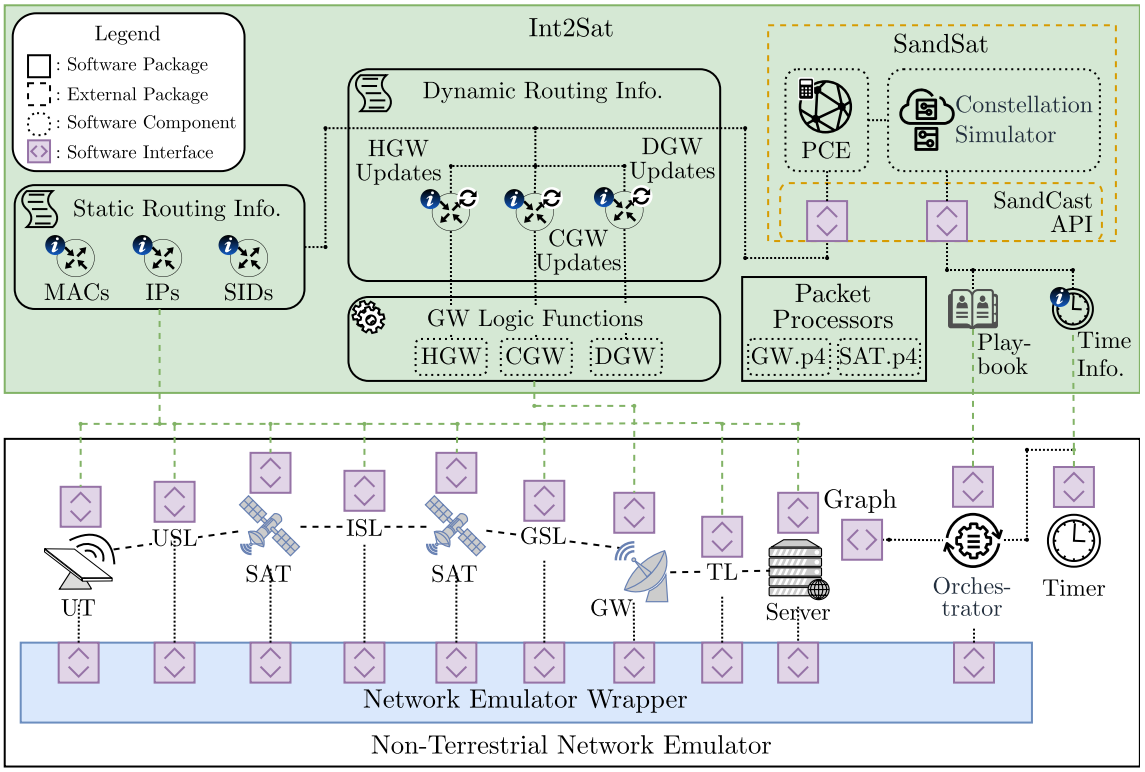



Figure 6.1: System architecture for the implementation of Int2Sat on the NTNE.

To provide realistic satellite dynamics, Int2Sat integrates with the SandSat library, proposed in [6] (see the upper-right corner of the system architecture). SandSat supplies essential external data of an emulated NTN, including constellation state, time information, and topology changes. The SandSat constellation simulator is used to simulate satellite constellations. It produces time-stepped information for the connections between satellites, gateways, and user terminals. This information is used to build the playbook for the NTNE orchestrator, which contains the time-stepped topology changes. The Path Computation Element (PCE) in SandSat calculates paths between the NTN nodes. The produced paths are provided as node lists. These lists are combined with the static routing information from the Int2Sat package to generate the segment routing encapsulation headers for packets traversing Int2Sat's SR domain. Changes to these headers and to the SR forwarding tables are packaged as gateway logic updates. The gateway logic functions apply these updates to the P4-based packet processors in the NTN. All communication between SandSat and Int2Sat occurs through the SandCast API, depicted as the integration point between the two packages.

The lower part of the system architecture shows the NEW and the NTNE packages, which are described in detail in Chapter 4. These components implement the non-terrestrial network on which the Int2Sat routing architecture operates. The UTs, SATs, GWs, and servers are instantiated as emulated nodes connected by USL, GSL, ISL, and TL links. The NTNE orchestrator processes the topology changes in the supplied playbook, while the NTNE timer ensures synchronization with the

constellation time and routing updates of the Int2Sat package.

The following sections cover the implementation details of the presented software architecture. The next section focuses on selecting a suitable encoding for Int2Sat's segment routing approach, as Int2Sat itself does not imply a specifc segment routing encoding. The final section of this chapter describes the implementation of the Int2Sat package in accordance with the selected segment routing encoding.

## 6.2 Implementation: Segment Routing Encoding

The Int2Sat routing architecture uses an SR paradigm to steer packets through the LEO satellite network. However, the architecture intentionally leaves the specific segment routing encoding open to the implementer in order to remain flexible with respect to different deployment environments. This section analyzes the available segment routing data plane standards proposed by the Internet engineering task force (IETF) to determine the most suitable standard for implementation in this thesis. The analysis includes the following IETF SR standards:

- Segment routing with MPLS (SR-MPLS) published as RFC 8660 [12]
- Segment routing over IPv6 (SRv6) published as RFC 8986 [42]
- Compressed SRv6 (CSRv6) published as RFC 9800 [14]

The next subsections introduce the evaluation methodology used to analyze each segment routing standard. Then, the analysis results are presented, including the selection of the most suitable segment routing encoding for the Int2Sat routing architecture.

### 6.2.1 Methodology

A structured comparison is required to identify which of the IETF segment routing encodings is most suitable for the Int2Sat routing architecture. Since Int2Sat is designed around the resource constraints and operational characteristics of LEO satellites, the choice of a suitable segment routing encoding must align with these satellite-specific requirements. In particular, satellites are considered by Int2Sat to operate with limited memory capacity and restricted computational performance. Therefore, the size of the segment routing header and the number of bits that must be processed per packet directly impact the applicability of each SR approach. The methodology of this analysis is therefore driven by two central requirements: minimizing the memory footprint associated with storing and buffering packets, and minimizing the number of bits that must be inspected or manipulated during forwarding operations.

Because SR-MPLS, SRv6, and CSRv6 use fundamentally different header formats and SID encodings, a normalization step is required to establish a fair basis for comparison. Figure 6.2 shows the normalization for the SRHs and the SID encodings in all three standards.

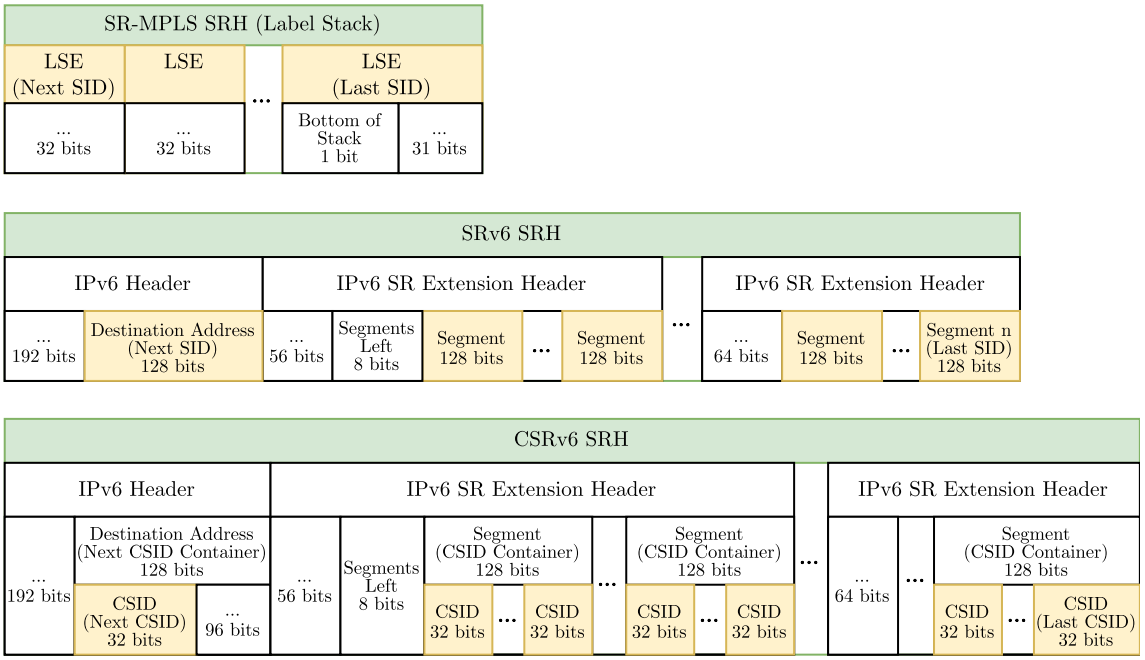



Figure 6.2: Normalization of the SRHs and SID encodings of SR-MPLS, SRv6 and CSRv6.

The SR-MPLS standard encodes the SRH as an MPLS label stack consisting of multiple label stack entries (LSEs). Each LSE consists of 32 bits and encodes one SID. The next SID is always contained in the LSE at the top of the label stack. An SR-MPLS router uses the next SID to identify the forwarding instruction for a received packet (e.g., forwarding the packet via a specific interface). Furthermore, the next SID identifies the operation performed on the label stack. These operations can be a push, pop or skip. The bottom-of-stack bit marks the last SID of the label stack. Therefore, it is sufficient for an SR-MPLS router to inspect only the topmost LSE, while the rest of the label stack does not need to be inspected [12].

In SRv6, the SRH is encoded using an IPv6 header and multiple IPv6 SR extension headers. The IPv6 header is positioned at the beginning of the SRv6 SRH. The destination address field of the IPv6 header contains a copy of the next SID. All SIDs of an SRv6 segment route are encoded in the IPv6 SR extension headers with a fixed size of 128 bits. Segment identifiers with a smaller size than 128 bits are padded with additional bits. To that end, the normalization of SRv6 and SR-MPLS consists of encoding each 32-bit SR-MPLS LSE as a 128-bit SRv6 SID with 96 bits of padding. RFC 8754 defines a reduced SRH, that omits the first SID from the IPv6 SR extension header because the SID is already included in the destination address field of the IPv6 header. This allows segment routes with a single SID to omit the IPv6 SR extension header, thereby reducing the overall size of the SRH. Processing an SRv6 SRH at an SRv6 router begins with inspecting the IPv6 header and performing a longest-prefix match in the router's forwarding information base (FIB) with the SID in the IPv6 header's destination address field. This identifies the forwarding instruction for the packet and the operation to perform on the SRH.

After the lookup, the SRv6 router inspects the IPv6 SR extension headers and copies the next SID to the destination address field of the IPv6 header. The next SID is identified using the segments-left field of the IPv6 SR extension header. This analysis assumes a worst-case scenario, wherein the next SID is found in the last IPv6 SR extension header [13], [42].

The CSRv6 segment routing standard aims to reduce the size of the SRv6 SRH by introducing compression flavors for the SRv6 SIDs. Figure 6.2 illustrates the basic principle for all compression flavors. Each flavor introduces a CSID and a format for encoding multiple CSIDs within the IPv6 SR extension header segments. In this context, these segments are also called CSID containers. The flavors do not impose a specific size for the CSIDs, instead, an implementer can choose the size for the CSRv6 domain. For the normalization of SR-MPLS and CSRv6, each CSID is assigned a size of 32 bits. This means that each CSID container can contain up to four MPLS LSEs. Processing a CSRv6 SRH is similar to the processing of an SRv6 SRH. The CSRv6 router performs a longest-prefix match of the CSID in the IPv6 header's destination address to determine the packet's forwarding instruction and the operation to perform on the SRH. Afterwards, the next CSID is copied from its CSID container to the topmost bits of the IPv6 header's destination address field, replacing the current CSID. Identification of the next CSID is based on the segments-left field in the IPv6 SR extension header and the remaining bits of the IPv6 header's destination address field. For the normalization with SR-MPLS, two bits are used to encode the index of the current CSID in a CSID container, allowing to address all four CSIDs in the container. As with SRv6, this analysis assumes the worst-case scenario for identifying the next CSID, i.e., that the next CSID is found in the last IPv6 SR extension header [14].

The described normalization allows the SR approaches to be evaluated using common metrics: (i) the SRH size, and (ii) the number of processed bits that contribute to the processing overhead at each hop. The SRH size metric quantifies the additional memory a satellite must allocate when receiving and forwarding an SR packet. The processed bits metric captures the number of bits that must be parsed, matched, or modified at each router. This metric reflects the computational load and forwarding latency on a satellite.

Using this consistent and implementation-independent methodology, the three segment routing standards can be compared objectively with respect to the resource limitations of LEO satellites.

### 6.2.2 Results

The results of this analysis are presented in Figures 6.3–6.4.

Figure 6.3 displays the SRH size metric for segment route lengths ranging from 1 to 50. SR-MPLS exhibits the smallest and most stable SRH size across all route lengths, increasing linearly with a slope of four bytes per additional segment and reaching 200 bytes at 50 segments. In contrast, SRv6 shows a very steep increase due to its 128-bit SID format and the required IPv6 and SR extension headers. This results in an SRH size of 856 bytes for 50 segments. The steps, observed in the SRv6 plot, originate from the fact that the IPv6 SR extension headers can contain

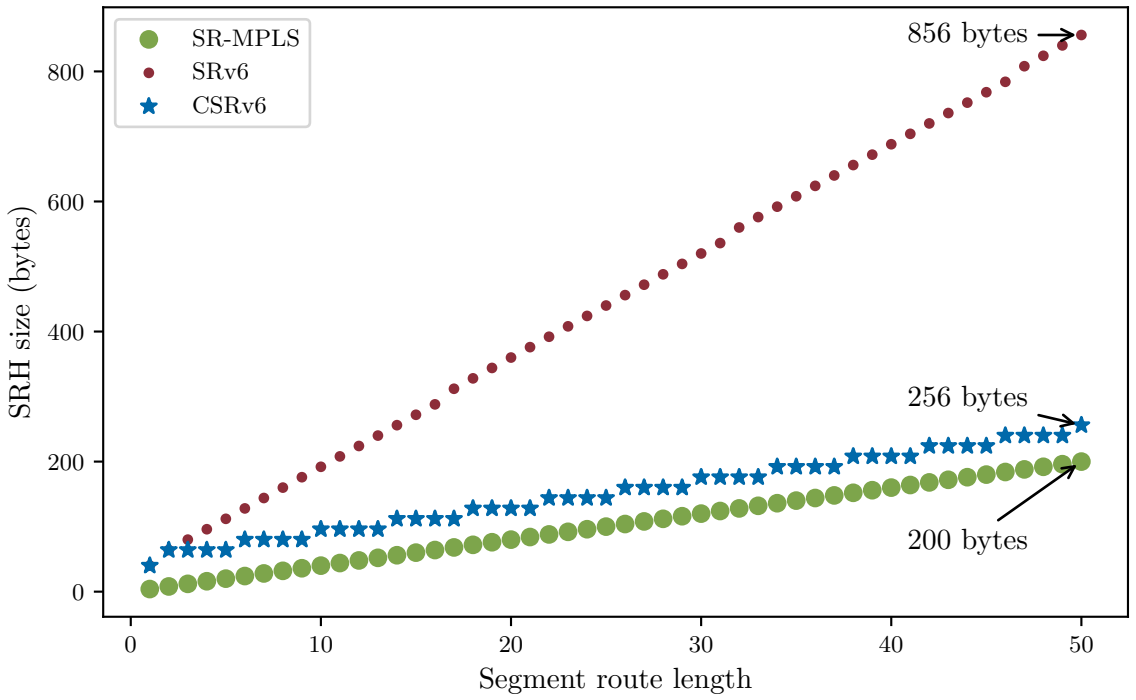



Figure 6.3: Comparison of the SRH sizes for the normalized SR-MPLS, SRv6 and CSRv6 standards for different segment route lengths.

only 15 segments at once. This requires an additional IPv6 SR extension header every 15 segments. A similar characteristic is shown in the CSRv6 plot. Here, the header size increases stepwise every four segments. This behavior stems from the structure of CSRv6's CSID containers, which can contain only four CSIDs at once. Compared to SRv6, CSRv6 substantially reduces the SRH size. However, even with its compressed SIDs, the SRH size still grows to 256 bytes at 50 segments and thus remains larger than the SRH size of SR-MPLS.

A similar behavior is observed for the number of bits that must be processed per hop, as shown in Figure 6.4. SR-MPLS requires 32 bits to be examined per hop, regardless of the segment route length, because forwarding decisions depend solely on the topmost 32-bit LSE. CSRv6 again provides a noticeable reduction compared to SRv6. However, it still reaches 322 processed bits at 50 segments, primarily due to the parsing of the IPv6 header and IPv6 SR extension header fields. SRv6 introduces the highest processing overhead, reaching 704 bits for 50 segments. As with the SRH size, the number of processed bits for SRv6 increases stepwise with the number of IPv6 SR extension headers required for the SID encoding. The smaller SRH sizes for SRv6 and CSRv6 for a segment route with only one segment, stem from the reduced SRH encoding, which omits the IPv6 SR extension header.

The results of both metrics consistently show that SR-MPLS has the smallest memory footprint and the lowest processing overhead. Given the strict memory and computational constraints of LEO satellites, SR-MPLS is the most resource-efficient and practically viable segment routing encoding for the Int2Sat routing architecture. Nevertheless, CSRv6's compression is a possible alternative worthy of

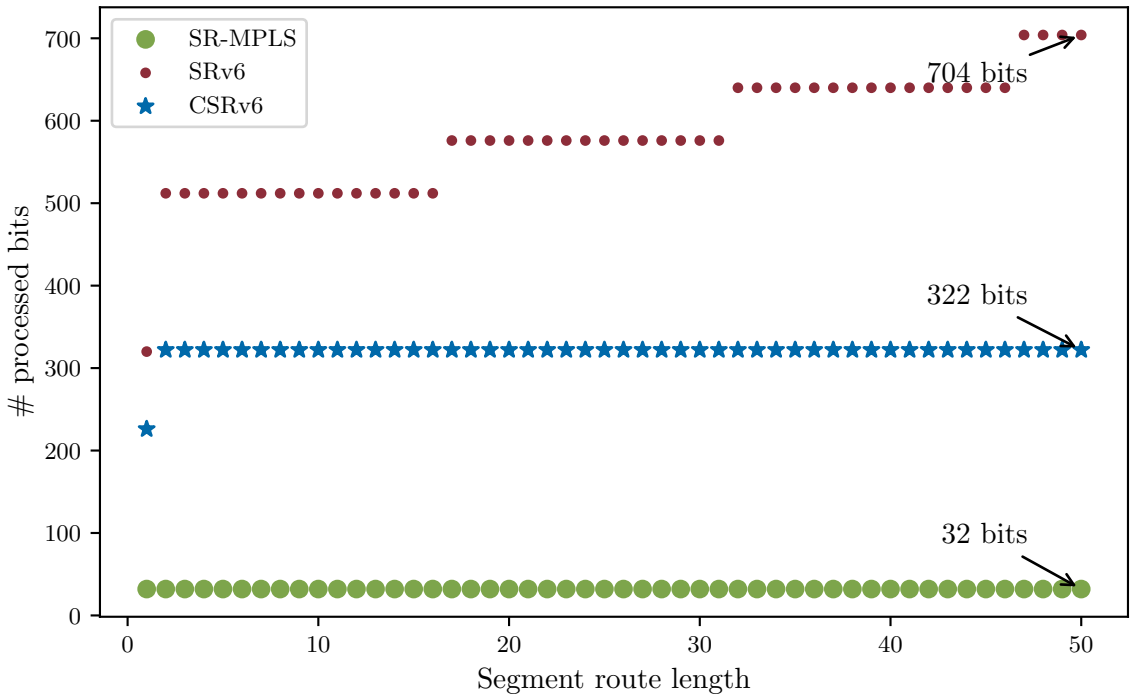



Figure 6.4: Comparison of the number of processed bits by a SR router for the normalized SR-MPLS, SRv6 and CSRv6 standards for different segment route lengths.

further research, as it could be suitable for deployments capable of IPv6 dataplane processing or that would benefit from IPv6-native tooling. However, due to the strict per-packet memory and processing constraints of LEO satellites, SR-MPLS is chosen for this implementation. Therefore, the remainder of this chapter describes the concrete Int2Sat implementation using SR-MPLS as the segment routing encoding.

## 6.3 Implementation: Int2Sat Package

This section describes the implementation of the Int2Sat package and covers the data models for static and dynamic routing information. It also explains the generation and scheduling of gateway update objects and how the gateway logic functions apply those updates to the switches in the Int2Sat routing domain. The last section details the implementation of the switches' P4-based packet processors.

### 6.3.1 Routing Information

The Int2Sat package distinguishes between two types of routing information, static and dynamic. Both types are essential for configuring and operating the emulated NTN. The following covers the implementation details for each type of routing information respectively.

#### 6.3.1.1 Static

Static routing information comprises all configuration elements required for routing and forwarding that remain constant throughout an emulation run. These elements include all layer 2 and layer 3 addressing information used for packet delivery within the NTN and across the Internet, namely MAC addresses, IP addresses, and segment identifiers (SIDs).

This static information establishes the foundational addressing and forwarding context for the P4-based packet processors. During system initialization, each network interface of every emulated node is assigned a MAC and IP address. MAC addresses are drawn from a predefined address pool, while IP addresses are allocated from dedicated IP-prefix blocks. The distribution of IP prefixes within the NTN follows a structured pattern: each emulated gateway (GW) receives its own prefix, and all user terminals (UTs) associated with a given home gateway (HGW) are assigned IP addresses from that gateway's prefix. In the real-world Int2Sat architecture, these prefixes would be announced via BGP to connected autonomous systems, providing global reachability for GWs and their assigned UTs [5]. In the emulation environment, this behavior is reproduced by configuring each server with the appropriate IP routing and ARP entries, ensuring correct end-to-end layer 2 and layer 3 connectivity. Similarly, each server receives its own prefix, and the forwarding tables of the GWs are provisioned accordingly to guarantee correct forwarding between gateways and servers.

Within the Int2Sat domain, segment routing is used to steer packets between gateways and user terminals across the satellite network. As discussed previously, this thesis adopts the SR-MPLS standard for encoding segment-routing information. SR-MPLS constructs a segment routing header (SRH) from one or more MPLS headers, each 32 bits in size and containing a 20-bit label field [43]. The Int2Sat package uses this label field to encode SIDs and stacks multiple MPLS headers to form a complete source route between the access satellites (AcSs) of user terminals and their associated gateways. The resulting sequence of SIDs specifies the individual NTN nodes and links a packet must traverse to reach its destination.

To support this mechanism, each gateway and satellite is assigned a global node SID, ensuring uniform addressing across the entire SR domain. User terminals, however, do not receive global node SIDs due to their mobility and the resulting variability in their serving access satellites. Instead, each access satellite maintains a set of local adjacency SIDs, one for each of its service links (USLs). Traffic destined for a UT is therefore guided through the SR domain using global node SIDs up to the current AcS, which then uses a local adjacency SID to forward the packet over the correct USL.

All global node SIDs and local adjacency SIDs are encoded within the 20-bit MPLS label field. The Int2Sat package reserves disjoint label ranges for global node SIDs, adjacency SIDs, and special-purpose SIDs (e.g., MPLS reserved labels defined in RFC 3032 [43]). A dedicated label allocator assigns concrete labels during initialization and publishes these assignments as part of the static routing information.

Together, the complete set of static routing information i.e., SIDs, MAC addresses, IP addresses, and IP prefixes, forms the baseline configuration on top of which all

dynamic routing information is determined. The next section covers the types of the dynamic routing information and the mechanisms responsible for maintaining and updating this dynamic routing state.

#### 6.3.1.2 Dynamic

Dynamic routing information is determined by the routing state in the NTN that evolves over time due to the topology dynamics inherent in LEO constellations. A single snapshot of the current routing state includes the configured source routes and forwarding information on all satellites and gateways within the Int2Sat SR domain. The source route state and the forwarding state change in correlation with the rotation of the satellite constellation around Earth. This imposes predictable dynamics for the links and nodes in the NTN.

Predictable dynamics include for example, changes in the source routes between UTs and GWs, originating from the periodic reselection of new AcSs of both GWs and UTs. Similarly to the soure route state, the forwarding state in the Int2Sat SR domain is also prone to prone to predictable dynamics, as forwarding entries must be changed in accordance with the current availability of the NTN links. The Int2Sat control plane can predict changes in the source route and forwarding state, as it knows the entire constellation and models, which allow calculating changes in advance.

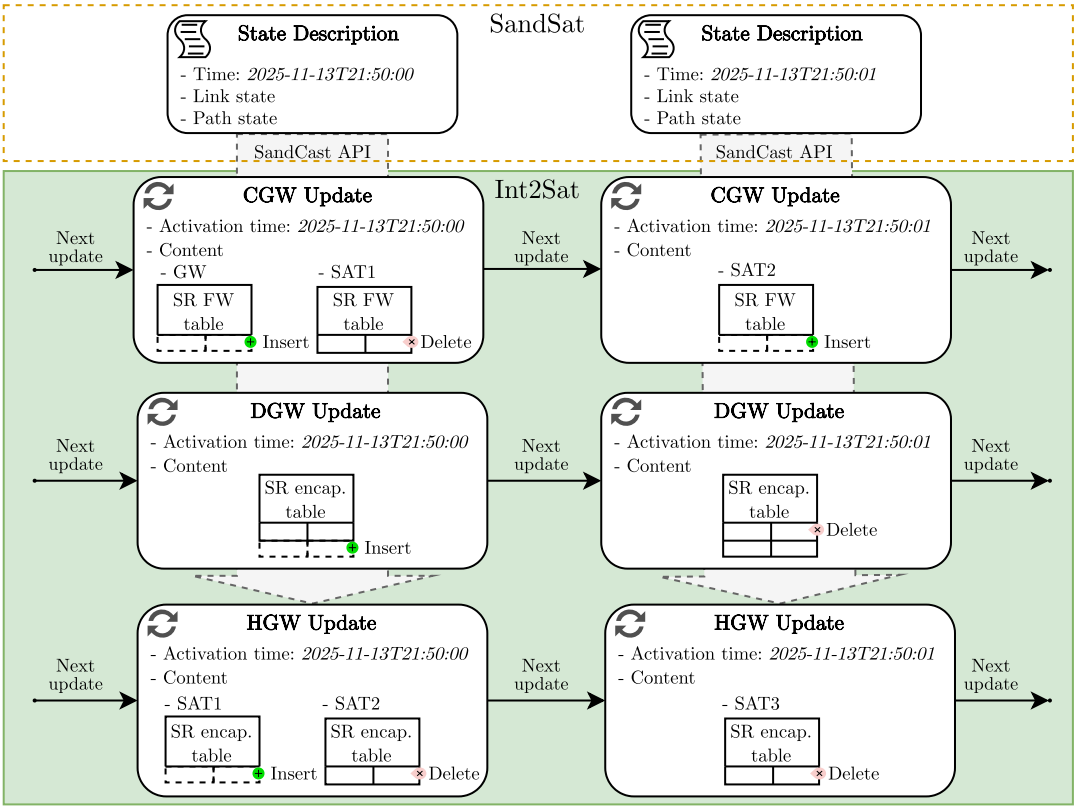



Figure 6.5: Process for deriving the update objects for each type of gateway (CGW, DGW, HGW) from the evolving dynamic routing information.

In the Int2Sat package, the evolving state information is provided by the SandSat library and its components, as shown in Figure 6.5. The SandSat constellation simulator is capable to simulate the evaluated satellite constellation. It derives the link availability for all NTN links from the orbital movement and visibility of the satellites in the constellation. The link state is internally used by the PCE component to calculate the path state between all NTN nodes. The link and path state information from the SandSat components are retrieved by the Int2Sat package via the SandCast API. The information is returned as a sequence of time-stepped description objects. Each description object contains the current link state and path state, derived from the simulated orbital positions and visibility relationships of the satellites.

The link and path states of each description are combined in the Int2Sat package with the static routing information to produce the CGW, HGW and DGW updates. Each update is associated with an activation time and contains concrete modification instructions for the SR-MPLS tables on the relevant nodes. The CGW is responsible for updating segment-routing forwarding tables (SR FW table) on gateways and satellites, while the HGW and DGW update the the SR encapsulation tables (SR encap. table) used at the ingress and egress of the Int2Sat domain. These updates may involve inserting new entries, deleting expired ones, or adjusting entries to reflect changes in adjacency or next-hop assignments.

The diagram in Figure 6.5 shows the stepwise, time-aligned nature of these updates. For each new constellation state description, corresponding CGW, DGW, and HGW updates are derived and scheduled. All updates of a specific gateway function are combined into a single-linked list where each update points to the next one. This ensures that all satellites and gateways maintain a forwarding and encapsulation state that is consistent with the current topology of the NTN. Consequently, the Int2Sat implementation can track the evolving satellite network with high temporal fidelity, enabling correct packet steering even under rapid and continuous topological change.

### 6.3.2 Gateway Logic Functions

The Int2Sat package registers an individual logic function for each type of gateway, i.e. CGW, DGW, and HGW at the emulated gateways in the NTNE. The general structure for each logic function is shown in Figure 6.6.

The updates derived from the dynamic and static routing information are applied by their specific logical gateway function. As illustrated in Figure 6.6, each gateway role in the Int2Sat architecture, i.e., CGW, HGW, and DGW, implements its logic function following a common structural pattern. Each function operates as a time-driven control loop that reacts to the evolving constellation state provided by the NTNE. First, the logic function queries the NTNE timer to obtain the current emulation time and determines whether a new update must be activated. If an update is pending, the function retrieves the corresponding gateway or satellite node from the NTNE graph and its switch-control interface, which is exposed through the NEW::SwitchControlApi.

After obtaining the correct node and control interface, the logic function applies

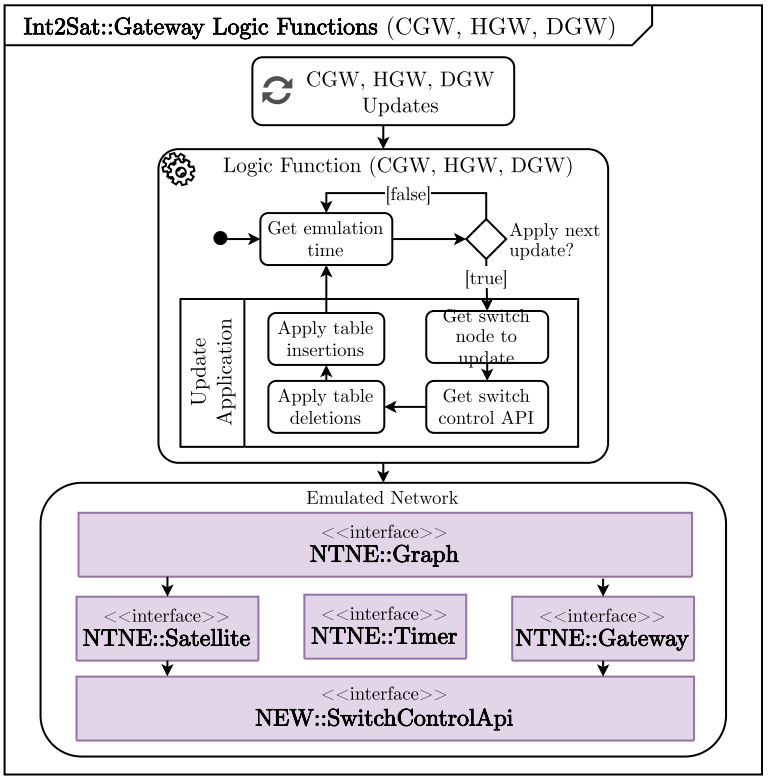



Figure 6.6: Common implementation structure for the logic functions representing Int2Sat's CGWs, HGWs, and DGWs

the update in two phases. First, obsolete forwarding or encapsulation rules are removed from the switch tables to prevent stale states from persisting across topology transitions. In the second phase, new table entries corresponding to the updated source routes or forwarding paths are inserted. This separation of deletion and insertion avoids transient inconsistencies in the data plane and mirrors the behavior of real-world SDN control plane systems.

Although all gateway logic functions share this structural workflow, they differ in the type of routing state they manipulate. The HGW maintains and updates the SR-MPLS encapsulation rules used for source routes toward UTs. The CGW installs forwarding entries on satellites and gateways based on the latest path computations. Finally, the DGW updates the encapsulation rules for the source routes on the AcSs of UTs toward the DGW itself. In this way, each gateway type contributes a distinct portion of the overall routing architecture while relying on the same standardized update mechanism.

The gateway logic functions update the data plane incrementally, ensuring that only the affected source routes and forwarding entries are modified after a topology change. This design minimizes processing overhead on the constellation and makes the architecture compatible with the strict resource limitations of LEO satellites.

### 6.3.3 Packet Processors

The packet processors used in the Int2Sat implementation are based on the P4 v1model architecture, which is implemented on the BMv2 software switches. As shown in Figure 6.7, the v1model provides a standard programmable pipeline consisting of a parser, checksum verifier, ingress control block, deparser, and checksum calculator. The Int2Sat packet processors map their forwarding logic onto these building blocks.

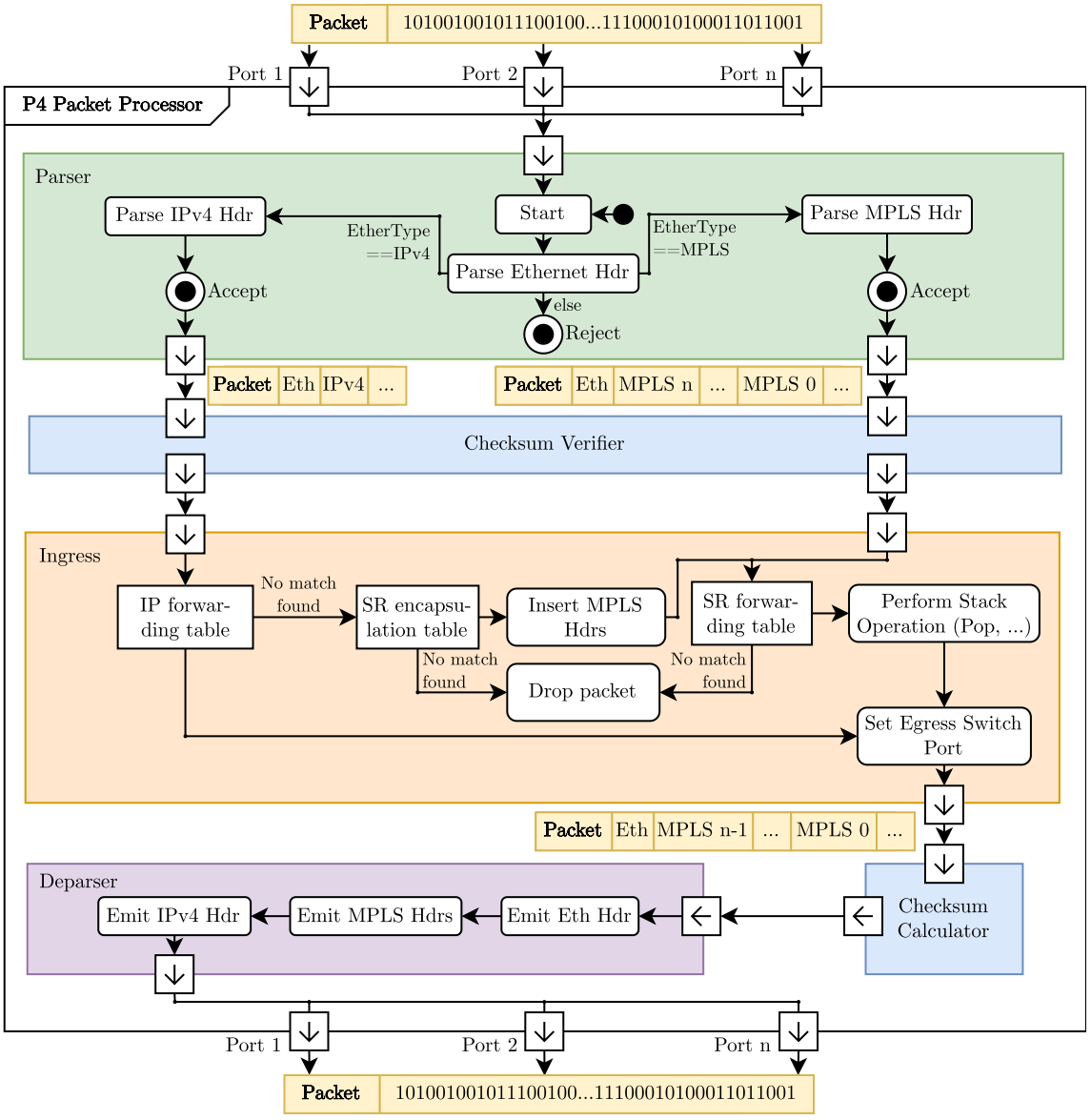



Figure 6.7: Common flow diagram for the satellite and gateway P4-based packet processors, implementing the data plane for the Int2Sat routing architecture. Building blocks of the underlying v1model architecture are highlighted in different colors. The egress match-action pipeline is omitted as it is not used.

The overall packet-processing pipeline consists of header parsing, checksum ver-

ification, ingress-table lookups for IP forwarding and SR encapsulation, and SR forwarding. This is followed by header reconstruction in the deparser and recalculation of the checksum prior to transmission of the packet on its determined egress port.

To reduce processing overhead, particularly on resource-constrained LEO satellites, the parser is implemented as efficiently as possible. It only extracts the headers that are strictly required for forwarding decisions. Ethernet headers are always parsed, followed by an IPv4 header if the EtherType field indicates IPv4, or by the single topmost MPLS header if the packet carries SR-MPLS traffic. No deeper MPLS stack is parsed during ingress, reducing memory accesses and lowering latency.

Although satellites and gateways use the same underlying P4 program, they differ in terms of how the packet processor leverage their tables. Satellite switches use the IP forwarding table to forward packets between multiple connected UTs of the same access satellite. Gateway switches, on the other hand, populate the IP forwarding table with routes that lead to Internet nodes, such as servers. The SR encapsulation table is also role-specific. Satellite switches configure this table with a default rule that forwards all packets toward the DGW, enabling satellites to primarily operate as transit nodes with minimal control logic. Additional entries may be configured if traffic engineering behaviour is required, such as load balancing or alternative failover paths. In contrast, gateway switches use the SR encapsulation table to install the source routes generated by the HGW for the user terminals associated with that gateway. These entries describe the exact MPLS label stacks that are pushed onto packets before they are injected into the satellite constellation. The forwarding table is used uniformly across satellites and gateways to implement MPLS-based segment forwarding. The table performs the necessary label-stack operations, such as popping the active label or continuing with the next label, and determines the packet's outgoing port. Packets that do not match a forwarding entry are dropped to prevent inconsistencies and forwarding loops.

# 7 Routing Architecture Evaluation

Following the implementation of the Int2Sat routing architecture in the previous chapter, this chapter evaluates its functional behavior and its integration with the NTNE. The evaluation has two goals: first, to verify that the implemented routing architecture operates correctly under dynamic LEO conditions, and second, to assess the effectiveness of the gateway logic functions and the applicability of the NTNE for scenarios of different scales. The following sections describe the evaluation methodology and present the results obtained from two representative emulation scenarios based on the Kuiper and Starlink constellations. The chapter concludes with a summary of all findings.

## 7.1 Methodology

The evaluation focuses on three central objectives. First, the functional correctness of the routing architecture shall be demonstrated. This includes showing that end-to-end communication between a UT and a server connected to a GW remains possible despite the dynamic topology of the satellite constellation. This demonstrates the functional feasibility of Int2Sat's basic Internet integration. Second, insights into the internal effectiveness of the implementation shall be provided, with particular focus on the control plane operations of the CGW, DGW, and HGW logic functions. These insights help quantify the computational load induced by routing updates and reveal how the implementation behaves during frequent topology changes. Third, the applicability of the NTNE for the functional evaluation of custom routing architectures and its scalability shall be demonstrated. To this end, the evaluation considers two emulation scenarios of different sizes and satellite densities to show that the emulator can handle varying network scales.

Two evaluation scenarios are considered, both based on prominent satellite constellations: one from Amazon's Kuiper project and the other from SpaceX's Starlink project. In the Kuiper scenario, an NTN with a UT placed in Tübingen and a GW including a connected server placed in Berlin are leveraged. The emulated NTN's constellation corresponds to Amazon's Kuiper system, consisting of 1,156 satellites at an altitude of approximately 630 km and an orbital inclination of 52°. The total emulation time is 10 minutes. Using the region-of-interest concept, no more than 20 satellites appear in any given scene, enabling a medium-scale evaluation of the routing architecture.

The Starlink scenario uses the same ground-node locations but employs a larger constellation: a Starlink shell of 1584 satellites at an altitude of approximately 550 km and an orbital inclination of 53°. The emulation time is again 10 minutes. Due to the region-of-interest limitation, a maximum of 55 satellites become active during the evaluation, providing a large-scale scenario for assessing scalability.

Two metrics are used to evaluate functional correctness, scalability and the internal control plane behavior. The first metric is the round-trip time (RTT) between the UT and the server connected to the GW. This metric directly reflects the underlying dynamic path length and thus provides evidence for correct data plane operation under topology changes. It is also used to evaluate the scalability of the NTNE by comparing RTT behavior across the two scenarios.

The second metric captures the operational activity of the gateway logic functions. It includes the number of forwarding-table deletions and insertions performed by the CGW, as well as the number of encapsulation-table operations performed by the HGW and DGW. These operations quantify the workload induced on the control plane and demonstrate the effectiveness with which the implementation handles routing updates as satellite density increases.

Both scenarios are executed on a single host equipped with an Intel© Xeon© CPU E5-2683 v4 with 16 cores and 32 GB of RAM. To evaluate the NTNE performance, the gateway and satellite switches within the emulator are configured to leverage the BMv2 software switch in its release variant. This excludes any logging or debugging functionalities on the switches, enabling the assessment of the maximum achievable NTNE performance on the given host.

## 7.2 Results

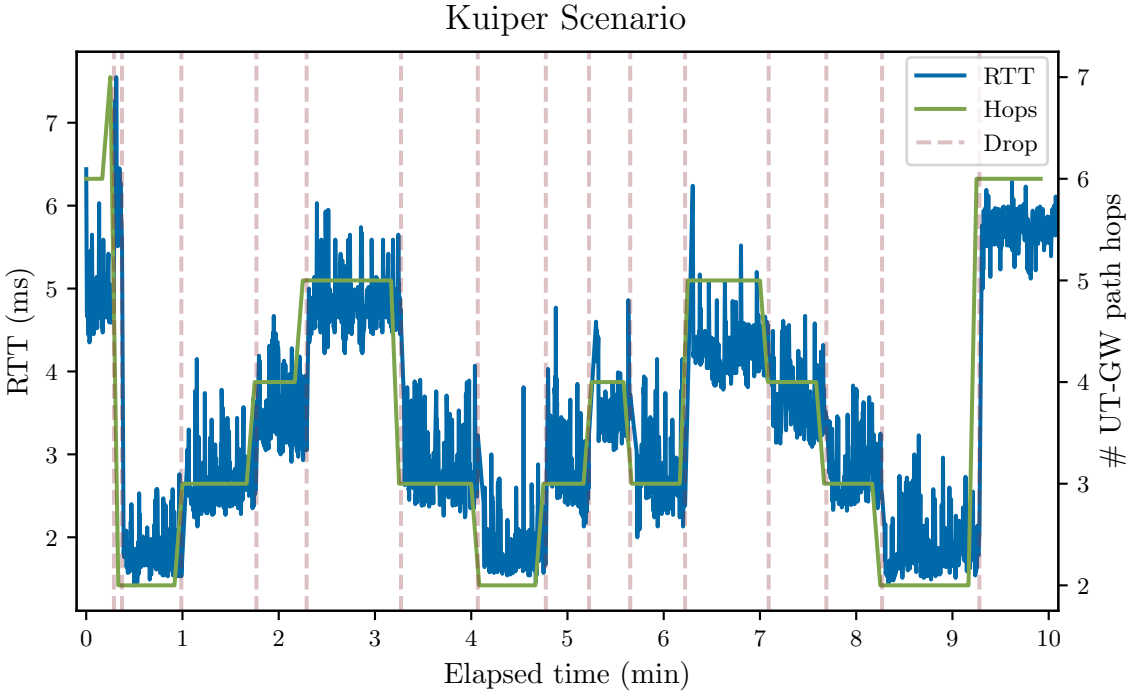



Figure 7.1: Measured RTT between the UT and the server connected to the GW (blue) and the corresponding hop count of the UT-GW path (green) over time in the Kuiper scenario. The start points for dropped packet intervals are marked in red.

Figures 7.1–7.2 show the measured RTT between the UT and the server, as well as the corresponding UT-GW hop count, for the Kuiper and Starlink scenarios, respectively. In both scenarios, changes in the constellation topology are reflected by rising or falling RTT values. These fluctuations closely correlate with the number of hops along the UT-GW path: increases in path length lead to increased RTT, and vice versa. Each topology change triggers a brief interval of packet drops, marked in red in the figures. Over the full emulation period, the Kuiper scenario experiences a packet loss of 5.28 % (2853/3012 received/sent packets), while the Starlink scenario shows 9.30 % (2742/3023 received/sent packets). These losses occur exclusively during topology transitions, confirming that packets are forwarded correctly during all steady-state periods.

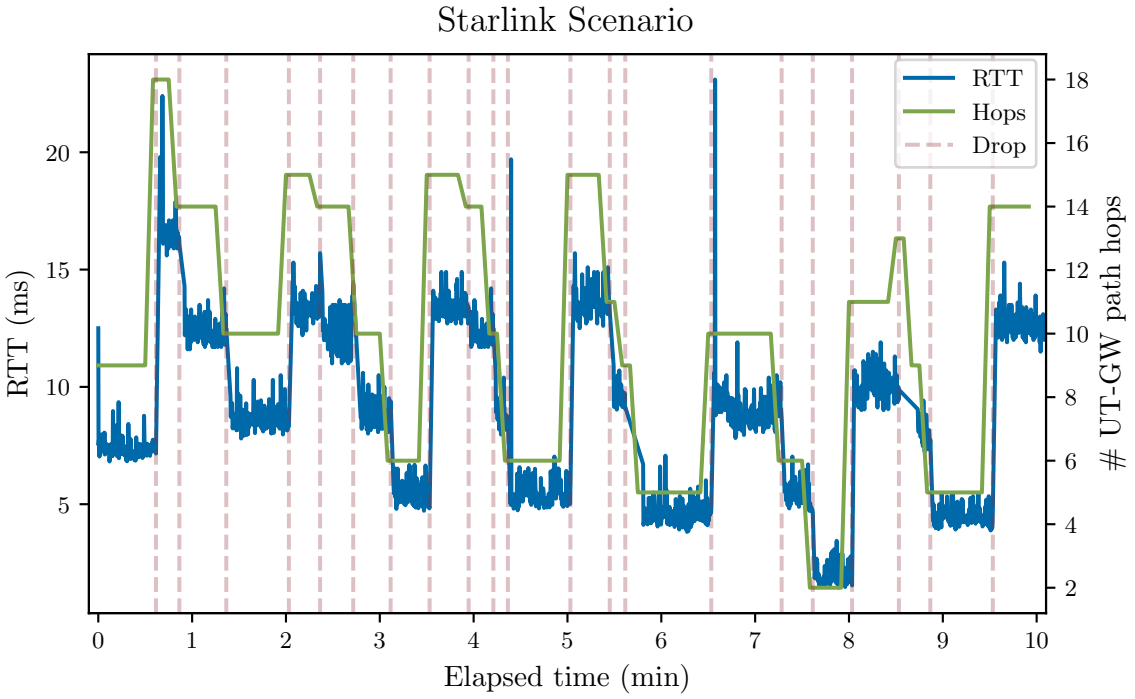



Figure 7.2: Measured RTT between the UT and the server connected to the GW (blue) and the corresponding hop count of the UT-GW path (green) over time in the Starlink scenario. The start points for dropped packet intervals are marked in red.

The measured RTT statistics further support the functional correctness of the Int2Sat implementation. In the Kuiper scenario, the average RTT is 2.965 ms, while the minimum and maximum RTTs are 1.402 ms and 7.551 ms, respectively. In the Starlink scenario, the corresponding values for the minimum, average, and maximum RTT are 1.438 ms, 8.582 ms, and 23.121 ms, respectively. Although these values illustrate the expected behavior of the routing architecture, they do not reflect real-world satellite RTT performance, as the current NTNE implementation does not emulate physical link characteristics such as delay, jitter, or bandwidth limitations. The measured RTTs therefore represent the maximum achievable performance of the emulator on the given host.

A comparison between both scenarios shows similar minimal RTTs of approximately 1.4 ms, but noticeably higher average and maximum RTTs in the Starlink scenario. The RTT standard deviation of 4.023 ms in the Starlink scenario also exceeds the 1.511 ms observed in the Kuiper scenario. These differences align with expectations, as the Starlink scenario involves a larger number of active satellites and thus experiences more frequent routing and path-length changes. Despite this increased dynamism, the figures demonstrate that the paths generated by the SandSat library remain relatively stable, with maximum stable-path durations of approximately 60 seconds in the Kuiper scenario and 30 seconds in the Starlink scenario.

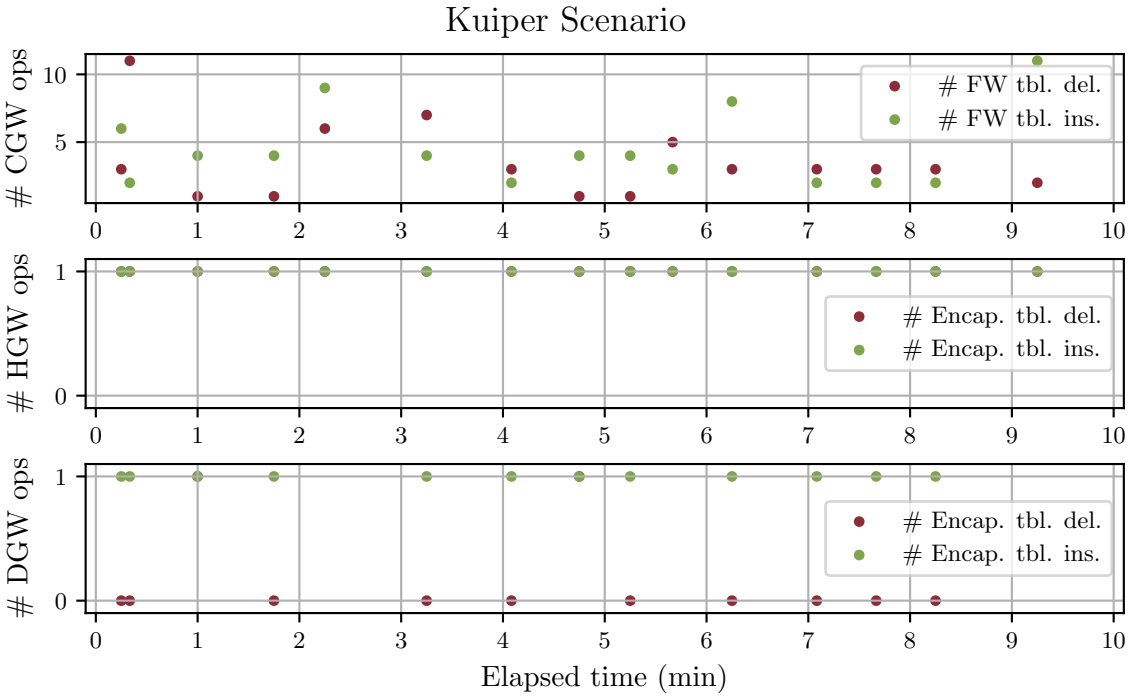



Figure 7.3: Operational activity of the CGW, HGW, and DGW control plane functions over time in the Kuiper scenario. It shows the number of forwarding-table deletions and insertions for the CGW and the number of encapsulation-table updates for the HGW and DGW.

The operational activity of the CGW, HGW, and DGW logic functions is shown in Figures 7.3–7.4. In both scenarios, the CGW performs the majority of control plane work, reflecting its responsibility for maintaining the forwarding state of all satellites and gateways. The figures demonstrate that each topology change imposed by the constellation triggers a burst of CGW operations, consisting of coordinated deletion and insertion events in the SR forwarding tables. In the Kuiper scenario, these bursts occur relatively infrequently and involve a limited number of table updates due to the smaller number of active satellites. In contrast, the Starlink scenario exhibits a noticeably denser and more irregular pattern of CGW activity, with a significantly higher number of operations per update interval. This increase corresponds to the larger number of satellites participating in the region of interest, requiring the CGW to update more forwarding rules whenever path changes occur.

The HGW and DGW traces in both figures remain comparatively sparse. Since both gateways only manage the encapsulation rules associated with the single emulated UT, their operational activity is limited to at most one deletion and one insertion at a time. This behavior is visible in both plots as isolated, low-frequency update points. This confirms that the workload of the HGW and DGW scales with the number of user terminals rather than with the constellation size. The consistency of their traces in the two scenarios also shows that the SR-encapsulation logic remains stable and is unaffected by increasing satellite density.

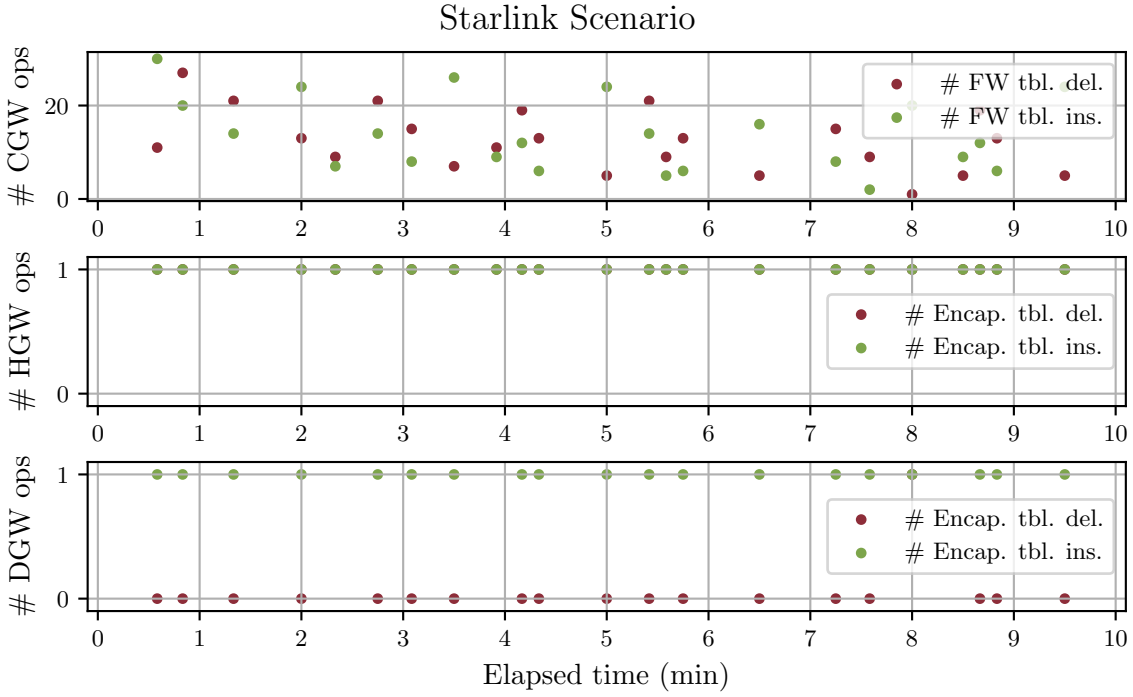



Figure 7.4: Operational activity of the CGW, HGW, and DGW control plane functions over time in the Starlink scenario. It shows the number of forwarding-table deletions and insertions for the CGW and the number of encapsulation-table updates for the HGW and DGW.

## 7.3 Summary

The evaluation demonstrates that the implemented Int2Sat routing architecture operates correctly and efficiently under realistic LEO satellite dynamics. The RTT measurements obtained in both the Kuiper and Starlink scenarios confirm that end-to-end communication between user terminals and gateways remains functional throughout continuously changing topologies. Observed RTT variations directly correlate with the evolving UT-GW hop count, validating the correctness of the installed source routes and forwarding entries. Packet losses occur exclusively during topology transitions, which is consistent with the expected behavior of a system undergoing frequent link and node changes. Countermeasures to reduce the packet loss during these predictable topology transitions are subject to further research.

The analysis of the gateway control plane activity further highlights the implementation's internal effectiveness. The CGW exhibits an increased operational load in larger constellations, reflecting its central role in installing and updating forwarding entries across the network. Distributing CGW functionality across multiple GWs, may be a suitable solution for higher CGW loads in larger constellations. In contrast, the activity of the HGW and DGW remains minimal and largely constant, because their workload scales with the number of user terminals rather than the constellation size. This behavior confirms that the gateway logic functions are implemented efficiently and adhere to the architectural design principles established in earlier chapters.

Finally, the evaluation demonstrates the applicability and scalability of the NTNE. By limiting the emulated topology to a region of interest, the emulator can handle both medium- (Kuiper) and large- (Starlink) scale constellation scenarios while maintaining stable performance throughout the experiments. These results validate the NTNE as a flexible and suitable platform for evaluating routing architectures under diverse non-terrestrial network conditions.

Overall, the results confirm that the Int2Sat implementation and its basic Internet integration is functionally correct, computationally efficient in its control plane design, and fully compatible with the NTNE as an evaluation environment. These findings provide a solid foundation for further experimentation, optimization, and extension of routing mechanisms for LEO-based non-terrestrial networks.

# 8 Discussion

The previous chapter evaluated the Int2Sat implementation on the NTNE, demonstrating its functional correctness, operational efficiency, and applicability across medium- and large-scale LEO satellite scenarios. While the results confirm that the routing architecture and emulator form a robust foundation for research on Internet-NTN integration, they also reveal opportunities for further development and refinement. This chapter discusses these future directions for both the Int2Sat implementation and the NTNE itself.

## 8.1 Future Work: Int2Sat

Although the implemented Int2Sat mechanisms demonstrate correct behaviour under dynamic constellation conditions, the following aspects of Int2Sat present valuable opportunities for extension.

A first extension in future work could be the implementation and evaluation of Int2Sat's protection switching mechanisms. The current implementation reacts to predictable handover events driven by LEO mobility, but it does not yet provide explicit mechanisms for fast recovery from unexpected link or node failures. Evaluating these mechanisms under controlled failure injections in the NTNE would provide insights into packet loss reduction and control plane responsiveness.

Another promising direction is the implementation and evaluation of Int2Sat's traffic engineering (TE) use cases. These include latency-aware path selection, traffic distribution across multiple disjoint routes, and congestion-avoidance mechanisms. Evaluation outcomes could include the functional feasability for each TE mechanism, the cost in control plane resources and responsiveness, and the stability and fairness under realistic dynamic conditions. These results will guide which TE features are practical for operational Int2Sat deployments and how the NTNE should be extended for large-scale TE experimentation.

## 8.2 Future Work: NTNE

The NTNE successfully enabled the evaluation of Int2Sat within realistic constellation scenarios, yet the system architecture offers several natural extension points. The future work items and how they integrate into the current emulator architecture is illustrated in Figure 8.1.

One extension is the introduction of a graphical user interface and a remote controller. These components would simplify experiment configuration, provide visualization of constellation dynamics, and support remote orchestration of emulation runs. This extension requires an API called the NTNE protocol (NTNEP) and a

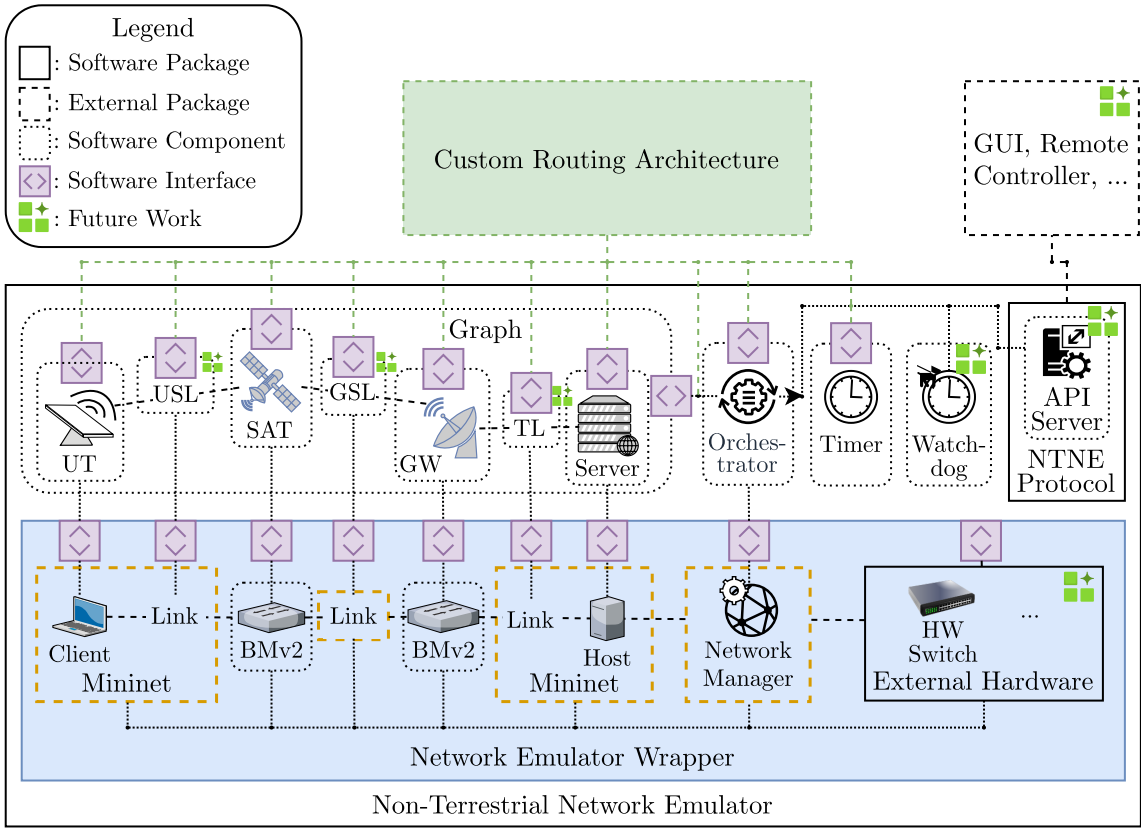



Figure 8.1: Future work for the NTNE.

server that implements the API and interacts with the emulated network of the NTNE.

The figure further identifies the extension of the NTNE with a watchdog mechanism. This component would monitor the health of all emulated nodes, links, and processes, automatically detecting failures or stalled components. Integrating a watchdog would significantly increase the robustness of long-running experiments.

Another extension is the integration with external programmable hardware switches and other external hardware. Hardware-in-the-loop capability enables hybrid experiments and improves the relevance of performance measurements for real-world deployments.

The last extension of the NTNE concerns the realistic emulation of NTN link characteristics. The current implementation models topology dynamics and connectivity state but does not yet emulate physical layer effects such as propagation delay, jitter, bandwidth limitations, or packet error rates. As a result, measured RTTs primarily reflect processing and routing behavior rather than real satellite communication performance. Incorporating configurable link-delay models, based on geometric propagation distances, modulation and coding schemes, weather effects, or queueing behavior, would enable significantly more realistic end-to-end performance evaluations.

# 9 Conclusion

The primary objectives of this thesis were the implementation and evaluation of the Int2Sat routing architecture. This work provided a complete reference implementation of Int2Sat's data and control plane. This implementation confirms the feasability of Int2Sat's basic integration of non-terrestrial networks leveraging LEO constellations into the Internet. The data plane was realized using SR-MPLS as the underlying segment-routing encoding and was implemented on P4-based packet processors tailored for satellite and gateway functionality. The control plane was developed as a modular Python library that manages dynamic routing state and configures the programmable data plane accordingly. To support the systematic evaluation of this implementation, this thesis introduced the non-terrestrial network emulator (NTNE), a time-dynamic emulator. The NTNE enables modeling of NTN topologies, hereby being capable of reproducing LEO scenarios in a controlled environment while allowing the integration of custom routing architectures.

The evaluation results confirm that Int2Sat operates correctly and efficiently under realistic constellation dynamics. End-to-end connectivity between user terminals and gateways was maintained despite frequent topology changes, validating the correctness of the implemented routing behavior. Furthermore, the results demonstrate that the modular control plane design is computationally lightweight and well suited to LEO satellite constraints. The NTNE successfully supported emulation scenarios of varying size and complexity, illustrating both its scalability and its suitability as a research platform for future investigations in non-terrestrial network routing.

## 9.1 Future Research

Beyond the concrete implementation extensions outlined in Chapter 8, several broader research topics emerge from this thesis. The most relevant directions are outlined in the following subsections.

### 9.1.1 Predictive Control Plane Mechanisms

A promising research direction is the development of control plane mechanisms that more effectively exploit the predictable dynamics of LEO constellations. Since satellite positions, link states, and handover events can be forecast accurately, future work could investigate proactive update strategies, prediction-guided scheduling, and techniques to reduce control plane signalling overhead. Such mechanisms may improve responsiveness, minimise packet loss during transitions, and further reduce the computational burden on constrained satellite nodes.

### 9.1.2 Evaluation of Alternative Segment-Routing Encodings

A second research direction concerns a comparative assessment of the SR-MPLS encoding used in this thesis and alternative segment-routing approaches, particularly CSRv6. While SR-MPLS was selected based on its compact header structure and favourable processing characteristics, further research should examine additional criteria such as extensibility, alignment with IPv6 infrastructures, and runtime efficiency under realistic LEO constellation conditions. Implementing and benchmarking CSRv6 and related encodings within the NTNE would provide empirical insights into their suitability for future non-terrestrial routing architectures.

# Bibliography


[1] J. Krause, *Technologies Deep Dive - Non-Terrestrial Networks*, Accessed: 18.10.2025, Sep. 2024. [Online]. Available: https://www.3gpp.org/technologies/ntn-overview.

[2] N. Chuberre and S. Chenumolu, *TR 38.811: Study on New Radio (NR) to support non-terrestrial networks*, Jun. 2018.

[3] S. Spengler, *The role of geostationary satellite networks in meeting the rural connectivity challenge*, Accessed: 18.10.2025, May 2024. [Online]. Available: https://www.broadbandcommission.org/insight/the-role-of-geostationary-satellite-networks-in-meeting-the-rural-connectivity-challenge/.

[4] European Space Agency, *Low Earth Orbit*, Accessed: 19.10.2025, Mar. 2020. [Online]. Available: https://www.esa.int/ESA_Multimedia/Images/2020/03/Low_Earth_orbit.

[5] M. Menth and M. Flüchter, "Int2Sat: A Routing Architecture with Internet Integration for Satellite Constellations with Inter-Satellite Links", Dec. 2025.

[6] M. Braun, "Graph-Based Simulation and Evaluation of Resilient Inter-Satellite Paths in LEO Mega-Constellations", M.S. thesis, Dec. 2025.

[7] European Space Agency, *Types of orbits*, Accessed: 28.11.2025, Mar. 2020. [Online]. Available: https://www.esa.int/Enabling_Support/Space_Transportation/Types_of_orbits.

[8] O. Kodheli, E. Lagunas, N. Maturo, *et al.*, "Satellite communications in the new space era: A survey and future challenges", *IEEE Communications Surveys & Tutorials*, vol. 23, no. 1, pp. 70–109, 2021. DOI: 10.1109/COMST.2020.3028247.

[9] F. Rinaldi, H.-L. Maattanen, J. Torsner, *et al.*, "Non-terrestrial networks in 5G & beyond: A survey", *IEEE Access*, vol. 8, pp. 165 178–165 200, 2020. DOI: 10.1109/ACCESS.2020.3022981.

[10] G. Geraci, D. López-Pérez, M. Benzaghta, and S. Chatzinotas, "Integrating terrestrial and non-terrestrial networks: 3D opportunities and challenges", *IEEE Communications Magazine*, vol. 61, no. 4, pp. 42–48, 2023. DOI: 10.1109/MCOM.002.2200366.

[11] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, *Segment Routing Architecture*, RFC 8402, Jul. 2018. DOI: 10.17487/RFC8402. [Online]. Available: https://www.rfc-editor.org/info/rfc8402.

[12] A. Bashandy, C. Filsfils, S. Previdi, B. Decraene, S. Litkowski, and R. Shakir, *Segment Routing with the MPLS Data Plane*, RFC 8660, Dec. 2019. DOI: 10.17487/RFC8660. [Online]. Available: https://www.rfc-editor.org/info/rfc8660.

[13] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima, and D. Voyer, *IPv6 Segment Routing Header (SRH)*, RFC 8754, Mar. 2020. DOI: 10.17487/RFC8754. [Online]. Available: https://www.rfc-editor.org/info/rfc8754.

[14] W. Cheng, C. Filsfils, Z. Li, B. Decraene, and F. Clad, *Compressed SRv6 Segment List Encoding*, RFC 9800, Jun. 2025. DOI: 10.17487/RFC9800. [Online]. Available: https://www.rfc-editor.org/info/rfc9800.

[15] F. Hauser, M. Häberle, D. Merling, *et al.*, "A survey on data plane programming with P4: Fundamentals, advances, and applied research", *Journal of Network and Computer Applications*, vol. 212, p. 103 561, Mar. 2023, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2022.103561. [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2022.103561.

[16] *White Paper: Software-Defined Networking - The New Form for Networks*, Accessed: 27.11.2025, Apr. 2012. [Online]. Available: https://opennetworking.org/wp-content/uploads/2011/09/wp-sdn-newnorm.pdf.

[17] The P4 Language Consortium, *P4-16 Language Specification (version 1.2.5)*, Oct. 2024. [Online]. Available: https://p4.org/wp-content/uploads/sites/53/2024/10/P4-16-spec-v1.2.5.html.

[18] *P4Lang: v1model*, Accessed: 16.11.2025. [Online]. Available: https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4.

[19] *P4Lang: Behavioral Model Version 2 (BMv2) Repository*, Accessed: 16.11.2025. [Online]. Available: https://github.com/p4lang/behavioral-model.

[20] The P4.org API Working Group, *P4-16 Language Specification (version 1.4.1)*, Oct. 2024. [Online]. Available: https://p4.org/wp-content/uploads/sites/53/2024/10/P4Runtime-Spec-v1.4.1.html.

[21] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks", in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX, Monterey, California: Association for Computing Machinery, 2010, ISBN: 9781450304092. DOI: 10.1145/1868447.1868466. [Online]. Available: https://doi.org/10.1145/1868447.1868466.

[22] G. Di Lena, A. Tomassilli, D. Saucez, F. Giroire, T. Turletti, and C. Lac, "Distrinet: A mininet implementation for the cloud", *SIGCOMM Comput. Commun. Rev.*, vol. 51, no. 1, pp. 2–9, Mar. 2021, ISSN: 0146-4833. DOI: 10.1145/3457175.3457177. [Online]. Available: https://doi.org/10.1145/3457175.3457177.

[23] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim, "CORE: A real-time network emulator", in *MILCOM 2008 - 2008 IEEE Military Communications Conference*, 2008, pp. 1–7. DOI: 10.1109/MILCOM.2008.4753614.

[24] A. Stockmayer, C. Kindermann, and M. Menth, "VITO: Virtual testbed orchestration for automation of networking experiments", in *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, ser. VALUETOOLS 2017, Venice, Italy: Association for Computing Machinery, 2017, pp. 113–118, ISBN: 9781450363464. DOI: 10.1145/3150928.3150954. [Online]. Available: https://doi.org/10.1145/3150928.3150954.

[25] F. Windisch, K. Abedi, T. Doan, T. Strufe, and G. T. Nguyen, "Hybrid testbed for security research in software-defined networks", in *2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2023, pp. 147–152. DOI: 10.1109/NFV-SDN59219.2023.10329614.

[26] M. Kerrisk, *Linux man-pages 6.15: cgroups - Linux control groups*, Accessed: 04.11.2025, May 2025. [Online]. Available: https://man7.org/linux/man-pages/man7/cgroups.7.html.

[27] M. Kerrisk, *Linux man-pages 6.15: network_ namespaces - Overview of Linux network namespaces*, Accessed: 04.11.2025, May 2025. [Online]. Available: https://man7.org/linux/man-pages/man7/network_namespaces.7.html.

[28] M. Kerrisk, *Linux man-pages 6.15: veth - Virtual Ethernet Device*, Accessed: 04.11.2025, May 2025. [Online]. Available: https://man7.org/linux/man-pages/man4/veth.4.html.

[29] M. Flinders and I. Smalley, *What are Linux containers?*, Accessed: 04.11.2025, Nov. 2025. [Online]. Available: https://www.ibm.com/think/topics/linux-containers.

[30] M. Mahalingam, D. Dutt, K. Duda, *et al.*, *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*, RFC 7348, Aug. 2014. DOI: 10.17487/RFC7348. [Online]. Available: https://www.rfc-editor.org/info/rfc7348.

[31] J. Elischer and A. Cobbs, *FreeBSD man-pages 14.3 - netgraph*, Accessed: 07.11.2025, Sep. 2021. [Online]. Available: https://man.freebsd.org/cgi/man.cgi?netgraph(4).

[32] L. Donnerhacke, *FreeBSD man-pages 14.3 - ng_pipe*, Accessed: 07.11.2025, Oct. 2019. [Online]. Available: https://www.gsp.com/cgi-bin/man.cgi?section=4&topic=NG_PIPE.

[33] X. Cao and X. Zhang, "SaTCP: Link-layer informed tcp adaptation for highly dynamic leo satellite networks", in *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, 2023, pp. 1–10. DOI: 10.1109/INFOCOM53939.2023.10228914.

[34] A. Conole, A. Serdean, A. Atteka, *et al.*, *Open vSwitch: Project Documentation*, Accessed: 06.11.2025, Nov. 2025. [Online]. Available: https://docs.openvswitch.org/en/latest/.

[35] L. Zeqi, L. Hewu, D. Yangtao, *et al.*, "StarryNet: Empowering researchers to evaluate futuristic integrated space and terrestrial networks", in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA: USENIX Association, Apr. 2023, pp. 1309–1324, ISBN: 978-1-939133-33-5. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/lai-zeqi.

[36] Filip, Ondrej, *The BIRD Internet Routing Daemon*, Accessed: 06.11.2025, Nov. 2020. [Online]. Available: https://bird.network.cz/.

[37] W. Lu, Z. Wang, S. Zhang, Q. Meng, and H. Luo, "OpenSN: An open source library for emulating leo satellite networks", in *Proceedings of the 8th Asia-Pacific Workshop on Networking*, ser. APNet '24, Sydney, Australia: Association for Computing Machinery, 2024, pp. 149–155, ISBN: 9798400717581. DOI: 10.1145/3663408.3663430. [Online]. Available: https://doi.org/10.1145/3663408.3663430.

[38] *etcd: Project Documentation (v3.6)*, Accessed: 06.11.2025, Jun. 2025. [Online]. Available: https://etcd.io/docs/v3.6/.

[39] M. Ottens, J. Deutschmann, K.-S. Hielscher, and R. German, "Proto2Testbed: Towards an integrated testbed for evaluating end-to-end security protocols in satellite constellations", in *2025 12th Advanced Satellite Multimedia Systems Conference and the 18th Signal Processing for Space Communications Workshop (ASMS/SPSC)*, 2025, pp. 1–8. DOI: 10.1109/ASMS/SPSC64465.2025.10946051.

[40] *Starlink - Updates*, Accessed: 12.11.2025. [Online]. Available: https://starlink.com/updates.

[41] T. Kohnstamm, *Kuiper - What is Amazon Project Kuiper?*, Accessed: 12.11.2025, Jun. 2025. [Online]. Available: https://www.aboutamazon.com/news/innovation-at-amazon/what-is-amazon-project-kuiper.

[42] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li, *Segment Routing over IPv6 (SRv6) Network Programming*, RFC 8986, Feb. 2021. DOI: 10.17487/RFC8986. [Online]. Available: https://www.rfc-editor.org/info/rfc8986.

[43] D. Tappan, Y. Rekhter, A. Conta, *et al.*, *MPLS Label Stack Encoding*, RFC 3032, Jan. 2001. DOI: 10.17487/RFC3032. [Online]. Available: https://www.rfc-editor.org/info/rfc3032.

# List of Acronyms

**3GPP** 3rd Generation Partnership Project

**AcS** access satellite

**BMv2** behavioral model version 2

**CGW** control gateway

**CSID** compressed segment identifier

**DGW** default gateway

**GEO** geostationary orbit

**GSL** gateway-satellite link

**GW** gateway

**HGW** home gateway

**IR** intermediate representation

**ISL** inter-satellite link

**LEO** low Earth orbit

**MEO** medium Earth orbit

**NEW** network emulator wrapper

**NTN** non-terrestrial network

**NTNE** non-terrestrial network emulator

**P4** programming protocol-independent packet processor

**PCE** Path Computation Element

**RoI** region of interest

**RTT** round-trip time

**SandSat** simulation and satellites

**SAT** satellite

**SDN** software-defined networking

**SID** segment identifier

**SR** segment routing

**SRH** segment routing header

**TE** traffic engineering

**TL** terrestrial link

**USL** user-terminal-satellite link

**UT** user terminal

# List of Figures

# List of Tables