
Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (XXXVII cycle)

# Integrating Artificial Intelligence Techniques for the Management of Softwarized Networks

By

**Antonino Angi**
******

**Supervisor:**
Prof. Guido Marchetto

Politecnico di Torino
2025

# Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Antonino Angi
2025

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Vorrei dedicare questa tesi e questi anni di lavoro a tutte le persone che mi sono state accanto.

Per cominciare, i miei genitori, Antonella e Giuseppe, che con il loro amore e le innumerevoli videochiamate non mi hanno mai fatto sentire solo, nonostante i tantissimi chilometri che ci separavano. Mia sorella Katia, che è sempre stata lì, in quello spazio in alto a sinistra, dove risiede il cuore. La nostra famiglia è la mia comfort zone più grande, il mio nido in cui torno quando più ne ho bisogno.

Ai miei migliori amici Federico, Marco, Giuseppe, Sara, Santi, Simone, Stefano, Elisabetta, Salvatore, Silvia, Sonam, Fiona, Mace, Erika, Lory, Mary, Cosimo, che sono stati un incredibile supporto durante questa avventura, offrendomi ospitalità e affetto in questi anni, ascoltandomi, comprendendomi, e, a volte, anche rimproverandomi, ma sempre facendomi sentire profondamente amato. Se oggi sono la persona che sono, lo devo in parte anche a voi. Grazie.

Al mio ragazzo, Lucas, che è sempre stato al mio fianco in tutti gli alti e bassi di questa esperienza, ascoltando lunghissimi messaggi vocali, le mie lamentele, i miei drammi, ma anche condividendo le gioie, le risate e i viaggi che hanno reso tutto più sopportabile. Obrigado por tudo o que você fez por mim, obrigado pela sua paciência, obrigado por ficar aqui comigo.

To my second family, Gina and Darryl, whom I have loved since the very first days we met. I have always found it incredible how destiny brought us together, and among all the people I could have met, I am deeply grateful to have encountered such amazing, intelligent, and kind-hearted individuals with whom I can always share laughter, travels, and meaningful conversations.

Ai miei tutori, Alessio e Guido, che mi hanno sempre supportato, rendendo l'esperienza di dottorato più facile e scorrevole e, soprattutto, mettendo il mio futuro

e la possibilità di fare tante esperienze davanti ai soli interessi di pubblicazione o di ricerca personale. Grazie per aver creduto in me e per avermi permesso di costruirmi un futuro ricco di esperienze curriculari. Grazie per il supporto in ogni cosa, dalle prime rejection alle varie richieste di missione: sapevo che per ogni novità potevo contare su di voi, un supporto fondamentale in questo viaggio.

# Abstract

Nowadays, the complexity of networks is continuously increasing due to the growing number of connected devices and a greater demand for data-intensive applications that often require low latency and high throughput. These demands pose critical challenges for complex and highly connected topologies, such as data centers, where big volumes of traffic data need to be processed and transferred at fast rates. Alongside this, traditional static network architectures often lack the flexibility to adapt to dynamic traffic patterns or support real-time management, resulting in degraded performance, increased latency, and potential bottlenecks.

An approach to support these challenges is provided by the concept of Software-Defined Networking (SDN), which simplifies network deployment and enables real-time adjustments, ensuring automated adaptability to dynamic network conditions. In this context, recent management approaches for softwarized networks leverage Artificial Intelligence (AI) and Machine Learning (ML) techniques to analyze traffic conditions, aiming to further reduce network reaction time and improve the accuracy of automated decisions. To intersect these advantages, data-plane programming has been introduced to enable fine-grained control over packet processing, allowing more efficient packet customization at the hardware level. However, while these techniques have shown great capabilities in improving network performance, they still lack structured mechanisms for efficiently distributing traffic loads in critical scenarios (*e.g.,* network congestion) or for appropriately integrating AI/ML methods that, due to their resource-intensive nature, can negatively impact the network's forwarding efficiency.

In this thesis, we present SDN solutions that leverage AI/ML methods and data-plane programmability to enable intelligent network routing for dynamic traffic demands on complex topologies, such as data centers. We begin by introducing Howdah, a profiling mechanism that, by performing traffic classification, offers

adaptation to different traffic classes under different network congestion scenarios. Building on this, we propose two advanced approaches for more autonomous traffic management: ROAR and ART. The former employs a Multi-Agent Reinforcement Learning (MARL) mechanism to implement adaptive routing policies based on real-time network conditions. By embracing the In-Network ML paradigm, the latter translates Deep Reinforcement Learning (DRL) decisions into lighter Decision Trees (DT), offering a more efficient solution that can be implemented directly inside switches without requiring external units.

Nevertheless, while network programming provides this high degree of customization, it is also known to be challenging, even for experts, requiring trial-and-error approaches with iterative debugging and log file analysis. This process is not only time-consuming but also prone to errors, slowing down the entire network customization process. To address this challenge, in this thesis, we also explore the Intent-based Networking (IBN) paradigm, which simplifies network programming by allowing programmers to specify high-level objectives (*i.e.,* intents) and have them translated into low-level network configurations. Thus, we propose two separate solutions: NLP4 and NAIL. The former combines Natural Language Processing (NLP) techniques with a MultiLayer Perceptron (MLP) model to translate high-level intents into mid-level policies, which are then integrated into network configurations through an API. The latter is a network transpiler architecture that adopts NLP and a specialized API to insert, monitor, and remove intents from a network without restarting the system.

In conclusion, we believe the proposed solutions contribute to enhancing programmable networks, addressing the growing complexity that modern networks face by providing greater flexibility and adaptation to dynamically changing conditions.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In the last decades, communication networks have evolved to accommodate the ever-growing traffic complexity and meet the demands for greater flexibility and efficiency. These advancements have made networks more customizable and programmable, allowing them to better adapt to the dynamic requirements of modern applications and services. One of the leading factors driving this evolution is the concept of "network softwarization" with the Software-Defined Networking (SDN) paradigm. By separating the control plane from the data plane, SDN has reshaped network design and revolutionized how networks are managed and operated [1].

This separation initially facilitated control plane programming by enabling the customization of centralized controllers, which led traditional SDN implementations to widely adopt OpenFlow as their standard control plane technology [1, 2]. However, while OpenFlow allowed this control plane customization, interactions with external controllers negatively impacted the overall network performance. To address this limitation, SDN evolved to also support data plane customization, with the emergence of data plane programming languages, such as P4 (Programming Protocol-Independent Packet Processors) [3] that supports the deployment of different applications [4] and led to a more specialized control-based APIs, such as P4Runtime [5].

The advent of data-plane programmability enabled a higher level of customization, allowing fine-grained control over packet-processing directly within network devices. For instance, P4 has allowed network programmers to customize their networks to different use-cases with the combination of advanced techniques, such

as Artificial Intelligence (AI) and Machine Learning (ML), that paved the way for more dynamic and intelligent network management [6–8]. Through the integration of specialized APIs (*e.g.,* P4Runtime), data-plane programmability has started addressing dynamic network demands in an automated way, allowing real-time optimization [9], network monitoring [10, 11], load balancing/profiling [12, 13], and network security [14, 15].

However, this integration comes with significant challenges. Deploying resource-intensive and complex AI/ML algorithms in the constrained, latency-sensitive environment of network packet processing is a demanding task. This integration requires careful optimization to ensure these models do not negatively impact overall network performance while still delivering benefits to the packet forwarding process. In addition to that, achieving this higher degree of flexibility and customization introduces significant challenges. While powerful, data-plane programmability, such as P4, is known to be difficult even for experts [16, 17] due to the language's limited abstraction capabilities and low-level hardware resources. Moreover, debugging data-plane programs is particularly challenging, often requiring network programmers to rely on iterative "trial and error" or analyze long log files to identify and resolve issues.

These challenges highlight the need for a more intuitive and streamlined approach to network management and customization, which led to the emergence of the Intent-based Networking (IBN) paradigm. IBN makes network programming accessible to less-experienced users by allowing the specification of high-level requirements (*i.e.,* an intent) in human-language without low-level details on how to reach them. In this context, we argue that effective integration of AI/ML techniques can not only enhance network performance by optimizing traffic distribution but also simplify network customization for less-experienced programmers through the combination of IBN techniques. This integration has the potential to further advance the "network softwarization" paradigm, essential for the efficient management and customized adaptation of modern network infrastructures.

**Our contribution.** Aware of the benefits brought by adopting AI/ML methods, in this dissertation, we explore their potential to enhance network traffic optimization in SDN by enabling intelligent decision-making for congestion-aware and adaptive management of networks. Furthermore, we leverage the advanced language understanding capabilities of these techniques to enable seamless network customization through the IBN paradigm, translating high-level intents into automated network

configurations and allowing even less-experienced users to have a network that aligns with their requirements. Together, these approaches aim to create more flexible, customizable, and reactive networks that can adapt to the dynamic traffic scenarios of complex topologies, such as data centers.

## 1.1 Summary of Contributions

Starting with the definition an intelligent management of traffic routing, we present Howdah, a traffic profiling mechanism that adopts a Decision Tree (DT) to predict and classify network traffic flows in two main categories: “elephant” and “mouse” flows. This classification is performed by the senders, which embed it into the packet before sending it. The switch or router receives the packet and forwards it to according to the classification flag. The "elephant" flows, characterized by large size and potential to cause congestion, are routed through the fastest paths using a Least Recently Used (LRU) criteria to minimize their impact on network performance. The “mouse” flows, being smaller and less likely to induce congestion, are forwarded with the widely adopted Equal-Cost Multi-Path (ECMP) algorithm. We demonstrated that Howdah is able to outperform other state-of-the-art solutions by reducing Round-Trip Time (RTT) and Flow Completion Time (FCT), showing adaptability to different network congestion levels and avoiding bottlenecks even at the highest load (Chapter 2).

With a similar goal of better dealing with network congestion, we developed ROAR, a Deep Reinforcement Learning (DRL)-based solution that performs adaptive routing according to the varying network conditions. Differently from traditional state-of-the-art solutions, ROAR runs directly within the switches in a distributed fashion by leveraging the In-Network ML paradigm. As traffic demand increases and the network becomes congested, our adaptive solution shows its benefits by achieving lower RTT, higher throughput, and decreased packet loss when compared to relevant solutions. Our evaluation, conducted in a simulated Mininet environment, highlights ROAR’s efficacy in dynamic and high-demand scenarios. We also showed that ROAR can be effectively implemented on real switches equipped with a Data Processing Unit (DPU), as its hardware requirements, in terms of CPU and RAM, are minimal (Chapter 3).

To avoid relying on external units while still embracing and taking advantage of the In-Network ML paradigm, we proposed ART, a Multi-Agent Reinforcement

Learning (MARL)-solution for adaptive traffic routing. In ART, each switch interacts with an external controller to exchange traffic metrics, which are used to train a DRL model. Following the teacher-student approach, the trained model is then distilled into lightweight DTs, making it efficient to be embedded directly within the switches. Our evaluations demonstrate that ART outperforms traditional state-of-the-art routing solutions by reducing RTT, increasing throughput, and reducing packet loss, especially at high network loads (Chapter 4).

Aware of the difficulty posed by network programming, we then propose NLP4, an architecture that leverages the IBN paradigm for translating intents, expressed in the form of human-language, into network configurations. When the network programmer specifies an intent, it goes through a Natural Language Processing (NLP) pipeline composed of preprocessing techniques to tokenize and reduce each word to its base form. After this phase, the intent goes through a Multi-Layer Perceptron (MLP) that identifies the involved network elements and the main goal specified by the intent. An API will then take this output and generate the appropriate configuration files that will customize the network. We showed that NLP4 is able to correctly translate the human-language intent and create the appropriate network configuration, ensuring the goals are fully respected across the network. Additionally, our evaluations showed that NLP4 maintains a limited execution time, even when processing intents for larger networks, highlighting its scalability and practicality for real-world scenarios (Chapter 5).

Further leveraging the IBN paradigm, we proposed NAIL, a transpiler architecture that bridges the gap between high-level intent and real-time network management. While NLP4 effectively translates human-language intents into network configurations by adding the required rules into the configuration files, it requires a system restart, which can be convenient when the network topology completely changes. However, in scenarios where only minimal changes and immediate responses are needed (*i.e.,* dynamic network environments or real-time traffic management), this approach can introduce delays. To address this limitation and allow real-time adaptability to new intents, NAIL enhances NLP4 capability by enabling network programmers to express desired behaviors in natural language and have them translated directly into executable code through a specialized transpiler without requiring a system restart. More in detail, when a network programmer introduces an intent in human-language, specialized NLP techniques parse it, identifying the involved network elements and the intent main goal. Finally, the transpiler leverages

the P4Runtime API that injects the rules to have the behavior applied to the device specified by the intent. Differently from other transpilers, NAIL also supports real-time intent updates and metric collection to validate the correctness of the intent. In our evaluation, we showed that NAIL requires fewer lines of code than other state-of-the-art transpilers while requiring only a few seconds to preprocess and inject the intent into the network. We also analyzed different possible intents that can be specified, showing correct rules installation that satisfy the requested behavior while also minimizing the RTT and reaction time when an update is requested (Chapter 6).

## 1.2 Thesis Objective and Research Methodology

The primary objective of this dissertation is to improve network traffic management by developing and evaluating novel routing and intent-based networking solutions that leverage AI techniques (*e.g.,* Decision Tree, Reinforcement Learning, Natural Language Processing). Specifically, we aim to address challenges related to traffic profiling, network congestion, adaptive routing, and intent-driven network configuration by proposing innovative mechanisms that enhance performance while maintaining efficiency and scalability.

To achieve this goal, the research is structured around the following sub-objectives:

1. **Traffic Classification and Efficient Routing**: Develop *Howdah*, a solution that efficiently classifies network traffic flows and improves routing decisions based on their characteristics;

2. **Adaptive Routing with DRL**: Design and evaluate *ROAR*, a DRL-based approach that dynamically adjusts routing strategies in response to network conditions;

3. **In-Network ML for Adaptive Routing**: Develop *ART*, a MARL-based solution that translates learned policies into decision tree rules, which are then distilled into switches for efficient, controller-independent traffic routing;

4. **Intent-based Networking (IBN) for Human-defined Rules**: Develop *NLP4*, an NLP-powered architecture that translates human-language intents into network configurations;

5. **IBN for Real-Time Network Configuration**: Enhance IBN with *NAIL*, a real-time transpiler that eliminates system restarts and allows immediate rule updates.

The research follows a structured methodology combining theoretical analysis and modeling, algorithm design, simulation-based evaluation, and SoTA comparison:

- **State-of-the-Art (SoTA) Analysis and Comparison**: Each objective is analyzed against recent SoTA solutions, showing their limitations and comparison with the proposed approach, which is furthermore benchmarked against mentioned routing and intent-based networking solutions to demonstrate improvements;
- **Algorithm and Model Design**: Each proposed solution is based on analyzing network requirements, existing limitations, and AI methodologies;
- **Simulation, Real-World Feasibility, and Trace-Based Evaluation**: Solutions are tested in Mininet, a widely used network simulation environment, to analyze performance metrics such as Round-Trip Time (RTT), throughput, packet loss, and scalability. CPU and RAM usage are also evaluated to assess real-world feasibility, and real trace-based evaluation is conducted to ensure practical applicability on realistic workload scenarios.

We summarize the contributions of this dissertation in Table 1.1, where we present the AI/ML methods applied to each specific networking problem, along with the corresponding chapters. We believe these solutions can serve as a foundation for developing more efficient automated systems and highly customizable networks, offering greater flexibility even to less-experienced network programmers.

| **Networking Problem** | **Chapter** | **AI/ML Method** |
|---|---|---|
| Traffic Profiling | Ch. 2 | Classification |
| Adaptive Routing | Ch. 3, Ch. 4 | MARL, DRL |
| Network Customization | Ch. 5, Ch. 6 | NLP, MLP |

Table 1.1 Summary of contributions and thesis organization.

# Chapter 2

# Congestion-aware Traffic Profiler

## 2.1 Introduction

Over the past decades, data centers have adapted their topologies to handle the growing demands of network applications, which always require data at a faster speed and a lower latency. To meet these demands, researchers have proposed new architectures, not only for optimizing ingress and egress traffic but also for improving the orchestration of internal traffic within the data center. Modern data center networks have focused on topologies like multi-rooted leaf-spine structures or, more commonly, fat trees. A key feature of these architectures is the use of multiple source-destination paths, designed to handle high traffic volumes, which introduces the challenge of implementing routing strategies that can effectively manage varying traffic loads, aiming to prevent congestion, minimize delays, and maintain high performance.

One critical challenge of these architectures is achieving balanced traffic distribution and ensuring that no single link becomes overutilized, which could lead to congestion, packet loss, or latency. However, the problem is further complicated by aspects such as uneven link bandwidth, different quality of service requirements, and traffic priorities across flows. In these cases, routing solutions must be capable of behaving differently according to these traffic types and handle their specific demands. These solutions are referred to as load-profiling routing algorithms [19, 20].

The work presented in this chapter has been partially published in [18].

A widely used routing strategy is Equal-Cost Multi-Path (ECMP), which assigns flows to paths through static hashing. However, ECMP does not account for network congestion or link failures, often resulting in uneven flow distribution and poor performance [21]. Researchers attempted to overcome such ECMP limitations by proposing valid solutions, however still introduced additional overhead, *e.g.,* [22, 23] or failed to apply efficient per-packet logic, such as in [24, 25]. Furthermore, while centralized schemes, such as Hedera [24], B4 [25], FastPass [26] and SWAN [27], can perform congestion-aware decisions, they require extensive control-plane traffic and react too slowly to dynamic data center traffic patterns. On the other hand, recent distributed approaches, such as CONGA [22] and HULA [23], rely on periodic feedback mechanisms, which can increase overhead and contribute to network congestion.

Building on these efforts, the research question we are addressing in this chapter is: *"Can we profile the network traffic's load over uncongested paths without the need to define elaborate protocols for exchanging information among switches or between the switches and a centralized controller?"*

In this chapter, we answer this question by proposing *Howdah*, a programmable data-plane architecture that enables load profiling by using distributed, partially congestion-aware logic for forwarding decisions. The idea behind Howdah is joint optimization: reducing flow collisions and maintaining efficient utilization within the data center network.

In Howdah, P4-programmed network switches implement a data-driven load profiling strategy that operates on flowlets rather than entire flows. Flowlets, described as bursts of packets in a flow split by a sufficiently large time gap, are preferred because they avoid packet reordering issues (since all packets within a flowlet follow the same path) and require no modifications to the TCP stack, as demonstrated in previous work [28].

To further optimize path selection, Howdah dynamically applies forwarding actions according to the traffic type of each packet. In Howdah, sending hosts within the data center and peripheral gateways run a supervised Machine Learning (ML) model to classify if each flow contains a large amount of data, *elephant flow*, or a small amount, *mouse flow*. This classification is then embedded into the packet itself in an in-band fashion, enabling intermediate switches to make quick routing decisions. Since elephant flows are the most responsible for network congestion,

they are routed over the fastest paths by selecting the least utilized path from the switch's perspective. In contrast, mouse flows, which cause a lower network load, are routed using a weighted version of the ECMP algorithm with flowlet grouping.

An ML classifier can help generalize over diverse traffic patterns while reducing the classification time compared to other statistical traffic classifiers. For this reason, in designing Howdah, we looked for an ML model that would minimize resource overhead—requiring less RAM, CPU usage, and training time—while also producing lower carbon emissions compared to other tested alternatives (*i.e.,* Support Vector Machines, k-means clustering, Random Forests, and Neural Networks). Based on our evaluation of these models using real-world traffic data, we selected a Decision Tree model for its simplicity, transparency, and ease of interpretability.

Then, we studied how different protocols impact Howdah's performance when transmitting in-band traffic classification information. Among different tested protocols (*i.e.,* IPv6, MPLS, New IP), we found that IP type-of-service fields provide negligible overhead, while also being a valid first implementation of our proposed architecture to transport traffic class information. We evaluated this implementation in data center network scenarios and compared its performance to relevant state-of-the-art solutions. Our results that Howdah significantly reduced both Round-Trip Time (RTT) and Flow Completion Time (FCT), particularly in scenarios involving high congestion and large data flows (elephant flows).

## 2.2 Related Work

Efficiently balancing or profiling traffic loads across available paths is known to be a challenge, especially in high-demand networks like data centers. Recent studies started to address this problem, proposing solutions designed to maximize the utilization of available bandwidth. While traditional local routing strategies, such as standard ECMP, are still widely adopted in practice, they are not optimal for data center environments, due to their local and stateless decisions, resulting in traffic division without accounting for potential network congestion [24, 29, 30].

Recent local approaches, such as DRILL [31], Clove [32], and PRESTO [33], attempt to address ECMP's limitations by refining decision-making at the individual switch level. DRILL, for example, bases forwarding decisions on real-time local

queue occupancy, allowing it to operate on packet-level timescales. PRESTO [33] takes advantage of ECMP's near-optimal performance in symmetric Clos topologies with small flows by splitting flows into "mice" and source-routing them across paths without requiring load awareness. However, both solutions often face challenges with TCP reordering, which is further complicated in networks with asymmetric topologies.

An alternative strategy involves delegating traffic management to centralized controllers, which can make congestion-aware decisions by leveraging global network information. Systems like B4 [25], F10 [34], Mahout [35], MicroTE [36], and Hedera [24] have demonstrated strong performance in managing inter-data center wide area networks (WANs). However, these approaches are less effective in highly dynamic data center environments due to their reliance on coarse-grained control timescales, while also not being designed to depart from balanced loads. A more recent centralized solution is Tiara [37], a three-tier system combining programmable switches for packet encapsulation and decapsulation, an FPGA for match-action table processing, and x86 servers for load-balancing software in slower paths. While this architecture offers improved efficiency and scalability, its reliance on specialized hardware significantly increases deployment costs, which may limit its practicality in some scenarios.

To achieve microsecond-level performance while still using global network information, CONGA [22] operates directly within the data plane by adopting a distributed approach for optimizing resource allocation and enabling rapid adjustments to network asymmetries. Specifically, CONGA uses a leaf-to-leaf mechanism where switches at the edge (described as leaves in Clos networks) collect and analyze congestion feedback from remote switches to estimate further congestion on fabric paths in real-time. This mechanism is then combined with the flowlet switching strategy to optimize performance. Results brought by CONGA highlight the efficiency of flowlet-based traffic management, especially in the context of data centers, where it helps maximize network performance. However, while valuable, this approach has two main limitations: first, a global congestion awareness at edge switches can negatively impact switch memory as the network grows; second, its implementation relies on custom hardware.

These limitations are explicitly addressed by HULA [23], a data-plane load balancing algorithm applied to P4-based programmable switches. Instead of tracking

congestion across all paths, HULA's leaf switches monitor the best path to each destination through their immediate neighbors, reducing memory demands and eliminating the need for custom hardware. Specifically, HULA uses probes to collect information about network conditions, such as link failures or topology changes and uses this data to update internal routing tables.

Similarly, our solution adopts the same principles for a data-plane load profiling strategy. However, rather than sending probes via Top of Rack (ToR) switches, we embed additional information directly into packets, allowing each switch to better handle traffic congestion. This leads to enhanced congestion control and improved overall network performance. A recent solution as CONTRA [38] provides a performance-aware routing that can adapt to traffic changes at hardware speed, allowing the users to specify network policies to rank network paths given their real-time performance. After verifying these policies, the CONTRA compiler translates them into P4 programs customized to the network topology. Despite its flexibility, HULA's policy remains the default and highest-performing option within this framework.

Leveraging the customized forwarding decisions brought by these solutions, we design our solution with some out-of-the-box load profiling actions that the users can extend and adapt to achieve specific network performance goals. However, unlike other load profiling solutions, our local congestion-aware routing approach does not rely only on switches but also on host machines for traffic type identification. This host-based traffic classification is then embedded into the packet in an in-band fashion and used by the switches for forwarding decisions across the network.

Other recent approaches have been studied, such as Application Aware Networking (APN) [39], which allow senders to transmit flow-specific information for fine-granularity traffic steering and network resource management. However, APN relies on multiple additional entities, such as controllers, edge nodes, and mid-points, which complicates deployment. In addition, it assumes that clients provide accurate metadata and behave in good faith, which may not always be the case. In contrast, our approach avoids these assumptions. For external traffic, metadata is inserted by the network provider's nodes, while internal machines are under control. We summarize in Table 2.1 the main differences between Howdah and the mentioned state-of-the-art solutions.

| | Local Decision | Hardware Independent | Traffic Type Distinction | Data-center Optimized | Distributed-only Implementation |
|---|---|---|---|---|---|
| ECMP | ✓ | ✓ | ✗ | ✗ | ✓ |
| DRILL [31] | ✓ | ✓ | ✗ | ✓ | ✓ |
| PRESTO [33] | ✓ | ✓ | ✓ | ✓ | ✗ |
| Hedera [24] | ✗ | ✓ | ✓ | ✓ | ✗ |
| Tiara [37] | ✓ | ✗ | ✗ | ✓ | ✓ |
| CONGA [22] | ✗ | ✗ | ✗ | ✓ | ✓ |
| HULA [23] | ✓ | ✓ | ✗ | ✓ | ✓ |
| CONTRA [38] | ✓ | ✓ | ✗ | ✓ | ✗ |
| **Howdah** | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 2.1 Comparison of related works with Howdah.

## 2.3 Architecture and Protocol Design

Howdah is a programmable data-plane architecture specifically designed for load profiling. It works by enabling switches to optimize network resource usage through a combination of load profiling techniques and flow classification, allowing the system to differentiate forwarding actions based on the traffic type. The traffic classifier can be implemented either at the network provider's ingress or directly at the local sender. Either way, in the context of our solution, we refer to this element as *Howdah host*.

In Fig. 2.1, we report the main architectural components of our Howdah algorithm: one running on local machines or gateways, and one running on P4-enabled switches. As visible from the figure, the host adopts a decision tree (D-Tree) to classify traffic before sending it out throughout the network and embeds the classification label into a specific field in the packet header (as described in Section 2.4). This classification process is applied to both encrypted and unencrypted packets, including flows with encrypted payloads. Upon receiving a packet, an intermediate switch examines the classification label to determine the appropriate forwarding action. For smaller "mouse" flows, forwarding decisions rely only on the packet header without requiring switches to update their statistics. However, for larger "elephant" flows, which significantly influence network congestion, switches adopt a Least Recently Used (LRU) mechanism to select the next hop port. The rest of this section motivates this design choice and describes the algorithms of both the Howdah host and P4-enabled switches.

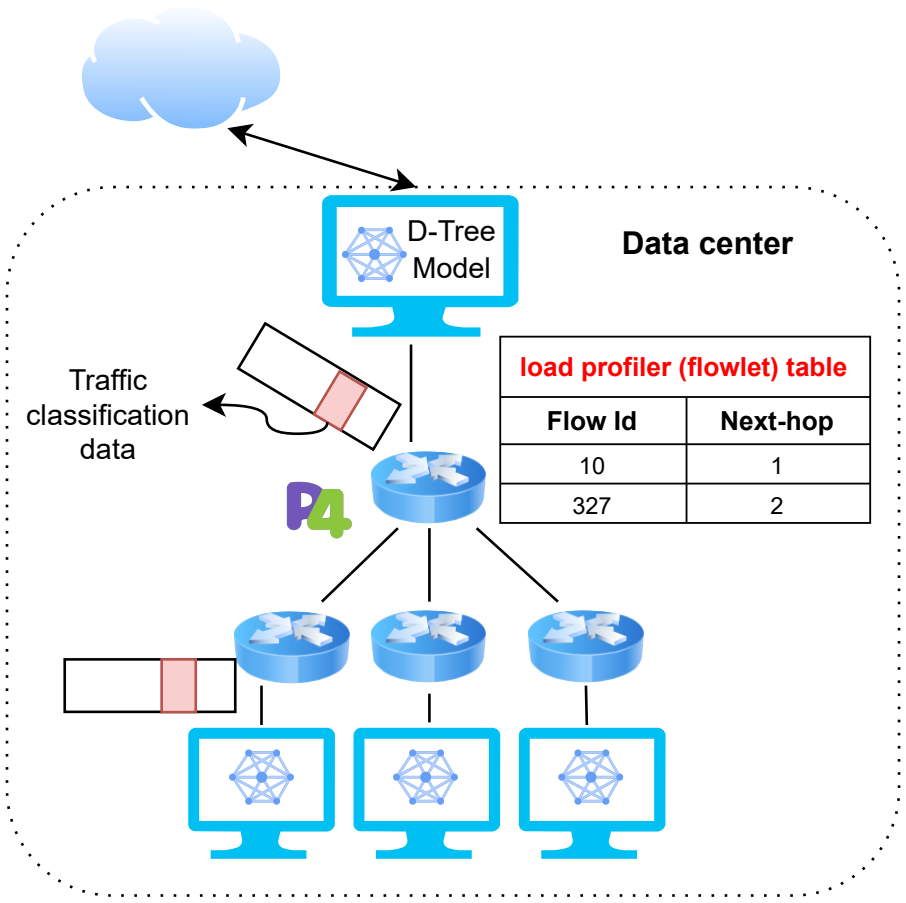



Fig. 2.1 Howdah's Overview and Components.

### 2.3.1 Host-based Traffic Classification within Howdah Hosts

In our architecture, while switches handle load profiling and traffic management decisions, the responsibility for traffic classification lies with senders and gateways. Given the data center topology of our network, we assume that, for East-West (internal) traffic, sending hosts can be configured to classify traffic using an ML model, which is then embedded directly into the packet header (see Section 2.5 for more details).

Conversely, we assume external hosts may not implement any ML classification logic for the North-South traffic (to/from outside). Most importantly, external traffic classification could not be trusted. For this reason, when packets originate from outside and enter our network, our gateway performs the traffic classification using the same ML algorithm and packet filtering, a common procedure in a data center.

For traffic leaving the data center, any classification metadata added within the network is removed before the packet exits.

**Howdah for traffic classification.** Data centers host a wide range of services, each generating distinct types of network traffic. Among them, we can cite on-demand video delivery, storage and file sharing, web search, social networks, cloud computing, financial services, recommendation systems, and interactive online tools [40, 41]. These applications present different traffic characteristics and distribution of flow arrivals, flow sizes, and flow duration [42]. For one, web search queries typically produce small, short-lived data flows, while batch-processing tasks like those in high-performance computing (HPC) environments often involve transferring big datasets. An example of this case is the shuffle phase in Hadoop's MapReduce jobs, which can move petabytes of data across the network [43]. Moreover, the rise of latency-sensitive and real-time applications, such as video streaming and voice communication, has driven a growing demand for efficient handling of interactive and time-critical traffic within data center networks.

Such various applications lead to the emergence of both long-lived connections and short microbursts in the same network [44, 45]. As typical in the network management literature [35, 46], we refer to long-lived flows as "elephants" and short microbursts as "mice".

The common goal of a load profiling solution is *to provide high bisection bandwidth for throughput-sensitive and latency-sensitive flows without unduly delaying remaining flows by distributing "available" bandwidth across a set of candidate routes to match the characteristics of incoming QoS requests.* In line with recent studies [47, 48], which emphasizes the importance of distinguishing between "elephants" and "mice", we argue that identifying long-lived flows is crucial for taking appropriate actions and handling traffic management. Properly classifying and distributing such flows can improve application performance and reduce congestion by leveraging redundant links more effectively. Recent studies have pointed out that East-West traffic within a data center typically generates traffic volumes an order of magnitude higher than North-South traffic [49]. Avoiding bottlenecks is thus extremely important for all the traffic flowing in the topology.

It can be noted that, while this chapter primarily focuses on two traffic categories, the Howdah framework and its P4-based implementation offer the flexibility

to accommodate more traffic classifications (*e.g.,* bandwidth *vs*. delay-sensitive applications, or web *vs*. database *vs*. HPC traffic).

**Why host-based classification.** One potential location for packet classification is within the switch itself. Unfortunately, the hardware design of network nodes is not suited for the training process of machine learning models, leading to poor performance. Furthermore, given the strict packet scheduling of switches in data centers, the integration of ML models could either impact packet forwarding efficiency or require extensive hardware and software changes to handle the extra computational demands.

To ensure high-speed packet processing, the literature has proposed approaches where switches collect flow metrics but move the ML training to centralized controllers [24, 50, 51, 25, 52]. However, these approaches face significant scalability challenges due to limited bandwidth between switches and the controller, which creates a bottleneck for transmitting statistics in this traffic management strategy. Additionally, collecting statistics for every flow individually would heavily impact switch resources, while sampling detection, which analyses only a portion of the packets, would only be able to identify elephant flows accurately after processing around $10,000$ packets [53]. In light of this, we argue that the most effective locations for identifying elephant flows in data centers are the host machines and gateways. The application layer of data center programs with the Howdah layer is well-suited to the unified software environment and centralized management typical of data centers. Moreover, hosts in these environments often have GPUs or powerful general-purpose CPUs, which are better suited to handle ML classification compared to the limited processing capabilities of standard network nodes. Finally, hosts and gateways have good visibility into the traffic patterns generated by applications, making them ideal for this goal.

**Why ML-based classification.** Traditionally, elephant and mouse flow classification has relied on statistical methods. However, we argue that adopting ML for this scope offers both greater speed and higher accuracy. As mentioned earlier, traffic classification can take place either at the host level or within the network's infrastructure. Host-based approaches, such as those in Mahout [35], typically rely on metrics like buffer occupancy during the decision-making process, meaning that any packet has to wait while the buffer is examined, which slows down the transmission process. However, as demonstrated in Section 2.6.8, our classification takes only

$\mu s$ to execute, outperforming the *ms* needed by Mahout. Other host-side detection methods often require communication to begin and traffic to accumulate before the flow size can be determined.

On the other hand, for a network decision, classifications made by SDN controllers, as demonstrated in ZOOM [54], use real-time metrics such as the number and size of active flows in the network. While this approach ensures accuracy by considering current network conditions, it requires periodic polling and results in delays of several seconds during states.

Adopting innovative data-driven algorithms, like those used in Howdah, reduces the entire process (for both classification and label stack) while delivering notable levels of accuracy, as shown in 2.6.2.

### 2.3.2 P4-compatible Switches

The main task of the switch is to profile flowlets – bursts of packets belonging to the same flow separated by a significant time interval – to avoid possible issues at the destination. Studies have demonstrated that routing flowlets along the same path ensures packets arrive in order, avoiding the need for additional buffering and mitigating potential Quality of Experience issues [28]. Moreover, flowlet-based decisions (rather than flow-based) allow higher granularity while providing better performance [22].

To maintain flexibility and adaptability across various protocols (as outlined in Section 2.4), we instruct our switches with P4, a programming language for protocol-independent packet processes [3]. Such a language enables the programming of packet processing pipelines in packet forwarding ASICs, allowing users to define customized parsing rules and implement new protocol logic. Its control framework aligns with the SDN architecture, employing a distinct control plane to deliver commands directly to network devices. This programmable approach offers significant benefits compared to fixed hardware implementations: users can adjust variable sizes and register capacities to match the network topology and traffic patterns. For example, since Howdah can work with different packet header formats (see Section 2.4), the packet parsing can be smoothly adapted to meet the desired header policy. In addition, P4 provides a hardware-agnostic abstraction for switches,

enabling portability: P4 programs are initially compiled into a generic representation (frontend) and later recompiled for specific platforms, *e.g.,* NetFPGA [55].

**Howdah switch forwarding.** In our solution, we customize P4 tables to apply match-action entries that execute our load-profiling functions. In general, P4 tables enable switches to define actions such as determining the next hop, managing multicast groups, or performing Layer-2 forwarding based on MAC addresses. With Howdah, once the hosts have inserted the information about the traffic type, our P4-enabled switches make forwarding decisions based on both port utilization and the provided traffic type data. In particular, we use a table stored in the P4 switch registers, which contains the hash of the incoming flowlet along with the output port and the last time this flowlet (from the same flow) was processed. This timestamp helps calculate the time difference between the new flow's arrival and the last recorded one. When the difference falls below a threshold $T$, chosen according to other state-of-the-art techniques [23], the switch forwards the flowlet to the previously selected best-hop. If the difference exceeds $T$, the switch recognizes the start of a new flowlet, calculates a hash based on a 5-tuple that includes the protocol, IP source and destination addresses, and TCP source and destination ports, and then selects the best next hop for the current 5-tuple. It is worth noting that the 5-tuple hashing is performed directly within the switch without involving any controllers, which helps minimize the delays that would typically arise from controller interactions [56].

The term *load profile* refers to the distribution of load across outgoing links of a switch, allowing users to define how traffic should be spread across these paths [19]. In many cases, this translates to balancing the traffic load evenly across available links; but other circumstances may demand uneven traffic distribution if links have different characteristics (*e.g.,* link capacity) or the traffic has different priorities. Our P4-enabled switches can be easily adapted to accommodate the load profiling policy the user specifies. More formally, consider a system with $N$ different paths between a given source and destination, and let $W$ denote the overall system load. In a load-balanced system, the goal is to distribute the load equally across all paths, bringing each path's bandwidth closer to $W/N$. In contrast, a load-profiled system aims to distribute the load in such a way that the likelihood of meeting the QoS requirements for incoming traffic requests is maximized. One simple algorithm for load profiling is based on assigning a weight $q_i$ to any switch's port $i$, representing the probability of choosing the path. The switch, then, performs a weighted choice when selecting the output links, ensuring that ports with higher weights are chosen

more often than those with lower weights, thereby distributing traffic in line with the desired load profile [57].

In our solution, forwarding rules are based on top of a flowlet-based version of ECMP packet forwarding with weights. Similar to traditional ECMP, the next-hop selection is made by hashing the 5-tuple, but here, the decision is based on flowlets rather than individual flows. For smaller flows (mice), the switch simply forwards the packet to the best next-hop according to the weighted flowlet-based version of ECMP, *i.e.,* it chooses a path with a weighted probability to avoid congesting a path quickly. Otherwise, for larger flows (elephants), the next-hop selection also considers the least recently used (LRU) port. Ports with higher usage are more likely to face congestion, so the switch also takes into account the frequency of path selections for each port. To do so, the switch updates the network's utilization statistics with every incoming packet, alongside the hash computation. Despite being simple, this LRU criterion is effective in preventing congestion and reducing delays by distributing the flowlet across less utilized ports, avoiding overlap with other large flows.

## 2.4 In-Band Traffic Knowledge Policy

Recent studies have shown that embedding additional network data within packets can increase their size, making some of them among the largest on the network [58–60]. Telemetry metrics can not excessively harm application data, even if used to monitor congestion and verify network performance requirements. A notable solution to this challenge is adopting in-band network measurement, which has become prevalent in network management tasks. This technique embeds network-related information directly into the packets, either in the payload or the header, to facilitate operations without creating excessive overhead.

For this reason, in our solution, we leverage in-band network management to guide packet-forwarding decisions at the switch level by using the embedded data within each packet. By combining in-band flow information with P4, we reduce the need for traditional SDN control traffic, such as the flow rule exchanges between switches and controllers seen in architectures like OpenFlow (see Section 2.6.6 for a numerical comparison). As visible from Fig. 2.1, the control traffic is now carried in packet headers. To demonstrate the feasibility of this architecture, we propose and evaluate three distinct algorithms, each designed to support real-time networking

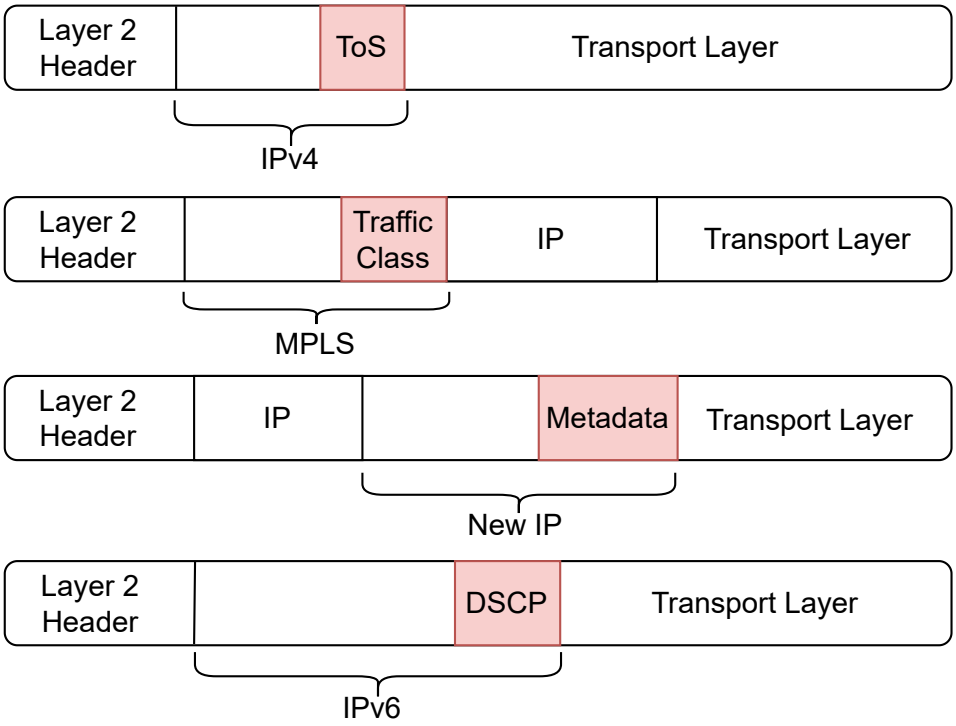



Fig. 2.2 Analyzed protocols for Howdah's packet header.

requirements without the need for specialized hardware or centralized controllers. Specifically, we analyze these approaches, outlining their benefits and potential trade-offs.

**IP Type of Service.** The Type of Service (ToS) field of IPv4 has been designed to indicate the priority of a datagram and request a route for a low-latency, high-throughput, or highly reliable service. This 8-bit field is divided into two parts: six bits for Differentiated Services Code Point (DSCP) and two bits for Explicit Congestion Notification (ECN). Although routers are not explicitly required to act in a specific way based on these values, ToS definitions are widely supported in Unix-based systems, making them a practical choice for embedding traffic classification data when combined with programmable switches and leading to minimal overhead (Section 2.6). However, the limited number of available bits limits the scalability and the ability to support diverse application needs or detailed switch action specifications.

**MPLS.** Multiprotocol Label Switching (MPLS) works by prefixing packets with a label stack, where each label includes four fields, one of which is a 3-bit Traffic Class

field commonly used for Quality of Service (QoS). An MPLS-compatible version of Howdah could leverage this field to encode traffic flow information. Possibly, paths per flow are reserved in advance by means of the Label Distribution Protocol (LDP), and also profile information can be easily carried. This method provides greater flexibility for traffic engineering and more precise control over network behavior. However, it introduces additional complexity, including the need for an extra packet header and a protocol like LDP for label distribution, as well as the administrative effort of preconfiguring paths on network devices. Moreover, there are also new proposed variants of MPLS that would allow for the encoding and processing of metadata as part of a label stack [61].

**New IP.** Other protocols capable of supporting ancillary data, which could carry classification details and more, include New IP [62], using a flexible packet contract, and its precursor Big Packet Protocol (BPP) [63]. These protocols introduce an extendable framework for modifying packet-based network behavior through the use of a "contract": a block of data (metadata and forwarding instructions) carried with the header and user payload that can be used to inject ancillary information that provides guidance to intermediate switches on how to process these packets. Within the context of our solution, the metadata field could encapsulate traffic classification details as well as other relevant data, such as parameters related to service-level guarantees. Furthermore, the ancillary data fields could be expanded to simultaneously enable additional functionalities, such as telemetry collection, which can help enhance and refine traffic profiling strategies.

**IPv6.** IPv6, the most recent version of IP, includes an 8-bit Traffic Class field in its header, which is further divided into two sub-fields for managing traffic and congestion: the 6-bit Differentiated Services Code Point (DSCP) for traffic classification and the 2-bit Explicit Congestion Notification (ECN) for congestion control.

In an IPv6-compatible implementation of Howdah, we propose embedding traffic information within the DSCP field, as its functionality aligns with the ToS field in IPv4. The IPv6 header is 320 bits long, twice the size of the IPv4 header, due to the 128-bit length of IPv6 addresses. While IPv6 adoption is constantly growing and almost 50% of Google connections occur over IPv6 [64], only 29.2% of networks listed in the global BGP routing table currently support it [65]. Our approach to embedding traffic data in the IPv6 header can work together with

Segment Routing over IPv6 (SRv6), which leverages IPv6 extension headers [66] to encode parameters for specialized routing operations. For instance, Segment Routing Headers (SRH) [67], a well-known example of such extension, enable the transmission and processing of additional parameters. This flexibility suggests strong compatibility between our architecture and SRv6, making the integration of traffic classification data into IPv6 both feasible and practical.

**Howdah essentials.** Given the building blocks of Howdah mentioned earlier, we designed our solution to be compatible with different protocols for embedding traffic information. In this chapter, we focus on selected few protocols that can inform switches about traffic types, as shown in Fig. 2.2. The figure shows, in red, the specific areas within different protocols (IPv4, MPLS, New IP, IPv6) where Howdah can insert classification labels, while other protocol fields are purposefully omitted for clarity. However, we argue that the flexibility of our P4-enabled switches allows for the use of alternative protocols, such as VXLAN, without compromising functionality. The essential idea of our approach does not change regardless of which protocol carries the classification result, and our solution involves hosts-switches cooperation towards optimized forwarding decisions. Before transmitting a packet, the host adds a flow-type bit to distinguish between elephant flows and mouse flows, informing the switches on how to handle the traffic. The switch uses the Howdah header for distinguishing between flows: “0” if mouse, “1” if elephant. This header field embeds metadata directly into the packet to provide guidance through the network, where our switches use this value to perform load profiling at the flowlet granularity.

## 2.5 Howdah Traffic Classification

One important aspect at our system’s core is traffic classification, as it impacts how packets are forwarded. This section describes the process implemented on host machines to perform such classification task.

### 2.5.1 Decision Tree Model

In Howdah, we classify each flow using a decision tree, a predictive ML model that uses a tree-like structure to make decisions based on various input variables [68].

The purpose of a decision tree classifier is to predict the class of input by building a tree-like structure where each internal node represents a feature, and each leaf node represents a class, using logical tests at each node to identify relevant features and provide accurate predictions based on trained data. Decision trees are versatile and can handle both classification and regression tasks, making them suitable for datasets with variables that have many possible values. One key parameter in decision trees, often shortened as D-Tree, is the tree's depth. Setting an appropriate depth is crucial to avoiding overfitting, a situation where the tree becomes overly complex with too many branches and leaves [69]. Overfitting can result in poor performance on new data, as the decision tree may not generalize well to unseen data. By optimizing the tree's depth, the decision tree can balance complexity and simplicity, ensuring accurate predictions for new scenarios. To achieve it, we used a $k$-fold cross-validation technique, which evaluates the optimal maximum depth for the dataset by dividing it into $k$ equal-sized subsets. Each subset is then used as a validation set while the model is trained on the remaining $k$-1 subsets. This iterative process checks the D-tree model's performance on different data subsets, minimizing bias and variance while improving the reliability of the decision tree's predictions.

### 2.5.2 Howdah Classifier Methodology

As mentioned, we used a decision tree to classify the traffic type for three main reasons. First, the structure of a D-tree resembles the decisions made by many networking systems, such as flow scheduling [70] and adaptive bitrate (ABR) algorithms [71], which rely on logical decision-making. Second, decision trees are computationally efficient, making them well-suited for networking environments. As detailed in Section 2.6.8, our implementation achieves accurate traffic classification without causing significant resource or latency overhead on the host machine. Third, decision trees are both expressive and interpretable, thanks to their non-parametric nature, allowing them to model complex policies effectively. Model interpretation is an important part of an ML process, as it enables insights into the model's decision-making process, ensuring fairness and accuracy. This transparency is particularly useful for monitoring and troubleshooting network operations, as further discussed in Section 2.6.3.

In Howdah, the host classifies traffic before transmission by applying a decision tree algorithm and embedding the classification label directly into the packet. Our

supervised classifier operates on an input space of $1 \times N$, where 1 refers to the fact that it just considers a single packet and $N$ is the cardinality of features considered. In particular, our decision tree model is trained on a features list composed of five elements: source and destination IP address, source and destination port number, and transport protocol (*i.e.,* TCP or UDP). This feature set is easily accessible through a packet interceptor and remains valid even for encrypted traffic, which is common in data center applications. The classification process outputs a binary label: "0" for a "mouse" flow or "1" for an "elephant" flow, guiding network decisions accordingly.

In our data center setup, both the hosts and the gateway are required to run an adapted version of either kernel-level network services or application-level socket instances. Inspired from previous work that demonstrated the advantages of implementing a shim layer on end hosts [35, 42], our prototype also adopts this approach, which validates the choice as shown in Section 2.6.

To further simplify the operations on the host machines, we apply the classification process only when needed: protocols known to contribute little to network congestion, such as ICMP, are automatically labeled as mouse flows. On the other hand, for transport protocols that could easily contribute to congestion, such as TCP and UDP, as well as for unrecognized protocols, Howdah classifies the packet before transmission. The resulting classification label is then embedded directly into the packet header. It is important to note that while forwarding decisions are made at the flowlet level, the classification is made at the flow level, reducing the number of times the classification is executed. Once the flow is classified, the resulting label is included in every packet within that flow, ensuring efficiency without compromising the forwarding strategy.

**Clarifying example.** When a connection is established, the host classifies the flow using a pre-trained Decision Tree (D-Tree) model before transmitting any data. In the IPv4 implementation of Howdah, this classification result is embedded within the header's Type of Service (ToS) field. After receiving a flowlet, the switch computes its hash based on the 5-tuple (protocol, IP source & destination address, TCP source & destination port), and stores the hash along with the current timestamp in a flowlet table. The switch then determines the output port according to the type of traffic: flowlets from smaller flows (mice) are forwarded using a weighted variant of the ECMP algorithm, while larger flows (elephants) are routed based on the least recently used (LRU) strategy. If a subsequent flowlet from the same flow is received, the

switch calculates the time elapsed since the previously stored timestamp. If this time difference is less than or equal to a threshold value $T$, the flowlet is forwarded using the same output port stored in the table. If the time difference exceeds $T$, the switch recalculates the hash, updates the timestamp, and selects a new output port based on the LRU method. This concept is particularly important for latency-sensitive applications, where being routed over the less congested path is crucial. Short-lived applications, sometimes encapsulated in protocols diverse from TCP/UDP, are thus used to balance the overall network congestion.

## 2.6 Evaluation

In this section, we illustrate the evaluation results that helped us develop our solution confirming Howdah's benefits. We begin by summarizing the experimental setup used in our testing. Next, we analyze the performance of the classifier and explain its behavior. Then, we assess the overall performance of Howdah and compare it against a centralized implementation. Finally, we measure the resources consumed when running our proposed ML model.

### 2.6.1 Evaluation Settings

To evaluate our solution in a simulated data center environment, we implemented Howdah on Mininet, a network emulator designed for creating virtual networks and enabling fast simulations. Designed for SDN networks, Mininet can be used in combination with the behavioral model version 2 (bmv2) to configure P4-programmable switches in a simulated environment. By compiling P4 programs into packet-processing logic for C++11-based software switches, Mininet allows developers to test and debug their network configurations before deploying them on physical hardware. For this study, we focused on a load profiling challenge.

The simulation environment was set up using a leaf-spine topology, consisting of 10 server racks connected to leaf switches, which in turn were connected to four spine switches each, as illustrated in Fig. 2.3. In this load profiling scenario, to the orange links (150 Mbps), we assign a weight of 2 and to the blu links (100 Mbps) a weight of 1, to favor faster links. Traffic workloads were generated using the $iperf3$

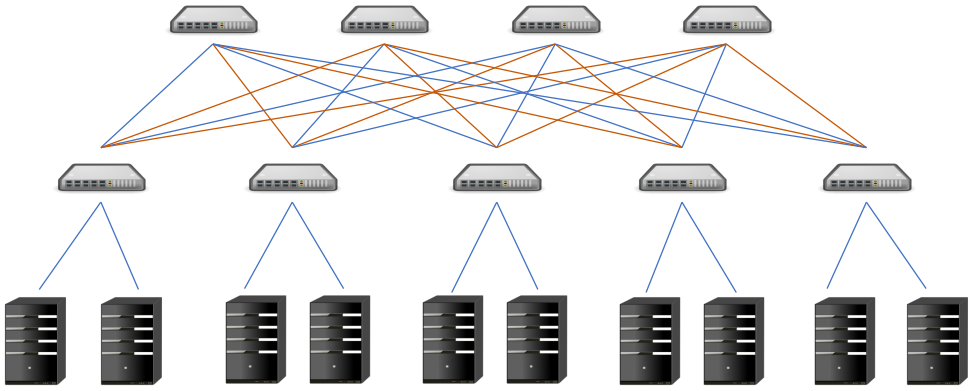


Fig. 2.3 Network topology for the experimental results: blue links with 100 Mbps bandwidth and orange links with 150 Mbps bandwidth.

tool to simulate different loads and induce also network congestion, enabling us to verify network performance under different conditions.

We then tested the traffic classifiers when the input is composed of three realistic workloads, taken from publicly available datasets [72]. We extracted three different datasets and stored them in a .pcap file, corresponding to three captures obtained from the same data center over different times of the day. We mentioned them: "US-UNV-1" with 887,647 items, "US-UNV-2" with 913,026 items, and "US-UNV-3" with 887,647 items. We analyzed these files to extract the essential features for each flow, and then assigned a flow label based on the total bytes transferred: if the flow exceeds a threshold of $D$ bytes or lasts longer than $L$ seconds, it is classified as an elephant; otherwise, it is considered a mouse. As in [23], we set the threshold $D$ to 1700 bytes and $L$ to 10 seconds, as these values proved to be practical and provided a balanced label distribution.

Additionally, as in [22, 23], we set the flowlet gap to $100\mu s$, which we found to be valid in distinguishing flowlets belonging to the same flow. In these experiments, hosts H1 and H4 generated background traffic using the *iperf3* tool, continuously adjusting the bandwidth to increase the overall network load and induce congestion. At the same time, H1 sent trace-driven packets to the other hosts (H2 through H10) and modifies the packet size based on the flow type (elephants or mice) while collecting and showing all the relevant metrics, such as Flow Completion Time (FCT) and Round-Trip Time (RTT), to evaluate network performance.

| | US-UNV-1 | | | | US-UNV-2 | | | | US-UNV-3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ***Accuracy*** | ***Precision*** | ***Recall*** | ***F1-score*** | ***Accuracy*** | ***Precision*** | ***Recall*** | ***F1-score*** | ***Accuracy*** | ***Precision*** | ***Recall*** | ***F1-score*** |
| **SVM** | 0.94 | 0.96 | 0.94 | 0.97 | 0.94 | 0.96 | 0.94 | 0.96 | 0.99 | 0.99 | 0.99 | 0.99 |
| **NN** | 0.93 | 0.92 | 0.99 | 0.96 | 0.93 | 0.93 | 0.99 | 0.95 | 0.99 | 0.99 | 0.99 | 0.99 |
| **k-means** | 0.83 | 0.99 | 0.83 | 0.91 | 0.84 | 0.99 | 0.84 | 0.91 | 0.97 | 0.99 | 0.98 | 0.99 |
| **RF** | 0.99 | 0.99 | 0.99 | 0.99 | 0.93 | 0.93 | 0.93 | 0.93 | 0.97 | 0.97 | 0.97 | 0.97 |
| **D-Tree** | 0.99 | 0.98 | 0.99 | 0.98 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.97 | 0.98 |

Table 2.2 Performance comparison in classifying data center traffic with different ML models across three datasets.

**Traffic classifier benchmarks.** We evaluated our Howdah classifier by benchmarking it against four popular and widely used ML models: First, a Support Vector Machine (*SVM*) model technique as in our previous work [13], leverages a combination of the SVM technique with stochastic gradient descent (SGD), which helps handle large datasets effectively while minimizing computation time. Second, we considered a Neural Network (*NN*) classifier, composed of three fully connected layers. The first two hidden layers consisted of 12 and 8 nodes, respectively, and employed the rectified linear unit (ReLU) activation function. The output layer consisted of a single node with a sigmoid activation function. We experimented with different configurations of layers and nodes for our *NN* classifier to finally find the optimal combination that maximizes the performance metrics of our classification problem. Additionally, a relevant study [48] explores both supervised and unsupervised ML methods to identify flow types based on traffic characteristics. Its prediction proposes an unsupervised ML solution that uses a clustering technique as $k-means$, to predict classes, labeling each flow as either "elephant" or "mouse". Finally, a Random Forest (*RF*) model-based technique as in [47] to classify network flows. The goal of this model is to improve performance metrics such as the incast completion time, particularly in buffered-switch environments where traffic patterns are dominated by elephant flows.

**Load profiler benchmarks.** We evaluate our approach by comparing it to two of the most recent solutions: CONGA [22] and HULA [23]. It is worth mentioning that the more recent CONTRA [38] also relies on HULA as its primary method. Unlike these approaches, we avoid using out-of-band probes, which generate additional overhead traffic, and instead inject network information directly into the packet itself. Finally, ECMP is used as a baseline.

### 2.6.2 Traffic Classification Accuracy

To estimate the performance of our model, we use the standard notation: TP for true positive, TN for true negative, FP for false positive, and FN for false negative. In particular, we pair TP as the numbers of elephants correctly predicted (TP) or mice correctly predicted (TN); FP as the numbers of elephants misclassified as mice (FP) and mice erroneously predicted as elephants (FN). Since Howdah simplifies the task to a binary classification problem, differentiating between "elephants" and "mice", this naturally defines the positive and negative classes. To compare different ML methods, we computed the most relevant performance metrics for these algorithms: accuracy, precision, recall, and f1-score, according to the definitions: (i) Accuracy: the proportion of correct predictions among all predictions: $accuracy = \frac{TP+TN}{TP+TN+FP+FN}$. (ii) Precision: the fraction of correctly identified positives out of all predicted positives: $precision = \frac{TP}{TP+FP}$. (iii) Recall: the proportion of actual positives correctly identified: $recall = \frac{TP}{TP+FN}$. Finally, (iv) F1-score: the harmonic mean of precision and recall: $F1\text{-}score = \frac{2*TP}{2*TP+FP+FN}$.

After training each classifier on 80% of the samples from the dataset $US-UNV-1$, we evaluated their performance on the remaining 20% and on the other two datasets. The results of this evaluation are summarized in Table 2.2. For the unsupervised $k$-means approach, we combined it with SVM to align it with the classification task and obtain comparable metrics. Despite providing high precision, we can observe how this unsupervised approach performs poorly compared to other supervised alternatives. Among the supervised methods, both Random Forest (RF) and Neural Network (NN) classifiers achieved strong results across all datasets but did not match the accuracy and F1-score of our decision tree (D-Tree) model. Notably, RF performed well on the dataset it was trained on but exhibited reduced generalization to other datasets.

On the other hand, our enhanced D-Tree model delivered consistently strong metrics across all workloads, demonstrating not only high accuracy but also robust generalizability, even when applied to different data center scenarios.

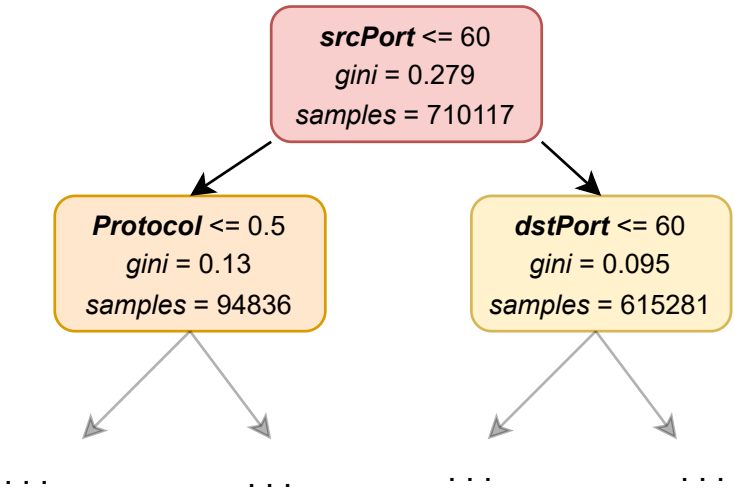



Fig. 2.4 The initial layers of the trained decision tree.

### 2.6.3 Interpreting classification with a decision tree

One reason behind our choice of D-Tree for traffic classification is the rich expressiveness of such models that allows their interpretation [73]. In classification, interpretability refers to understanding and explaining how the model makes predictions. This capability helps users identify the key factors influencing decisions, identify biases, and explain the model's predictions. There are different ways to interpret a classification model, depending on the type of model and the specific techniques used. Focusing on D-Tree models, they can be interpreted by following the path that an input takes through the tree to arrive at a prediction.

We report our D-tree's first levels of depth after having trained it over the training set (80%) of $US-UNV-1$ in Fig. 2.4. The figure shows how the root node, *i.e.,* the node that starts the tree, which initiates the tree, divides it into two branches based on the source port of the transport protocol. This first branch, along with the second level, shows the key features influencing the classification decisions. The decision-making process within the tree is driven by the Gini index, a measure used to assess the quality of the splits. In detail, the Gini index is calculated by subtracting the sum of the squared probabilities for each class at a node from 1, showing the most effective attribute to split on during each step of the learning process [74]. A lower Gini index indicates more uniformity within a node, while higher values suggest greater class diversity. The D-tree aims to reduce the Gini index with each

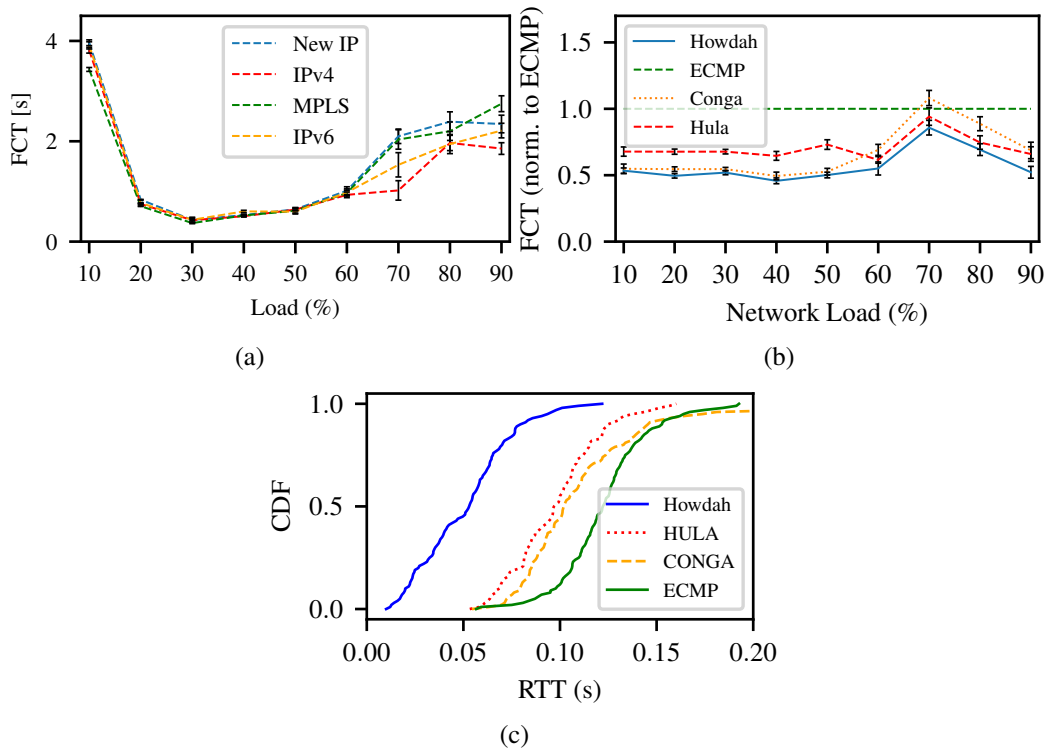



Fig. 2.5 FCT analysis of (a) packet headers, (b) SoTA load profiling techniques (b), and (c) CDF on RTT at 70% network load.

split, improving both the accuracy and interpretability of the model. The figure also shows the number of samples contained within each node. For instance, the root node contains 710,117 samples, representing the 80% of $US-UNV-1$ dataset (with 887,647 samples in total). More in detail, all categorical fields were converted into numerical values for ML processing, as seen in the first level of depth, where the transport protocol (*i.e.,* TCP, UDP) was converted into numerical values (*i.e.,* 0, 1). We can observe how the source and destination ports, as well as the protocol, are the major factors that dictate the decision process. This aligns with rule-based decision-making methods as in [35], but the data-driven approach of the D-Tree offers enhanced adaptability and generalization across various traffic patterns.

### 2.6.4 Packet Header Impact

As discussed in Section 2.3, Howdah is compatible with different protocols that incorporate additional information directly into packet headers. Among them, in

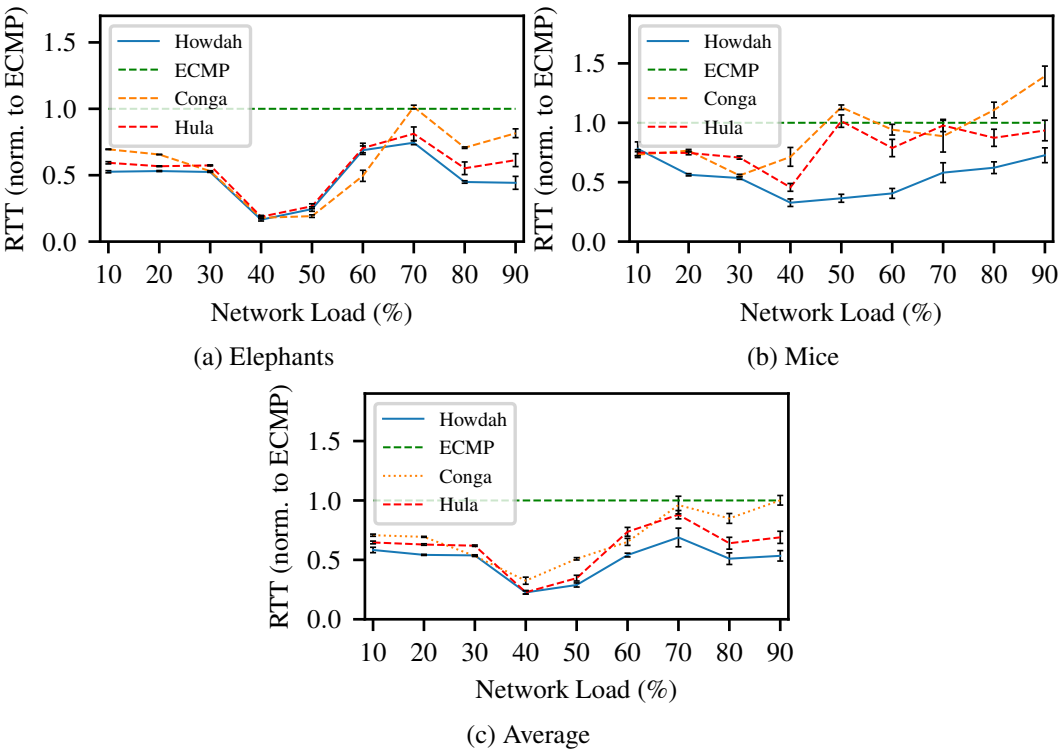



Fig. 2.6 RTT under varying loads for (a) elephant flows, (b) mouse flows, and (c) their average.

this chapter, we mainly focus on IP, New IP, MPLS, and IPv6, even if more options are available. In Fig. 2.5a, we study the diverse header format's impact on FCT for different network loads. FCT, a critical performance metric for congestion, is defined as the time interval from when the first packet of a flow is sent to when the last one is received [75]. The error bars in the graph refer to the 95% confidence intervals.

The first visible advantage of the proposed load profiling technique can be seen when the network load is at 20%. While congestion is negligible at a 10% load, as traffic increases, our method effectively distributes flows across available paths. When network congestion decreases and traffic levels fall below a 50% load, the differences between header formats become minimal, with all approaches bringing similar FCTs. This suggests that the burden introduced by additional bytes in the packet header is minimal. However, at higher traffic loads over 70%, the advantage of using IPv4, with no added header bytes, becomes visible as it achieves the lowest FCT. While other header options remain valid and offer additional flexibility, the analysis shows that IP headers introduce minimal overhead. For this reason, we

adopt the IP header as the default protocol in the following evaluations, as it requires only a minimal modification on the host side.

### 2.6.5 Load Profiling Effectiveness

After evaluating our predictive model and the impact of different packet header formats, we assessed the load-profiling effectiveness in a data center scenario by comparing Howdah against the other benchmark solutions. The 10 servers in Fig. 2.3 send packets to replicate data center workloads as described in [29]. To simulate varying network conditions, we progressively increased the network load by adjusting the number of receiving servers from 1 to 9.

We started by comparing the FCT achieved by Howdah and the benchmark solutions for load-profiling. To ensure consistency, all results were normalized against the ECMP baseline. As shown in Fig. 2.5b, Howdah is able to minimize FCT across all network loads. While HULA showed strong performance under heavy network loads and CONGA performed better at lighter loads, Howdah outperformed both by maintaining the lowest FCT across all scenarios, demonstrating its ability to maintain a less congested network under different conditions.

We then evaluated another critical metric, the RTT, focusing on a network load of 70%. We analyzed the cumulative distribution function (CDF) of RTT to better understand its distribution, especially the tail values. As visible in Fig. 2.5c, our solution not only reduces the RTT on average compared to state-of-the-art but also lowers the RTT of the transmission of the most long-lived packets. In particular, all responses were received within 0.12 seconds of the request, showing the fastest performance among the alternatives.

Moreover, to further generalize our findings, we examined RTT behavior across varying network loads, distinguishing between “elephant” and “mouse” flows. Fig. 2.6a shows the normalized RTT for elephant flows, highlighting that Howdah’s benefits become more visible as network load increases. While CONGA slightly outperformed Howdah for loads between 50% and 60%, it does not handle higher loads effectively. When evaluating RTT for mouse flows (Fig. 2.6b), CONGA’s limitations were even more visible, occasionally performing worse than ECMP. On the other hand, Howdah delivered better performance, particularly for mouse flows, where the improvements were most significant.

Finally, by averaging the results for both flow types (Fig. 2.6c), we observed that under low network loads (10% to 40%), congestion was minimal, and Howdah, CONGA, and HULA achieved similar RTTs. However, as the load increased, Howdah's advantages became more evident. These findings reinforce the FCT results and demonstrate how Howdah's traffic classification and dynamic switch actions lead to better overall performance.

### 2.6.6 Centralized Approach

To evaluate the effectiveness of our distributed schema, we compared it to a centralized approach where an SDN controller takes forwarding decisions. In this situation, while the centralized method allows for more advanced decision-making, it has a slower control loop due to the need for frequent interactions with the controller. For this comparison, we implemented a centralized version of Howdah using OpenFlow [76], one of the most widely adopted SDN protocols. In this centralized setup, packets without predefined routes are forwarded to the SDN controller, which determines their handling based on pre-configured rules. These rules may take into account various factors, such as source and destination addresses, transport layer properties, packet flags, and network conditions. While earlier sections addressed the parameters influencing packet forwarding, in this analysis we focus specifically on the performance of the centralized Howdah, termed Howdah-centralized. This centralized version was developed using the Ryu framework [77] and, as the distributed one, deployed within the Mininet simulation environment. Since OpenFlow switches cannot support hash-based routing [78, 79], the hash calculations and storage were managed by the SDN controller. In particular, the controller running the ML model classifies the incoming flow. When elephant flows are recognized, it checks the flowlet hash table. If an entry exists, the controller installs the corresponding rule on the switch using the destination IP and port as match criteria. If not, it calculates the hash, updates the table, and then pushes the rule to the switch. These flow routes are removed once their prefixed timeout expires.

Fig. 2.7 shows the comparison between the distributed Howdah (implemented in P4) and the centralized version (based on OpenFlow). The evaluation considers flow types (elephants and mice) as well as the overall average performance, using varying network loads and congestion levels generated by the *iperf3* tool. In particular, we

computed the FCT for all flows in terms of seconds at a varying network load and level of network congestion induced with the *iperf3* tool.

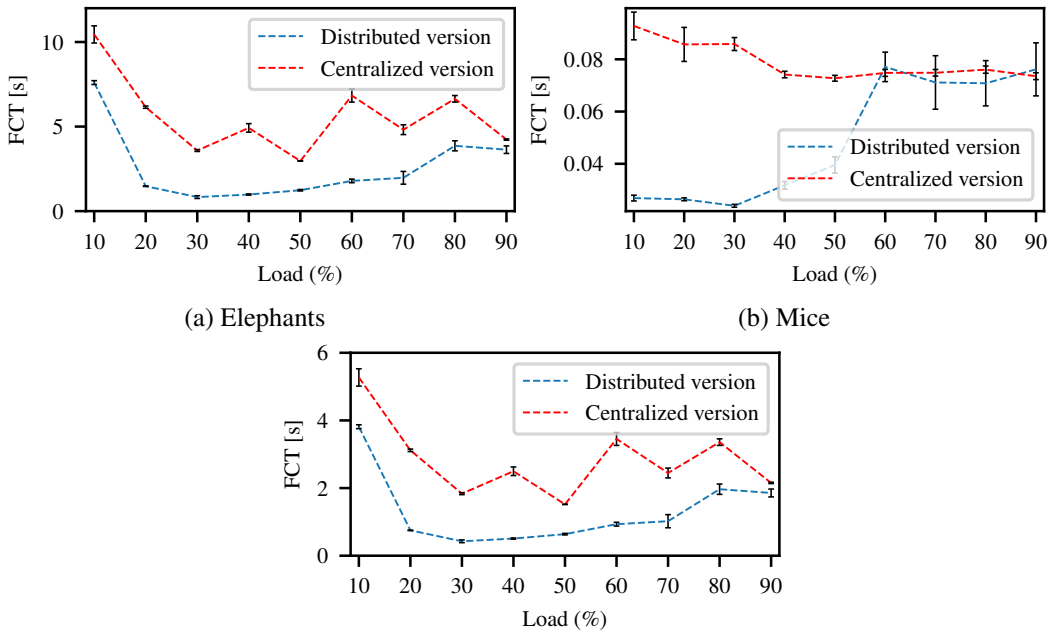
We can observe how our distributed version performs better than the distributed one across all flow types. For elephant flows (Fig. 2.7a), even under low network loads (10%–50%), the controller interaction of the centralized model brings higher FCT. When the network starts to get congested, and the load is greater than 50%, we can see that both versions start to perform similarly, and with a congested network (90%), even a distributed solution achieves almost the same FCT as the centralized one. This behavior is even more visible in the mice flows (Fig. 2.7b) where just as for the elephant flows, when the network is not congested, the interaction with the controller leads to a slower forwarding time and, consequently, a greater FCT. Meanwhile, both versions achieve almost the same FCT when the network becomes more congested.

However, when considering the average FCT across all flow types (Fig. 2.7c), the distributed model demonstrates a significant reduction in FCT and operational overhead compared to the centralized version. These findings confirm the advantages of our distributed design, highlighting the value of host-based flow classification and in-network data-plane forwarding.

### 2.6.7 Howdah in Conjunction with CC algorithms

In evaluating Howdah, we also considered how it behaves when an in-network congestion control (CC) algorithm is present, taking for this experiment the well-known data-center TCP (DCTCP) [29]. DCTCP works by enabling switches to flag packets on the Explicit Congestion Notification (ECN) field whenever the buffer occupancy at a switch exceeds a predefined threshold. These flags serve as signals to the sender about emerging network congestion: the sending host adjusts its transmission rate, scaling it down proportionally to the percentage of marked packets (*i.e.,* the higher the proportion, the greater the reduction in the sending rate).

We compared two implementations under varying network loads: one running Howdah alone and another combining Howdah with DCTCP. The results, shown in Fig. 2.8, highlight the impact of these setups on FCT for different flow types. In Fig. 2.8a, which focuses on large elephant flows, both configurations have almost identical FCTs across all network loads. This suggests that Howdah's load profiling is




(a) Elephants (b) Mice

(c) Average

Fig. 2.7 FCT at increasing network load vs. a centralized OpenFlow version for (a) elephant flows, (b) mouse flows, and (c) the average.

effective enough and able to collaborate with other control mechanisms like DCTCP. Similarly, for smaller mouse flows (Fig. 2.8b), the two approaches show comparable performance. A minor improvement is observed for Howdah + DCTCP between 50% and 80% network load, where it outperforms Howdah alone by a margin of 2.106ms. Overall, as shown in Fig. 2.8c, both implementations perform similarly on average, demonstrating that Howdah performs as a good profiler both alone and in combination with other mechanisms.

### 2.6.8 Resource Consumption

Finally, we consider the impact of the traffic classifier on the host machines. One of the challenges faced by the design and implementation of Howdah is the efficiency in terms of processing time, especially onboard host machines, which are typically running resource-consuming processes. A lightweight yet accurate classifier is thus essential. To this end, we study the memory and CPU usage of different ML models during both the training and execution phases, reporting results in Fig. 2.9. The

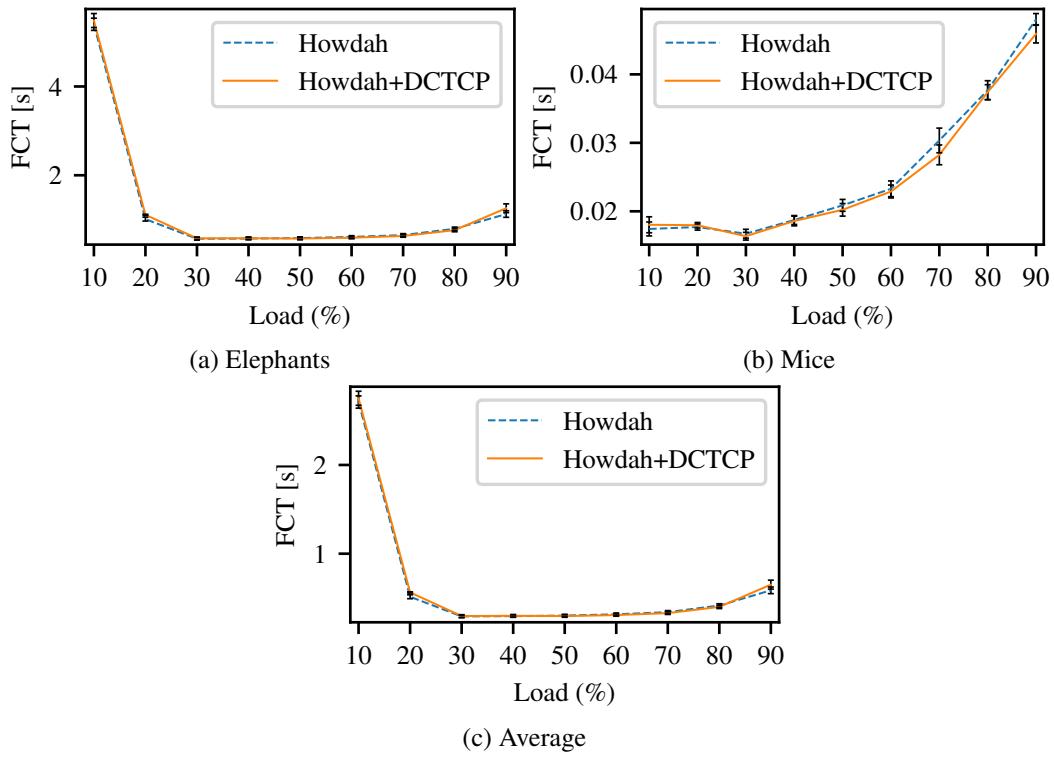



Fig. 2.8 FCT under varying load is compared for DCTCP coexistence vs. Howdah-only for (a) elephant flows, (b) mouse flows, and (c) average.

considered machine featured a 2.6 GHz 6-core CPU and 16GB RAM. We can observe that, while our D-tree classifier shows the the highest RAM consumption (Fig. 2.9a), the memory required remains minimal and is also available even on devices with limited resources. Regarding CPU usage (Fig. 2.9b)the D-tree model performs comparably to other algorithms and demonstrates significantly lower consumption than the Neural Network model. These results confirm that even without specialized hardware, the D-tree's computational requirements align well with Howdah's design goals, supporting its deployment on host machines.

In addition to memory footprint and computation resources, we analyzed the ML models' overhead in terms of training and execution (Table 2.3). We report the time, $CO_2$, RAM, and CPU for both training and classification processes, where the training occurs over 80% of the $US-UNV-1$ dataset and the classification over the remaining 20% of it. Classification time refers to the time taken to process and classify an unknown flow before sending it. As shown in Table 2.3, the Decision Tree, similarly to k-means and SVM, requires a limited training time, while neural

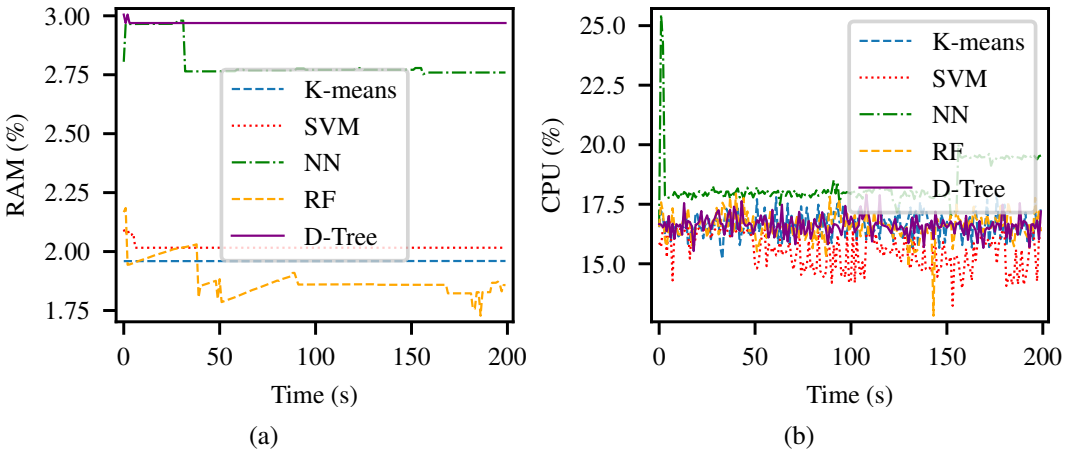



Fig. 2.9 (a) RAM usage (%) and (b) CPU utilization of classifiers during host-side execution.

| | Tr. time [s] | Class. time [$\mu$s] | $CO_2$ (T) [mg] | $CO_2$ (C) [mg] | RAM (T) [%] | RAM (C) [%] | CPU (T) [%] | CPU (C) [%] |
|---|---|---|---|---|---|---|---|---|
| **SVM** | 0.927 | 0.0264 | 1.933 | 0.0417 | 2.013 | 2.022 | 15.554 | 14.523 |
| **NN** | 170.869 | 4.081 | 416.6 | 1.8391 | 2.766 | 2.758 | 18.328 | 18.207 |
| **k-means** | 0.647 | 0.07974 | 2.013 | 0.0728 | 1.959 | 1.954 | 16.621 | 15.583 |
| **RF** | 83.439 | 10.562 | 192.4 | 3.1312 | 1.872 | 1.851 | 16.58 | 16.983 |
| **D-Tree** | 1.0202 | 0.107 | 1.733 | 0.0481 | 2.969 | 2.93 | 16.619 | 16.506 |

Table 2.3 Training (T) and Classification (C) resource usage of analyzed ML models.

networks and RF take significantly longer to train. Moreover, the Decision Tree's classification time is negligible, making it an ideal choice for real-time flow processing. On the contrary, despite its accuracy, RF exhibits slower performance during classification.

Concerning the energy efficiency of the models, different studies [80, 81] have proven that training ML models are highly polluting, especially when there are many parameters and massive datasets. To quantify this for our models, we used the CodeCarbon Python library [82] to estimate carbon emissions in milligrams of $CO_2$. This library calculates emissions based on the product between the carbon intensity of electricity (gCO2/kWh) and the power consumed during computation (kWh), considering also the carbon intensity specific per country. The data shows that models with longer training times, such as NN and RF, result in higher carbon emissions. In contrast, the Decision Tree model is one of the most environmentally friendly options, with $CO_2$ emissions comparable to SVM. Quite surprisingly, RF exhibited the highest emissions even during classification, exceeding those of deep learning models. This is likely due to the complexity introduced by managing multiple trees. Lastly, when averaging RAM and CPU consumption across models,

the findings align with Fig. 2.9, where the RAM usage of D-tree is the highest among alternatives but still limited, and the CPU is comparable to benchmarks.

## 2.7 Conclusion

This chapter presented Howdah, an innovative in-band load profiling solution centered on the collaboration between the host and network switches. The host leverages a specialized machine learning model, specifically a Decision Tree, to classify outgoing traffic and embed this classification directly into the packet header. On the other hand, switches, programmed using P4, make forwarding decisions by considering both the traffic type and real-time network conditions. By enabling switches to make local, per-packet decisions, Howdah ensures resilience to link failures and dynamic adaptation to changes in network topology. Throughout the chapter, we also explored possible protocols that can be used to include in-band information about the ongoing traffic type. Experimental results show that Howdah outperforms state-of-the-art techniques, particularly under high network loads, by significantly reducing both Round-Trip Time and Flow Completion Time. Moreover, the lightweight nature of the Decision Tree model minimizes system overhead, validating our design of delegating the classification task to the host process. In the future, in order to improve load profiling, traffic classification, and forwarding decisions further, we plan to explore new strategies that use a more fine-grained classification for a multi-class classification that identifies more applications (*e.g.,* video streaming, interactive call) or incorporating additional data within the packet beyond the traffic type classification.

# Chapter 3

# Adaptive Routing via Multi-Agent Reinforcement Learning

## 3.1 Introduction

In recent years, the rapid rise of new applications has significantly increased the demands on communication frameworks, particularly on emerging technologies like 5G and 6G, while also presenting complex challenges for Internet infrastructure. Each of these applications imposes its own stringent requirements concerning factors such as latency, jitter, bandwidth, and packet loss. As networks evolve, finding effective routing methods becomes crucial.

A notable trend is the integration of Machine Learning (ML) and Deep Learning (DL) techniques in routing with the aim of leveraging information about past traffic conditions to optimize routing strategies for future scenarios [84]. The adaptability of Software-Defined Networking (SDN) offers both reactive and proactive network management, allowing SDN controllers to monitor network conditions closely and adjust according to traffic changes [85, 50].

Among ML approaches, Reinforcement Learning (RL) stands out as particularly well-suited for routing due to its capability to learn and improve autonomously, seeking to find the optimal routing policy [86]. In recent research, RL and Deep

The work presented in this chapter has been partially published in [83].

Reinforcement Learning (DRL) have been widely explored within SDN environments to enhance routing efficiency [87, 88].

However, centralized controller-based approaches struggle to keep up with rapid and fine-grained traffic changes, such as traffic bursts. Even when routing decisions happen locally on switches, the options are often limited and not fast enough [89].

To overcome these limitations, programmable data planes have recently emerged as a solution, with various studies developing mechanisms that operate directly within the data plane for real-time adaptability [90, 91]. These solutions can deliver considerable performance benefits over more static mechanisms and centralized approaches using fine-grained performance information on hardware timescales. However, these techniques tend to rely on performance-based policies that lack the ability to adaptively learn, making them less effective in handling diverse traffic scenarios. Moreover, as the complexity of the network grows, finding optimal routing strategies to manage congestion and enhance performance is becoming both critical and challenging.

To bring autonomous routing directly to network devices, we designed the Reinforcement Learning for Autonomous Routers (ROAR) system. Using P4-programmable switches [3] and general network programmability, ROAR enables distributed, intelligent routing decisions via Deep Reinforcement Learning (DRL).

In this setup, each network switch acts as an agent in the DRL system, using the algorithm to choose the best forwarding port for each incoming packet. This decision is based on two key factors: the next hop toward the packet's destination, determined by the network's known topology, and the port's outgoing queue, to be minimized. By continuously learning from their environment, the switches train the model periodically, allowing it to evaluate the impact of different routing choices on performance and make decisions based on learned routing strategies.

Designing a DRL model with P4 is known to be challenging since the architecture does not support loops, complex arithmetical operations, or if-else conditions in action blocks, which are essential for the DRL algorithm. To overcome this limitation, we modified the P4-16 compiler to integrate with an external C++ module, which serves as an intermediary between the P4 application and the DRL algorithm. This module acts as an ML-based network controller, enabling dynamic packet-forwarding decisions based on real-time network conditions. It interacts with the Behavioral Model (bmv2) via socket communication, maintains an RL-based reward system,

and manages MAC address mappings and buffer states. Additionally, it facilitates communication with external Python scripts for the DRL analysis. This extension enhances the adaptability of the P4 pipeline by incorporating intelligent control mechanisms, without requiring deep modifications to the core P4 program [92].

We evaluated our solution on an emulated network over Mininet, showing that when the network starts being congested, the benefits of ROAR can be observed in the increment of throughput and delay reduction.

## 3.2 Related work

With the increasing number of connected devices and more demanding applications, *e.g.,* Tactile Internet, meta-verse, the volume of data traffic flowing through networks has grown exponentially. This has created a pressing need for efficient routing mechanisms to optimize network performance, reduce latency, and ensure reliable data transmission. Researchers are responding to this challenge by proposing different automated and reactive approaches, usually combining ML techniques, such as RL and DRL, to predict traffic for load balancing and routing optimization [89].

One example is QR-SDN [93], where the authors use tabular RL (Q-learning) techniques to reduce network latency by optimizing multipath routing. An SDN centralized controller routes packets using a flow-preserving strategy that aims to minimize the latency of transmissions. Another study is RSIR [94], which implements a knowledge plane linked to the management plane to store data, enabling the SDN centralized controller to determine the shortest routing paths and balance network load. The study compares its results to those achieved with the traditional Dijkstra algorithm.

As the authors suggest, adopting a centralized controller with a global view brought a low response time in case of topology changes. However, while this approach works effectively in smaller networks, larger ones with high volumes of flows experience issues. The frequent interactions with a centralized controller can lead to reduced throughput and increased delays, as it struggles to maintain reliable packet transmission at scale.

Over the last few years, Deep Reinforcement Learning (DRL) methods have also been quickly adopted in many fields of networking, from the Internet of Things

| | Distributed-only Implementation | Internal-only Execution | Without Manual Intervention |
|---|---|---|---|
| QR-SDN [93] | ✗ | ✗ | ✓ |
| RSIR [94] | ✗ | ✗ | ✓ |
| DROM [95] | ✓ | ✗ | ✗ |
| SmartCC [96] | ✓ | ✗ | ✓ |
| **ROAR** | ✓ | ✓ | ✓ |

Table 3.1 Comparison of related works with ROAR.

(IoT) to concurrent multipath transfer data scheduling, mobile-edge computing, and heterogeneous networks, focusing especially on routing optimization and congestion control [87]. One example is DROM [95], a routing optimization system based on deep policy gradient algorithms, which uses neural networks to boost network performance. However, this approach requires manual intervention for customizing network strategies and maintaining the reward function [89].

Routing optimization mechanisms were also essential in heterogeneous networks where various devices with different capabilities, such as cell phones, laptops, and tablets, are connected, and efficient resource utilization was vital to avoid delays and maximize the throughput. An example of an application in a heterogeneous network is SmartCC [96], a multipath congestion control approach based on DRL where an asynchronous RL framework learns a set of congestion rules and adapts the congestion windows accordingly. Several attempts to alleviate network congestion and balance network load utilizes DRL methods running without [97] or with an SDN controller, such as DRLS [98] and IQoR-LSE [99], where great focus is brought to the collection of network measurements.

Although programmable switches have enabled some computation to shift within network devices (*e.g.,* [100, 101]) to the best of our knowledge, there is no current routing solution that runs a data-driven DRL algorithm directly on the device yet. As illustrated in Fig. 3.1, unlike centralized systems that encounter the issue with data collection, we introduce a routing optimization mechanism designed as a flexible, general solution that can adapt to various network setups and any P4-compatible switch. We also summarize in Table 3.1 the main differences between ROAR and the mentioned state-of-the-art solutions.

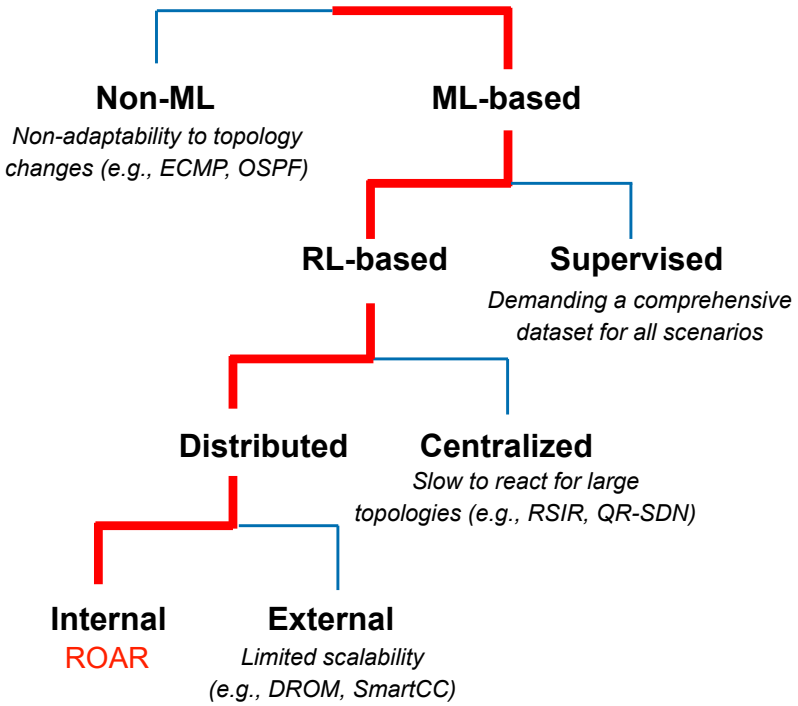



Fig. 3.1 Adaptive routing techniques highlighting the difference between ROAR and other relevant studies.

## 3.3 System Architecture and Components

In this section, we describe the design of our solution, as well as the principles behind this definition. As represented in Fig. 3.2, ROAR revolves around leveraging P4 switches to control the forwarding plane directly inside the network device, adopting forwarding decisions according to the outcome of a deep reinforcement learning approach. Each switch of our network is composed of three main blocks: $(i)$ the P4 application, which constitutes the logic of the network device, $(ii)$ the Deep Reinforcement Learning (DRL) module, responsible for running the learning algorithm to route the packets, $(iii)$ the Inter-process communication (IPC) module, which is composed of the high-level modules and data structures needed to connect the P4 application to the DRL module.

### 3.3.1 DRL Module in ROAR

In ROAR, every switch of the network is an agent of the DRL, making the solution a Multi-Agent Reinforcement Learning (MARL) [102] scenario that applies forwarding decisions according to the result of the DRL model in an environment where other agents are also trying to reach the same goal. We consider a system of $M$ agents (routers) within a shared environment without a centralized controller responsible for gathering rewards or making decisions on behalf of the agents. In this setup, the collection of agents is represented as $\mathscr{M}_t$, where $M$ denotes the cardinality of the set, and every agent has the capability to communicate with all other agents. In particular, the composition of the agent set can possibly vary over time (*e.g.,* failures) and is defined as $\mathscr{M}_t$ at a given time $t \in \mathbb{M}$.

The time-varying MARL process described in ROAR is defined as a tuple $\langle \{S^i\}_{i\in\mathscr{M}}, \{A^i\}_{i\in\mathscr{M}}, P, \{R^i\}_{i\in\mathscr{M}}, \{\mathscr{M}_t\}_{t\geq 0}\rangle$, where $S^i$ denotes the local state space of agent $i$ in $\mathscr{M}_t$, and $A^i$ is the action set that agent $i$ can execute. Besides, $A = \prod_{i=1}^{M} A^i$ is the joint action space of all agents, also referred to as the global action profile. We then proceed by defining the local reward function of agent $i$, denoted as $R^i : S \times A \to \mathbb{R}$, and the state transition probability function $P : S \times A \times S \to [0,1]$. In this setup, we assume that the states and actions have a global impact but are locally observable, as well as the rewards, which are only observed locally. At each time step $t$, given the state $s_t \in S$ and the joint actions of the agents $a_t = (a_t^1, \ldots, a_t^M) \in A$, each agent receives an individual reward $r_{t+1}^i$. This reward is given by an equation that captures the incentive that the learning model wants to model and is determined by $R^i_{(s_t,a_t)}$. Additionally, the system transitions to a new state $s_{t+1} \in S$ with a probability of $P(s_{t+1}|s_t, a_t)$.

Our model is entirely decentralized, with each agent acting *independently* by receiving local rewards and making its own decisions. Different from a centralized SDN scenario, where the controller has a global view of the network, our approach distributes decision-making across the network. The key idea is to train each agent separately, minimizing the need for coordination between routers and reducing the overhead of frequent updates. Routers only exchange basic topology information to build a shared virtual view of the network. In this scenario, agents operate independently without sharing network state details or model parameters with each other.

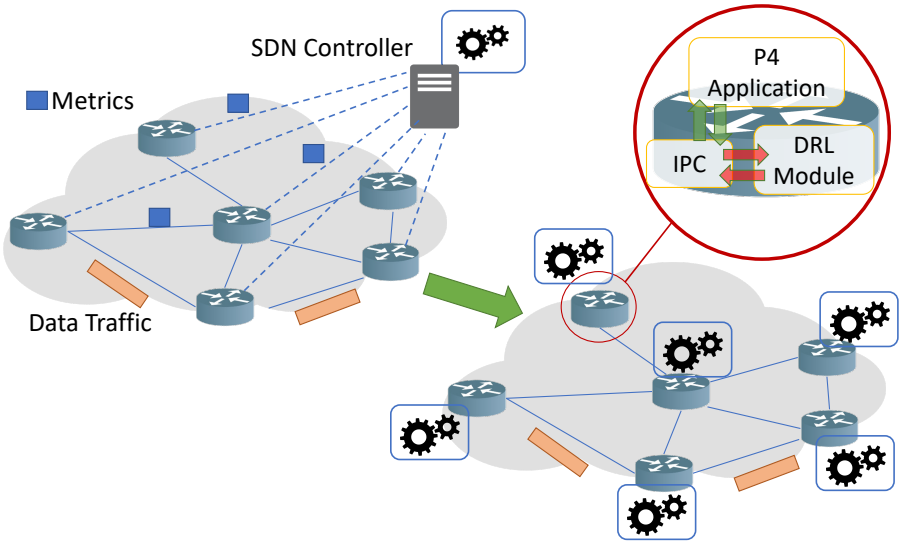



Fig. 3.2 ROAR's Overview and Components.

In each ROAR's agent, $i$, the *action* $A^i$ is a discrete number ranging from 1 to $N$, where $N$ is the number of ports the switch uses. The *state* $S^i$, instead, is composed of three elements: $(i)$ current destination, which is the destination IP address of the packet being processed, $(ii)$ future destinations, which is a list of $L$ next packet's destination IP addresses that follow the same route as the current one, $(iii)$ action history, which is a list of the last $k$ actions adopted for the current packet's destination.

Every time a given action for a certain state has been performed (*e.g.,* a packet has been forwarded towards a specific port), the reward function is evaluated to update the expected cumulative reward (Q-value) for that state-action couple. This function considers two primary factors: $(i)$ queuing time, which is the time every packet has spent in the output queue, and $(ii)$ the distance of the chosen next hop from the final destination. While the goal is to minimize queuing times, the distance of the next hop from the final destination is the only information the agent knows about the global topology. The *reward* function $R^i$ also considers two indicators: delivered, $\sigma_1$, set to 1 if correctly routed, 0 otherwise; and dropped, $\sigma_2$, set to 1 if the packet has been dropped, 0 otherwise. Their value is always set to 0 in the case of spine switches, as they are not directly connected to any destination host.

Summarizing, the reward function for each agent $i$ is:

$$R^i = \lambda_1 * \sigma_1 - \lambda_2 * q - \lambda_3 * \sigma_2 - \lambda_4 * \sigma_3 * d \tag{3.1}$$

where: $(i)$ the $\lambda$ values are the model's hyper-parameters set during the training to check the performance of the algorithm, $(ii)$ $q$ is the time that the packet has spent in the queue before being sent, $(iii)$ $\sigma_3$ is a parameter indicating whether the switch is a spine one and is multiplied by $d$, *i.e.,* the distance to the destination switch.

**Our Neural Network.** In ROAR, we determined the optimal number of input features for our NN (*i.e.,* the state space), by computing their mean reward and selecting the list length for which they achieve the highest value. After testing different settings, we chose to input the last two taken decisions (*i.e.,* the lengths for the "future destination" and "action history" states). Being categorical features, they must be converted into numerical values. To this goal, we used the One-Hot Encoding, which employs *dummy* variables to represent categories and has shown stronger performance than other encoding methods based on the precision-recall area under the curve (PR-AUC) metric [103].

With these encoded features prepared, they serve as inputs for our Neural Network model. We tested several NN structures on both leaf and backbone switches, finding that an architecture composed of 3 hidden layers of 128, 64, and 32 neurons can achieve the best performance.

It is important to note that our DRL algorithm does not run every time a packet arrives at a switch, as the processing overhead would be excessive. The resulting trade-off is to adopt a static routing guided by the DRL algorithm by means of periodical updates. Our tests showed that updating the route to a specific destination for every $10,000$ packet achieved the best network performance.

### 3.3.2 P4-based Actions

When a ROAR's router receives a packet, it performs two main operations: $(i)$ inserts the current packet's IP destination address inside the "future destinations" data structure, accessible by the IPC module, $(ii)$ chooses the output port given by the DRL module.

The egress module carries out two additional tasks: $(i)$ it forwards packets based on a First-In First-Out (FIFO) criteria and removes the destination IP from the "future destinations" data structure, and $(ii)$ it interacts with the IPC module to compute the reward for that forwarding decision, based on the time the packet spent in the outgoing queue.

Due to P4 limitations that prevent implementing any ML method inside the default P4 compiler, we customized it to enable the use of *"extern"* instances, which allow external methods to be integrated outside the P4 program [92]. While the P4 program can reference these extern objects and pass input to them, the inner workings of the objects remain hidden from the P4 program. This approach simplifies the separation between the control and data planes, enabling the P4 code to focus on packet handling logic, while the extern objects handle the specific, hardware-level interactions.

### 3.3.3 IPC module

In ROAR, each switch is equipped with an IPC module that acts as an intermediary between the P4-based network application and the DRL algorithm. This module facilitates communication by allowing the P4 application to send packet counters to the DRL system and enabling the DRL to communicate chosen actions (*i.e.,* the next hop) to the P4 forwarding plane.

As mentioned previously, our IPC module also handles preprocessing and data transformation to make information compatible with the DRL algorithm, particularly for the Neural Network model and monitors system performance to provide feedback for the reward function. More in detail, our IPC module is implemented using a socket abstraction in C++ to create a communication channel for exchanging data with the DRL module.

## 3.4 Evaluation

This section describes our experimental settings and results obtained using a virtual testbed like Mininet, with a focus on comparing ROAR to a conventional routing protocol and a centralized SDN solution.

### 3.4.1 Evaluation settings

To validate ROAR's benefits, we used Mininet to emulate a topology composed of 10 servers connected to their switches, which are consequently connected to other 4

switches, in a leaf-spine fashion, with all the links of the topology with 100 Mbps bandwidth. Furthermore, we adopted *iperf3* for performance measurements.

Each server of the network sends packets to any other server, varying the number of receiving servers (from 1 to 9) and replicating the workload as described in [29]. For each network load, we then computed the average of the obtained results, *i.e.,* Round Trip Time (RTT), Flow Per Second (FPS), throughput and packet loss, and drew the two-tailed confidence interval at 95%.

For the context of this work, we compare our results against two alternatives: a traditional routing protocol implementation as OSPF, which uses longest-prefix-match tables to route packets independently of the current network load; and a centralized SDN solution, QR-SDN [93], that routes packets according to a tabular reinforcement learning (RL) algorithm.

### 3.4.2 Random Traffic Generation

To evaluate our solution's performance, we generated network traffic using iperf3, which allowed us to manage network congestion levels and compare ROAR's results with those of QR-SDN and a standard OSPF routing implementation (Fig. 3.3).

As shown in Fig. 3.3a, under low load (10% to 20%), the network is not congested and QR-SDN achieves the lowest RTT, while ROAR and OSPF perform similarly. This might be because ROAR uses DRL to choose the optimal route and, for every $10,000$ packet, the IPC module interacts with the DRL one to retrieve information about the best port to forward the packet. This interaction can indeed impact the network performance, decreasing the overall throughput. However, as the network load increases (20% to 40%), the difference becomes more visible with OSPF, while QR-SDN still achieves a lower RTT than ROAR. When our network is highly congested (from 50% to 90% of network load), we can clearly identify the benefits brought by ROAR, as the RL-driven approach enables it to select paths that minimize congestion and reduce RTT effectively. At the same time, QR-SDN's interactions with an SDN controller begin to negatively affect the network performance.

Throughput results, shown in Fig. 3.3b, show a similar behavior. Under low congestion (10% to 20%), ROAR performs comparably to the traditional OSPF, while QR-SDN initially achieves higher throughput. However, as the network load

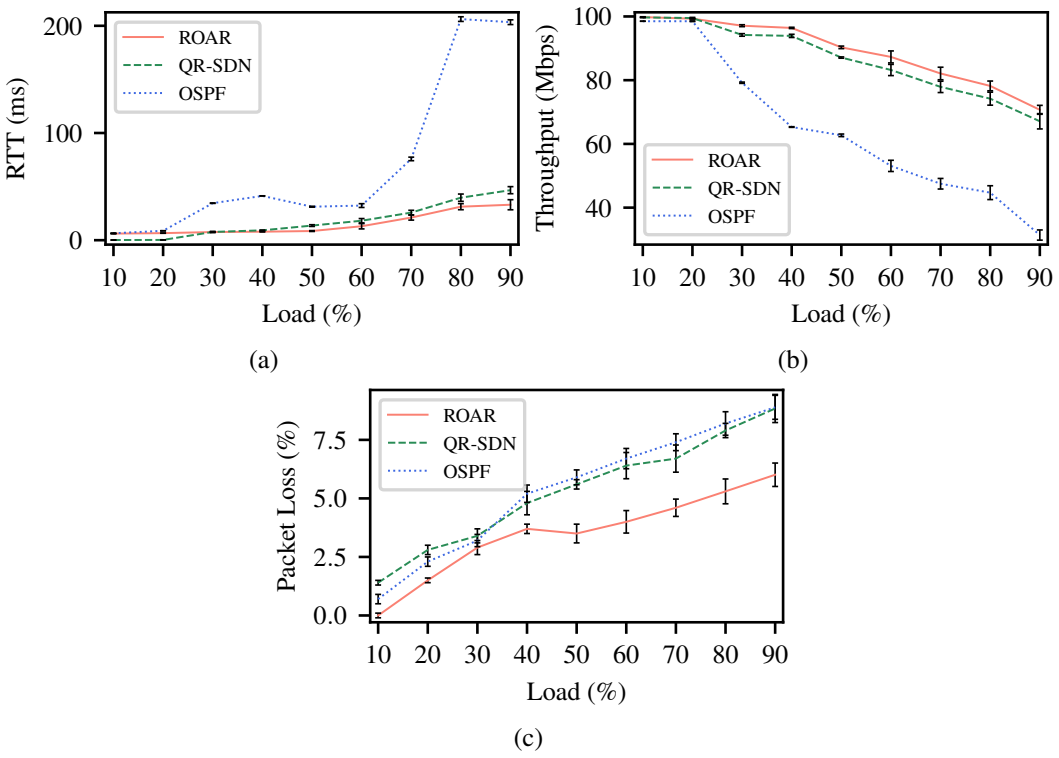



Fig. 3.3 ROAR, OSPF, and QR-SDN comparison for (a) the evolution of RTT, (b) throughput, and (c) packet loss as network load varies.

grows, ROAR begins to outperform both alternatives, showing its ability to adapt to increased traffic and manage congestion effectively.

Packet delivery is another relevant metric when evaluating a solution, as it is an indicator of how well the network responds to the implemented strategy. High packet loss indicates poor transmission quality and necessitates packet retransmissions, which reduces network efficiency. For this reason, we report in Fig. 3.3c the packet loss comparisons between ROAR, QR-SDN, and OSPF when using UDP, chosen to isolate routing effects from those of TCP congestion control, which could otherwise interfere. As visible from the figure, ROAR consistently minimizes packet loss compared to the other solutions, particularly under higher congestion. By promptly reacting and adapting to the next hop, ROAR significantly reduces packet loss, a critical factor for delay-sensitive applications where retransmissions must be minimized to maintain performance.

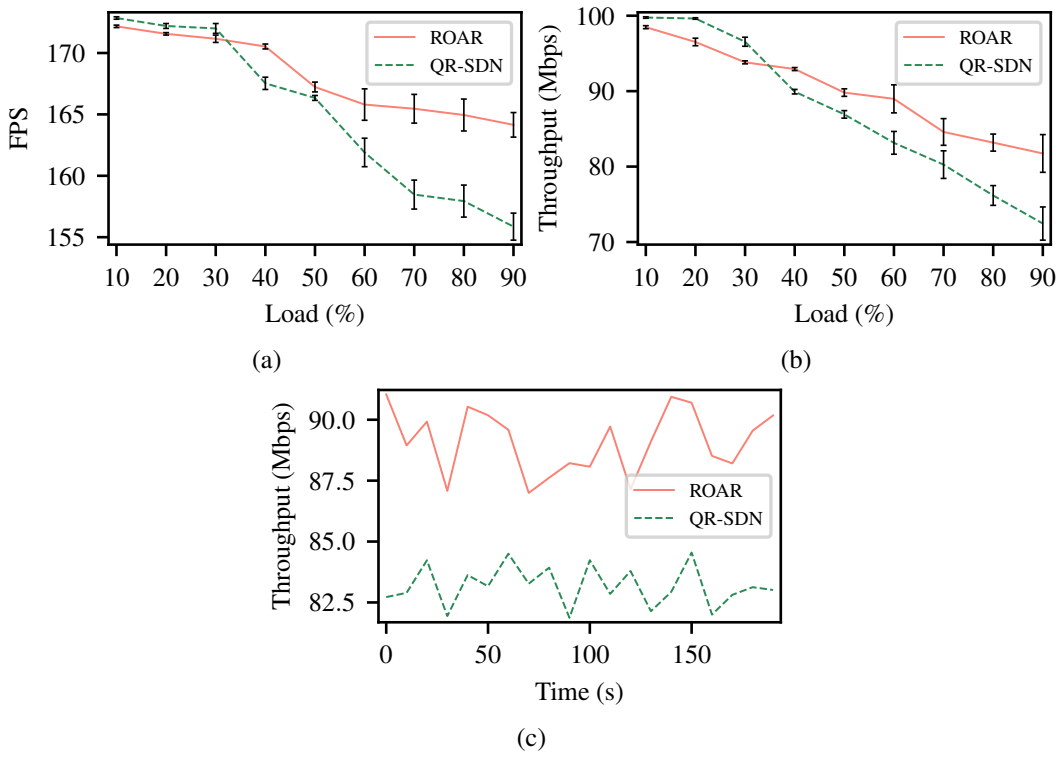



Fig. 3.4 ROAR and QR-SDN comparison at realistic traffic conditions for (a) FPS and (b) throughput as network load varies. (c) Throughput over a 200*s* period with network load at 60%.

### 3.4.3 Trace-based Evaluation

To evaluate our solution under a realistic workload scenario, we used captures taken from publicly available datasets [72] and replayed them in our network using the well-known *tcprelay* tool. The file, representing traffic collected from a data center, was analyzed to replicate the flows in our topology by adjusting IP addresses as necessary. We reported in Fig. 3.4 the results of our evaluation.

This result aligns with prior observations: as network load grows, the RL module in ROAR adapts effectively, selecting more optimal paths.

We start in Fig. 3.4a by counting the FPS that the network can handle while running the file capture. When our network is not congested, and the network load is low (10% to 30%), QR-SDN manages to achieve higher FPS than our solution. However, as the network load increases (from 40% and up), ROAR demonstrates better performance, maintaining a high FPS rate even with a congested network.

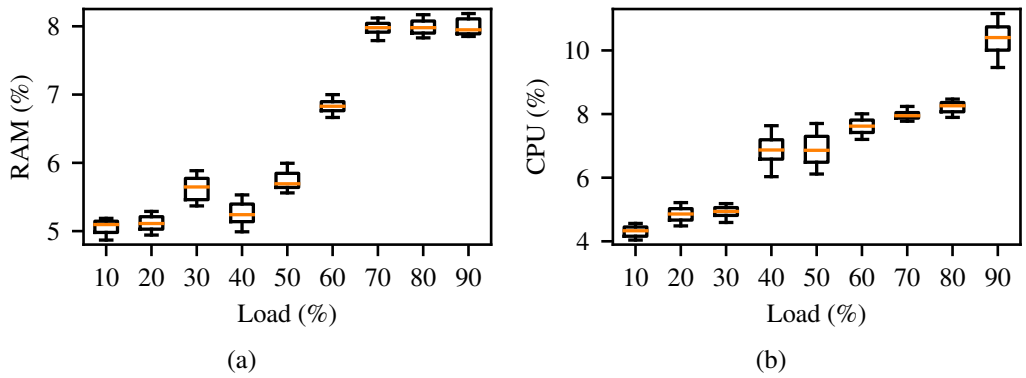



Fig. 3.5 (a) Memory usage and (b) CPU consumption as a percentage of the total available resources on a standard Intel Tofino switch.

This result aligns with prior observations: as network load grows, the RL module in ROAR adapts effectively, selecting more optimal paths.

A similar behavior is visible when evaluating the throughput in both our solution and QR-SDN (Fig. 3.4b). While QR-SDN achieves higher throughput at lower network loads, ROAR is able to outperform when the network is fully congested.

Finally, we considered the first 200 seconds of execution at a fixed network load of 60% and reported the result in Fig. 3.4c. Throughout this period, ROAR consistently delivers better throughput than QR-SDN, further validating our earlier results. These results are extremely important to assess the validity of local performance-aware routing not only with synthetic traffic, but also with more realistic traffic patterns.

### 3.4.4 Can ROAR run over real switches?

Aware of the impact on resource consumption that a DRL model might cause when implemented on physical switches (*e.g.,* FPGA, Tofino), we computed the RAM and CPU usage of ROAR at a varying network load. To do so, we took as reference the X308P-48Y-T programmable switch [104], combined with its embedded Data Processing Unit (DPU), proportioning the results to its computing power. We reported the results in Fig. 3.5. In Fig. 3.5a, we can see how our solution consumes a low amount of RAM when the network load is low, and the network is congested. This amount increases only up to 2% at the highest load (70%-90%). The same behavior is visible in Fig. 3.5b, where the CPU consumption only increases at high

network loads. These tests prove that, despite the DRL module, the IPC module interaction, and the NN algorithm, ROAR requires small hardware resources, making it suitable to deploy on real programmable switches.

## 3.5 Conclusion

In this chapter, we introduce ROAR, a distributed ML-based solution that leverages Deep Reinforcement Learning (DRL) to enhance network routing by conducting computations directly within P4-programmable switches. This approach enables us to take into consideration the current link load, allowing packets to be rerouted dynamically along less congested paths. In the experimental results, we compared our solution to a traditional routing implementation and a centralized SDN solution. The results indicate that ROAR's decentralized architecture reduces RTT even under heavy network congestion and significantly improves throughput, especially at high load levels.

# Chapter 4

# Adaptive Routing via Distilled Decision Trees

## 4.1 Introduction

The programmability of the control and data planes has enhanced the efficiency of various network management services. Classical problems, such as traffic routing or congestion control, and recent ones, such as traffic prediction/classification or anomaly detection, find new ways of being deployed, leading to improved network performance and an easy way of managing network devices for the operators [106].

A growing trend in this field is the application of Machine Learning (ML) and Deep Learning (DL) models to solve these problems, with this chapter focusing specifically on routing solutions [50, 107]. Such data-driven approaches leverage historical traffic data to identify optimal routing strategies for present or future network flows [84, 85, 50].

In particular, different routing optimization strategies leveraging Reinforcement Learning have been introduced [107, 108, 87, 88, 109]. For example, QR-SDN [93] shows how Q-learning can be applied to minimize network latency effectively by optimizing multipath routing in a Software-Defined Networking (SDN) environment. Similarly, RSIR [94] introduces a collaborative knowledge plane within an SDN

The work presented in this chapter has been partially published in [105].

framework that determines the shortest routing path and allows the distribution of network load according to the link-state information representing the network's state.

While these centralized SDN solutions have strong conceptual foundations, their interaction with centralized controllers limits their responsiveness in case of traffic changes, reducing adaptability in dynamic conditions.

To address this, modern approaches leverage SDN programmability and data-plane flexibility to run computations directly on network devices, such as Switch-ASICs, network interface cards (NICs), FPGA-based devices, and P4-enabled switches [3]. This paradigm, known as *In-Network Machine Learning*, executes ML tasks, both training or inference, within the network itself. Such systems can meet the demands of increasing traffic by operating at a line rate and processing all traffic or specific subsets as required. This approach is promising since network devices are already part of the infrastructure and naturally handle data streams suitable for ML-based inference. We summarize in Table 4.1 the main differences between ART and the mentioned state-of-the-art solutions.

However, the potential of programmable switches is limited by current hardware constraints, restricting their use to either extracting critical features for models [110] or deploying pre-trained models for inference tasks [111]. To the best of our knowledge, no available solution can run adaptive ML-based routing decisions directly within network switches [112].

To address this limitation while also meeting hardware constraints, such as those of P4 switches, we introduce a novel approach called *Adaptive Routing with Trees (ART)*. ART leverages Deep Reinforcement Learning (DRL) to optimize traffic routing by learning from real-time network conditions. The algorithm leverages the telemetry capabilities of P4 switches to make routing decisions based on real-time network utilization. Measurements collected through P4 are provided to the DRL model, enabling dynamic adjustments to the routing tables.

This distillation process uses a teacher-student methodology, transforming the complex decision-making capabilities of the DRL's neural network into an interpretable and computationally efficient format suitable for hardware constraints.

The DRL model, once trained, is distilled into a lightweight Decision Tree (DT) that can be easily translated into flow rules compatible with P4 switches. This distillation process follows a teacher-student approach, transforming the complex

| | Distributed Execution | In-Network ML | Data-center Optimized |
|---|---|---|---|
| QR-SDN [93] | ✗ | ✗ | ✗ |
| RSIR [94] | ✗ | ✗ | ✓ |
| **ART** | ✓ | ✓ | ✓ |

Table 4.1 Comparison of most recent related works with ART.

decision-making capabilities of the DRL's neural network into an interpretable and computationally efficient format suitable for hardware constraints, *i.e.*, the Decision Tree. In particular, every P4 switch runs a learning agent that interacts with the DRL model to determine the appropriate forwarding port for incoming packets, based on the packet's IP destination and the real-time utilization of the switch ports. Experimental evaluations conducted on an emulated Mininet network confirm that ART effectively reduces congestion by dynamically adapting routes while minimizing the additional overhead. In turn, ART reduces packet delays and increases the throughput while minimizing packet losses.

## 4.2 System Design

A recent innovation in networking involves automatically determining packet routes using real-time traffic data. This process often employs reinforcement learning (RL) to find optimal paths for source-destination pairs by using only a few link-state metrics (*i.e.*, available bandwidth, loss, and delay) as inputs for the learning process. While RL-based routing offers many benefits, such as automation and adaptability to dynamic network conditions, its implementation is challenging on programmable switch architectures. To address these limitations without losing the advantages of RL-based routing, we have developed ART, represented in Fig. 4.1.

As shown in the figure, ART is composed of three main components: *(i)* an SDN controller, used to interact with P4-enabled switches, *(ii)* a Deep Reinforcement Learning (DRL) module, which implements the learning algorithm required for packer routing, *(iii)* a Decision Tree (DT) module that uses the trained DRL to generate routing rules, which are then incorporated into the P4 code.

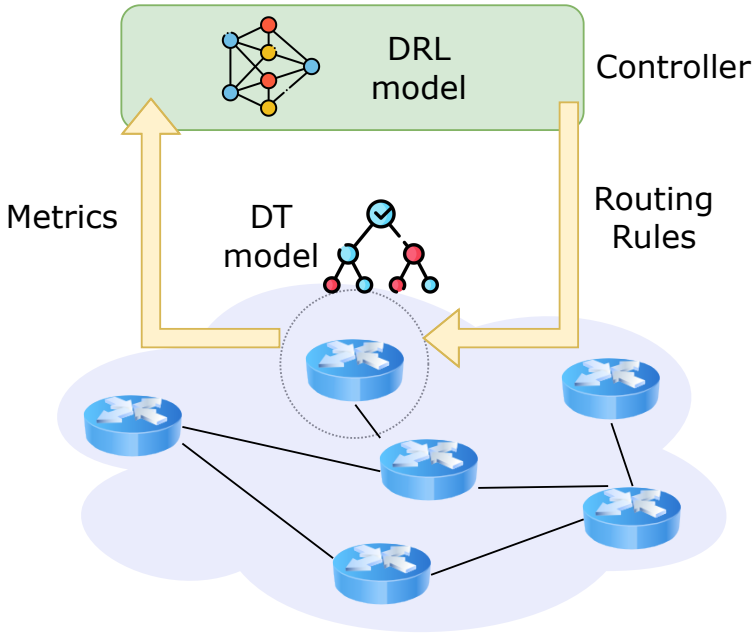



Fig. 4.1 ART's Overview and Components.

### 4.2.1 Switch-Controller Interaction

P4 provides the flexibility to specify how packets should be parsed, matched against rules, and modified, enabling precise control over the behavior of network devices. Despite its flexibility and efficiency, programming in P4 is often considered challenging due to several limitations: loops are not allowed, conditional statements (if-else) are restricted to specific areas, and debugging requires analyzing extensive log files that track every packet's movement through the system.

Additionally, P4 lacks built-in mechanisms for restarting or updating code seamlessly, making it difficult to adapt routing logic in response to changes in traffic or network conditions [113, 16]. Current solutions struggle to dynamically modify routing rules as conditions evolve [114]. To address this challenge, in ART we adopted the P4Runtime API [5], which facilitates real-time interaction between an SDN controller and switches programmed with P4. Using a *gRPC* connection, P4Runtime enables the controller to modify P4 program elements dynamically, such as adding, updating, or deleting table entries, and to monitor network performance by reading counter values. It even supports packet manipulation, allowing ART to adapt to changing network conditions without requiring a full system restart.

**Data Plane.** In ART, every time a packet arrives at the switch, three main values are extracted: the destination IP address, the packet size, and the timestamp. These fields are then sent to the controller via P4Runtime using the digest mechanism, ensuring only essential information is transmitted while avoiding unnecessary packet forwarding. The P4 switch, using its match-action paradigm, determines the outgoing port for the packet. The controller defines the mapping between IP prefixes and their corresponding next hops.

**Control plane.** The controller aggregates data from the switches, such as packet details and timestamps, along with the current network load to calculate metrics necessary for routing decisions. This information is passed to the DRL module, which evaluates the network conditions and computes a reward function to decide the best port for forwarding packets based on the link utilization and load capacity (more in Section 4.2.2). The match-action rule is sent to the P4Runtime API, which then updates a pre-configured table within the P4 configuration. When a packet needs to be forwarded, the P4-programmed switch accesses the table to select the correct egress port. Finally, timestamps collected from the switches are used to measure packet latency, a critical metric for assessing the effectiveness of routing decisions. While the interaction between the switches and the controller is essential for packet routing in ART, it can also introduce significant overhead. To mitigate the latency associated with constant communication with a centralized controller, the P4 switches are programmed to send digests asynchronously, enabling packet parsing to occur at the line rate. Additionally, the DT model used for routing is updated only after every $f$ packet is received. We set $f$ equal to $1,000$, as this value provides a good balance between system responsiveness and overall performance.

### 4.2.2 DRL module in ART

Despite the presence of an external SDN controller, we train a DRL model per switch, creating a decentralized architecture where every switch acts as an independent agent of the model. This transitions our approach into a Multi-Agent Reinforcement Learning (MARL) paradigm [102]. Choosing a MARL strategy over a single-agent one offers significant advantages. First, MARL enables different decision-making strategies among switches, ensuring robustness to changing traffic patterns and network dynamics. Second, this decentralized approach enhances scalability, allowing the

network to manage complex situations more effectively while optimizing resource utilization and delivering higher performance [115].

In this MARL setup, each agent (*i.e.,* switch) determines the forwarding port based on the reward it receives, operating in an environment where other agents are independently trying to achieve the same goal.

In this framework, each agent in our network topology, denoted as $i \in \mathscr{M}$, operates within a shared environment. For any given state $s \in \mathscr{S}^i$, the agent takes an action $a \in \mathscr{A}^i$ that transitions it to a new state $s' \in \mathscr{S}^i$ and calculates a reward function $r \in \mathscr{R}^i$. The goal is to determine the optimal policy $\pi^i$ for each agent $i \in \mathscr{M}$, where $\pi^i : \mathscr{S}^i \to \mathscr{A}^i$, such that it maximizes the local reward $r \in \mathscr{R}^i$. This reward is defined by the function $R^i : S^i \times A^i \to \mathbb{R}$, and the state transitions have a probability function $P : S^i \times A^i \times S^i \to [0,1]$.

At each time step $t$, given the current state $s_t \in S$ and the actions of all agents $a_t = (a_t^1, \ldots, a_t^M) \in A$, each agent calculates its individual reward $r_{t+1}^i$, which reflects the outcome of the action $a_t$ taken in the state $s_t$.

It is crucial to note that while each agent has a local view of the environment and independently determines its action and reward, the decision made by any single agent affects the overall behavior of the multi-agent reinforcement learning (MARL) system.

In this environment, for each agent $i$, the possible action $a_t \in A^i$ corresponds to a discrete value in the range $[1,N] \in \mathbb{N}$, representing the available switch ports. The set of states $S^i$ that an agent considers consists of three key elements: *(i)* the destination of the packet, *(ii)* the packet's size, and *(iii)* the current utilization of the port.

Our solution begins with running the distributed DRL module, which sets up a customized network environment via the gymnasium library. At initialization, this environment interacts with switches through the P4Runtime API. Once active, the API processes *digested* packets from the data plane and forwards them to the DRL module, which evaluates the reward based on three critical factors: *(i)* the packet's latency, *(ii)* the capacity of the links, and *(iii)* the utilization of those links. In ART, the reward function $R^i$ considers two values: the contribution given by the packet's latency $r_2^i$ and the one given by the chosen port's utilization $r_1^i$. In detail,

$$r_1^i = \frac{1}{1 + latency} \tag{4.1}$$

and

$$r_2^i = \frac{port\_utilization}{port\_capacity} \tag{4.2}$$

To summarize, the reward function for each agent $i$ is:

$$R^i = r_1^i - \lambda * r_2^i \tag{4.3}$$

where the $\lambda$ is the model's hyperparameter chosen according to the algorithm's performance.

According to the reward function, the DRL module decides the appropriate forwarding port, which is passed to the P4Runtime API to update a P4 table in the ingress pipeline and forward the packet to the chosen next hop. This operation is performed for each received packet in order to train the DRL module.

### 4.2.3 DT module in ART

Once every $f$ packet, the DRL model is distilled into a DT module, which predicts possible forwarding ports based on the same input space. This procedure helps create an interpretable DT model that constructs, retrieves, and organizes splitting criteria into match-action rules, which are directly embedded in the P4 code.

The choice of generating a DT model and injecting its rules inside the P4 code has two reasons. *(i)* First, network switches have limited computational resources (*e.g.,* CPU, RAM) [116]. Utilizing a DRL model within the switch may not be feasible, often requiring external processing modules like Data Processing Units (DPUs) for faster computation or distilling the DRL output into look-up tables (LUTs), as in [117]. In this latter case, however, LUT-based methods can cause memory usage to grow exponentially with increasing table sizes, potentially reducing network performance. *(ii)* Second, by embedding DT rules directly into the ingress pipeline's tables, the P4 code eliminates the need for frequent API interactions during packet processing, making the entire phase smoother. This conversion is possible because DRL-based routing systems can be viewed as rule-based decision systems, which DT models can represent effectively.

We achieve this DT conversion using a *teacher-student* training methodology [118], where the DRL model (teacher) guides the DT model (student) by sharing

$state - action$ pairs for each switch. In ART, during each training episode of the DRL model, actions are predicted based on the current state of the environment and until the model reaches its termination point. These predictions influence the network topology, generating the next state, rewards, and termination flags. The collected actions and states from these episodes are then used to train the DT model.

One challenge in this process is that traditional DT algorithms can generate very complex trees with long branches while trying to replicate the performance of Deep Neural Networks (DNNs). For this reason and to simplify the tree while maintaining accuracy, we applied two main strategies. First, effective policies often suggest the same action for many observed states. Using the DRL's outputs, the DT can consolidate branches, reducing complexity without losing critical decision-making ability. Second, different input-output pairs have different impacts on the policy's performance. To prioritize impactful decisions, we applied a specialized resampling technique [119], allowing the DT to prioritize actions that lead to optimal outcomes.

To summarize, in ART, we limit the depth of the decision tree (DT) to just two levels. This decision is mainly driven by implementation limitations and switch constraints, as well as further evaluations that show that the tree beyond this depth does not significantly improve performance. While one might assume that more rules would enhance accuracy, this is not always true: adding many rules and conditions in P4 can complicate the code and slow down processing, thus negatively impacting the overall performance [120].

When building the DT, the main values for splitting are the packet's destination address and its size. These values are inserted into the P4 code's ingress pipeline via the P4Runtime API, which allows for flexible network policy management. This API stores these values in specific registers corresponding to the DT's levels, which are then queried using a sequence of *if-else* conditions that guide the packet to the correct output port. An example of this process is shown in Fig. 4.2, where the condition at the root level compares the stored value with the packet's destination address. The right side of the figure shows an example of the possible DT's splitting criteria. At the root level, the condition evaluates whether the root value, pre-stored in a register by the P4Runtime API, is less than the packet's destination address.

It is important to note that the DRL model encodes the root value from the network topology, ensuring each destination address is linked to a unique value. During training, this encoded value is then passed to the DT. Initially, the packet is

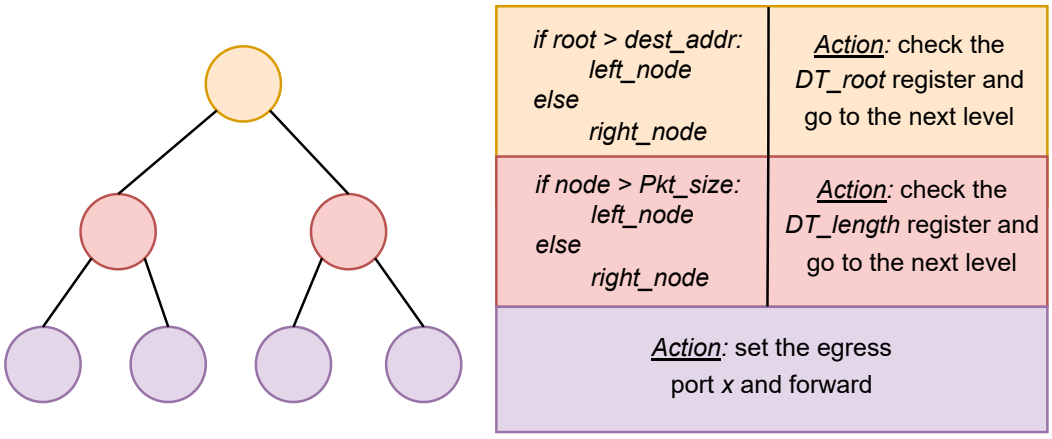



Fig. 4.2 Injected DT model: left shows DT structure trained by DRL, right shows *if-else* conditions, splitting rules, and actions.

checked for size at the first level to determine if it is smaller than the value provided by the API. Depending on the result, it moves to the next decision level, where the forwarding action directs it to the appropriate egress port represented by the DT's leaf output.

In conclusion, to ensure practicality and maintain efficiency, the DT generation and injection process is carried out every $f$ packet. This allows ART to interact with the centralized controller at a low frequency, keeping overhead minimal while maintaining up-to-date decision logic.

## 4.3 Evaluation

To assess the performance of ART, we implemented it in a simulated setup using Mininet, allowing network developers to run and debug their code, interact with the packet-processing logic of the C++11 software switch, and simulate its behavior as if it were running on actual hardware.

### 4.3.1 Experimental Settings and Benchmarks

We evaluated our solution over a data-center topology composed of 10 servers and 9 switches arranged in a leaf-spine fashion, with all links having a 100 Mbps bandwidth, as visible in Fig. 4.3. With *iperf3*, we were able to generate increasing

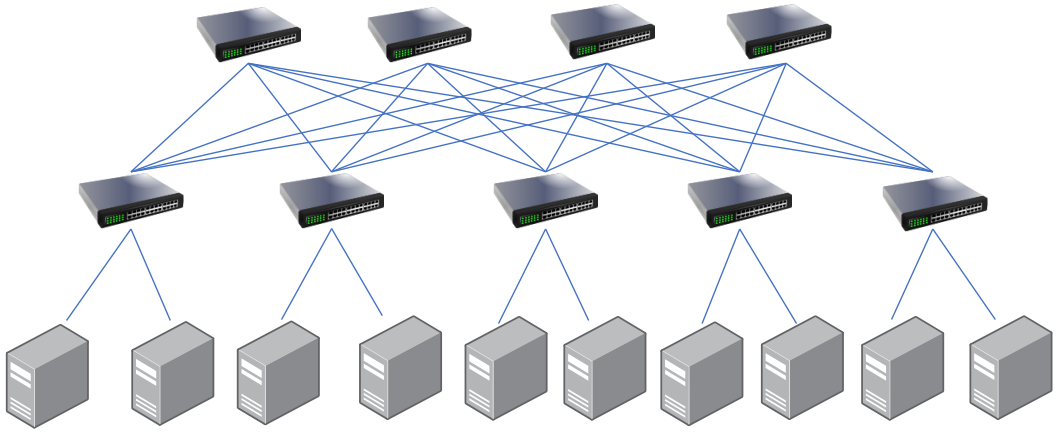


Fig. 4.3 Fat-tree topology adopted for our evaluation.

traffic loads by sending packets between the servers, enabling us to observe the network's behavior under congestion and replicate the transmission as in [29]. To analyze ART's performance under higher network loads, we measured metrics such as round-trip time (RTT), throughput, and packet loss, while also computing the 95% confidence interval. We then compared these findings against related studies, including a conventional routing protocol like OSPF, which adopts longest prefix matching for packet forwarding, and a centralized SDN approach, QR-SDN [93], which leverages a tabular reinforcement learning (RL) model for network routing; and IN-RL, an in-network RL solution where the RL model runs directly on the P4 switches using external C++ functions, as described in [92].

### 4.3.2 Evaluation Results

We started our evaluation by using *iperf3* to generate increasing traffic levels, allowing us to assess how the routing strategy performs under different network load conditions.

Our evaluation begins by comparing the fidelity of the distilled DT against the original DRL model for the top four switches in the network (Fig. 4.4). This metric is determined by counting how often the actions taken by the DT align with those of the original DRL model. This metric is computed by counting how often the actions taken by the DT align with those of the original DRL model. A high fidelity value indicates that the simplified DT is closely replicating the decision-making process of the more complex DRL model. We evaluate fidelity across three DT depth levels:

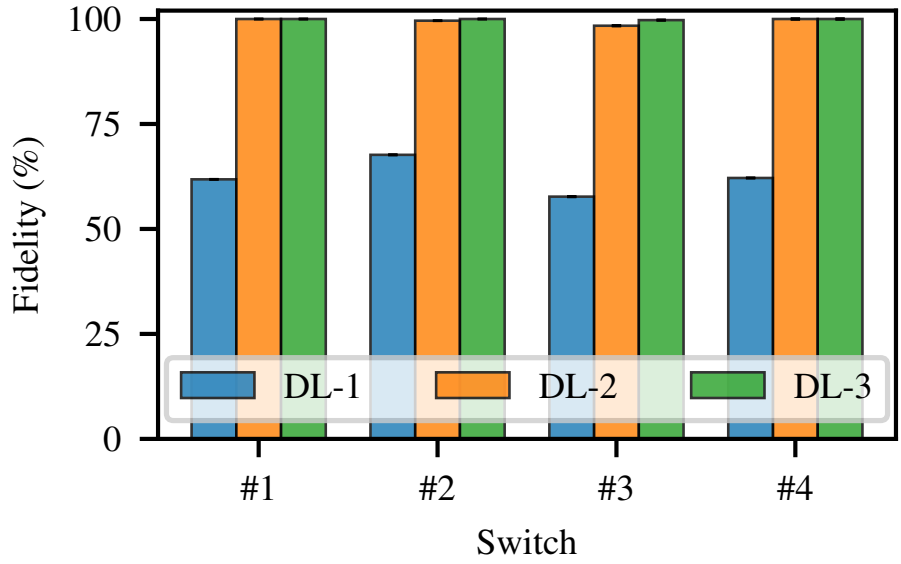
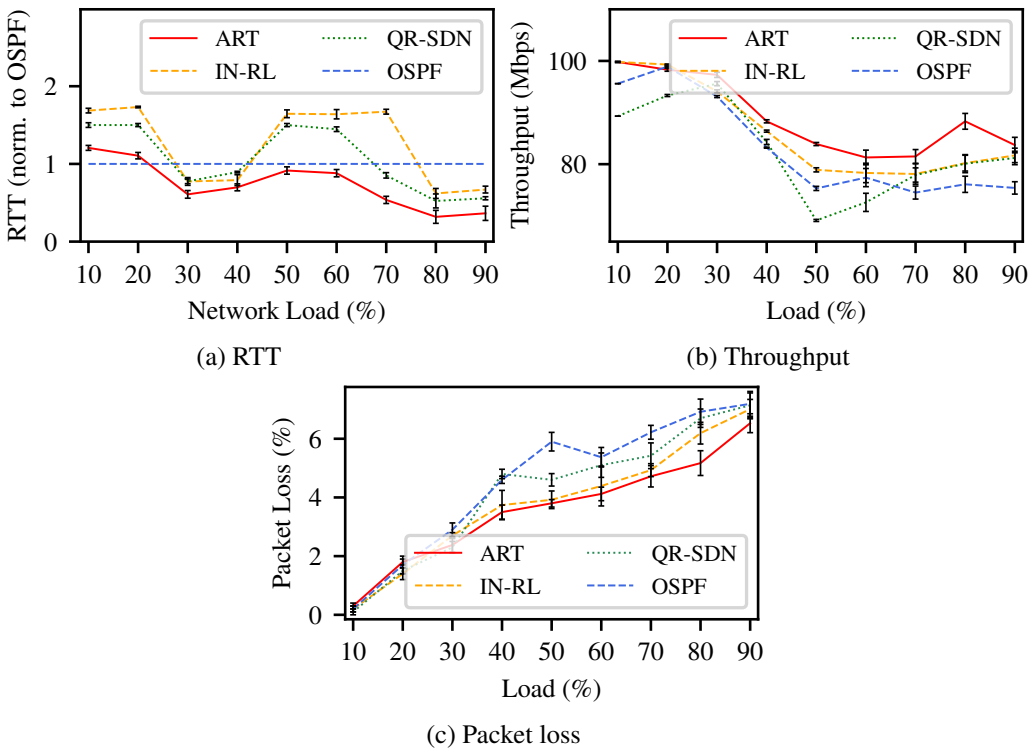



Fig. 4.4 Distilled DTs' accuracy, represented as fidelity percentile, in comparison to the original DRL models.

depth level 1 ($DL-1$), depth level 2 ($DL-2$), and depth level 3 ($DL-3$) when the tree has three layers.

As shown in the figure, a DT with a single layer achieves around 60% fidelity on average, while with two layers, the fidelity grows to about 99.9%. While adding more layers does not show further fidelity degradation, we chose to limit the tree to just the first two layers. This choice not only preserves the high accuracy but also helps contain the complexity of the generated P4 code. Each decision splits the DT into specific registers, and by restricting the tree's depth, we keep the P4 code more manageable, avoiding unnecessarily complex if-else statements, which could slow down decision-making, especially in time-sensitive network environments.

The comparisons of our solution against other benchmarks are then reported in Fig. 4.5. In Fig. 4.5a, we computed the Round Trip Time (RTT) for all the aforementioned solutions at increasing network load, normalizing the results to the traditional OSPF routing protocol. At lower network loads (from 10% to 20%), all the RL-based methods perform worse than the traditional OSPF, due to the added computational overhead of ML-driven approaches, which can significantly slow down forwarding decisions, resulting in higher RTTs. However, as the network becomes more congested, the advantages of the RL-based solutions become evident. As the load increases, these solutions outperform OSPF. In particular, ART consistently achieves




(a) RTT

(b) Throughput

(c) Packet loss

Fig. 4.5 ART vs. SoTA performance comparison on (a) RTT (norm. to OSPF), (b) throughput, and (c) packet loss as network load varies.

lower RTTs across all levels of network congestion, starting from 30% load. In contrast, the other benchmarks only surpass OSPF in certain conditions: either when the network load is moderate (30% to 40%) or when it is heavily congested (70% to 90%).

We then examine the average traffic throughput under different network conditions, as shown in Fig. 4.5b. The figure shows how ART consistently outperforms the other solutions in terms of throughput, even as network congestion increases. The trained DTs of the switch allow ART to dynamically react to increasing congestion levels and reroute packets along less congested paths. It is important to remember that throughput and RTT are not directly correlated, as throughput is independent of the number of packets sent or received. Therefore, this makes the throughput results particularly significant, as they demonstrate ART's ability to maintain TCP flow throughput under different load conditions. Additionally, the effectiveness of in-network machine learning is visible, as IN-RL achieves higher throughput than traditional centralized solutions like QR-SDN.

An important factor in evaluating a solution's effectiveness is packet delivery, as it indicates the network's ability to handle congestion properly. High packet loss can harm transmission quality, leading clients to resend packets, which negatively impacts both the host's performance and the usage of network resources. Fig. 4.5c shows packet loss percentages as the network load increases during UDP packet transmission. We chose UDP for this test because, unlike TCP, UDP does not have its own retransmission mechanism, which allows us to focus purely on the network's response.

The results show that when the network load is relatively low (10% to 30%), all solutions perform similarly, with only a few packet losses. However, as the traffic load increases (from 50% to 90%), ART outperforms the other solutions by reducing packet loss more effectively. This is due to ART's ability to quickly adapt to congestion, as the reactive mechanism embedded in the switches minimizes overhead related to telemetry and network reconfiguration.

## 4.4 Conclusion

In this chapter, we present ART, a decentralized approach that leverages machine learning, specifically Deep Reinforcement Learning (DRL), to optimize the routing decisions within the network. To handle the limitations of P4 programmable switches, a complex, resource-intensive model is simplified into a more lightweight Decision Tree (DT), which allows ART to adaptively consider real-time link congestion and redirect packets through less congested routes. In our experiments, ART is evaluated against three benchmarks: a conventional routing protocol, a centralized RL-based method, and a hybrid solution where the RL model operates within the switch. Preliminary results indicate that avoiding interaction with a centralized SDN controller, combined with an efficient and precise DT, lowers RTT and packet loss under congestion while delivering better throughput performance.

# Chapter 5

# Architecture for Intent-Driven Data Plane Programmability

## 5.1 Introduction

For several years, both academic and industry professionals in networking have aimed to enhance the customization of network infrastructure [122]. Recent advancements in data-plane programmability have facilitated this customization by enabling the deployment of high-performing and tailored network services directly onto programmable switches using specialized programming languages like the Programming Protocol-independent Packet Processors (P4 [3]).

While P4 offers significant potential by enabling operators to tailor network functions at line speed, developing effective P4 programs still demands a deep understanding of the programming language and presents a challenging learning process, even for experienced networking professionals. Researchers in programming languages, especially those focused on network programming, have aimed at enhancing programming safety [123–125] or simplify the process [16]. While some researchers focused on control plane programmability, such as OpenFlow [126], others investigated ways to make data plane programming more accessible and flexible [16, 127].

The work presented in this chapter has been partially published in [121].

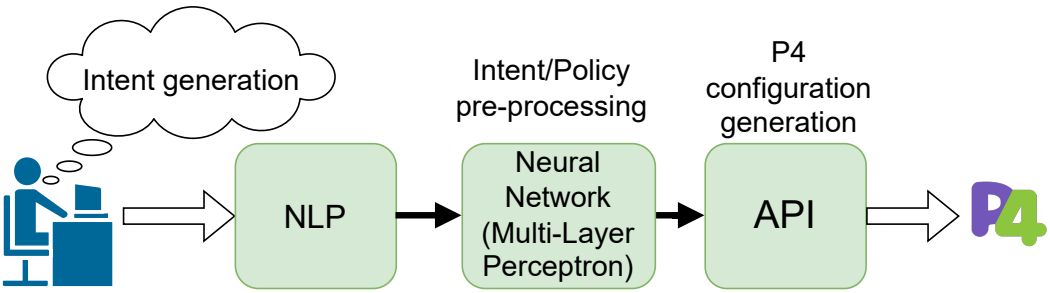



Fig. 5.1 NLP4's Overview and Components.

A key goal of intent-based networking is to facilitate the specification of network requirements, allowing users with little technical expertise to customize their networks. This new paradigm leverages Natural Language Processing (NLP) techniques to simplify programming tasks, including the specification of network intents [128]. Following this paradigm, an innovative method was proposed with P4I/O [129] for interpreting and translating network intents by leveraging the concept of Intent Definition Language (IDL).

Inspired by this approach, we extend it further by integrating Natural Language Processing (NLP) and define NLP4, a solution that translates user inputs into P4 programs, enabling users to customize networks according to the P4 criteria. NLP4 is composed of several blocks, as shown in Figure 5.1. First, a user generates an intent, which is preprocessed into a tokenized and encoded array using a predefined dictionary of words. Then, we adopt a MultiLayer Perceptron (MLP) model to map the array to specific network elements, such as servers for traffic control or switch ports for link load management. Regardless of the network component involved, this mapping identifies the main goal of the introduced intent. To achieve the desired outcome, NLP4 adjusts the behavior of key network forwarding elements by leveraging a tailored API that converts the encoded array into configuration files for P4-enabled switches.

Finally, we evaluate our method by implementing an initial prototype of NLP4 on a virtual network testbed and studied a load profiling use case [130]. Results demonstrate NLP4's ability to program P4-enabled switches, allowing users to customize the traffic profile of their networks.

## 5.2 Related work

Many studies have focused on maximizing network automation by proposing solutions that integrate frameworks with low-level policy translators [131, 132]. Some of them focus on enhancing the expressiveness of data-plane programming languages while also simplifying customization and enabling simulations across a variety of hardware platforms.

For example, Pyretic is a Python-based language that abstracts the possibly complicated network rules [133]. Another language, Merlin [134], was introduced to to handle network traffic by expressing rules as logical predicates. P4 is another recent programming language for data-plane programmable switches, which soon became one of the most used [3]. However, working with P4 has proven difficult due to its lack of high-level abstractions. Despite various efforts to make P4 more user-friendly, it remains challenging to code with, as it operates at a very low level of abstraction [16]. Besides, these languages are more focused on managing network administration issues rather than identifying, understanding, and translating application requirements.

To simplify coding in P4, researchers started introducing the concept of intents and, in particular, intent-driven networking techniques, allowing even inexperienced user to customize their networks, such as Nile [135] and Marple [136]. While the first adopts the human language to retrieve user intents, using feedback from the extracted text to improve other translations; the latter uses P4 to perform network monitoring and performance evaluation, translating dynamic queries into primitives where the result is stored. Despite being very successful and helpful for future studies, these works still do not help create the network topology used by P4 at startup.

Recently, organizations like IETF and ONF have been actively developing a NorthBound Interface (NBI) suitable for intent-based networking, and designed for use in Software-Defined Networking (SDN) contexts [137]. Researchers have also incorporated software techniques to simplify intent expression, such as using natural language input, as in [131]. Particularly, this work focuses on leveraging a Behaviour Driven Development framework in Python, called *behave*, and integrates it with intent-policy translators and interpreters. However, this work does not allow to customize a switch's forwarding rules using data-plane programming languages.

| | NLP-supported for Human Translation | Allows Switch Customization | Network Topology Independent |
|---|---|---|---|
| Nile [135] | ✓ | ✓ | ✗ |
| Marple [136] | ✗ | ✓ | ✗ |
| Behave [131] | ✗ | ✗ | ✓ |
| **NLP4** | ✓ | ✓ | ✓ |

Table 5.1 Comparison of intent-based related works with NLP4.

Other recent solutions apply Machine Learning (ML) techniques to abstract low-level network configurations from natural language inputs, particularly focusing on English text [138, 139].

Differently from previous work, we apply an Intent-Driven API to interpret the user requirements by converting them into P4 configuration files, which are directly applied to each switch of the network, customizing their forwarding rules. This is achieved through the integration of Natural Language Processing and MultiLayer Perceptron techniques. Our solution demonstrates that, through this combination, even users without P4 programming expertise can configure a network's data plane according to their needs without the need for manual coding. We summarize in Table 5.1 the main differences between NLP4 and the mentioned state-of-the-art solutions.

## 5.3 NLP4 Architecture Overview

This section describes the main components of NLP4, highlighting its design features. As shown in Figure 5.1, our solution is composed of three sequentially operating components: *(i)* an NLP-based text mining technique for initial preprocessing, *(ii)* a MultiLayer Perceptron (MLP) model to handle the processed data, and *(iii)* an API that integrates the intent specifications into the configuration files used by P4 switches. In particular, once the user defines the intent and the network's desired behavior, the NLP model processes the natural language input to produce a compact representation that expresses the context. This information is then used by the MLP to generate the mid-level policy that is retrieved by our API, responsible for populating the configuration file required by the P4 switches at startup.

### 5.3.1 Interpreting User Language via NLP

When a user specifies the intent, it can be done in English or other natural languages, as our solution is not language-dependent. However, the intent might contain irrelevant or ambiguous words that could increase the time required for interpretation or, worse, lead to misunderstanding. For this purpose, in NLP4, we adopt Natural Language Processing (NLP) techniques as the first part of our process: the preprocessing part.

NLP is a branch of Artificial Intelligence (AI) whose aim is to understand the natural language and extract the main information in order to perform activities such as content analysis, chatbot, sentiment analysis, and even preprocessing tasks. NLP offers two main functionalities: Natural Language Understanding (NLU) and Natural Language Generation (NLG). The former is used to convert the input into an easily analyzable representation. Therefore, NLU only focuses on catching the meaning of the human language rather than processing it. The latter generates text from various input representations, whether numerical or visual [140].

During this preprocessing phase, the initial step involves applying a text-cleaning function to organize and structure the input intents. This step removes unnecessary words and punctuation that might introduce noise into our Machine Learning model, returning a list of tokens. Next, these tokens are converted into their base form through normalization rules.

There are two common normalization methods discussed in the literature: lemmatization and stemming. The first uses vocabulary to reduce words to their dictionary form [141]; the latter brings all words characterized by the same root (stem) to their common form. A recent study [142] shows that stemming tends to be more accurate for shorter text inputs. Considering that our intent-based system primarily handles brief queries, we opted for stemming, and in particular, the widely-used Porter stemmer, known for its efficiency and popularity [141].

Lastly, since machines cannot directly comprehend human language, the NLP model needs to convert this knowledge into a numerical representation. It does so by counting the words present in the intent, tokenizing them, building a word dictionary, and generating an encoded vector as the final output. This vector will then serve as the input of the next MLP module.

### 5.3.2 MLP for Mid-level Policy Generation

A MultiLayer Perceptron (MLP) is an artificial Neural Network (NN) composed of one or more hidden layers between the input and the output ones. Particularly, an MLP is a fully connected NN that operates in a feed-forward manner, meaning it has no loops between nodes, and it learns through Back-propagation. The MLP model is commonly used due to its flexibility and strong performance in both classification and regression tasks, offering good parameter optimization and generalization across various data sets [143]. In an MLP model, the input layer contains neurons corresponding to the number of input features for the model, while the output layer has neurons equal to the number of classes [144].

In our implementation, the MLP model consists of two hidden layers. The first layer contains 20 neurons, chosen after a preliminary performance evaluation, and uses the rectified linear unit activation function. The second hidden layer has a number of neurons equal to the network elements in the topology (*e.g.,* servers, switches) and uses the sigmoid activation function.

The output from this MLP model is a mid-level policy, representing actions to the involved network elements. These values are then compressed into a vector and fed into our Intent-Driven API.

### 5.3.3 Intent-Driven API

In the final step of our workflow, we use an Application Programming Interface (API) to integrate the MLP model's output into the configuration files used by P4 at startup.

Our API is developed following the principle of *behave*, a Behaviour Driven Development (BDD) framework. BDD is an agile software technique designed to simplify high-level requirement specifications for non-technical users by abstracting away low-level implementation details. It leverages a structured language called Gherkin, which follows the *Scenario-Given-When-Then* (SGWT) format to outline requirements. BDD has been employed in various domains [145], including testing verification techniques for software-defined networks, despite its maintenance cost [146].

In NLP4, we use an Intent-Driven API in Python to convert the intent specification, already processed and analyzed by the NLP and MLP components, into a configuration file for P4-enabled switches. In this prototype, we mainly focus on the switch behavior and configure it accordingly, even if the intent is server-related. For example, if the user wants to disable a server, our solution would interact with the corresponding switch by putting a weight of zero on the port associated with that server. This choice makes the API accessible for individuals with minimal technical expertise in coding or network management.

Although P4 switches can support various operations beyond basic packet forwarding, in NLP4, we mainly focus on a load profiling use-case [130, 20], leaving the implementation of other actions as future work. In this load profiling scenario, we allow the user to specify how to split traffic over outgoing links of a switch and the desired load on these links. If the links have different characteristics or traffic has different priorities, an uneven load distribution might be convenient, where these distributions form the profiles. Our P4-enabled switches are thus programmed to read these profiles from the configuration file and implement the corresponding policies.

### 5.3.4 P4-enabled Switches in NLP4

In NLP4, we employed P4-programmable switches, a widely adopted programming language for data plane customization. However, given the complexity of P4's syntax, we believe that an intent-based approach to programming network switches has the potential to unlock new business opportunities and drive innovative research.

One of the known challenges with P4 is its complexity, making it difficult for beginners to work with. Since P4 commands must operate at a line rate, the language imposes several restrictions. For instance, P4 does not support loop cycles, requiring programmers to use switch-case or if-else structures instead. As a result, programmers have to rely on extensive trial and error, frequently reviewing log files to diagnose and fix errors, a process that is often time-consuming.

Researchers have worked on making P4 more abstract and easier to program by creating shared modules across various applications and switching hardware specifications. However, we are far from considering P4 as a high-level language that newbies can easily learn with no much effort [16]. There have also been studies that

allow users to specify requirements as high-level intents and have them translated to a more low-level language for data-plane programmable switches [129].

In our solution, we modified our P4 code based on the introduced intent. To do so, we stored the information contained in the switches' configuration files in registers that are then retrieved at P4 startup. We wrote the code according to P4 rules by replacing loop structures with a series of if-else conditions and ensuring that the register could be manipulated within match-action tables.

For the implemented load profiling use case, we created multiple match-action tables to manipulate the registers containing the intent specification. This was necessary because P4's limitations prevent the register from being manipulated within if-else conditions, so we moved these operations into the *apply* block within tables when a match was detected. This example underscores P4 complexity and further motivates an intent-based architecture as the one proposed with NLP4.

## 5.4 Illustrative Examples

This section presents two simple use cases and outlines the steps NLP4 takes to process them, with a focus on its individual components.

### 5.4.1 Load Profiling

An example of intent specification could be for a load profiling use case, where NLP4 is applied to adjust the behavior of P4 switches based on the network operator's desired profiles.

However, the network operator does not always need to define the exact profiles; instead, they can specify certain objectives. For instance, a high-level intent might be something like "Disable server 2 for maintenance." Once NLP preprocessing is complete, we would obtain a word dictionary from this input as shown in Listing 5.1.

Listing 5.1 The mapped elements generated from the NLP preprocessing phase for the load profiling use case.

```
{'disabl': 1, 'server': 2,
'2': 3, 'mainten': 4}.
```

Next, we applied our MultiLayer Perceptron (MLP) model to generate a mid-level policy corresponding to the specified intent. The MLP model returns an array corresponding to the network elements interested in the intent.

In our example, considering a topology composed of 4 servers, the output would be: "[1 0 1 0 0]", where the first value indicates the intent main goal (presuming that 1 is linked to "disable" in our dictionary), and the other values correspond to the server, which had to be applied. This vector is then processed by our API to generate the configuration files for all switches in the network, ensuring that traffic is redirected to other servers while the switch connected to S1 disables its port.

Although this example is focused on servers, the approach can be extended to other network elements, such as switches or ports. The API identifies the network element through the first value in the vector and configures the switches accordingly to meet the user's intent.

### 5.4.2 Traffic priority

NLP4 can also be used to prioritize traffic associated with a specific server, whether the traffic is incoming or outgoing. For example, if the user wants to prioritize traffic related to a particular server, such as $S1$, the intent could be something like "I want to prioritize Server $S1$'s traffic on my network."

Our NLP preprocesses this intent by first tokenizing the words and then applying the Porter stemming algorithm. In this case, the sentence would be transformed into stemmed tokens like "network, priorit, traffic, server, s1" after preprocessing.

After this process, the tokens are mapped to numerical values to enable our Machine Learning algorithm to work with them. This conversion is done using an encoded vector that leverages a predefined dictionary of words, as shown in Listing 5.2.

Listing 5.2 The mapped elements generated from the NLP preprocessing phase for the traffic priority use case.

```
{'network': 1, 'priorit': 2,
'traffic': 3, 'server': 4,
's1': 5}.
```

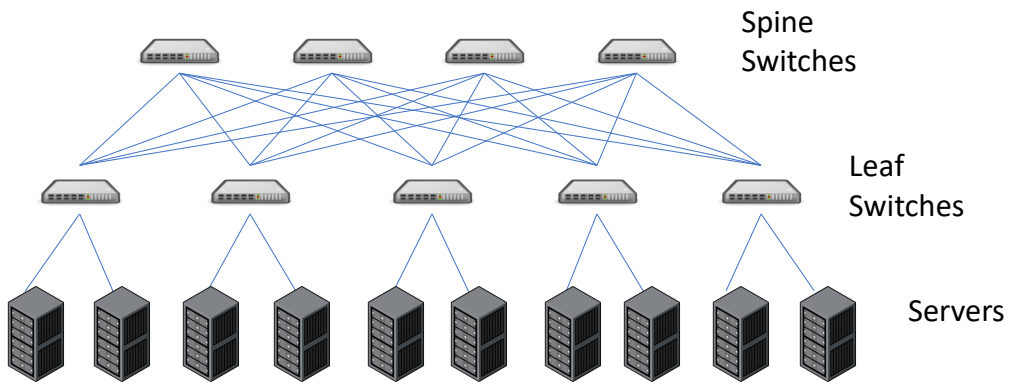



Fig. 5.2 Network topology adopted for the evaluation results.

Once the tokens are encoded, our MLP model predicts the output value for the encoded tokens, returning an array where each entry corresponds to one of the network's servers. Still assuming the network consists of 4 servers, the output would be "[2 2 1 1 1]", where the first value represents the intent's goal derived from the dictionary, and the remaining values indicate the traffic priority assigned to each server.

After this phase, our Intent-Driven API processes the generated vector and appends the traffic priorities to configuration files, which are then used and modified by P4 at startup. At runtime, the switch leverages these priority weights to manage packet forwarding, giving higher priority to traffic to or from $S1$.

## 5.5 Evaluation

This section describes our experimental settings and the results obtained over a virtual testbed, with a particular focus on NLP4's functionality within the context of intent-based load profiling.

### 5.5.1 Evaluation settings

In Mininet, we configured a datacenter-like network topology, with 10 servers connected to their respective switches, which are then connected to an additional 4 switches in a leaf-spine configuration, as shown in Figure 5.2. We set the links in the topology to have 100 Mbps bandwidth.

For the language interpretation task, we employed a MultiLayer Perceptron (MLP) model composed of two hidden layers. The first hidden layer contains 20 neurons and applies the rectified linear unit (ReLU) activation function, while the second layer utilizes the sigmoid function. The number of neurons in our MLP model corresponds to the number of network elements to be profiled, such as servers, switches, or specific switch ports. In this case, since we focused on profiling the 10 servers within the network topology, the output layer of the MLP is set to 10 neurons. The architecture of our model, including the 2 layers and their respective number of neurons, was chosen based on an initial performance analysis focusing on the model's accuracy.

### 5.5.2 Evaluation results

In the following sections, we evaluate how NLP4 performs when dealing with different traffic profiles within the network.

After converting the load profiling intents into weights as specified in the configuration file, our P4 switches select output links using a weighted choice algorithm, matching the traffic profile desired by the user. One common application of traffic profiling is load balancing, where the network is configured to evenly distribute traffic across multiple links. However, in more general scenarios, the switch directs outgoing traffic based on the assigned port weights: ports with higher weights are selected more frequently than those with lower weights.

To verify if the network nodes can maintain the desired traffic profile, we simulate network activity by sending 1000 ping packets between two servers that belong to different leaf switches.

Figure 5.3 shows a comparison between the traffic profile intended by the user and the actual profile observed during the experiments. The desired link loads, which are assigned to the leaf switches for testing purposes, are randomly selected. From the figure, it is visible that the weighted port selection aligns with the expected traffic profile across each port, demonstrating that the desired traffic distribution is effectively maintained.

We then measure the overhead introduced by NLP4 and measure the program's execution time across varying network sizes, as shown in Figure 5.4. The results indicate that as the number of switches increases, NLP4 continues to maintain low

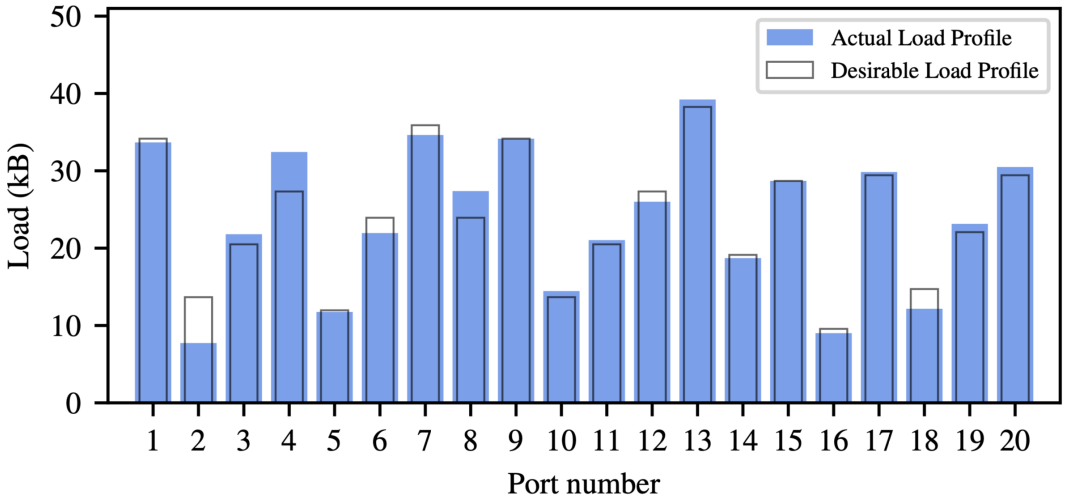



Fig. 5.3 Desirable vs. Actual Load Profile across network switches' ports.

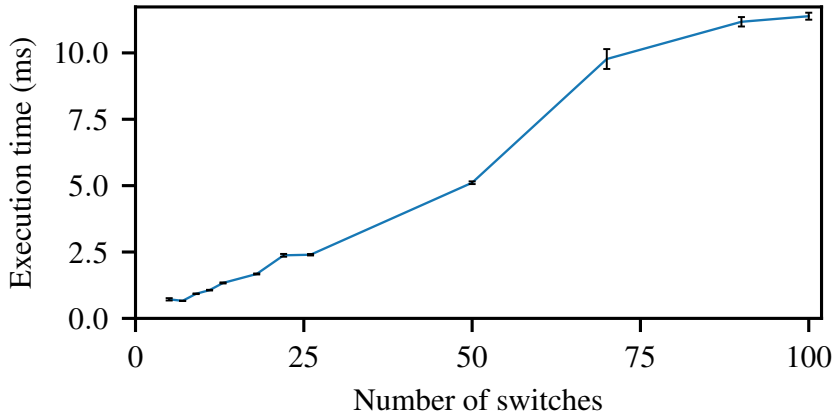



Fig. 5.4 Execution time of the intent parser as the number of P4 switches grows.

execution times. This suggests that, even in real data-center environments with numerous switches, the execution time remains within milliseconds. It is important to note that the program only needs to run when there is a need to adjust the load profile, which is not a frequent event.

## 5.6 Conclusion

This chapter introduces NLP4, an intent-based solution designed to simplify P4 data-plane programmability by leveraging a combination of Natural Language Processing and MultiLayer Perceptron methods. The result is an Intent-Driven framework

that enables users, even with limited network expertise, to create tailored network configurations by directing programmable switches to operate according to the specified intent. Initial results confirm the effectiveness of this method in handling load profiling intents. Future work will explore additional use cases, expand the set of potential user intents, and extend the solution to support a wider range of switch actions.

# Chapter 6

# Transpiler for Deploying Human Intents into Network Configurations.

## 6.1 Introduction

Recent advances in data-plane programmability have enabled network developers to directly inject rules into programmable switches, tailoring networks to specific needs. Among the various programmable switch architectures, the Programmable Independent Switch Architecture (PISA) stands out for its versatility and pipeline-based design that has become widely adopted in the telecommunications and networking sectors. This adaptability gives operators the flexibility to adapt network functions to meet operational requirements. With the advent of programmable switches, networking has transitioned into a new era of vendor-independent runnable software, providing network operators with greater flexibility, portability, and consistency. Researchers have shown that adopting PISA can lower costs and improve scalability for high-demand applications, such as Next Generation Firewall (NGFW) or Deep Packet Inspection (DPI) [147]. Different network programming approaches have been developed, protocol-based – *e.g.*, using SRv6, where labels are interpreted as packet instructions or evolved MPLS, currently under standardization - but also data-plane based such as P4 and OpenFlow. P4, in particular, has attracted significant industry attention by enabling the customization of packet-processing pipelines that were previously hard-coded, allowing the implementation of custom packet-

The work presented in this chapter has been partially published in [113].

processing behavior. To address P4's limitations, researchers are working on flexible data structures that simplify switch programming, making P4 code more accessible for network developers. One relevant example in this research area is P4All [16], where the authors developed an elastic data structure that adjusts automatically to the hardware capabilities of each switch, enabling more portable and modular modules. Another recent work, Lucid [17], introduces a language that allows PISA switches to be customized and managed through P4, making it accessible to users who are not familiar with network programming languages. Despite such valuable improvements, data-plane programming, especially with P4, remains a complex task for many that would benefit from higher-level abstractions.

To address this, many researchers have proposed systems that integrate frameworks with low-level policy translators, helping to automate network tasks and reduce manual coding. At the same time, Intent-Based Networking (IBN) has gained attention, centered around the concept of *intent*, which represents a high-level specification of desired goals and objectives that a network should meet. An example of a possible intent could be: "Link utilization in every link should be less than 70%" [148]. As a result, Intent-based networking does not only include proper rendering of intent as code on a per-switch basis but also incorporates deployment aspects such as the "zero-touch networks" paradigm. Following the intent paradigm, researchers have explored how software methodologies can help define, interpret, and implement intent on programmable switches. An example is P4I/O [129], where the authors introduced an Intent Definition Language designed to translate high-level intents into specific low-level P4 code. While valuable, this study does not allow users to customize their network using an interactive API on-demand, but only through an Intent Definition Language (IDL). Researchers have also studied a way of translating high-level code into low-level rules using programming languages. This is the case of P4HLP [149], which uses E-Domino, a high-level C-like programming language, to generate P4 code. Another relevant work is Lumi [150], which allows users to ask questions about the network of a college as intents and translates them into data-plane programming language commands. Although promising, Lumi's architecture relies on a specific IDL, which, as the authors suggest, would need to be adapted for broader application across various network settings. We summarize in Table 6.1 the main differences between NAIL and the mentioned state-of-the-art solutions.

| | NLP-supported for Human Translation | Interactive API | Network Topology Independent |
|---|---|---|---|
| P4I/O [129] | ✗ | ✗ | ✓ |
| P4HLP [149] | ✗ | ✗ | ✓ |
| Lumi [150] | ✓ | ✓ | ✗ |
| **NAIL** | ✓ | ✓ | ✓ |

Table 6.1 Comparison of intent-based related works with NAIL.

The rise of Large Language Models (LLMs) like ChatGPT [151] and Llama [152] has opened up new possibilities for automating the translation from high-level language to low-level code, allowing users to prompt models to generate purpose-built code. Despite these advancements, human review is essential to ensure the generated code meets requirements and functions as intended [153]. In our own testing, for example, ChatGPT occasionally suggested code using functions that do not exist in the original libraries but instead are found only in developer-specific branches. This demonstrates that generated code often requires deep know-how and reverse-engineering integrating procedures to be fully operational. Intersecting these efforts and leveraging such recent advances, this chapter introduces NAIL, a network management framework designed to convert high-level intents into specific data-plane programming rules. NAIL includes a transpiler that uses natural language processing to interpret human-generated intents and translate them into data-plane programming code, specifically P4 in this implementation. The system also features a network management API, which provides the necessary methods for NAIL to interact with network components. With NAIL, high-level intents are converted into working programs, such as P4 table entries, using the API, which interacts with NLP methods and databases. This setup enables network programmers to customize network behavior for specialized functions (*e.g.*, load profiling, stateful firewall). To provide more flexibility, the framework also supports dynamic rule management, allowing rules to be added, modified, or removed as needed. Finally, NAIL allows the network programmer to collect statistics about demanded intents or a switch for troubleshooting purposes. We demonstrated NAIL's performance with some use cases, also focusing on the lines of code (LoC) and the updating rules reaction time. Our results show that NAIL simplifies the implementation of network intents, requiring fewer LoCs than other state-of-the-art solutions, while also fast reacting to any rule update.

## 6.2 Overall Architecture Design

This section describes our network management system, which consists of a network management object model, a transpiler, and supporting databases to facilitate intent manipulation.

### 6.2.1 Intent to data plane program workflow

When a network programmer specifies an intent, it can be expressed in any language (*e.g.,* English, Italian, Chinese) as we designed our parser to be language-agnostic. However, the intent may include ambiguous or complex terms, potentially introducing noise that could lead to misinterpretations. To address this, we employ preprocessing techniques combined with Natural Language Processing (NLP), a field of AI focused on analyzing and extracting information from human language. This involves a text-cleaning phase that removes all stop words and reduces them to their root form, effectively removing any noise and irrelevant information. After this NLP phase, we obtain a refined list of keywords, which our dictionary then interprets to determine the main purpose of the specified intent (*e.g.,* load profiling, firewall) and identifies all network components involved in such intent (*e.g.,* switches, links, port numbers).

An overview of NAIL is shown in Fig. 6.1. Four main components form our architecture:

- Converter: this component includes NLP software libraries, connected with a parser and a dictionary to generate intent objects;
- API: this includes all methods that allow NAIL intent object manipulation, *i.e.,*: (1) *create*, (2) *delete*, (3) *update*, (4) *get_stats()*;
- Management Information Base (MIB): this is a logically centralized database that maintains all NAIL states, including network elements, their ids, and subnets. This database also stores all introduced intents enabling future modifications like updates and deletions as needed.
- NLP Datasets: these datasets support the NLP processing phases, containing data for stopword identification and stemming processes.

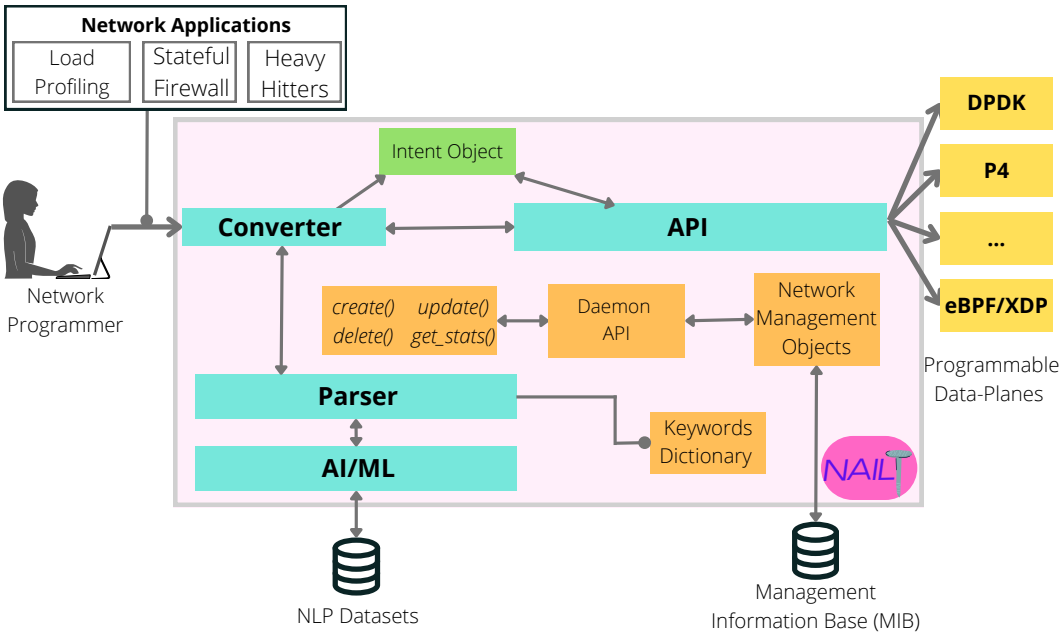



Fig. 6.1 NAIL's Overview and Components.

As shown in Fig. 6.1 (yellow blocks), our NAIL API can interact with different data-plane programming languages and frameworks, *e.g.*, P4, DPDK, or eBPF/XDP.

### 6.2.2 NAIL API: The User Perspective

NAIL allows network programmers to define an intent using the *create* method and manipulate it using three other operations: *delete*, *update*, and *get_stats*. These four operations enable network programmers to deploy new data-plane instances programs without knowing how to program in a specific data-plane programming language, empowering the *zero-touch* network paradigm.

Network programmers can define an intent using the *create()* API call:

$$\texttt{intent\_id} = \texttt{create}(\texttt{intent})$$

Once defined, the intent undergoes preprocessing with NLP techniques and a keyword dictionary analysis to identify the involved network elements. Finally, NAIL's interpreter determines the necessary actions based on the specified network elements and the intent's main objective, such as rerouting traffic around a specific

switch, prioritizing traffic from a specific IP address, or blocking traffic between designated groups.

The *create()* function returns a unique `intent_id`, corresponding to the intent object that contains the involved network elements and any configuration states applied to satisfy the intent's objective.

Network programmers can also retrieve statistics on a particular intent using the *get_stats()* API, which accepts two optional parameters, such as *intent_id* and *switch_id*, for troubleshooting purposes.

$$\texttt{get\_stats}(\texttt{intent\_id}, \texttt{switch\_id})$$

The first parameter enables the collection of statistics for a particular intent running on the network (*e.g.,* the count of packets processed by each switch involved), while the latter provides details specific to a particular switch (*e.g.,* queue on ports, dropped packets). To modify a previously added intent, it is possible to use the *update()* method. This function accepts either an *intent_id* of a previously configured intent or a new intent, in which case it triggers *create()*, as before:

$$\texttt{update}(\texttt{intent\_id}, \texttt{new\_intent}).$$

If network programmers want to remove an existing intent, they can use the *delete()* API, which takes the id of the specified intent object as input:

$$\texttt{delete}(\texttt{intent\_id})$$

This function deletes all the network configurations associated with the intent and the corresponding intent object from the system.

### 6.2.3 NAIL API: The Inner Workflow

To describe the process that NAIL follows after the user inputs the desired intent, we start from the example described in Listing 6.1, where the API is used to configure a stateful firewall.

```
int_1 = ‘‘Block traffic from GroupA to GroupB’’
intent_id = create(int_1)
#NLP techniques and keywords recognition
#Write into the switch’s table for this action
int_2 = ‘‘Block traffic from GroupA to GroupC’’
update(intent_id, int_2)
#To add a new intent update() calls create()
intent_id = create(int_1)
#To get statistics from an intent
get_stats(intent_id)
#To get statistics from a specific switch
get_stats(switch_id)
#To delete an intent
delete(intent_id)
```

Listing 6.1 The user perspective of a stateful firewall using the NAIL API.

When the network programmer requests an intent to block traffic between two subnet groups through the *create()* method, NAIL performs a pre-processing phase. In this phase, NAIL uses NLP techniques and a pre-configured dictionary containing regex patterns and keyword recognition to separate the intent's main objective from the network elements involved. Such NLP pre-processing is broken down into three steps: *i)* the intent is converted into tokens resulting in "["block", "traffic", "from", "groupa", "to", "groupb"]"; *(ii)* we remove all the stopwords from the list of tokens, resulting in "["block", "traffic", "groupa", "groupb"]"; *iii)* The remaining tokens are manipulated using the Porter stemmer, which brings each token to its base form. This third step does not change the content of our tokens as they already are in their canonical form. It is important to notice that these two last steps use the information contained in the NLP Dataset to retrieve the necessary words to remove canonical forms. Once pre-processed, these elements are matched with the MIB database, which provides details like IDs, number of ports, and IP addresses. In this example, "group A" and "group B" are matched to two specific subnets in the network, each with a unique IP range. To confirm the intent's goal, the API then scans the tokens against its dictionary, identifying the keyword "block".

After completing the initial processing phase, the system moves into the *deploy* phase to generate an intent object, which includes a unique incremental ID (provided to the user), the specified network elements (such as the subnets in GroupA and GroupB), and the action goal ("block traffic groupa groupb"). NAIL then customizes a source P4 code template, which includes minimal functionalities (*i.e.,* IPv4 for-

warding), along with registers and tables, by adding the necessary blocking rules. By using P4Runtime library methods (as detailed in Section 6.3.1), table entries are injected into the switch connected to GroupB's subnet. While this approach allows reducing possible errors in compilation, we also deployed the *get_stats()* method to allow network programmers to verify the network behavior and perform two different actions: *i*) retrieve statistics from the customized network, *ii*) troubleshoot the network after the insertion of an intent to verify that the injected rules are correct.

## 6.3 Prototype Implementation and Evaluation

We evaluated NAIL by implementing several use-cases (described in Table 6.2 below) and selecting P4 as our data-plane programming language. To do so, we compiled the intent using the p4c compiler and adopted the behavioral model version 2 (bmv2) as the target for our software switch. Finally, we simulated and tested our P4-generated programs using a leaf-spine datacenter topology composed of 10 servers and deployed in Mininet.

### 6.3.1 NAIL Integration with P4Runtime

We selected P4 as our prototype implementation due to its compatibility with both software and hardware-based platforms [147]. While P4 provides the potential to customize the network to suit diverse use cases, modifying the P4 program requires stopping the execution of the current program and restarting it with an updated one, making real-time updates infeasible. To address this limitation, in NAIL, we incorporated P4Runtime into our API, enhancing the flexibility and efficiency of key operations such as inserting, deleting, and updating intents.

While P4Runtime already includes a built-in mechanism for adding entries to P4 tables, it does not natively support methods for modifying or removing these entries after. To overcome this, we extended its functionality by introducing three additional methods: one for updating entries, another for deletion, and a third for retrieving statistics. These enhancements grant network administrators and developers greater control and adaptability in managing network behavior.

| | LoC in P4 | LoC in P4All | LoC in P4I/O | LoC in NAIL (user perspective) | LoC in NAIL (internal code) | Installation time [s] |
|---|---|---|---|---|---|---|
| **IPv4 Forwarding** | 197 | 217 | 416 | 5 | 241 | 0.558 |
| **Stateful Firewall** | 207 | 217 | 477 | 11 | 294 | 1.714 |
| **Load Profiling** | 294 | 286 | N/A | 8 | 305 | 1.784 |
| **Heavy hitters det.** | 316 | N/A | 477 | 6 | 298 | 1.762 |
| **DDoS Attack det.** | 233 | N/A | 477 | 9 | 298 | 1.883 |
| **BeauCoup** | 1500 | 541 | N/A | 10 | 320 | 1.707 |
| **PRECISION** | 283 | 266 | N/A | 9 | 297 | 2.487 |
| **SketchLearn** | 366 | 88 | N/A | 15 | 284 | 2.644 |

Table 6.2 Lines of code (LoC) comparison between NAIL and some SoTA solutions for both the user perspective and internal code.

Once the entry is ready, NAIL invokes the *WriteTableEntry* method to add it to the table. A similar process applies when programmers wish to modify or delete an existing intent. In these cases, NAIL leverages the *UpdateTableEntry* or *DeleteTableEntry* methods to replace the old entry with an updated one or remove the corresponding intent altogether.

When NAIL processes a new intent and creates the corresponding intent object, it uses the *buildTableEntry* method in P4Runtime to prepare the table entry. This method requires the table name, action name, and action parameters from the intent object to identify the correct table for the specified action and its parameters. Once the entry is ready, NAIL invokes the *WriteTableEntry* method to add it to the table. The same procedure is visible when network programmers want to update or delete the rules of a previously added intent. In these cases, NAIL calls the *UpdateTableEntry* or the *DeleteTableEntry* function, respectively, and simply replaces the old entry with the new one or deletes the corresponding intent.

### 6.3.2 NAIL Prototype: Design and Evaluation Metrics

In evaluating our solution, we began by examining the lines of code (LoC) that comprise NAIL and comparing them to other relevant works. LoC serves as a simple yet effective software size validation metric, allowing us to more accurately assess the scale and complexity of our implementation [154]. For this reason, during the development of our use cases, we report how many lines of code NAIL generated and compared it with the LoC generated by P4, P4All [155], and P4I/O [129]. As shown in Table 6.2, in our implementation, we considered eight use-cases: an IPv4 forwarding, a stateful firewall, a load profiling, a heavy hitter detector, a

DDoS attack detector, BeauCoup, PRECISION, and SketchLearn. BeauCoup is a system that monitors the network through queries, PRECISION is an algorithm that uses probabilistic recirculation to find top flows (*e.g.*, detect heavy hitters) on switches, SketchLearn uses multi-level sketches to identify flows that are statistically responsible for causing traffic conflicts. The LoC for these three applications was directly taken from [155].

An interesting observation is that NAIL's internal code for most applications is shorter than that of other intent-based architectures, such as P4I/O. In addition to the internal LoC, we also evaluated the LoC required for network programmers to customize their networks. This is shown in the "user perspective" column where we assumed that the network programmer uses at least all the methods *create(), delete(), get_stats(), update()* multiple times.

The "Installation time" column in Table 6.2 represents the time NAIL requires to configure the network after receiving an intent.

In general, the process of translating an intent to code depends on two main stages: mapping the intent to specific network policies and generating the corresponding code to implement these policies. The table shows that NAIL achieves relatively low installation times for all tested applications, demonstrating its efficiency in parsing and deploying intents. Unlike traditional approaches that require hours or even days to manually write and implement P4 programs, NAIL can process and deploy intents in just a few seconds. This efficiency highlights NAIL's potential to significantly streamline the workflow for network programmers.

### 6.3.3 Load Profiling (LP)

Load profiling refers to the process of having different priorities for different traffic demands. Studies have shown how critical it can be to deploy an efficient load profiler, as the whole traffic optimization depends on it, especially when it is adopted in big topologies such as datacenters. By analyzing traffic patterns and associating a specific profile, the network administrator can identify bottlenecks, efficiently allocate resources, and improve the network's overall performance. For this reason, researchers have studied different algorithms and techniques of integrating load profiling into the data-plane of SDN programmable switches, using P4 as programming language [13].

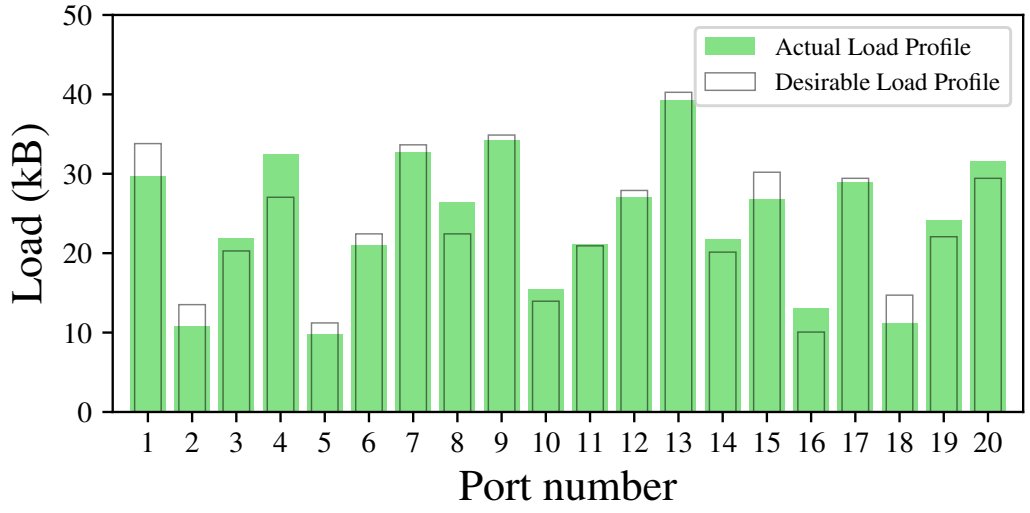



Fig. 6.2 Desired vs. Actual Load Profile across network switches' ports.

In NAIL, we considered a use-case of a load profiler that sets different weights on each switch's port. To test our use-case and evaluate if our network respects the desired profile, we sent and received 1000 ICMP packets between two servers belonging to different leaves. Fig. 6.2 compared the generated load profile: the obtained one (in green) and the desired one (in white). It is noticeable that the actual load profile coherently respects the desired one throughout all ports of our switches, demonstrating the effectiveness of the implemented solution.

### 6.3.4 Stateful Firewall (SFW)

We considered a stateful firewall (SFW) as a use-case scenario given the fact that nowadays companies and individuals widely use a firewall to protect their networks from unsolicited traffic, and researchers have introduced firewall applications in their studies [17, 147]. As firewalls are used to categorize and filter traffic, we developed our solution in the data plane using a bloom filter, a probabilistic data structure mainly used for its fast computation time in small memory space. Although bloom filters may occasionally result in false positives during computation, they remain a popular choice in various network security and privacy systems.

Our SFW operates entirely in the data plane, eliminating the need for constant interaction with the control plane, which can otherwise introduce delays and increased response times. In our implementation, each flow is processed by our PISA switches

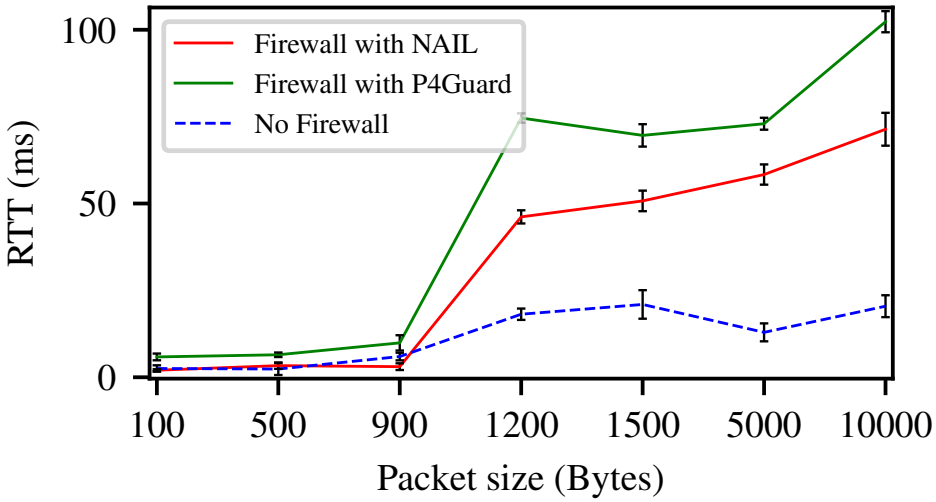



Fig. 6.3 RTT for different packet sizes (in bytes) with an installed stateful firewall.

that calculate a bloom filter hash based on a 5-tuple composed of the source and destination IP addresses, source and destination TCP ports, and the protocol in use. The result is stored in registers for a subsequent lookup function, which determines whether incoming packets are from the internal network or external and, in this latter case, discard them. Additionally, we track the number of packets going through each switch, including dropped ones, for statistical and troubleshooting purposes. One common problem in implementing a stateful firewall is ensuring the correct sequence of rule installation and evaluation. If rules are not ordered correctly in a stateful firewall, it can lead to unintended consequences, such as blocking desired traffic. NAIL addresses this by prioritizing rule application based on the time of intent specification and by enabling ongoing monitoring, which allows users to verify that rules are applied correctly and that the desired security objectives are met.

In Fig. 6.3, we evaluated how the round trip time (RTT) reacts when the packet size increases and compared it to another software firewall, P4Guard [156], and to a baseline case when there is no SFW installed. The figure shows that the three cases achieve almost the same performance when the packets are small (100-900 Bytes). However, as packet size grows (900-10,000 Bytes), NAIL's firewall outperforms P4Guard despite both using bloom filters. This demonstrates that NAIL achieves promising results with minimal input from network programmers, thanks to its *zero-touch* data-plane coding approach. It is important to notice that P4Guard uses the *P*4_14 version of P4 instead of the last one, *P*4_16, which is known to perform better than the previous version. When our firewall is compared to a case with no

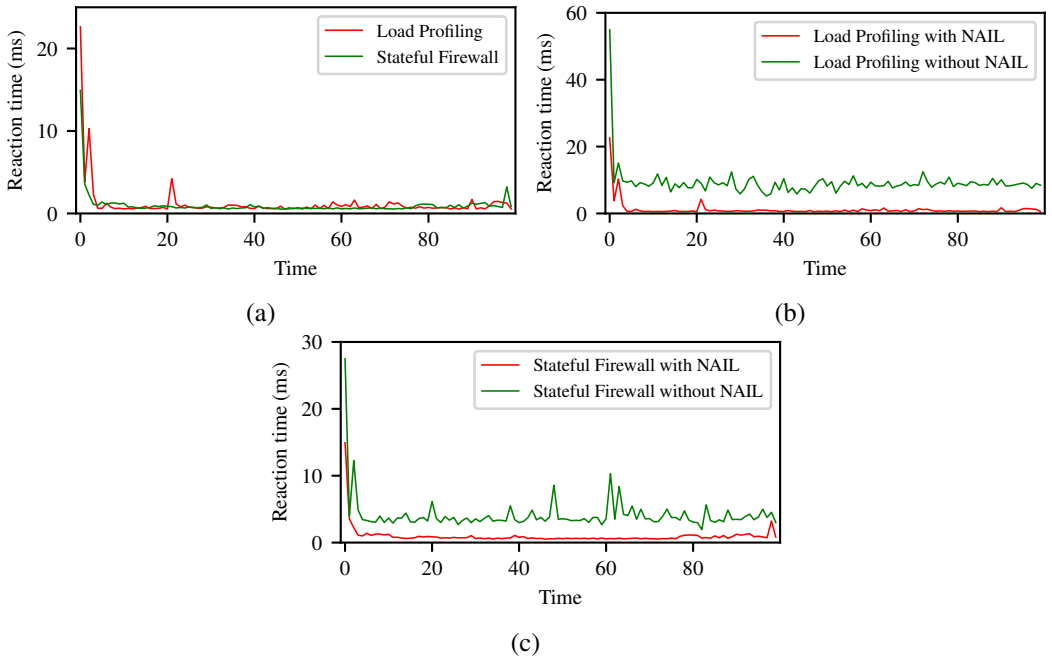



Fig. 6.4 (a) Response time of *update()* for load profiling and a stateful firewall. (b, c) Comparison with/without NAIL for load profiling and firewall.

firewall installed, the overhead is minimal and likely attributed to the hash functions employed in the firewall implementation.

### 6.3.5 Updating Rules

Another important aspect of NAIL is the possibility of updating an existing table entry by just using the *update()* method. As mentioned earlier, NAIL can dynamically modify any forwarding table, adding new entries or even modifying existing ones, in response to changes in the network topology, traffic patterns, user needs, or other factors. This adaptability allows the network to respond to real-time conditions and optimize its performance without restarting the whole configuration. This dynamic modification of a table entry has to be performed as quickly as possible, since there might be situations in which a rerouting needs to be performed for a failed link or other reason, making the reaction time to an update a critical factor.

In NAIL, we evaluated the reaction time of updating a table entry in Fig. 6.4, focusing on the applications that we deployed: load profiling (Fig. 6.4b), stateful

firewall (Fig. 6.4c), and a combined scenario where both applications run simultaneously (Fig. 6.4a). Looking at Fig. 6.4b, we evaluated the reaction time required to modify an existing rule in a load profiling setup, such as updating weights assigned to specific switch ports. The performance was compared to the conventional method of editing a text file with the new rule and restarting the network configuration.

From the figure, we can notice that NAIL, thanks to the *update()* performed in real-time, always achieves lower reaction times (in *ms*) than traditional approaches, allowing the network programmer to quickly modify the profile for its network according to its needs, to failed links or traffic pattern. It is also visible that the highest reaction time is achieved when the function is called for the first time; meanwhile, the reaction time is even lower for all the next calls. Similarly, Fig. 6.4c shows, the reaction time for updating rules in a stateful firewall. NAIL outperforms the traditional method, achieving up to 83.8% faster reaction times on the initial update and maintaining superior performance for subsequent updates. Finally, we evaluated a scenario where both load profiling and a stateful firewall were active, and updates were required for both applications. As shown in Fig. 6.4a, the reaction time remained consistently low, with updates for load profiling rules taking up to 22.6 *ms*, likely influenced by the need to install new weights for each switch port.

## 6.4 Conclusion

This chapter introduced NAIL, an architecture that translates network intents into programs for programmable entities, providing a management layer that is both flexible and accessible, even to users with minimal technical expertise. While NAIL is compatible with various programmable switches, this work focuses specifically on switches built on the PISA architecture and their native programming language, P4. We validated NAIL's capabilities using well-known use cases, illustrating how it can handle a variety of intents, from traffic prioritization to detecting security threats.

That being said, there is still significant room for improvement and future development. One area of expansion involves simplifying even more network engineers' workloads by incorporating automated deployment strategies, such as zero-touch deployment (ZTD), to facilitate continuous code updates without manual intervention. Secondly, enhancing the natural language interface by integrating NAIL with modern chatbots could provide a more intuitive experience. This integration could

enable users not only to submit intents but also to engage in dynamic dialogues to clarify their needs, learn what the network can realistically achieve, and address potential conflicts between overlapping intents.

# Chapter 7

# Conclusions

The ever-increasing demand for advanced digital services and applications has presented significant challenges for current network infrastructures, which must handle these stringent requirements such as ultra-low latency and high throughput.

To handle these challenges, there is increasing interest in equipping networks with autonomous, real-time decision-making capabilities by leveraging Artificial Intelligence (AI) and Machine Learning (ML) techniques. These advanced techniques need to be accurately incorporated within the network to provide efficiency and adaptability to different network conditions. However, this integration comes with different complexities.

First, these algorithms need to be trained with metrics and measurements collected from the network. This step needs to be carefully studied and executed as errors or inaccuracies can severely affect runtime performance, delaying packet forwarding. Secondly, these algorithms are known to seriously impact hardware resources, *e.g.,* CPU and RAM, hence, selecting and optimizing the appropriate methods is critical to prevent bottlenecks that may slow down routing and other network operations. Third, these methods need to address sub-optimal behaviors via network programmability, where many complications join together. A key enabler in this context is data-plane programmability, which allows for the implementation of advanced, application-specific logic directly within the data plane of network devices. Despite its advantages, network programmability, especially data-plane programming, presents its own challenges, in particular for less-experienced pro-

grammers due to its complicated structure and debugging procedure, often leading to many attempts and trial-error processes.

In this dissertation, we explored novel methodologies aimed at solving some classic networking management problems by considering AI/ML algorithms that can effectively be used in optimizing network operations.

We started by presenting Howdah, a load profiler mechanism that leverages host-switch cooperation for intelligent routing through the use of classification flags embedded in packets. After evaluating different protocols for embedding the classification flag and analyzing the advantages and disadvantages of different ML algorithms, Howdah employs a Decision Tree algorithm on hosts to classify traffic and embeds the resulting classification flag into the IPv4 protocol. Despite the network congestion, especially at high network loads, Howdah shows outperforming results compared to state-of-the-art solutions.

With the same goal of enhancing routing in SDN via a programmable data plane, we proposed ROAR, an adaptive routing solution that accommodates the traffic demand and handles network congestion by leveraging the In-Network ML (INML) paradigm where each switch is an agent of a MARL architecture. The Deep Reinforcement Learning implemented within the switches efficiently forwards packets based on the network conditions and traffic patterns, outperforming state-of-the-art techniques.

Enhancing further the adapting routing paradigm, we proposed ART, a decentralized ML solution that integrates the forwarding logic directly into switches with little impact. In ART, switches interact with a controller to exchange traffic metrics, which the controller uses to train DRL methods. These methods are then distilled into lightweight decision trees (DTs) that are embedded directly within the switches. This approach minimizes continuous interaction with the external controller and enables reactive routing, even during network congestion.

Then, aware of the difficulties brought by network programming and with the desire to obtain more customizable and flexible networks, we embraced the intent-based network (IBN) paradigm and presented NLP4. With NLP4, network programmers can insert intents in human-language that will be parsed and converted by our Artificial Intelligence techniques (*i.e.,* Natural Language Processing and Multi-Layer Perceptron) into network configurations. This architecture proved that networks could be customized and spanned the way for more complex IBN-based network

architecture, such as NAIL. This architecture allows network engineers to articulate desired network behaviors at a higher, more expressive layer and then translate such behaviors into executable code using a transpiler that acts as a network intent translator.

While the proposed solutions have addressed only a subset of existing challenges, we hope future research will address the remaining problems by expanding the INML paradigm even further, which could pave the way for a more autonomous and reactive network. Among possible challenges, one critical issue is dealing with limited data, which can negatively impact the learning process. At the same time, adopting interpretable ML models could improve our understanding of the decision-making logic, allowing us to understand which input is critical for the final decisions. Moreover, strategies to mitigate the performance impact of these intelligent models, which by their nature can impose computational overhead, need to be further analyzed. Therefore, allowing more customizable networks can enable network programmers and even less-experienced users to have networks that can meet their requirements. This can be achieved by exploiting the IBN paradigm and by intersecting it with GenAI strategies, such as Large (or Small) Language Models, to improve intent understanding and translation. However, while promising, this approach presents notable challenges. For one, these models are resource-intensive to train and may require specialized techniques, such as prompting or fine-tuning, that align with specific network topologies. Additionally, the computational demands of these methods are not negligible, necessitating in-depth analysis to balance accuracy with the hardware capabilities.

# List of Acronyms

**Acronyms / Abbreviations**

| | |
|---|---|
| *ABR* | Adaptive Bitrate |
| *AI* | Artificial Intelligence |
| *API* | Application Programming Interface |
| *APN* | Application Aware Networking |
| *BDD* | Behaviour Driven Development |
| *BMV*2 | Behavioral Model Version 2 |
| *BPP* | Big Packet Protocol |
| *CDF* | Cumulative Distribution Function |
| *CO*2 | Carbon dioxide |
| $D-Tree$ | Decision Tree |
| *DCTCP* | Data-center TCP |
| *DL* | Deep Learning |
| *DNN* | Deep Neural Network |
| *DPI* | Deep Packet Inspection |
| *DPU* | Data Processing Unit |
| *DRL* | Deep Reinforcement Learning |

| | |
|---|---|
| *DSCP* | Differentiated Services Code Point |
| *DT* | Decision Tree |
| *ECMP* | Equal-Cost Multi-Path |
| *ECN* | Explicit Congestion Notification |
| *FCT* | Flow Completion Time |
| *FIFO* | First-In First-Out |
| *HPC* | High-Performance Computing |
| *IBN* | Intent-Based Networking |
| *IDL* | Intent Definition Language |
| *IoT* | Internet of Things |
| *IPC* | Inter-process communication |
| *kWh* | Kilowatt-hour |
| *LDP* | Label Distribution Protocol |
| *LLMs* | Large Language Models |
| *LoC* | Lines of Code |
| *LP* | Load Profiling |
| *LRU* | Least Recently Used |
| *LUT* | Look-Up Table |
| *MARL* | Multi-Agent Reinforcement Learning |
| *MIB* | Management Information Base |
| *ML* | Machine Learning |
| *MLP* | Multi-Layer Perceptron |
| *MPLS* | MultiProtocol Label Switching |

| | |
|---|---|
| *NGFW* | Next Generation Firewall |
| *NICS* | Network Interface Cards |
| *NLP* | Natural Language Processing |
| *NN* | Neural Network |
| *P*4 | Programming Protocol-Independent Packet Processors |
| *PISA* | Programmable Independent Switch Architecture |
| $PR-AUC$ | Precision-Recall Area Under the Curve |
| *QoS* | Quality of Service |
| *ReLU* | Rectified Linear Unit |
| *RF* | Random Forest |
| *RL* | Reinforcement Learning |
| *RTT* | Round-Trip Time |
| *SDN* | Software Defined Networking |
| *SFW* | Stateful Firewall |
| *SGD* | Stochastic Gradient Descent |
| *SRH* | Segment Routing Headers |
| *SVM* | Support Vector Machine |
| *ToR* | Top of Rack |
| *ToS* | Type of Service |
| *WAN* | Wide Area Networks |
| *ZTD* | Zero Touch Deployment |

# References


[1] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. A survey on software-defined networking. *IEEE Communications Surveys & Tutorials*, 17(1):27–51, 2014.

[2] Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy. Network innovation using openflow: A survey. *IEEE communications surveys & tutorials*, 16(1):493–512, 2013.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[4] Alessio Sacco, Flavio Esposito, and Guido Marchetto. On control and data plane programmability for data-driven networking. In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6. IEEE, 2021.

[5] P4Runtime Spec. Accessed: 2024-12-6.

[6] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, 212:103561, 2023.

[7] Anteneh A Gebremariam, Muhammad Usman, and Marwa Qaraqe. Applications of artificial intelligence and machine learning in the area of sdn and nfv: A survey. In *2019 16th International Multi-Conference on Systems, Signals & Devices (SSD)*, pages 545–549. IEEE, 2019.

[8] Alexandru Vulpe, Cosmin Dobrin, Apostol Stefan, and Alexandru Caranica. Ai/ml-based real-time classification of software defined networking traffic. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–7, 2023.

[9] Apoorv Shukla, Kevin Nico Hudemann, Artur Hecker, and Stefan Schmid. Runtime verification of p4 switches with reinforcement learning. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pages 1–7, 2019.

[10] Junjie Geng, Jinyao Yan, Yangbiao Ren, and Yuan Zhang. Design and implementation of network monitoring and scheduling architecture based on p4. In *Proceedings of the 2nd International Conference on Computer Science and Application Engineering*, pages 1–6, 2018.

[11] Bowei Guan and Shan-Hsiang Shen. Flowspy: An efficient network monitoring framework using p4 in software-defined networks. In *2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall)*, pages 1–5. IEEE, 2019.

[12] Chih-Heng Ke and Shih-Jung Hsu. Load balancing using p4 in software-defined networks. *Journal of Internet Technology*, 21(6):1671–1679, 2020.

[13] Antonino Angi, Alessio Sacco, Flavio Esposito, Guido Marchetto, and Alexander Clemm. Howdah: Load profiling via in-band flow classification and p4. In *2022 18th International Conference on Network and Service Management (CNSM)*, pages 46–54. IEEE, 2022.

[14] Ya Gao and Zhenling Wang. A review of p4 programmable data planes for network security. *Mobile Information Systems*, 2021(1):1257046, 2021.

[15] Linyih Teng, Chi-Hsiang Hung, and Charles H-P Wen. P4sf: A high-performance stateful firewall on commodity p4-programmable switch. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5. IEEE, 2022.

[16] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. Elastic switch programming with P4All. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 168–174, 2020.

[17] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 731–747, 2021.

[18] Antonino Angi, Alessio Sacco, Flavio Esposito, Guido Marchetto, and Alexander Clemm. Load profiling via in-band flow classification and p4 with howdah. *IEEE Transactions on Network and Service Management*, 2023.

[19] I. Matta and A. Bestavros. A load profiling approach to routing guaranteed bandwidth flows. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications*, volume 3, pages 1014–1021, 1998.

[20] Agnese V. Ventrella, Flavio Esposito, and L. Alfredo Grieco. Load profiling and migration for effective cyber foraging in disaster scenarios with formica. In *4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 80–87, 2018.

[21] Siddhartha Sen, David Shue, Sunghwan Ihm, and Michael J Freedman. Scalable, optimal flow routing in datacenters via local link balancing. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 151–162, 2013.

[22] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, et al. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication (SIGCOMM '14)*, page 503–514, New York, NY, USA, 2014. ACM.

[23] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research (SOSR '16)*, pages 1–12. Association for Computing Machinery, 2016.

[24] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 89–92. San Jose, USA, 2010.

[25] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[26] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized" zero-queue" datacenter network. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication (SIGCOMM '14)*, pages 307–318. ACM New York, NY, USA, 2014.

[27] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication (SIGCOMM '13)*, pages 15–26. ACM New York, NY, USA, 2013.

[28] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic Load Balancing without Packet Reordering. *ACM SIGCOMM Computer Communication Review*, 37(2):51–62, March 2007.

[29] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[30] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. Per-packet load-balanced, low-latency routing for clos-based data center networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies (CoNEXT '13)*, pages 49–60, 2013.

[31] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. DRILL: Micro Load Balancing for Low-Latency Data Center

Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*, page 225–238, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-aware load balancing at the virtual edge. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '17)*, pages 323–335, 2017.

[33] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. *ACM SIGCOMM Computer Communication Review*, 45(4):465–478, 2015.

[34] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 399–412, 2013.

[35] Andrew R Curtis, Wonho Kim, and Praveen Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM '11)*, pages 1629–1637. IEEE, 2011.

[36] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the seventh conference on emerging networking experiments and technologies (CoNEXT '11)*, pages 1–12, 2011.

[37] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, et al. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1345–1358, 2022.

[38] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 701–721. USENIX Association, 2020.

[39] Zhenbin Li, Shuping Peng, Daniel Voyer, Cong Li, Peng Liu, Chang Cao, and Gyan Mishra. Application-aware Networking (APN) Framework. Internet-Draft draft-li-apn-framework-06, Internet Engineering Task Force, September 2022. Work in Progress.

[40] Alessio Sacco, Flavio Esposito, Guido Marchetto, Grant Kolar, and Kate Schwetye. On Edge Computing for Remote Pathology Consultations and Computations. *IEEE Journal of Biomedical and Health Informatics*, 24(9):2523–2534, 2020.

[41] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, 2020.

[42] Mohammad Noormohammadpour and Cauligi S Raghavendra. Datacenter traffic control: Understanding techniques and tradeoffs. *IEEE Communications Surveys & Tutorials*, 20(2):1492–1525, 2017.

[43] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[44] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, et al. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM Conference on Data communication (SIGCOMM '09)*, pages 51–62. Association for Computing Machinery, 2009.

[45] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement (IMC '09)*, pages 202–208, 2009.

[46] Yi Li, Hong Liu, Wenjun Yang, Dianming Hu, Xiaojing Wang, and Wei Xu. Predicting inter-data-center network traffic using elephant flow and sublink information. *IEEE Transactions on Network and Service Management*, 13(4):782–792, 2016.

[47] Kokouvi Benoit Nougnanke, Yann Labit, and Marc Bruyere. ML-based Incast Performance Optimization in Software-Defined Data Centers. In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6. IEEE, 2021.

[48] Anshuman Chhabra and Mariam Kiran. Classifying elephant and mice flows in high-speed scientific networks. *Proc. of the International Workshop on Innovating the Network for Data Intensive Science (INDIS '17)*, pages 1–8, 2017.

[49] Cisco Systems Inc. Cisco global cloud index: Forecast and methodology, 2016–2021, 2021.

[50] Alessio Sacco, Flavio Esposito, and Guido Marchetto. RoPE: An Architecture for Adaptive Data-Driven Routing Prediction at the Edge. *IEEE Transactions on Network and Service Management*, 17(2):986–999, 2020.

[51] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, et al. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM Conference on Data communication (SIGCOMM '10)*, pages 339–350. Association for Computing Machinery, 2010.

[52] Alessio Sacco, Matteo Flocco, Flavio Esposito, and Guido Marchetto. Supporting Sustainable Virtual Network Mutations with Mystique. *IEEE Transactions on Network and Service Management*, 18(3):2714–2727, 2021.

[53] Tatsuya Mori, Masato Uchida, Ryoichi Kawahara, Jianping Pan, and Shigeki Goto. Identifying elephant flows through periodically sampled packets. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement (IMC '04)*, pages 115–120. Association for Computing Machinery, 2004.

[54] Steffen Gebert, Stefan Geissler, Thomas Zinner, Anh Nguyen-Ngoc, Stanislav Lange, and Phuoc Tran-Gia. Zoom: Lightweight sdn-based elephant detection. In *28th International Teletraffic Congress (ITC 28)*, volume 2, pages 1–6. IEEE, 2016.

[55] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The p4-> netfpga workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 1–9, 2019.

[56] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in sdn-enabled switches. In *Proceedings of the 1st ACM SIGCOMM symposium on software defined networking research*, pages 1–6, 2015.

[57] Ibrahim Matta, Azer Bestavros, and Marwan Krunz. Load profiling based routing for guaranteed bandwidth flows. *European Transactions on Telecommunications*, 10(2):165–181, 1999.

[58] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic In-Band Network Telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication (SIGCOMM '20)*, pages 662–680. ACM New York, NY, USA, 2020.

[59] Alessio Sacco, Matteo Flocco, Flavio Esposito, and Guido Marchetto. Partially Oblivious Congestion Control for the Internet via Reinforcement Learning. *IEEE Transactions on Network and Service Management*, 20(2):1644–1659, 2022.

[60] Mario Baldi, Guido Marchetto, and Yoram Ofek. A Scalable Solution for Engineering Streaming Traffic in the Future Internet. *Computer Networks*, 51(14):4092 –4111, 2007.

[61] Stewart Bryant and Alexander Clemm. Token cell routing: A new sub-ip layer protocol. In *2021 17th International Conference on Network and Service Management (CNSM)*, pages 153–159. IEEE, 2021.

[62] Richard Li, Kiran Makhijani, and Lijun Dong. New ip: A data packet framework to evolve the internet. In *2020 IEEE 21st International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2020.

[63] Richard Li, Alexander Clemm, Uma Chunduri, Lijun Dong, and Kiran Makhijani. A New Framework and Protocol for Future Networking Applications. In *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*, NEAT '18, page 21–26, New York, NY, USA, 2018. Association for Computing Machinery.

[64] Ipv6 statistics. Accessed: 2023-1-7.

[65] Ipv6 usage. Accessed: 2023-1-7.

[66] Steve Deering and Robert Hinden. RFC8200: Internet protocol, version 6 (IPv6) specification, 2017.

[67] C Filsfils, D Dukes, S Previdi, J Leddy, S Matsushima, and D Voyer. IPv6 segment routing header (SRH): RFC8754, 2020.

[68] Sotiris B Kotsiantis. Decision trees: a recent overview. *Artificial Intelligence Review*, 39(4):261–283, 2013.

[69] Max Bramer. Avoiding overfitting of decision trees. *Principles of data mining*, pages 119–134, 2007.

[70] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-Agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, pages 455–468. USENIX Association, 2015.

[71] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28(4):1698–1711, 2020.

[72] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[73] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial intelligence*, 101(1-2):285–297, 1998.

[74] Suryakanthi Tangirala. Evaluating the impact of gini index and information gain on classification using decision tree classifier algorithm. *International Journal of Advanced Computer Science and Applications*, 11(2):612–619, 2020.

[75] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.

[76] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[77] Ryu controller. Accessed: 2024-12-6.

[78] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-Based server load balancing gone wild. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 11)*, 2011.

[79] Mike Schlansker, Yoshio Turner, Jean Tourrilhes, and Alan Karp. Ensemble routing for datacenter networks. In *2010 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–12. IEEE, 2010.

[80] Josh Cowls, Andreas Tsamados, Mariarosaria Taddeo, and Luciano Floridi. The ai gambit: leveraging artificial intelligence to combat climate change—opportunities, challenges, and recommendations. *Ai & Society*, pages 1–25, 2021.

[81] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. Towards the systematic reporting of the energy and carbon footprints of machine learning. *Journal of Machine Learning Research*, 21(248):1–43, 2020.

[82] Victor Schmidt, Kamal Goyal, Aditya Joshi, Boris Feld, Liam Conell, Nikolas Laskaris, Doug Blank, Jonathan Wilson, Sorelle Friedler, and Sasha Luccioni. Codecarbon: estimate and track carbon emissions from machine learning computing, 2021.

[83] Antonino Angi, Alessio Sacco, Flavio Esposito, and Guido Marchetto. Roar: Routing packets in p4 switches with multi-agent decisions logic. In *2024 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, pages 63–68. IEEE, 2024.

[84] Alessio Sacco, Flavio Esposito, and Guido Marchetto. Resource Inference for Sustainable and Responsive Task Offloading in Challenged Edge Networks. *IEEE Transactions on Green Communications and Networking*, 5(3):1114–1127, 2021.

[85] Yan-Jing Wu, Po-Chun Hwang, Wen-Shyang Hwang, and Ming-Hua Cheng. Artificial intelligence enabled routing in software defined networking. *Applied Sciences*, 10(18):6564, 2020.

[86] Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Ying-Chang Liang, and Dong In Kim. Applications of deep reinforcement learning in communications and networking: A survey. *IEEE Communications Surveys & Tutorials*, 21(4):3133–3174, 2019.

[87] Tingting Fu, Chaoyu Wang, and Nan Cheng. Deep-learning-based joint optimization of renewable energy storage and routing in vehicular energy network. *IEEE Internet of Things Journal*, 7(7):6229–6241, 2020.

[88] Chenyi Liu, Mingwei Xu, Yuan Yang, and Nan Geng. DRL-OR: Deep reinforcement learning-based online routing for multi-type service requirements. In *IEEE INFOCOM - IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[89] Rashid Amin, Elisa Rojas, Aqsa Aqdus, Sadia Ramzan, David Casillas-Perez, and Jose M Arco. A Survey on Machine Learning Techniques for Routing Optimization in SDN. *IEEE Access*, 9:104582–104611, 2021.

[90] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 701–721. USENIX Association, 2020.

[91] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication (SIGCOMM '20)*, pages 296–309. ACM New York, NY, USA, 2020.

[92] Jeferson Santiago da Silva, François-Raymond Boyer, Laurent-Olivier Chiquette, and JM Pierre Langlois. Extern Objects in P4: An Rohc Header Compression Scheme Case Study. In *4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 517–522. IEEE, 2018.

[93] Justus Rischke, Peter Sossalla, Hani Salah, Frank HP Fitzek, and Martin Reisslein. QR-SDN: Towards Reinforcement Learning States, Actions, and Rewards for Direct Flow Routing in Software-Defined Networks. *IEEE Access*, 8:174773–174791, 2020.

[94] Daniela M Casas-Velasco, Oscar Mauricio Caicedo Rendon, and Nelson LS da Fonseca. Intelligent Routing Based on Reinforcement Learning for Software-Defined Networking. *IEEE Transactions on Network and Service Management*, 18(1):870–881, 2020.

[95] Changhe Yu, Julong Lan, Zehua Guo, and Yuxiang Hu. Drom: Optimizing the routing in software-defined networks with deep reinforcement learning. *IEEE Access*, 6:64533–64539, 2018.

[96] Wenzhong Li, Han Zhang, Shaohua Gao, Chaojing Xue, Xiaoliang Wang, and Sanglu Lu. Smartcc: A reinforcement learning approach for multipath tcp congestion control in heterogeneous networks. *IEEE Journal on Selected Areas in Communications*, 37(11):2621–2633, 2019.

[97] Sai Shreyas Bhavanasi, Lorenzo Pappone, and Flavio Esposito. Dealing with changes: Resilient routing via graph neural networks and multi-agent

deep reinforcement learning. *IEEE Transactions on Network and Service Management*, 20(3):2283–2294, 2023.

[98] Lei Zhao, Jiadai Wang, Jiajia Liu, and Nei Kato. Routing for Crowd Management in Smart Cities: A Deep Reinforcement Learning Perspective. *IEEE Communications Magazine*, 57(4):88–93, 2019.

[99] Bin Dai, Yuanyuan Cao, Zhongli Wu, and Yang Xu. IQoR-LSE: An Intelligent QoS On-Demand Routing Algorithm With Link State Estimation. *IEEE Systems Journal*, 16(4):5821–5830, 2022.

[100] Chengxiao Yu, Wei Quan, Deyun Gao, Yuming Zhang, Kang Liu, Wen Wu, Hongke Zhang, and Xuemin Shen. Reliable cybertwin-driven concurrent multipath transfer with deep reinforcement learning. *IEEE Internet of Things Journal*, 8(22):16207–16218, 2021.

[101] Amedeo Sapio, Marco Canini, et al. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*, pages 785–808, 2021.

[102] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications-1*, pages 183–221, 2010.

[103] Cedric Seger. An investigation of categorical variable encoding techniques in machine learning: binary versus one-hot and feature hashing, 2018.

[104] 48x25gb+8x100gb, intel tofino p4 programmable bare metal switch: Asterfusion, July 2022.

[105] Antonino Angi, Alessio Sacco, Flavio Esposito, and Guido Marchetto. Routing with art: Adaptive routing for p4 switches with in-network decision trees. In *2024 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2024.

[106] Ons Aouedi, Thai-Hoc Vu, Alessio Sacco, Dinh C Nguyen, Kandaraj Piamrat, Guido Marchetto, and Quoc-Viet Pham. A Survey on Intelligent Internet of Things: Applications, Security, Privacy, and Future Directions. *IEEE Communications Surveys & Tutorials*, 2024.

[107] Sai Shreyas Bhavanasi, Lorenzo Pappone, and Flavio Esposito. Dealing with changes: Resilient routing via graph neural networks and multi-agent deep reinforcement learning. *IEEE Transactions on Network and Service Management*, pages 1–1, 2023.

[108] Zoubir Mammeri. Reinforcement Learning Based Routing in Networks: Review and Classification of Approaches. *Ieee Access*, 7:55916–55950, 2019.

[109] Daniela M Casas-Velasco, Oscar Mauricio Caicedo Rendon, and Nelson LS da Fonseca. DRSIR: A Deep Reinforcement Learning Approach for Routing in Software-Defined Networking. *IEEE Transactions on Network and Service Management*, 19(4):4807–4820, 2021.

[110] Francesco Musumeci, Ali Can Fidanci, Francesco Paolucci, Filippo Cugini, and Massimo Tornatore. Machine-Learning-Enabled DDoS Attacks Detection in P4 Programmable Networks. *Journal of Network and Systems Management*, 30:1–27, 2022.

[111] Yung-Sheng Lu and Kate Ching-Ju Lin. Enabling Inference Inside Software Switches. In *20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 1–4. IEEE, 2019.

[112] Changgang Zheng, Xinpeng Hong, Damu Ding, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. In-Network Machine Learning Using Programmable Network Devices: A Survey. *IEEE Communications Surveys & Tutorials*, 2023.

[113] Antonino Angi, Alessio Sacco, Flavio Esposito, Guido Marchetto, and Alexander Clemm. NAIL: A Network Management Architecture for Deploying Intent into Programmable Switches. *IEEE Communications Magazine*, 62(6):28–34, 2023.

[114] Elie F Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE access*, 9:87094–87155, 2021.

[115] Doriana Monaco, Alessio Sacco, Enrico Alberti, Guido Marchetto, and Flavio Esposito. A collaborative and distributed learning-based solution to autonomously plan computer networks. In *2023 26th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 1–5. IEEE, 2023.

[116] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. Switches for hire: Resource scheduling for data center in-network computing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 268–285, 2021.

[117] Lorenzo De Marinis, Emilio Paolini, Rana Abu Bakar, Filippo Cugini, and Francesco Paolucci. Cascaded look up table distillation of p4 deep neural network switches. In *GLOBECOM 2023- IEEE Global Communications Conference*, pages 2111–2116. IEEE, 2023.

[118] Z. Meng, M. Wang, J. Bai, M. Xu, H. Mao, and H. Hu. Interpreting Deep Learning-Based Networking Systems. In *Proc. of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20)*, pages 154–171, 2020.

[119] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable Reinforcement Learning via Policy Extraction. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31, 2018.

[120] Henning Stubbe, Sebastian Gallenmüller, Manuel Simon, Eric Hauser, Dominik Scholz, and Georg Carle. Keeping up to date with p4runtime: An analysis of data plane updates on p4 switches. In *International Federation for Information Processing (IFIP) Networking 2023 Conference (IFIP Networking 2023)*, 2023.

[121] Antonino Angi, Alessio Sacco, Flavio Esposito, Guido Marchetto, and Alexander Clemm. Nlp4: An architecture for intent-driven data plane programmability. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pages 25–30. IEEE, 2022.

[122] Andrew T Campbell, Herman G De Meer, Michael E Kounavis, Kazuho Miki, John B Vicente, and Daniel Villela. A survey of programmable networks. *ACM SIGCOMM Computer Communication Review*, 29(2):7–23, 1999.

[123] Timothy L Hinrichs, Natasha S Gude, Martin Casado, John C Mitchell, and Scott Shenker. Practical declarative network management. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 1–10, 2009.

[124] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *ACM Sigplan Notices*, 46(9):279–291, 2011.

[125] Gal Barel and Ralf Herwig. Netcore: a network propagation approach using node coreness. *Nucleic acids research*, 48(17):e98–e98, 2020.

[126] Justus Rischke and Hani Salah. Chapter 6 - software-defined networks. In Frank H.P. Fitzek, Fabrizio Granelli, and Patrick Seeling, editors, *Computing in Communication Networks*, pages 107–118. Academic Press, 2020.

[127] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, April 2022. USENIX Association.

[128] Engin Zeydan and Yekta Turk. Recent advances in intent-based networking: A survey. In *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, pages 1–5. IEEE, 2020.

[129] Mohammad Riftadi and Fernando Kuipers. P4I/O: Intent-based Networking with P4. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 438–443. IEEE, 2019.

[130] Ibrahim Matta and Azer Bestavros. A load profiling approach to routing guaranteed bandwidth flows. In *Proceedings. IEEE INFOCOM'98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No. 98*, volume 3, pages 1014–1021. IEEE, 1998.

[131] Flavio Esposito, Jiayi Wang, Chiara Contoli, Gianluca Davoli, Walter Cerroni, and Franco Callegati. A behavior-driven approach to intent specification for software-defined infrastructure management. In *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–6. IEEE, 2018.

[132] Yifan Li, Chengjun Jia, Xiaohe Hu, and Jun Li. Mahjong: A generic framework for network data plane verification. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, pages 52–58, 2021.

[133] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 1–13, 2013.

[134] Robert Soulé, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Managing the network with merlin. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013.

[135] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. Refining network intents for self-driving networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, pages 15–21, 2018.

[136] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[137] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review*, 45(4):29–42, 2015.

[138] Azzam Alsudais and Eric Keller. Hey network, can you understand me? In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 193–198. IEEE, 2017.

[139] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. {Net2Text}:{Query-Guided} summarization of network forwarding

behaviors. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 609–623, 2018.

[140] Albert Gatt and Emiel Krahmer. Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. *Journal of Artificial Intelligence Research*, 61:65–170, 2018.

[141] Vimala Balakrishnan and Ethel Lloyd-Yemoh. Stemming and lemmatization: a comparison of retrieval performances. 2014.

[142] Ivan Boban, Alen Doko, and Sven Gotovac. Sentence retrieval using stemming and lemmatization with different length of the queries. *Advances in Science, Technology and Engineering Systems Journal*, 5(3):349–354, 2020.

[143] Matt W Gardner and SR Dorling. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment*, 32(14-15):2627–2636, 1998.

[144] Hassan Ramchoun, Youssef Ghanou, Mohamed Ettaouil, and Mohammed Amine Janati Idrissi. Multilayer perceptron: Architecture optimization and training. 2016.

[145] Leonard Peter Binamungu, Suzanne M Embury, and Nikolaos Konstantinou. Maintaining behaviour driven development specifications: Challenges and opportunities. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 175–184. IEEE, 2018.

[146] Tim Storer and Ruxandra Bob. Behave nicely! automatic generation of code for behaviour driven development test suites. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 228–237. IEEE, 2019.

[147] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with P4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, 212:103561, 2022.

[148] Aris Leivadeas and Matthias Falkner. A Survey on Intent-Based Networking. *IEEE Communications Surveys & Tutorials*, 25(1):625–655, 2022.

[149] Zijun Hang, Mei Wen, Yang Shi, and Chunyuan Zhang. Programming Protocol-Independent Packet Processors High-Level Programming (P4HLP): Towards Unified High-Level Programming for a Commodity Programmable Switch. *Electronics*, 8(9):958, 2019.

[150] Arthur Selle Jacobs, Ricardo J Pfitscher, Rafael Hengen Ribeiro, Ronaldo A Ferreira, Lisandro Zambenedetti Granville, Walter Willinger, and Sanjay G Rao. Hey, Lumi! Using Natural Language for Intent-Based Network Management. In *USENIX Annual Technical Conference*, pages 625–639, 2021.

[151] Introducing ChatGPT. Accessed: 2024-12-6.

[152] Introducing Llama: A foundational, 65-billion-parameter language model. Accessed: 2024-12-6.

[153] Ali Borji. A categorical archive of chatgpt failures. *arXiv preprint arXiv:2302.03494*, 2023.

[154] Hongyu Zhang. An Investigation of the Relationships Between Lines of Code and Defects (ICSM). In *2009 IEEE International Conference on Software Maintenance*, pages 274–283, 2009.

[155] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular Switch Programming Under Resource Constraints. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–15, 2022.

[156] Rakesh Datta, Sean Choi, Anurag Chowdhary, and Younghee Park. P4Guard: Designing P4 Based Firewall. In *2018 IEEE Military Communications Conference (MILCOM)*, pages 1–6. IEEE, 2018.

# Appendix A

# List of Publications

The following is the complete list of publications carried out during the Ph.D.

- Angi, Antonino, Alessio Sacco, Flavio Esposito, and Guido Marchetto. "Routing with ART: Adaptive Routing for P4 Switches With In-Network Decision Trees." In 2024 IEEE Global Communications Conference (GLOBECOM). IEEE, 2024.
- Angi, Antonino, Alessio Sacco, Flavio Esposito, and Guido Marchetto. "ROAR: Routing Packets in P4 Switches With Multi-Agent Decisions Logic." In 2024 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN), pp. 63-68. IEEE, 2024.
- Angi, Antonino, Alessio Sacco, Flavio Esposito, Guido Marchetto, and Alexander Clemm. "NAIL: A Network Management Architecture for Deploying Intent into Programmable Switches." IEEE Communications Magazine (2023).
- Sacco, Alessio, Antonino Angi, Guido Marchetto, and Flavio Esposito. "P4FL: An Architecture for Federating Learning with In-Network Processing." IEEE Access (2023).
- Angi, Antonino, Alessio Sacco, Enrico Alberti, Guido Marchetto, and Flavio Esposito. "RLVNA: a Platform for Experimenting with Virtual Networks Adaptations over Public Testbeds." In 2023 IEEE International Mediterranean Conference on Communications and Networking (MeditCom), pp. 406-411. IEEE, 2023.

- Angi, Antonino, Alessio Sacco, Flavio Esposito, Guido Marchetto, and Alexander Clemm. "Load Profiling via In-Band Flow Classification and P4 With Howdah." IEEE Transactions on Network and Service Management (2023).
- Sacco, Alessio, Antonino Angi, Flavio Esposito, and Guido Marchetto. "HINT: Supporting congestion control decisions with P4-driven in-band network telemetry." In 2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR), pp. 83-88. IEEE, 2023.
- Angi, Antonino, Alessio Sacco, Flavio Esposito, Guido Marchetto, and Alexander Clemm. "Howdah: Load profiling via in-band flow classification and P4." In 2022 18th International Conference on Network and Service Management (CNSM), pp. 46-54. IEEE, 2022.
- Angi, Antonino, Alessio Sacco, Flavio Esposito, Guido Marchetto, and Alexander Clemm. "NLP4: An architecture for intent-driven data plane programmability." In 2022 IEEE 8th International Conference on Network Softwarization (NetSoft), pp. 25-30. IEEE, 2022.