P4eBalancer: Leveraging P4 and eBPF for Optimized Load Balancing with Network and Host Insights

Cleiton Puttlitz[1], **Alberto Schaeffer-Filho**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{cputtlitz,alberto}@inf.ufrgs.br

Abstract. *Load balancing solutions monitor the infrastructure and redistribute network traffic or application requests in order to adapt to variations in network or server loads. However, these schemes rely on monitoring mechanisms that observe only specific segments of the infrastructure, i.e., network core or end-hosts, which may lead them to make suboptimal decisions. In this paper, we present P4eBalancer, a system that leverages an advanced monitoring mechanism capable of observing the entire distributed infrastructure to make load-balancing decisions. P4eBalancer incorporates an intelligent offline control loop, operating in the control plane, making global decisions with a reinforcement learning agent, and an online control loop in edge switches, which makes quick decisions and reacts to network variations. We implemented a prototype using the BMv2 P4 software switch. Our results show that P4eBalancer can make better load balancing decisions using end-to-end metrics.*

1. Introduction

Data centers typically host a wide range of applications. To ensure high performance and reliability, these services are replicated across multiple servers and connected through multiple paths. A load balancer system must appropriately distribute traffic across these multiple components (such as links and end-servers), which provide the same functionality or service [Zhang et al. 2018]. Traditional approaches, such as Equal-Cost MultiPath (ECMP) and Weighted-Cost MultiPath (WCMP) [Zhou et al. 2014], fail to adapt their strategies to traffic dynamics, leading to underutilized resources and an unbalanced load [Benson et al. 2010]. The emergence of programmable data planes [Bosshart et al. 2014] and in-kernel processing [eBPF 2024] has introduced new opportunities to gain insights into distributed network infrastructures and provide better conditions for load balancers to adapt to traffic dynamics [Katta et al. 2016].

Most load-aware solutions operate in a *centralized* manner in the control plane [Robin and Khan 2022, Al-Fares et al. 2010, Casas-Velasco et al. 2022]. These solutions rely on a comprehensive view of the infrastructure to make global changes, such as selecting the best routes. However, they suffer from high latency, and do not react quickly to network conditions. On the other hand, *decentralized* schemes make local decisions on hosts or switches [Alizadeh et al. 2014, Katta et al. 2016, Hsu et al. 2020b, Tajbakhsh et al. 2022, Tajbakhsh et al. 2024]. While they quickly adapt to dynamic changes in the network, collecting global information remains challenging.

In addition, the approaches discussed above typically rely on monitoring mechanisms that observe only specific segments of the infrastructure:

network core [Alizadeh et al. 2014, Katta et al. 2016, Hsu et al. 2020b] or end-hosts [Tajbakhsh et al. 2022, Eisenbud et al. 2016]. For network monitoring, telemetry-based solutions such as INT (In-band Network Telemetry) [Kim et al. 2015] leverage specialized headers added to packets as they traverse the network to collect fine-grained telemetry data in real-time about network routing devices. More recently, eBPF [eBPF 2024] has been used to enable the monitoring of a wide range of system events at end-hosts, providing instant visibility into all host activities. Based on monitoring systems that observe only specific segments of the infrastructure, current load balancing solutions do not obtain a comprehensive view of the entire distributed infrastructure. Without end-to-end visibility, these solutions may make inaccurate decisions, such as ignoring the condition of a potentially overloaded server.

In this work, we present P4eBalancer, a load balancing mechanism for data centers based on information from both the network and the end-servers. The system consists of two main components: (1) an intelligent *offline control-loop* operating in the control plane to make global decisions, and (2) an *online control-loop* in edge switches to react quickly to network variations. P4eBalancer relies on a telemetry subsystem that uses INT to collect information from the switches and eBPF to gather metrics from end servers [Puttlitz et al. 2024]. In particular, the offline control-loop employs a reinforcement learning agent that receives these infrastructure metrics and selects a set of *top-k* best paths, which are then installed in the switches. Moreover, the online control-loop stores a subset of the latest metrics for the installed paths and uses this information to rapidly redirect requests to optimal paths as network or server conditions change.

Our contributions can be summarized as follows: *(i)* A novel load balancing system that employs reinforcement learning that considers *end-to-end metrics* to select optimal paths. *(ii)* A *decentralized mechanism* capable of making intelligent global decisions in the control plane while quickly detecting and reacting to network changes in the data plane. *(iii)* A prototype implementation and evaluation that shows the benefits of combining network and host metrics in load balancing.

The remainder of this paper is organized as follows: Section 2 provides the background context of this work. Section 3 outlines the proposed approach for load balancing. Section 4 presents implementation details and experimental results. Section 5 discusses the related work. Finally, Section 6 presents the concluding remarks.

2. Background

2.1. Programmable Data Plane and INT

P4 (Programming Protocol-Independent Packet Processors) [Bosshart et al. 2014] is a domain-specific programming language that describes how a packet should be processed by the data plane of a programmable forwarding device. This flexibility facilitates the creation of custom functions such as In-band Network Telemetry (INT) [Kim et al. 2015]. INT is a network monitoring framework that collects and reports network information directly from the data plane. At each hop, the switch inserts an *INT header* containing its telemetry information, such as ID, queue size, link utilization, and hop latency. Before the packet is delivered to the destination, that is, on the last hop, the original packet is restored and sent to the destination. In addition, a cloned packet containing only the INT

metadata is sent to an *INT Engine*, which analyzes the collected data to provide insights into the state of the data plane.

2.2. Extended Berkeley Packet Filter

eBPF (Extended Berkeley Packet Filter) [eBPF 2024] is a technology that enables running small, custom programs inside the kernel, modifying its behavior in real-time. eBPF programs are executed in response to an event in the kernel using *hooks*. Specifically, in this work, we are interested in the network events captured using the XDP and TC hooks (Figure 1) to perform operations on the packet (e.g., redirect, drop, modify). The *XDP hook* enables the execution of an eBPF program immediately upon packet reception, directly into the network interface card (if it supports this function), or even before the kernel initiates any other processing, ensuring high performance [Abranches et al. 2021]. In turn, programs attached to the *TC hook* can be triggered from entry and exit points in the network stack and have metadata added by kernel processing. Programs attached at different points in the kernel or in the user space can store and share information using *Maps*, which is a generic key-value data structure that supports multiple data types.

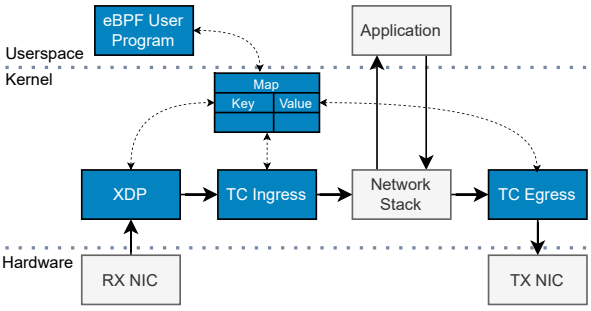


Figure 1. Overview of the eBPF network hooks

2.3. Reinforcement Learning

Reinforcement learning (RL) is a machine learning technique that allows agents to learn to achieve a goal in an interactive environment through trial and error, without requiring a training dataset [Sutton and Barto 1998]. At each iteration, the agent observes a state $s \in S$ and selects an action $a \in A$ to be executed. After executing the action, the environment transitions to a new state $s' \in S$, and the agent receives a reward $r \in R$ that evaluates the effectiveness of the action. The agent aims to learn an optimal policy $\pi : S \rightarrow A$ that maximizes the expected cumulative reward over time. To achieve this, the agent initially explores the environment, tests different actions, and exploits the acquired knowledge to select the most rewarding actions.

3. P4eBalancer Design

In this section we describe the design of P4eBalancer, whose key insight is to consider information from both the network and end-servers in order to perform load balancing. First, we present the fundamental challenges that P4eBalancer must overcome. Next, we present the overview and workflow of our approach. Finally, we present details on how the control plane and data plane work, as well as how they cooperate to form a hybrid strategy for load balancing in data centers.

3.1. Challenges

Challenge #1: Comprehensive load balance using end-to-end metrics. Network-level load balancing aims to balance traffic uniquely in terms of network links. However, they do not consider the condition of the destination host, which may be overloaded. To address this, we employ an efficient monitoring system capable of gathering metrics from both the network and the end-servers [Puttlitz et al. 2024]. Firstly, an INT module appends monitoring information to client request traversing the network. Secondly, when a server receives a request, an eBPF module extracts the INT data and gathers metrics from the end-host, and, when the response is issued, the eBPF module will insert both the INT data and the host information into the response packets. Finally, as the response traverses back to the client, INT information is also added to the packet. By integrating these network and host metrics, we can make more informed load balancing decisions.

Challenge #2: React quickly to performance variations. INT enables load balancing solutions to gain real-time insights into the network. When applied in centralized load balancer approaches, it provides full network visibility but it is slow to react to network variations. Conversely, distributed approaches can make quick decisions but have only local visibility. To address this challenge, we employ a hybrid solution: an *online control-loop* at the switches themselves which employs a heuristic mechanism for fast decision-making to react to network variations, and an *offline control-loop* that is periodically executed by the controller which uses a reinforcement learning agent to analyze the state of the entire infrastructure and take global actions.

Challenge #3: Distributed telemetry information processing. Incorporating network and server metrics into routing decisions means that a large amount of data needs to be stored and analyzed. This makes it difficult to perform this task on network devices, which are subject to several constraints such as limited memory and stages in the pipeline. On the other hand, the control plane provides sufficient computing and storage resources to accomplish this. We explore a division of labor between switches and the controller: while the controller receives *all telemetry data* obtained by the data plane to make decisions about the optimal distribution of traffic using a reinforcement learning agent, the switch maintains efficient data structures (sketches) to store *only the latest metrics* for the installed *top-k* paths. This information is akin to a "cache" and is used to rapidly redirect requests to optimal paths as the condition of the network or servers changes.

3.2. Overview and Workflow

Figure 2 presents an overview of our approach. P4eBalancer is a hybrid system that operates both in the control and data planes: the control plane receives the telemetry information and applies a reinforcement learning agent to select the *top-k* best paths, which are subsequently installed in the switch; the data plane in an edge switch keeps the latest metrics for the *top-k* installed paths and uses this information to split traffic according to the load of links and end-servers.

P4eBalancer relies on an end-to-end monitoring system [Puttlitz et al. 2024] that uses INT and eBPF to collect network and end-server metrics associated with a client request/response (e.g., this request could be a machine learning job or another

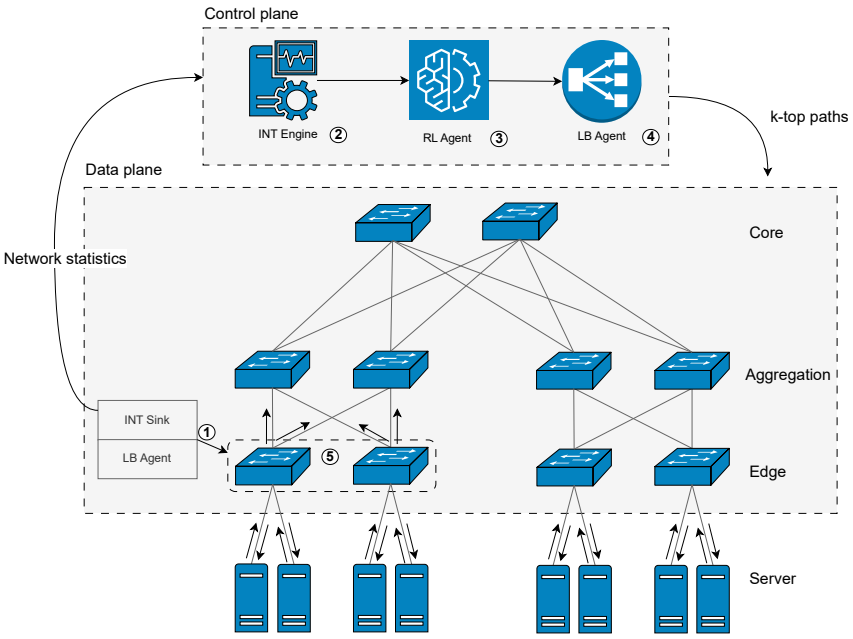


Figure 2. P4eBalancer overview

data center application request). As the client request/response completes, the *INT Sink* switch extracts the end-to-end telemetry information from the packet and forwards it to the *INT Engine* (Step ①). The *INT Engine* processes this data and generates a new state representation, which serves as input to the reinforcement learning agent (Step ②). This state representation includes information about the current network and end-host state. The RL agent uses this input to select an action, specifically, choosing the k best paths from the set of candidate paths (Step ③). Once selected, these paths are installed in the switches (Step ④). Finally, the edge switch dynamically chooses one of the configured paths to forward subsequent client requests based on real-time metrics (Step ⑤).

3.3. Intelligent Control Plane

Figure 3 provides an overview of the control plane in P4eBalancer. The *INT Engine* receives the end-to-end telemetry data collected through our telemetry subsystem [Puttlitz et al. 2024], and uses this information to represent the current state of the infrastructure. The reinforcement learning agent then analyzes this state and selects the *top-k* best paths, which are subsequently installed in the data plane.

Reinforcement Learning Agent. The reinforcement learning agent is modeled by a set of states S, a set of actions A, and a reward function R. These elements form the basis of the agent's learning process, defining how it perceives the environment, makes decisions, and evaluates the effectiveness of its actions.

State. We model the state based on telemetry information collected from each switch and end-host in the network. The state S is represented by a matrix $Sw_{i,j}$ for each switch i and metric j:

$$Sw = [sw_{1,1}, \ldots, sw_{1,j}, \ldots, sw_{i,1}, \ldots, sw_{i,j}]$$

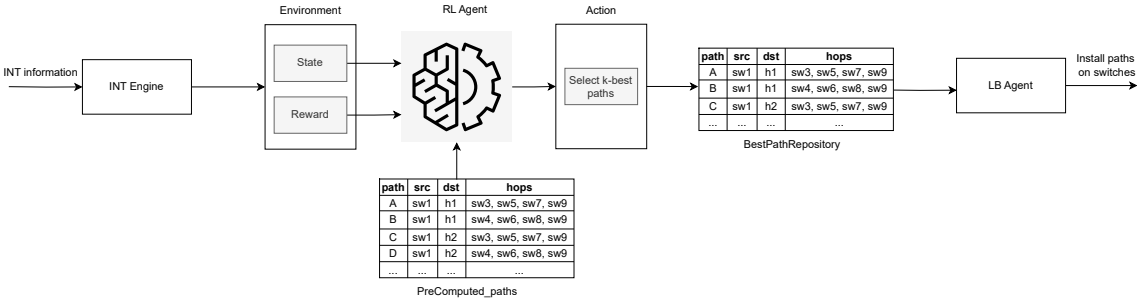


Figure 3. Control Plane overview

combined by a matrix $H_{m,n}$ of each host m and metric n:

$$H = [h_{1,1}, \ldots, h_{1,n}, \ldots, h_{m,1}, \ldots, h_{m,n}]$$

In our proof-of-concept prototype, we consider the switch metric `queue_depth` (number of packets in the queue) and the host metric `cpu_occupancy`. These metrics were chosen for their ability to represent the operational state of the components: `queue_depth` indicates potential congestion in the switch, while `cpu_occupancy` indicates the current processing capacity of the host. However, P4eBalancer is based on an extensible subsystem for collecting end-to-end telemetry, and other metrics could be considered in the future.

Actions. The action space A consists of a predefined set of end-to-end paths for each pair of source edge switch and destination host. At each decision step, the agent selects the $top - k$ paths from this predefined list. To generate the paths, we use the algorithm proposed in [Yen 1971]. By limiting the action space to a predefined set of paths, we ensure that the number of possible actions remains controlled and well-defined.

Reward. Since our goal is to achieve a balanced utilization of network resources, we design a multi-objective reward function that assigns higher values to states representing a more uniform utilization of network and end-host resources. The reward $R(s, a)$ is inversely proportional to the standard deviation of switch queue sizes and the standard deviation of end-host CPU occupancy, defined by Equation 1:

$$R(s,a) = \alpha \cdot \frac{1}{stdev(queue\_depth)} + \beta \cdot \frac{1}{stdev(cpu\_occupancy)} \tag{1}$$

The values α, β, $\in$ [0, 1] are tunable parameters useful for assigning a weight value to a specific metric in the calculation of the reward. In particular, they can be used to prioritize a more balanced utilization of *network* resources or, alternatively, a more balanced utilization of *end-host* resources. The network state may take some time to update and reflect the action taken. Thus, in order for the new state to represent the consequences of the action taken, the agent receives a reward after t time has elapsed after the action is carried out.

Best paths update. During system initialization, the `PathTable` table is populated with all precomputed paths available for selection by the agent. The

`PreComputed_paths` consists of a tuple containing the path ID, source, destination, and a set of hops. An example of an entry is: (src = sw1, dst = h1, hops = [sw2, sw3, sw4, sw5]). In this way, when the agent selects the paths, the *LB Agent* uses a SouthBound API to send the selected paths and their corresponding weights to the switch. This update is applied to the `WCMPTable` table, where the IDs of the selected paths are inserted. The update imposes that all new paths initially have the same weight. This update, however, does not affect active flows, which continue to be forwarded through the initially selected path, as detailed below.

3.4. Reactive Data Plane

Figure 4 presents the layout of the P4eBalancer switch data plane and details the steps required for load balancing. Each edge switch distributes flows over multiple paths according to path weights. A small amount of in-band telemetry metrics collected from each path is stored on the switch and used to update the weights of each path and quickly react to traffic conditions.

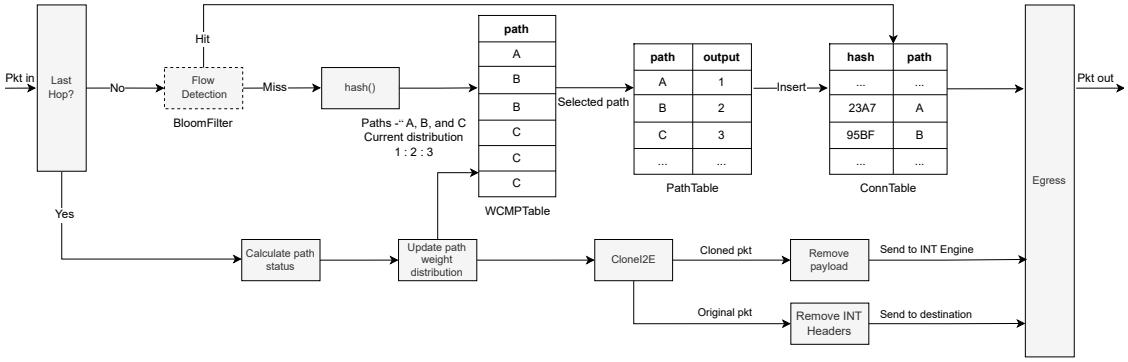


Figure 4. Data Plane overview

Path Selection. P4eBalancer selects a destination path for each incoming packet, initially identifying its flow by computing the hash of the connection identifier (i.e., the 5-tuple in the packet header). A bloom filter [Tarkoma et al. 2012] is used to check whether it is a new flow or not in a highly efficient way. If the packet is part of an existing flow (hit in the bloom filter), the packet will be forwarded to the previously selected path for the flow. The switch remembers the path by which the first packet of a flow was sent by using the connection table (`ConnTable`). Additionally, ensuring that all packets in a flow are delivered to the same destination server ensures per-connection consistency (PCC) [Barbette et al. 2020]. On the other hand, if the packet is the first packet of a new flow (miss in the bloom filter), P4eBalancer chooses a path among multiple paths according to a set of weights assigned to each path in the `WCMPTable`.

The `WCMPTable` table is populated with entries proportional to the weight of each path. Paths with lower loads are more likely to be chosen, while paths with high demand can also be selected, but with a lower probability. When a packet arrives, a hash value is computed from the header fields. The result value is an index of the `WCMPTable` table and the packet is sent along the `path` at the `WCMPTable` table entry indexed by the hash value. In both cases, the switch inserts the selected `path_id` into the packet to specify the entire path where the traffic is forwarded. Switches along the selected path

identify the next hop by this field, querying the `PathTable` table on what should be the output port for the given path ID.

Update Path Weight. Each path is continuously monitored using our telemetry subsystem. The switch uses this information to calculate the path state and adjust the weight of each path in the `WCMPTable` table. The path state is computed by a heuristic that considers metrics about the network and end-host. In our work, we calculate a path utilization using `queue_depth` (number of packets in the queue) and host `cpu_occupancy`:

$$path\_state = \alpha \cdot queue\_depth + \beta \cdot cpu\_occupancy \quad (2)$$

The values α, β, $\in$ [0, 1] are tunable parameters useful for providing a weight value for a specific metric in the calculation of the path state and can be used to add more weight to the utilization of *network* resources or utilization of *server* resources.

The switch maintains the last information collected for each path. To determine the current state of the path, we employ an Exponentially Weighted Moving Average (EWMA), considering a smoothing parameter λ. The current utilization of the path is a weight combination of the previous value, stored in the switch, and the new value obtained by the system monitor. Using the calculated path state, the `WeightTable` is queried to obtain the new path weight value. This table contains values inversely proportional to path usage, i.e., less used paths have a greater weight than more used paths.

To handle hardware constraints in the data plane, which make it difficult to update path weights directly on the switch, our solution is based on the Data-plane Adaptive Splitting with Hash threshold (DASH) data structure proposed by [Hsu et al. 2020b]. The main idea behind DASH is leveraging a multi-stage pipeline and stateful ALUs to update the path weights at line rate. Furthermore, a small number of region boundaries need to be updated instead of many table entries in the traditional WCMP-based algorithms. We will give an example below of how the path weights are updated in the data plane.

Example. Figure 5 presents an example of how P4eBalancer performs path selection and updates the path weight. In this example, there are three paths, A, B, and C and the current distribution is 3:3:3, which means that all paths have the same hashing space of the total hashing range (Figure 5.a). When a packet arrives, the switch calculates the hash value of its headers. The path that will be associated with the packet is obtained by comparing the hash value obtained with the limits of each path. Initially, the hash value is compared with the region A boundary. If the calculated value is less than the boundary of path A (B_0=3), the corresponding packet is forwarded along this path. Otherwise, the hash value is compared against the other boundaries, in an ascending order, and will be forwarded accordingly. In our example, the hash value is 4, so the packet is forwarded through path *B*.

The weight of a path is dynamically adjusted based on the INT telemetry carried by packets (Figure 5.b). The INT headers are extracted, and together with the previously collected values stored in the switch, they form the current state of the path. The boundaries are updated sequentially by adding the cumulative sum of the previous path boundaries with the current path weight size. In our example, the switch should change the weight of path *B* to 4, and the new distribution of paths A, B, and C will be 3:4:3. Path

A boundary remains unchanged (B_0=3). Path B boundary is equal to the sum of path A boundary and its weight size, which now is 4. So the switch performs B_1=B_0+4. Finally, path C boundary also needs to be updated and will be equal to the sum of the path B boundary and its weight size 3 (B_2=B_1+3).

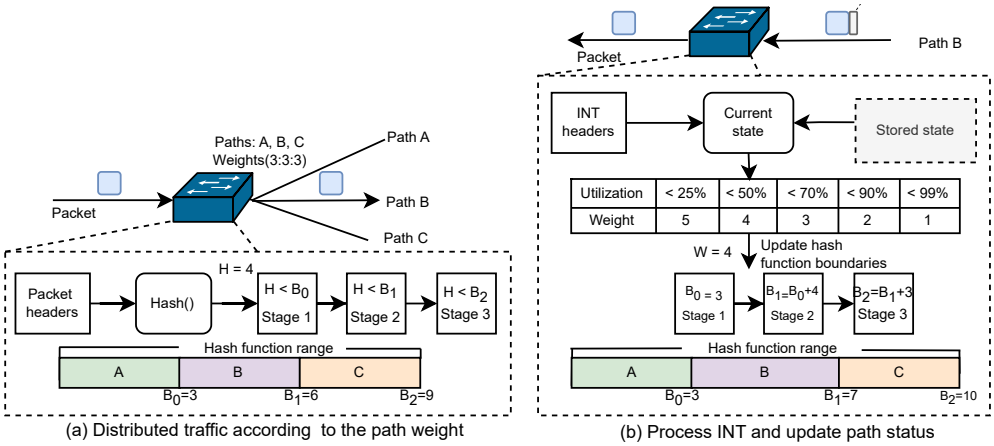


Figure 5. Path selection and update path weight example

Send Telemetry Data. The edge switch represents the final telemetry phase of outgoing packets within the network. Response packets received in this switch go through a clone operation (`CloneI2E`) that replicates the packet from the ingress to the egress pipeline. Cloning the packet has two goals: (1) One copy of the packet is forwarded to the client, and (2) the other is used to proactively send the collected information to the INT Engine. To ensure transparency to the client application, the entire INT header stack is removed before delivering the response to the client. In contrast, the cloned packet has its payload removed, carrying only telemetry data that is sent to the INT Engine.

4. Implementation and Evaluation

4.1. Experimental Setup

Prototype. We implemented a prototype of P4eBalancer[1] including the data plane, the control plane, and the software of the RL agent. The switch data plane code is written in P4-16. The controller is written in Python and uses Scapy to send and receive packets between the controller and the software switches. Finally, the reinforcement learning agent is implemented using the Deep Q-Learning (DQN) algorithm from stable_baselines3 [Raffin et al. 2021] in a custom environment created with Gym.

Evaluation Setup. We conducted experiments in an emulated environment consisting of BMv2 switches in Mininet running on a Linux VM with Ubuntu 20.04 and kernel version 5.4. The virtual machine was configured with 4 cores at @ 2.7GHz and 8GB RAM. We use a fat-tree topology, shown in Figure 6, with 10 switches. Among these, four are edge switches and have a host directly connected to them running a client/server application. The link speeds are limited to 5 Mb/s due to hardware limitations in the setup used in our experiments. At each step, the RL agent selects 4 out of 12 pre-computed

[1] Available at: https://github.com/cleitonputtlitz/P4eBalance

paths for each source/destination. The selected paths are configured on the switch with a weight of 5 (maximum possible value). The switches collect telemetry data and update the path weights every 100 ms.

Workloads. We select hosts *h1* and *h2* to act as clients and hosts *h3* and *h4* to act as servers. Servers can handle multiple connections simultaneously, while clients send 200 requests per second. The load balancer mechanism selects a path that includes the network hops and a server to handle the client request. We consider scenarios with different values for α and β, used in Equations 1 and 2 to calculate the reward of the RL agent and the current path state in the switch. Table 1 presents the values used in the different scenarios, where α is used to determine the weight for network metrics and β for server metrics.

Table 1. Parameters

Scenario	Network (α)	Host (β)
1	1	0
2	0.5	0.5
3	0.8	0.2
4	0	1

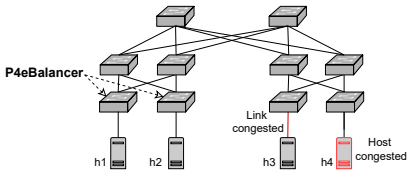


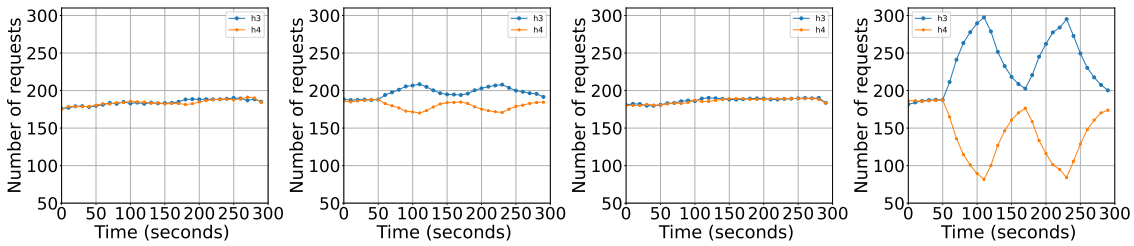
Figure 6. Topology

4.2. Evaluation Results

The experiments aim to observe whether the developed mechanism can identify and avoid paths that have very congested hosts or links. In addition, we seek to analyze the influence of the parameters in Table 1 on the decisions taken to select the best paths.

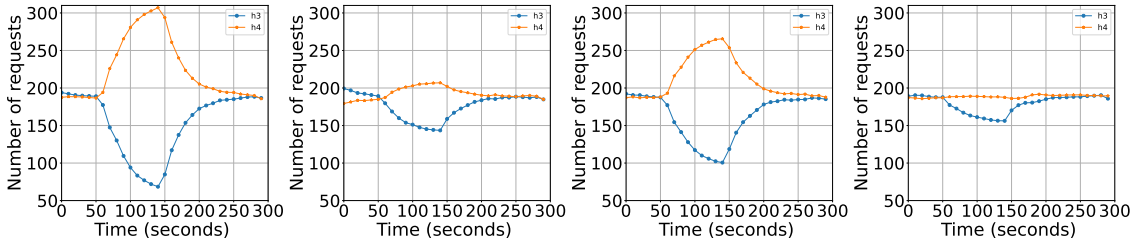
Host congested. To simulate the load on servers, we developed a script that runs periodically on host *h4* and consumes a high CPU rate. The script runs for 30 seconds at a one-minute interval. In scenarios 1 (Figure 7.a) and 3 (Figure 7.c), where the weight of network metrics is greater than the weight of host metrics, it can be observed that the distribution of requests occurs equally between the two servers, even in periods when *h4* has an overloaded CPU. In scenario 2 (Figure 7.b), where the network and host metrics have the same influence on the composition of the state of the paths, in periods when server *h4* presents an increase in CPU consumption, there is a decrease in the number of requests forwarded to it. Conversely, we observe an increase in the number of requests sent to server *h3*, which has low CPU usage. This operation is more evident in scenario 4 (Figure 7.d), where the system only considers host metrics: when identifying an increase in CPU consumption on server *h4*, P4eBalancer prioritizes paths that send requests to *h3*, demonstrating the correct functioning of the mechanism.

Link congested. To simulate link congestion, we generate a high volume of traffic to host *h3* during the time period 60 to 150. In scenarios 1 (Figure 8.a) and 3 (Figure 8.c), where the weight of network metrics is greater than the weight of host metrics, it can be observed that a larger proportion of requests is sent to *h4*, conversely reducing the number of requests sent to *h3* due to the congestion of the link. In scenario 2 (Figure 8.b), where network and host metrics have the same influence on the composition of the path state, there is a small increase in the number of requests sent to *h4* compared to *h3* during



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3 (d) Scenario 4

Figure 7. Host congested in different scenarios



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3 (d) Scenario 4

Figure 8. Link congested in different scenarios

the period when congestion occurs. However, *h3* continues to receive a large number of requests compared to scenario 1. In scenario 4 (Figure 8.d), the system only considers host metrics, disregarding the congestion of the link to *h3*. In this way, requests are sent in a balanced manner to both hosts. The difference in the number of requests between *h3* and *h4* is explained by the fact that drops occur in the switch due to the queue being full on the path to *h3*. When the congested link condition ends, we observe in all scenarios that requests are forwarded in a balanced manner between hosts.

5. Related Work

In this section, we discuss different strategies used for load balancing, summarized in Table 2, including those that act directly on the data plane, control plane, hybrid approaches that combine both, and those that use reinforcement learning algorithms.

Approaches such as [Katta et al. 2016, Hsu et al. 2020a] act directly on the data plane, updating its state and directing flows based on traffic conditions. However, they only consider one best path at a given time, which can lead to suboptimal network utilization and easily congest the selected path. Other approaches split traffic over multiple paths [Zhou et al. 2014, Hsu et al. 2020b, Robin and Khan 2022, Benet et al. 2018, Ye et al. 2018]. Each path has an associated weight to distribute the incoming load over these multiple paths, and the decision about which path to use considers this distribution. Paths with higher weights are more likely to be selected. To respond to the dynamically changing network, these approaches rely on periodic probes to collect path status. The path weight distribution is then updated either directly in the data plane [Hsu et al. 2020b, Ye et al. 2018] or with the intervention of the control plane [Zhou et al. 2014, Robin and Khan 2022].

Other schemes delegate load balancing decisions to a centralized controller [Al-Fares et al. 2010, Katta et al. 2017]. Although these solutions pro-

vide a comprehensive view of the network infrastructure, they are slow to react to dynamic traffic conditions. Hybrid solutions combine centralized and distributed planes [Hyun et al. 2018, Pizzutti and Schaeffer-Filho 2019, Coelho and Schaeffer-Filho 2023]. In this approach, the centralized plane provides a global view of the network infrastructure and makes global changes, such as selecting the best routes. In turn, rapid decisions occur at the switch, identifying congested routes and switching between paths generated by the controller.

Reinforcement learning has been applied as a promising solution for load balancing at the control plane, making global decisions about the network, and improving current strategies limited to heuristic-based. Telemetry is an important part of these approaches, providing detailed information about the network. Analyzing the collected telemetry data enables informed decision-making, which can then be applied to the network. QCMP [Zheng et al. 2023] leverages INT-collected information across the network and applies a Q-learning algorithm to adjust the weight of each path. In [Hyun et al. 2018], a self-driving network architecture is proposed, capable of operating and managing the network without operator intervention. Other approaches rely on an RL agent to optimize the selection of routing algorithms [Coelho and Schaeffer-Filho 2023, Casas-Velasco et al. 2022] that can be used by the switches to send traffic.

Our work shares several features with previous research, but is innovative in its use of a telemetry subsystem that provides end-to-end metrics for more informed load-balancing decisions. By employing a reinforcement learning agent, we can select the best paths by prioritizing network metrics, server metrics, or a combination of both. Furthermore, the fast decision loop in the data plane enables quick reactions to transient congestion on installed paths.

Table 2. Related work

Decision plane	Examples	Comment
Data Plane	HULA [Katta et al. 2016], MP-HULA [Benet et al. 2018], CONTRA [Hsu et al. 2020a]	Quickly reacts to network changes. However, requires the propagation of information.
Control plane	HEDERA [Al-Fares et al. 2010], WCMP [Zhou et al. 2014], CLOVE [Katta et al. 2017]	The control plane calculates and configures the available paths. However, the response time to react to changes in the network is high.
Hybrid	[Pizzutti and Schaeffer-Filho 2019] CLB [Robin and Khan 2022]	Cooperation between the control and data plane. The centralized plane makes global changes while rapid decisions occur at the switch.
RL Agent	DRSIR [Casas-Velasco et al. 2022], QCMP [Zheng et al. 2023], CROSSBAL [Coelho and Schaeffer-Filho 2023]	The RL agent analyzes the collected telemetry data to select best paths or adjust the weight of the paths.

6. Concluding Remarks

This work proposed a new intelligent and decentralized load balancing scheme, called P4eBalancer. With active In-band network telemetry, P4eBalancer collects metrics about all distributed infrastructure (network and servers). A reinforcement learning agent uses this information to determine the best paths to be installed on the switches. P4eBalancer switches then distribute traffic proportionally on these paths, based on weights. Additionally, by storing a small amount of the collected INT information, P4eBalancer updates the weights of each path directly at the switch, allowing for quick reaction to changing network conditions. By assigning different weights to network and host metrics, our results

show that better load balancing decisions can be made using end-to-end metrics, avoiding congested links or servers. In future work, we intend to explore alternatives that can make the agent robust to topology changes, without the need for retraining. Additionally, we plan to incorporate a larger set of metrics to improve agent decision-making.

Acknowledgments

This work was financed in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and grant #88887.954253/2024-00, CNPq grants #311276/2021-0 and #405940/2022-0, and FAPESP grants #2020/05152-7, #2023/00673-7 and #2023/00764-2.

References

Abranches, M., Michel, O., Keller, E., and Schmid, S. (2021). Efficient network monitoring applications in the kernel with ebpf and xdp. In *IEEE NFV-SDN*.

Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. (2010). Hedera: dynamic flow scheduling for data center networks. In *USENIX NSDI*.

Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Lam, V. T., Matus, F., Pan, R., Yadav, N., and Varghese, G. (2014). Conga: distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM*.

Barbette, T., Tang, C., Yao, H., Kostić, D., Jr., G. Q. M., Papadimitratos, P., and Chiesa, M. (2020). A High-Speed Load-Balancer design with guaranteed Per-Connection-Consistency. In *USENIX NSDI*.

Benet, C. H., Kassler, A. J., Benson, T., and Pongracz, G. (2018). Mp-hula: Multipath transport aware load balancing using programmable data planes. In *ACM NetCompute*.

Benson, T., Akella, A., and Maltz, D. A. (2010). Network traffic characteristics of data centers in the wild. In *ACM SIGCOMM*.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*.

Casas-Velasco, D. M., Rendon, O. M. C., and da Fonseca, N. L. S. (2022). Drsir: A deep reinforcement learning approach for routing in software-defined networking. *IEEE TNSM*.

Coelho, B. L. and Schaeffer-Filho, A. E. (2023). Crossbal: Data and control plane cooperation for efficient and scalable network load balancing. In *CNSM*.

eBPF (2024). Dynamically program the kernel for efficient networking, observability, tracing, and security. https://www.ebpf.io/. Accessed in: 09.03.2023.

Eisenbud, D. E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingiroglu, A., Cheyney, B., Shang, W., and Hosein, J. D. (2016). Maglev: A fast and reliable software network load balancer. In *USENIX NSDI*.

Hsu, K.-F., Beckett, R., Chen, A., Rexford, J., and Walker, D. (2020a). Contra: A programmable system for performance-aware routing. In *USENIX NSDI*.

Hsu, K.-F., Tammana, P., Beckett, R., Chen, A., Rexford, J., and Walker, D. (2020b). Adaptive weighted traffic splitting in programmable data planes. In *SOSR*.

Hyun, J., Van Tu, N., and Hong, J. W.-K. (2018). Towards knowledge-defined networking using in-band network telemetry. In *IEEE/IFIP NOMS*.

Katta, N., Ghag, A., Hira, M., Keslassy, I., Bergman, A., Kim, C., and Rexford, J. (2017). Clove: Congestion-aware load balancing at the virtual edge. In *ACM CONEXT*.

Katta, N., Hira, M., Kim, C., Sivaraman, A., and Rexford, J. (2016). Hula: Scalable load balancing using programmable data planes. In *ACM SOSR*.

Kim, C., Sivaraman, A., Katta, N., Bas, A., Dixit, A., and Wobker, L. J. (2015). In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*.

Pizzutti, M. and Schaeffer-Filho, A. E. (2019). Adaptive multipath routing based on hybrid data and control plane operation. In *IEEE INFOCOM*.

Puttlitz, C., Parizotto, R., and Schaeffer-Filho, A. (2024). P4netintel: End-to-end network telemetry with ebpf and xdp. In *IEEE NFV-SDN*.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *JMLR*.

Robin, D. D. and Khan, J. I. (2022). Clb: Coarse-grained precision traffic-aware weighted cost multipath load balancing on pisa. *IEEE TNSM*.

Sutton, R. and Barto, A. (1998). Reinforcement learning: An introduction. *IEEE TNN*.

Tajbakhsh, H., Parizotto, R., Neves, M., Schaeffer-Filho, A., and Haque, I. (2022). Accelerator-aware in-network load balancing for improved application performance. In *IFIP Networking*.

Tajbakhsh, H., Parizotto, R., Schaeffer-Filho, A., and Haque, I. (2024). P4hauler: An accelerator-aware in-network load balancer for applications performance boosting. *IEEE TCC*.

Tarkoma, S., Rothenberg, C. E., and Lagerspetz, E. (2012). Theory and practice of bloom filters for distributed systems. *IEEE COMST*.

Ye, J.-L., Chen, C., and Huang Chu, Y. (2018). A weighted ecmp load balancing scheme for data centers using p4 switches. In *IEEE CloudNet*.

Yen, J. Y. (1971). Finding the k shortest loopless paths in a network. *Management Science*.

Zhang, J., Yu, F. R., Wang, S., Huang, T., Liu, Z., and Liu, Y. (2018). Load balancing in data center networks: A survey. *IEEE COMST*.

Zheng, C., Rienecker, B., and Zilberman, N. (2023). Qcmp: Load balancing via in-network reinforcement learning. In *ACM FIRA*.

Zhou, J., Tewari, M., Zhu, M., Kabbani, A., Poutievski, L., Singh, A., and Vahdat, A. (2014). Wcmp: weighted cost multipathing for improved fairness in data centers. In *ACM EuroSys*.