QCMP: Load Balancing via In-Network Reinforcement Learning

Changgang Zheng*
University of Oxford
changgang.zheng@eng.ox.ac.uk

Benjamin Rienecker
University of Oxford
benjamin.rienecker@bnc.ox.ac.uk

Noa Zilberman
University of Oxford
noa.zilberman@eng.ox.ac.uk

ABSTRACT

Traffic load balancing is a long time networking challenge. The dynamism of traffic and the increasing number of different workloads that flow through the network exacerbate the problem. This work presents QCMP, a Reinforcement-Learning based load balancing solution. QCMP is implemented within the data plane, providing dynamic policy adjustment with quick response to changes in traffic. QCMP is implemented using P4 on a switch-ASIC and using BMv2 in a simulation environment. Our results show that QCMP requires negligible resources, runs at line rate, and adapts quickly to changes in traffic patterns.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **In-network processing**; • **Computing methodologies** → **Reinforcement learning**.

KEYWORDS

Load Balancing, In-Network Computing; Distributed; Reinforcement Learning; Machine Learning; Programmable Switches; P4.

1 INTRODUCTION

Network traffic is dynamic by nature. The increasing use of cloud computing and the introduction of more and more networked services mean that traffic patterns are less predictable and more imbalanced than in the past [15]. Furthermore, network topologies are also complex and evolve over time, as a network grows, nodes fail and connectivity changes.

The dynamism of the network is of a special challenge when trying to satisfy capacity demands and to maintain service objectives. In networks where multiple paths exist between source and destination, *Load Balancing* aims to assign flows to different paths in a manner that optimizes network utilization [15], and consequently user experience.

The classic load balancing algorithm, Equal-Cost Multi-Path (ECMP) [13], evenly distributes traffic to all available paths using a hash function. While ECMP is simple and feasible within switches, it may suffer imbalance due to hash collisions and varying data rates [15], or global traffic imbalancing in networks with asymmetric topologies [3].

Network telemetry, and in particular in-band telemetry, and the rise of software defined networks, enabled new load balancing solutions such as CONGA [1] and HULA [5]. These solutions gained real-time insights into network utilization, using a variety of in-network techniques. However, they still use fixed load balancing policies. In addition, each solution has its limitations. For example, CONGA achieves global congestion awareness through end-to-end path feedback, requiring storing large amounts of path information and limiting large-scale deployment. HULA lacks global cooperation of switches, which may lead to congestion when multiple switches concurrently identify the same best path [6].

Unlike fixed-strategy policies, reinforcement learning (RL) has the ability to dynamically adapt to unknown environments. It improves policies by interacting with the environment to find the optimal policy. When an environment changes, reinforcement learning can gradually discover new optimal strategies, adjusted based on trial and error and environmental feedback. Its applicability to complex problems, interactive learning, and autonomous decision-making capabilities make it a promising solution for load balancing challenges. Previous works have applied reinforcement learning to host-based load balancing [8] and controller-based load balancing [18], but the exploration of reinforcement learning for switch-based load balancing remains limited [15].

In this work, we introduce QCMP, a reinforcement-learning based distributed load balancing solution for network switches. QCMP applies Q-learning to adjust the weight of each path, balancing the load across the network. QCMP uses In-Network Telemetry (INT) to collect congestion and utilization information across the network, and uses Q-learning to make decisions. QCMP's distributed switch-based load balancing ensures quick reactions and high scalability, overcoming the limitations of centralized controllers. The RL-based decision ensures continuous policy update by QCMP and adjusting to diverse environments. In summary, the main contributions of this work are as follows:

(1) We introduce two novel in-network Q-learning paradigms, one implemented entirely within the data plane, and the other using local switch CPU for the Q-table updates.
(2) We propose QCMP, an RL-based load balancing solution. QCMP is switch-based, distributed and model-free. It uses INT and Q-learning to continuously interact with the environment and update its policy.
(3) We implement QCMP in P4 and evaluate it on a swich-ASIC and in simulation. Our evaluation shows that QCMP is feasible on commodity hardware, and provides efficient load balance in dynamic scenarios.

2 BACKGROUND

Programmable Data Planes. Programmable network devices allow network operators to customize the forwarding behavior of network devices, providing high flexibility and adaptability. Using the P4 language, programmers can specify the functionality of the data plane, based on the architecture of the target device (e.g., PISA). Information can be extracted from packet headers, and used as keys for lookups in match-action (M/A) tables. While table entries cannot be changed directly from the data plane, register arrays maintain state across packets and allow operations such as read-modify-write. Updating table values is done via the control plane, often using the local CPU of a switch box [7].

*Changgang and Benjamin contributed equally to this work.

Load balancing. Load balancing distributes traffic across multiple links, to optimize system performance, maximize link utilization, and enhance overall reliability. In data centers, load balancing can be categorized into three main types: centralized controller-based load balancing, host-based load balancing, and switch-based load balancing [10]. Centralized load balancing can universally do the balance task but incurs relatively high network resource consumption, deployment overhead, and response latency [2, 18]. While host-based load balancing offloads balancing tasks from a centralized controller to the hosts, thereby reducing deployment overhead, it imposes additional challenges in modifying host protocol stacks, making deployment difficult. Switch-based load balancing is in comparison more responsive and easier to deploy than the previous two approaches [1, 5], with recent progress based on the advancement of programmable network devices [7]. The granularity of switch-based load balancing implementation can be based on packet, flow, or flowlet. In this work, the proposed QCMP algorithm is a flow-level switch-based load balancing solution.

Reinforcement Learning. RL optimizes strategies by interacting with the environment. The objective of RL algorithms is to obtain the maximum cumulative reward from a dynamic environment [4]. RL has the ability to adapt to environmental changes by maintaining an optimal strategy through interactions and trial errors. RL algorithms can be classified into model-based and model-free approaches. Model-free reinforcement learning refers to cases where the environment cannot be fully modelled and predicted, such as load balancing with complex and dynamic unpredictable traffic. Model-free reinforcement learning can further be categorized into value-based and policy-based methods. In value-based reinforcement learning, a value function is utilized to assist policy updates, while policy-based methods directly uses iterative updates to optimize the policy. Value-based RL methods can be further classified into offline or online learning. Online learning requires the policy used for making decisions to be consistent with the one that is improved (e.g. SARSA [9]), while offline learning allows decision-making based on non-latest learned policies (e.g. Q-learning [14]).

In-network Machine Learning. In-network machine learning (ML) offloads ML algorithms into the data plane (e.g., programmable routers, switches, and NICs) [16]. It lowers latency, achieves high throughput and reduces the load on servers and accelerator cards. Online processing of traffic using in-network ML enables real-time decision-making, and pushing functionality closer to the edge. Most in-network ML solutions focused on ensemble tree models, classic models and neural networks (e.g., [11, 17]). To date, the only attempt to implement in-network RL was on a smartNIC, heavily relying on the NIC's externs (hardware-specific additional functions) and had limited portability [10].

2.1 Motivation and Challenges

Load balancing is a complex problem. The network is dynamically changing, flows come and leave, and congestion at one end of the network may affect traffic at the other end of it. Reinforcement learning is ideal to solving such problems: it is dynamic, self-learns from changing situations and can detect trends that human planning and fixed policy may not.

Still, implementing RL within the data plane is a challenge. Unlike other forms of in-network ML, such as classification or clustering algorithms, it requires continuous computations and updates of values, contrary to the common data and control plane separation. The following Section 3 reports how we overcome the challenges and implement in-network RL, while the subsequent Section 4 introduces the application of in-network RL to load balancing.

3 IN-NETWORK REINFORCEMENT LEARNING

This section introduces our in-network RL solution. The algorithm needs to fit the primitives of the data plane's PISA architecture, e.g., M/A pipeline. It also need to tolerate hardware constraints on resources, operations and stages. Based on different aspects of RL properties, the requirements are (i) a model-free RL algorithm, where complex environmental modeling is not necessary. (ii) off-policy algorithm, meaning that policy updates are independent of the agent's actions. The availability of the "*action first update later*" paradigms fits the operation logic of programmable data planes (M/A table entries can only be updated by the control plane, not in-band).

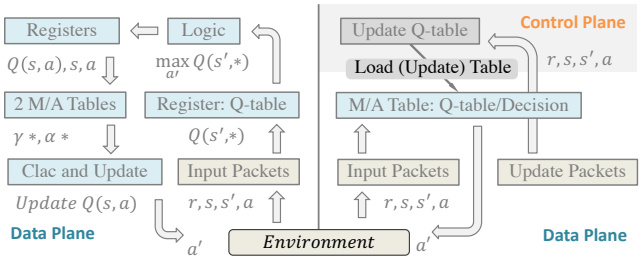
Q-learning [14] is a popular RL algorithm. Unlike other RL algorithms that are deep neural-network and hard to implement in a data plane [17], Q-Learning is suitable for mapping. As shown in Algorithm 1, the algorithm tries to solve the Markov decision process by learning an optimal policy (Q-table) through value iteration. It achieves this by iteratively updating the Q-value of state-action pairs (s-a and s'-a' are the current and next state-action pairs). To this end, it uses observed rewards (r) and the maximum expected future rewards (Algorithm 1 line 6, where α is the learning rate and γ is a discount factor). The algorithm explores the environment using an ϵ-greedy algorithm to balance exploration and exploitation, gradually converging towards an optimal policy that maximizes the expected cumulative reward over time. During the exploration process, each time Q-table update means a step and each time environment restart means an episode. The look-up mechanism used by Q-learning's Q-table fits the M/A table in the data plane well, and the limited action space inside the Q-table reduces memory requirements.

The main challenge to implementing in-network Q-learning is updating the Q-table. This includes where to store the Q-table, how to calculate Q-value, and how to maintain history action, state, and

Algorithm 1: Q-learning

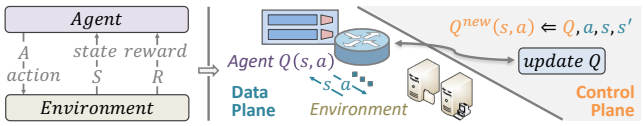
Initialize: $Q(s,a)$ arbitrarily

1 **Repeat** // for each episode
2 Initialize s;
3 **Repeat** // for each step of episode
4 $a \leftarrow Q(,)$ and s using policy e.g., ϵ-greedy;
5 Take action $a \rightarrow$ observe r and s';
6 $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$;
7 $s \leftarrow s'$;
8 **while** *step is not terminal*;
9 **while** *episode is not terminal*;



(a) Q-learning fully on the Data Plane (b) Q-learning partially on Data Plane

Figure 1: In-network Q-learning solutions. (a) Register-based Q-learning and (b) M/A table-based Q-learning.



(a) General RL (b) In-network RL

Figure 2: Comparison between traditional Q-learning and M/A-based in-network Q-learning in Figure 1 (b).

reward information. To address these challenges, we introduce two solutions: one purely in the data plane, and one combining data and control planes.

The first solution implements Q-learning entirely in the data plane using registers, as shown in Figure 1 (a). In this solution, the Q-table and some parameters (e.g., previous state, action, reward) are stored in register arrays. The workflow of this register-based Q-learning is as follows:

(1) When a new packet arrives (Input Packets in Figure 1 (a)), it is associated with environment's information including new state (s') and reward (r) (past state s and action a are optional), reflected in line 5 of Algorithm 1.
(2) The rewards $Q(s', *)$ are read from the Q-table stored in registers (shown in Figure 1 (a) step Register: Q-table).
(3) The new action (a') is calculated using logic based on an ϵ-greedy algorithm.
(4) The previous Q-value ($Q(s,a)$), state (s), and action (a) are read from registers and the registers are updated with the new values ($Q(s',a')$, s', a') (line 7 in Algorithm 1). The registers in this step can be changed to packet headers if the input packet contains these values.
(5) The Q-value update is done in the step *Calc and Update* (line 6 in Algorithm 1). For the update calculation, the multiplication result of the learning rate and discount factor is pre-calculated and stored in two M/A tables, removing the need for multiplication which is unsupported within the data plane.
(6) After the new Q-value is computed and the Q-table is updated, the packet (or another object or function) will return the new action (a') to the environment (line 4 in Algorithm 1).

This procedure will iterate, with the agent taking the new action and getting the reward from the environment.

The second solution moves complex operations from the data plane to the control plane, as shown in Figure 1 (b) and Figure 2. The workflow of this M/A table-based Q-learning is as follows:

(1) Like the first solution, an input packet is associated with the previous state & action (s & a) and the new (current) state & reward (s' & r).
(2) The reward of all actions is read from the Q-table, implemented as a M/A table, and the new action (a') is selected based on an ϵ-greedy algorithm.
(3) The new action (a') is sent back to the environment to guide the agent's behavior.
(4) Different from the registers used in the first solution, M/A tables can not be updated in-band. Instead, the control plane (e.g., the switch CPU) needs to update the entries. The current reward (r) and state (s') are brought to the control plane using update packets. The update packets can be generated from the switch CPU, sent by the environment, or generated (mirrored) by the switch.
(5) As the update process happens in the control plane, there is no need to calculate the new Q-value in the data plane. The new Q-value can be calculated in the control plane before the table update.
(6) After the Q-table has been changed, the control plane updates the changed table entry in the data plane.

The differences between these two approaches are further discussed in §7.

4 LOAD BALANCING SYSTEM DESIGN

There are two common classes of load balancing solutions: distributed and centralized. In centralized solutions (e.g., AuTO [2]), all switches are treated as a single agent. However, this approach results in an excessively large state and action space, unsuitable for Q-learning. Additionally, handling INT is more complex in a centralized solution, leading to increased response times. Therefore, QCMP opts for a distributed load balancing. Each programmable switch runs a Q-learning algorithm responsible for controlling the agent, i.e., the switch. This section provides an overview of the QCMP design workflow and operation.

4.1 QCMP Workflow

The implementation of QCMP follows the structure outlined in Figure 1, implementing in-network RL with the Q-table in the data plane. Most data centers use hardware switches for high throughput and low latency traffic forwarding. On hardware targets, the register-based solution is harder to implement and has limited scalability. Therefore, we focus on M/A-based Q-learning (Figure 1 (b)) as a leading example for QCMP.

A high-level view of QCMP's implementation is shown in Figure 3. The operation distinguishes between two types of packets: normal traffic and INT messages[1]. Normal network traffic, on the left of Figure 3, is forwarded based on path weights. Parsed packets are forwarded to ports based on routing tables (routing-related) and weights (load balancing-related). INT packets, on the right of Figure 3, contain queue lengths information, used to update path

[1]Use of dedicated INT packets is for illustration purposes

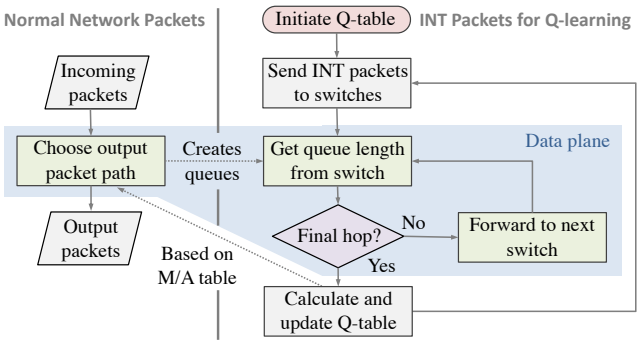


Figure 3: QCMP system workflow.

weights. This information is sent to the control plane for Q-value updates.

The Q-table and the match-action table which stores path weights, are initially set to equal weight. When INT packets pass through a switch, the switch updates queue length in the INT header. For coverage, we assume that source routing is used, as in [12]. INT packets are mirrored to the CPU and the controller calculates a new Q-value and updates the Q-table in the control plane. Based on the queue lengths from INT packets, path weights are updated in the M/A table.

4.2 Q-learning in QCMP

An effective implementation of load balancing using Q-learning depends on the several essential components: agents, states, actions and rewards.

4.2.1 States. The state space of QCMP is made from the queue lengths for each path from a switch. Each queue has a length in the range $[0, N_{queue}]$, but through state aggregation, the queue is mapped to integers in a smaller range $[0, N_q]$. The mapping means dividing the queue length by N_q and then rounding up to the nearest integer. This crucially allows a zero queue to be its own state, only possible if the incoming traffic rate is less than the outgoing traffic rate. The queue lengths updated frequently, so having a more detailed state space is not required for avoiding state space explosion, as the queue length fluctuations are small. Less frequent updates (less INT communication overhead) mean that the state space will grow. Similarly, the state space can be reduced by considering a subset of paths or a subset of hops from the switch. Two possible paths from a switch result in a state space of N_q^2 states, which can be controlled to an adequately small number without using state space compression methods.

4.2.2 Actions. The QCMP action controls the weight of each output port. The action space covers all possible permutations of increasing, decreasing and keeping constant the weights of each path. The sum of the path weights remains constant, ensuring a finite action space. In QCMP, we further constrain the actions to a form shown in Table 1, focusing on changing one port and consequently changing other ports to compensate for the change. As shown in Table 4, in any N_P paths, there are at most $2N_p + 1$ actions. In order to accelerate the convergence process of path weights, the constant k that controls the weight of each path's changes can be dynamic. This chosen value can depend on the episode number or other parameters that reflect the change of environment.

Action	Path 1 Weight	Path 1 Weight	...	Path n Weight
$\uparrow\downarrow \ldots \downarrow$	$+k(N_p-1)$	$-k$	...	$-k$
$\downarrow\uparrow \ldots \uparrow$	$-k(N_p-1)$	$+k$	...	$+k$
...	...	...	...	...
$\uparrow \ldots \uparrow\downarrow$	$+k$	$+k$	...	$-k(N_p-1)$
$- \ldots -$	+0	+0	...	+0

Table 1: QCMP actions.

4.2.3 Rewards. Rewards are based on the change in weighted average queue length for different paths from a switch and the difference in queue lengths. The weighted average queue length needs to be minimized, but so does the difference in queue lengths, preventing getting stuck in non-optimal weights. Without having a term to minimize the difference, a policy would get stuck in the minima where one path has a full queue but is assigned a zero weight, whilst the other paths have empty queues and share all weights. Combining the difference and the weighted average terms results in Equation 1, where r evaluates current load balancing performance, c: is a constant, L: is the queue length, w: is the path weight, $*$: indicates values of the previous step, and $i, j \in \{1, \ldots, N_p\}$.

$$\overbrace{(c - \max|L_i - L_j|)}^{\text{Difference term}} + \overbrace{(\sum_i L_i^* w_i^* - \sum_i L_i w_i)}^{\text{Weighted average term}} = r \tag{1}$$

To reduce (over) sensitivity of the system, Equation 2 is used to calculate the actual reward.

$$\text{Reward} = \begin{cases} +1 & \text{if } r > 0.5 \\ -1 & \text{if } r < -0.5 \\ \pm 0 & \text{if } -0.5 < r < 0.5 \end{cases} \tag{2}$$

5 IMPLEMENTATION

Both in-network Q-learning strategies are implemented in BMv2 software switch using P4 with v1model architecture. The M/A table-based Q-learning approach is also implemented on an Intel Tofino switch-ASIC (APS-Networks BF6064X, SDE 9.5.0) using TNA architecture. Emulation of QCMP uses Mininet. The switch control plane is written in Python and supports P4Runtime.

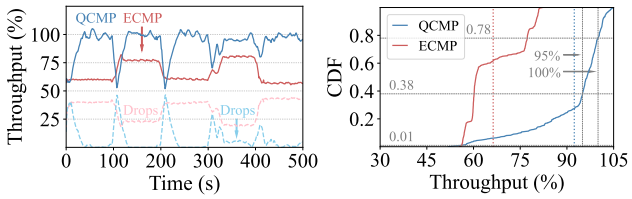
6 EVALUATION

QCMP hardware performance test is run on Intel Tofino. System level operation is evaluated using BMv2, and performance is compared with ECMP.

6.1 Simulation

Network Setup (Topology). QCMP is evaluated on a 3-tier Fat-Tree topology with two spine switches connected to a pod. Within a pod, two aggregation switches are connected to two Top of Rack (ToR) switches.

Experimental Setup. In the evaluation, the queue length rate on the simulator ranges from $[0, N_{queue} = 100]$ and we set $N_q = 10$. In



(a) Ratio of Throughput (b) CDF of Throughput

Figure 4: System performance comparision between QCMP and ECMP.

the evaluated network topology, each switch is connected to two other switches, which means $N_p = 2$. The path changes weight k is initialized to 5 and the constant x in the reward equation is set as 50. The model starts with a high learning rate and exploration rate for rapid explorative learning but reduces to lower values for a more exploitative approach that converges accurately to the optimal policy. The exploration rate is limited to a minimum of 0.1 to overcome non-optimal states, a discount factor of 0.25 to lay more emphasis on immediate reward and less on long-term reward.

Figure 4 shows the performance of QCMP compared to ECMP over a 500-second episode, with the output port queue rates changing every 100 seconds. A 10-second moving average is used for both sets of results in order to limit the effect of noise due to the random variations of the hash function. ECMP shows consistent performance over the episodes with an average throughput of about 60% or 75%, depending on queue rate configuration. This is the result of packets being evenly sent on paths with queue rates ratios of 3:1, leading to per-port throughput of 66% and 100% of port rate, respectively. Consequently, the larger the difference between the queue rates of the paths, the worse ECMP would perform. A small temporary increase in throughput can be observed when the queue rates change, as full queues rapidly send packets.

As shown in Figure 4 (a), QCMP's throughput starts at the same level as ECMP, with around 60%, but it learns the optimal path weights to achieve a throughput of 100% within 30 seconds. This occurs when one output port handles 75% of traffic and the other 25%. At 100 seconds, when the queue rates are switched, the throughput suddenly drops to around 50%. This is because the output port receiving 75% will now experience only a third of the previous packet rate, whilst the other output port experiences triple the previous packet rate. For this reason, every hundred seconds when parameters are reset, QCMP's performance drops and is slightly worse than ECMP for a short time. Note that this happens only due to the significant artificial change in port rates in our experiment. At 400 seconds, the optimal queue weights are re-found within 14 seconds. This is shorter than the first 3 times of learning because the Q-table already contains a trained policy[2]. These show that QCMP adapts to network conditions changes.

Figure 4 (a) also shows the number of packets dropped using ECMP and QCMP. The number of packets dropped and the number of packets arriving at the destination approximately matches the input packet rate, with slight difference due to the packet delays in queues. The graph clearly shows QCMP drops far fewer packets than ECMP. While there is a brief period where queue rates are switched and QCMP experiences an increase in packet drops, once it relearns the optimal path weights, no packets are dropped.

CDFs of QCMP and ECMP throughput are shown in Figure 4 (b). The average throughput of QCMP is 92.6% (including rate change events), whilst ECMP achieves 66%, a significant improvement. Our experiments show that the gap in performance grows as networks become more complex. QCMP achieves over 95% of the input packet rate 62% of the time, and matches 100% input packet rate 22% of the time, outperforming ECMP 99% of the time.

6.2 Hardware Test

QCMP's implementation on Tofino is tested using a snake configuration, utilizing $64 \times 100GE$ ports. Our measurement shows that QCMP achieves full line rate on all 64 ports, with no packet drops. The resources used on the switch are negligible, less than 1% of memory resources and two pipeline stages.

7 DISCUSSION

In-network RL in the data plane. To the best of our knowledge, this is the first work to introduce in-network RL within the data plane, implemented on switch ASIC. OPaL [10], which implemented in-network RL on a smartNIC, relied on externs for the RL functionality, using many minions for tile coding. The two Q-learning solutions presented in this work introduce trade-offs: the register-based solution will react faster to changes (sub-μs vs tens of μs) and can process updates at line rate. The M/A-table approach, on the other hand, can handle a larger state space, requires less resources, and can support more complex RL implementations. For the purpose of load balancing, the table-based approach is preferred over the register-based, as it offers better scalability and the reaction time required is (relatively) slow.

Packets reordering. All packets from the same flow will follow the same path, i.e. the output port will be the same for all packets of the same flow. Updates to output port selection, due to congestion along the path and policy updates, may lead to reordering, but as our results show the effect on performance was small.

Flowlets. While this work focuses on flow-level load balancing, similar approaches can be applied to flowlets. We leave it to future work.

8 CONCLUSION

This paper presented QCMP, a reinforcement-learning based solution to traffic load balancing, implemented within the data plane. We introduced two methodologies to enabling reinforcement learning within the data plane, one using M/A tables and the other registers. QCMP was implemented on an Intel Tofino Switch and in a simulation environment, and the evaluation shows that it can run at line rate, requires negligible resources and adapts quickly to changes in traffic patterns. Future work will explore the use of flowlets and extend the scope of evaluation.

Acknowledgements This work was partly funded by VMware. We acknowledge support from Intel. This paper complies with all applicable ethical standards of the authors' home institution.

[2]The scale of seconds is intentionally set, to avoid fluctuations.

REFERENCES

[1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, et al. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *ACM SIGCOMM*, pages 503–514, 2014.

[2] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization. In *ACM SIGCOMM*, pages 191–205, 2018.

[3] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *ACM SIGCOMM*, pages 350–361, 2011.

[4] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement Learning: A Survey. *JAIR*, 4:237–285, 1996.

[5] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable Load Balancing Using Programmable Data Planes. In *ACM SOSR*, pages 1–12, 2016.

[6] Jingling Liu, Jiawei Huang, Wanchun Jiang, and Jianxin Wang. Survey on Load Balancing Mechanism in Data Center. *Journal of Software*, 32(2):300–326, 2020.

[7] Oliver Michel, Roberto Bifulco, Gabor Retvari, and Stefan Schmid. The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications. *ACM Computing Surveys (CSUR)*, 54(4):1–36, 2021.

[8] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. *ACM SIGCOMM Computer Communication Review*, 41(4):266–277, 2011.

[9] Gavin A Rummery and Mahesan Niranjan. *On-Line Q-Learning Using Connectionist Systems*, volume 37. Citeseer, 1994.

[10] Kyle A Simpson and Dimitrios P Pezaros. Revisiting the Classics: Online RL in the Programmable Dataplane. In *NOMS, IEEE/IFIP Network Operations and Management Symposium*, pages 1–10. IEEE, 2022.

[11] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, et al. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *USENIX NSDI*, pages 513–533, 2022.

[12] Carl A Sunshine. Source Routing in Computer Networks. *ACM SIGCOMM Computer Communication Review*, 7(1):29–33, 1977.

[13] Dave Thaler and C Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. Technical report, 2000.

[14] Christopher JCH Watkins and Peter Dayan. Q-Learning. *Machine learning*, 8:279–292, 1992.

[15] Jiao Zhang, F Richard Yu, Shuo Wang, Tao Huang, Zengyi Liu, and Yunjie Liu. Load balancing in data center networks: A survey. *IEEE Communications Surveys & Tutorials*, 20(3):2324–2352, 2018.

[16] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. IIsy: Practical In-Network Classification, 2022.

[17] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Automating In-Network Machine Learning, 2022.

[18] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, et al. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *ACM EuroSys*, pages 1–14, 2014.