Intelligent Orchestration of High Performance, Scalable, and Correct Data Centres As a Network Service

Zhiyuan Yao

INSTITUT POLYTECHNIQUE DE PARIS

ÉCOLE POLYTECHNIQUE

IP PARIS

NNT : 2023IPPAX019

Thèse de doctorat

Autonomous Service Management in the Cloud

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à l'École polytechnique

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Réseaux, information et communications

Thèse présentée et soutenue à Palaiseau, le 17 mai 2023, par

ZHIYUAN YAO

Composition du Jury :

SARA BOUCHENAK Professeur, INSA Lyon, France	Rapporteur
RÉMI BADONNEL Professeur, Telecom Nancy, France	Rapporteur
GAËL THOMAS Professeur, Telecom SudParis, France	Examinateur
ERWAN LE PENNEC Professeur, l'École Polytechnique, France	Président
LAÉRCIO LIMA PILLA Charge de Recherche, LaBRI/CNRS, France	Examinateur
THOMAS HEIDE CLAUSEN Professeur, l'École Polytechnique, France	Directeur de thèse

Autonomous Service Management in the Cloud

Zhiyuan Yao

2022

 This work was conducted while the author was employed by Cisco Systems France. It benefited from support from French CIFRE grant #2019/0792.

Abstract

Applications and services have become more complex, while the Internet has become increasingly difficult to evolve both regarding its physical infrastructure, and its protocols and performance. Being responsible for policy configurations as well as network management and performance tuning, network operators are shifting towards the use of more and more automated tools to accomplish these tasks. The concept of "programmable networks" has emerged to alleviate the challenges, and to facilitate network evolution. This includes paradigms such as (i) software-defined networking (SDN) and (ii) network function virtualization (NFV), which decouple the forwarding hardware into the control plane and data plane, and seek to abstract network forwarding, and other networking functions, from the hardware. In the era of "big data" on cloud computing, these paradigms have enabled rich network traffic processing services, while having also reduced the granularity of task allocation in data centers. It has been recognized that shifting controllers from logically centralized to distributed will increase not only scalability but also robustness to inconsistency. Machine-Learning (ML)-based approaches have been proposed to deploy more intelligence in networks, when using decoupled control and data planes. In this context, the question explored in this thesis is whether, and how, it is possible to offer generic, data-driven networking functions in data center networks as services, for constructing autonomous networking systems which optimize networking performances with minimal human intervention and operational complexity. This thesis investigates the increasing scale, complexity, and heterogeneity of networking infrastructure, and protocols, as well as the demand for virtualization and cloud support services in terms of efficient resource management, rapid provisioning, and scalability present a set of new challenges in effective network organization, management, and optimization. This is accomplished by studying how certain network functions and primitives (traffic classification, auto-scaling, load balancing) can be reliably enhanced by various data-driven algorithms, while bearing in mind the in-production requirements in data center networks – high scalability, high throughput, low latency, and low overheads.

The characteristics of networking features in the context of in-production overlay networks are investigated first, which opens the discussion of the challenges of collecting measurements and deploying data-driven networking policies in real-world systems. To tackle these challenges, a generic tool to extract networking features from the data plane and deploy ML algorithms for various networking functions in real-world networking systems is built. A methodological framework is also designed and showcased, allowing for the developing of algorithms of different learning paradigms for networking problems. This thesis then dedicates the study to network load balancing problems in data center networks, on which a survey of state-of-the-art load balancers is provided. A hardware-based load balancing mechanism is proposed, achieving line-rate load-aware workload distribution by exploiting server load information embedded in packet headers as feedback signals. Finally, both an open-loop and a closed-loop learning load balancing algorithms are proposed based on learning algorithms, and they show better performance than state-of-the-art load balancing methods.

Keywords — Data-Center Networking, Data-Driven, Network Functions, Load Balancing, Learning Algorithms

Acknowledgement

First and foremost, I would like to express my heartfelt gratitude to my thesis advisor – Thomas Clausen – for his invaluable guidance and support throughout my thesis. As someone who introduced me to the field of computer networking, he played a pivotal role in shaping my research and academic journey. Despite the challenges posed by the pandemic, Thomas never failed to provide me with unwavering support and encouragement. His rigorous and constructive reviews of my papers, along with his humorous nattar, filling up every corner of the printed pages in red ink (and later in aqua ink, as he learned that it was my favorite color), were a constant source of motivation and inspiration. Mark Townsley, to whom I would like to extend my sincere thanks for providing me with the opportunity to conduct research at Cisco in the industrial context. Despite being remote and busy, Mark always made time to check in with me periodically and provided valuable guidance and support. I truly appreciate the freedom he gave me to pursue my research interests and explore the application of machine learning in computer networking in one of the best networking companies.

I am grateful to my thesis reporters, Sara Bouchenak and Remi Badonnel, for their careful reading of my manuscript, their detailed and relevant reports, and their willingness to travel from afar to participate in my defense. I also extend my thanks to my jury examinators, Gaël Thomas, Erwan Le Pennec, and Laércio Lima Pilla, for their presence at my defense. Their enthusiasm and pointed questions were invaluable in helping me to think critically about my research and to present my findings in the best possible light. I appreciate the time and effort that all of my committee members and jury members invested in my thesis, and I am proud to have had such a distinguished group of scholars and experts as my advisors and evaluators.

I would like to acknowledge the contributions of my colleagues at Cisco who have provided me with a stimulating and supportive research environment. I would like to particularly thank Yoann Desmouceaux, my mentor at Cisco, who has been instrumental in my journey as a researcher. His guidance, support, and enthusiasm have been invaluable to me since the beginning of my internship. As an excellent role model, he has taught me almost everything necessary for my PhD research, especially his patient explanations of the French holidays. I also appreciate the 3 years that I spent together with : Pierre Pfister for his guidance on how to mentor intern students, Mohammed Hawari for his strive for perfection, Jacques Samain for inviting me to the Thursday basketball club, and Thomas Feltin for being my PhD buddy in the neighboring group. Even only a brief period of time was spent in the office because of the pandemic, it is not possible for me to improve my babyfoot skills without the help of my dearest (for some, ex-) colleagues : Guillaume Solignac, Paul Ponchon, Aloÿs Augustin, Nathan Skrzypczak, Clemens Kunst. Thanks to everyone I have collaborated with or conversed with, with the hope that I have not inadvertently omitted anyone : Enzo Fenoglio, Wenqin Shao, Benoît Ganne, Jérôme Tollet, Éric Vyncke, David Gaumont, Michael Grad, Marouane Hamda, Jordan Augé, Frank Brockners, Mohsin Kazmi, Giovanna Carofiglio, Xuan Zeng, Malycia Ly, Luca Muscariello, André Surcouf, Ghislain Bourgin, Mauro Sardara, Michele Papalini, Angelo Mantellini, Gaetan Feige. During the three years, I am honored to have the chance to observe the growth and success of several generations of interns : Carmine Rizzi, Georges Hachem, Tristan Burgère, Leo Marché, Martin Litré, Arthur Van Tran, Edison Reshketa, Maxime Peim, Chinonso Ngwu, Hadi Rayan Ai-Sandid, Ismail Abdel Wahab, Chen-Yen Wu, and Augustin Jourdain. Ultimately, I must acknowledge the essential contribution of Carole Reynaud, whose expertise in navigating complex administrative issues help paved my way through the thesis in France.

I also want to express my gratitude to my colleague at LIX : Juan-Antonio Cordero-Fuertes for his insightful feedback on our papers, which always comes from his exceptional author's mindset; Alexandre Poirrier for being my rock during those Modals lectures when I was flooded with students' questions; Jean-Louis Rougier and Antoine Lavignotte for the intellectually stimulating discussions we had teaching together; Rim El Malki and Wael Iabidi for our wonderful conversations in the office and over lunch in the canteen. Thanks to my friends, both current and former, from École Polytechnique : Jiazi Yi for his mentorship starting from my master's program to his beautiful garden with a barbecue, Haowen Zhang for our intensive ping-pong matches and mathematical discussions during our RER-B commutes, Rui Yuan and Chenghao Lyu for the wonderful memories we shared on the basketball court and in various conferences. Thanks also Évelyne Rayssac, Fatima Pires Frances, Rimbourg Karima, for their help with the adminitrative process at X. Lastly, I must thank Vanessa Molina Magana for bringing me my first physical attendance in

a conference in Nice, which has been one of the most rewarding experiences during the past three years.

I express again my gratitude to all my co-authors for their meticulous effort and the extensive time they invested in scrutinizing and refining the manuscript. Their contributions were crucial in bringing to fruition the research works cited in this thesis. Although some have been previously acknowledged, I will mention them again here: Thomas Clausen, Yoann Desmouceaux, Juan-Antonio Cordero-Fuertes, Mark Townsley, Carmine Rizzi, and Ziyan Ding. Special thanks to Zihan Ding, one of my co-authors pursuin his Ph.D at Princeton University. Our collaboration started remotely on GitHub during the pandemic period, which has led to fruitful publications in NeurIPS and CIKM. I am very grateful for his invaluable contributions and wish him a bright future leading up to his final Ph.D defense.

In the end, I would like to thank my family and friends for their constant love, encouragement, and support throughout my doctoral journey. Despite the challenges posed by the pandemic, they have always been there for me, cheering me on and helping me stay motivated. I regret that I have not been able to visit them in China during the past three years due to the collision of my PhD with the pandemic, but I am grateful for their understanding and patience. Their immutable belief in me has been a source of motivation and inspiration. I also want to express my gratitude to my friends who came all the way to this small plateau in the south of Paris to attend my final defense. Special thanks go to Yang Liu, who traveled from the Netherlands to be here. Your presence meant a lot to me, and I appreciate your support. I hope that my friends who attended my final defense reached a consensus that, despite not fully grasping all of my slides, I have made a small contribution to our society.

Last but not least, I would like to express my unfeigned gratitude to my wife Qiuge Wang, my cat DaXiang, and my puppy FeiChang, for their unconditional love, patience, and understanding during this challenging time. Their support has been a constant source of strength and inspiration, and I could not have accomplished this without them. Thanks to Qiuge, for always supporting me throughout my thesis, even when she had concerns about my well-being. I am grateful for her care and love that kept me going, and I am proud to have made it through with only a single slightly herniated lumbar disc. Thanks to my uncommunicative cat, whose silent companionship helped me learn how to deal react when the paper reviewers ignore my passionate arguments and conscientious proposals. Finally, thanks to my energetic puppy. Our walks together during the pandemic provided a much-needed break from the isolation of research, and your playful nature helped me understand the nuances of reinforcement learning, a critical component of my thesis. Thank you all for being my family and for always being there for me.

Contents

Part I

Introduction

Chapter 1

Introduction

In the evolving digital world, the adoption of cloud services and management grows rapidly – accelerated, especially as the post-pendamic era crystallizes since 2020[1]. More workloads – computing, storing, and distributing huge amounts of data – are migrated to the cloud [1], and the amount of heterogenous applications – *e.g.,* web services, video streaming, big data processing, distributed storage – running in the cloud has been increasing [2]. With increasing demands for elastic software and hardware configurations, emerging technologies – including virtualization [3,4], software-defined networks (SDN) [5, 6], and programmable hardware devices [7, 8] – have offered improved programmability of data center network architectures, since programmable networks separate underlying hardware from control software and improve operational flexibility. The combination of these technologies allows deploying various network functions in "middleboxes", who can be composed into chains of network services and meet high-level service requirements and intents [9, 10]. Data center networks undergo continuous and frequent updates and changes at large scales [11]. This makes the dynamicity and complexity of the deployed applications in the cloud hard to be tracked manually in real-time [12]. Service providers and cloud operators are thus confronted with more challenges when orchestrating and managing all provided services, in order to provide scalable services and optimize quality of service (QoS) while constraining power consumption and minimizing operational overhead [13, 14].

Because of their complexity, large-scale, elastic, and heterogenous data center networks makes simple heuristics or human interventions result in sub-optimal performance [12]. Consequently, the development of data-driven algorithms – *e.g.,* machine learning (ML), which are capable of approximating complex interactions and dynamic systems – has given rise to an ambitious goal for realizing autonomous network orchestration and management [15]. Data-driven algorithms benefit from the data flowing in data center networks, to progressively learn and make informed decisions that offer improved performance for customized applications and tailored use cases [16, 17], for *e.g.,* traffic classification [18–20], intrusion detection [21, 22], congestion control [23–25], traffic optimization [26–28], and resource allocation [29, 30]. However, it is challenging to use these data-driven algorithms to drive management decisions and build autonomous networking systems which function *in production in real-time*. This, because it requires (i) to collect reliable and scalable task-specific networking measurements with minimal overhead, and (ii) to adapt appropriate learning algorithms to infer system states, predict traffic patterns, and generate informed control policies.

In this context, the question explored in this thesis is whether, and how, it is possible to offer generic, data-driven networking functions in data center networks as services, for constructing autonomous networking systems which optimize networking performance with minimal human intervention and operational complexity. This is accomplished by studying how certain network functions and primitives (traffic classification, auto-scaling, load balancing) can be reliably enhanced by various data-driven algorithms, while bearing in mind the in-production requirements in data center networks – high scalability, high throughput, low latency, and low overhead.

[1] According to a report in 2022 (https://www.gartner.com/en/newsroom/press-releases/2022-04-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-500-billion-in-2022), the worldwide public cloud services market is predicted to grow from $411 billion in 2021 to $495 billion in 2022. According to a technical survey (https://info.flexera.com/CM-REPORT-State-of-the-Cloud), the number of heavy cloud users – who run more than 25% of workloads in the cloud – increased from 59% in 2021 and 53% in 2020 to 63% in 2022.

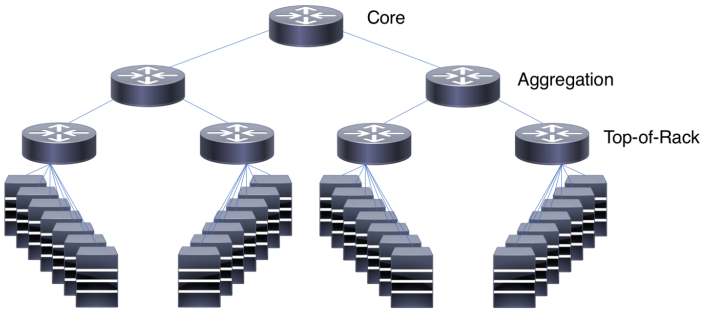


Figure 1.1: Three-tiered data center network topology [31]

The remainder of this introductory chapter is structured as follows. Section 1.1 introduces and reviews the foundamental concepts in data center networks, including the network protocols, topologies, management frameworks, emerging technologies, and key network functions that are studied in this thesis. Section 1.2 provides a brief survey over the applications of data-driven algorithms on networking problems, and identifies the challenges of constructing autonomous networking systems with ML algorithms in high-performance and large-scale cloud data center networks. Section 1.3 finally argues for the usage of these emerging technologies and adaptation of various data-driven methods to augment data center primitives and network functions, by enabling "middleboxes" to make informed decisions based on their observations in a distributed manner.

1.1 Background

This section presents background information on data centers: section 1.1.1 reviews data center network architectures, section 1.1.2 introduces traditional network protocols used in the Internet and in data centers, and section 1.1.3 introduces emerging technologies in data center networks that offers imrpvoed performance and programmability.

1.1.1 Data Center Architectures

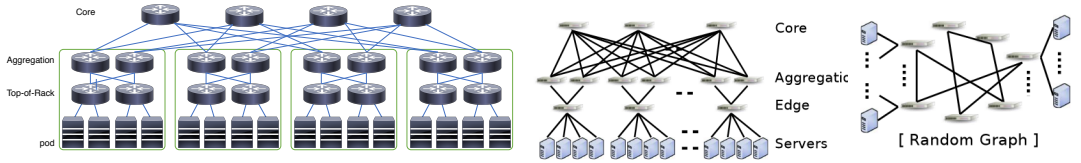
Behind the rise of cloud computing, data centers play a significant role, agglomerating mass compute and storage resources and providing various scalable services, *e.g.,* big data applications [32–36], high performance computing [37–39], and distributed and reliable storage [40–42]. In data centers, it is not surprising to find tens of thousands of servers interconnected by thousands of switches[2] [43, 44], which aim at meeting the throughput and latency requirements from different applications. To avoid creating bottlenecks at the interconnection level, the link speed of data center networks has increased from 25Gbps to more than 100Gbps because of the 100Gbps per-storage-node throughput and the microsecond-level access latency of NVMe SSD [45, 46]. Yet, the ever-growing scale of data centers and service-level demands of applications push network architects to design and study network topologies which deliver high-throughput capacities and low end-to-end latency, while constraining hardware and operational costs, and reducing environmental impacts [47, 48].

The throughput capacity of a data center network is evaluated by calculating the *bisection bandwidth*[3]. The *bisection bandwidth* is defined as the throughput between two bisected partitions of a network topology. The bisection of the two partitions is conducted so that the throughput between the two partitions is minimum. A topology is said to have full bisection bandwidth if its bisection bandwidth is no less than the throughput of half of the total servers. Such topologies that follow one family of data center network design – Clos-based designs [49] – allow for arbitrary application instance placements to achieve optimal throughput capacity.

Clos-based designs – *e.g.,* fat-tree [50], VL2 [51], Jupiter [52], Facebook Fabric [44], and F10 [53] – are switch-centric, and have bi-regular, tree-based hierarchical structures, where leaf server nodes

[2] In this section, the word "switch" is used to designate devices that can perform Layer-2 and/or Layer-3 forwarding, indifferently.

[3] Beware that the bandwidth here actually refers to the throughput in networking systems.

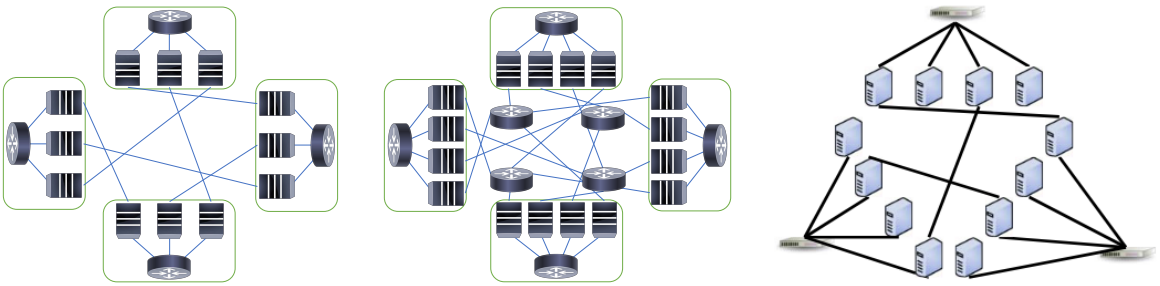


(a) A 4-ary Fat-tree data center network topology [50]

(b) VL2 topology [51]

(c) Jellyfish topology [54]

Figure 1.2: Switch-centric data center network topology



(a) DCell [55] with $n = 3$ and $\ell = 1$

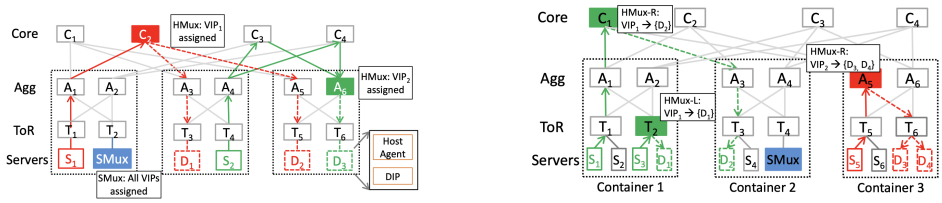
(b) BCube [56] with $n = 4$ and $\ell = 1$

(c) Level-1 FiConn [57] with $n = 4$

Figure 1.3: Recursive data center network toplogies

are interconnected by low-level switches, which are then interconnected by high-level switches. As depicted in figure 1.1, a 3-tier tree-based architecture consists the following components. Servers are grouped as *racks*. Each rack is connected to a *Top-of-Rack* (ToR) switch, which is further connected to an *aggregation* switch. The top-tier *core* switch finally connects the aggregation switches. Connections between 2 arbitrary servers in the data center network thus traverse at most 5 switches. This topology may create over-subscription, where the total server-facing throughput surpasses the total uplink-facing throughput. For instance, 8 10Gbps servers connected by a 40Gbps ToR switch give a $2:1$ over-subscription ratio, which makes the ToR switch a bottleneck in the network. In tree-based topologies, higher-tier switches and links are required to have higher throughputs so as to avoid creating bottlenecks, especially among core and aggregation switches. However, switches with high throughput capacities are expensive and power-consuming. The fat-tree topology (figure 1.2a) replaces the high-throughput top-tier switches by interconnecting multiple identical commodity switches, so as to achieve full-bisection bandwidth, while providing higher scalability and removing single point-of-failure [50]. A k-ary fat-tree can connect $k^3/4$ servers with $k(k+1)$ identical k-port switches. VL2 [51] (figure 1.2b) uses a topology similar to fat-tree, yet, instead of connecting each aggregation switch in a pod to a portion of the core switches, VL2 connects the aggregation switches and the core switches as a complete bipartite graph. By connecting each rack to 2 aggregation switches, and leveraging high speed uplink-facing links, VL2 has lower cabling complexity than fat-tree. Though both fat-tree and VL2 topologies rely on centralised management and incur high operational costs.

To reduce operational costs and improve the network scalability, alternative designs adopt expander-graphs to interconnect switches, and propose incremental installation, where every switch connects to a group of servers – unlike aggregation switches and core switches which connect to other switches. Jellyfish [54] (figure 1.2c) reserves several ports on each ToR switch to interconnect switches and creates a degree-bounded random regular graph. It achieves incremental expansion by purging a randomly selected link between 2 ToR switches, to which the added ToR switch is linked. Connections between 2 arbitrary servers traverse less hops than in a fat-tree. DCell [55] (figure 1.3a) is a recursively constructed topology, which offloads the interconnection and routing responsibility to servers, instead of switches. At level-0, a rack of n servers are connected to a ToR switch. A level-1 DCell is then constructed with $n+1$ level-0 DCells, so that each level-0 DCell is connected to every other level-0 DCell via links between servers, as a full mesh. In a level-$(l+1)$ DCell, $s_l + 1$ level-l DCells can be found, where s_l is the number of servers in a level-l DCell. This recursive design offers high scalability – the number of servers grows doubly exponentially with the port number of each server. BCube [56] (figure 1.3b) is similar to DCell, sharing the same structure



(a) Suboptimal VIP and DIP configuraiton [58] **(b)** VIP and DIP configuration exploiting locality [59]

Figure 1.4: Service deployment in a fat-tree topology

of level-0 DCell. At level-$(l+1)$, BCube interconnects n level-l BCubes with n^{l+1} n-port switches, each of which is connected to 1 server in every level-l BCube. FiConn [57] further limits the port number on each server to 2. Sharing the same topology of level-0 DCell, a level-$(l+1)$ FiConn is constructed with $b/2+1$ level-l FiConns, where b is the number of server with available ports in each level-l FiConn. These 3 recursive data center network topology requires, however, to deploy a custom forwarding module in each server, which may incur additional operational complexity.

Atop these *physical* data center network topologies, virtualization technologies allow different tenants[4] sharing physical resources (*e.g.,* CPU, disk, memory, network) in a data center or in the same host machine (operating system kernel) in the form of *virtual machines* (VMs) [60], containers [61], or using the Lambda model [62]. These technologies motivate the development and deployment of microservices [63] on commodity hardware, which reduces the expenses for dedicated hardwares and improves the service scalability by elastically expanding or decreasing provisioned instances [64–66]. The development of virtualization, where computers are emulated and/or sharing an isolated portion of the hardware by way of Virtual Machines (VMs), or run as isolated entities (containers) within the same operating system kernel, has accelerated the commoditization of compute resources. Therefore, the gigantic in-production data center networks – which may comprise thousands of servers – are partitioned into small pods (server clusters), where different services are hosted. Running over various data center network architectures, virtualization technologies improve service transparancy not only for clients but also for service providers [67], which can deploy their applications without worrying about server management. However, this elastic computing scheme may lead to several issues, *e.g.,* suboptimal configurations [59], resource contention [68], and colocated-workloads [69]. Suboptimal performance can happen when, *e.g.,* malicious tenants plunder shared resources to improve their network performance [70], or the same configuration is applied in heterogenous architectures [71, 72].

Similalry, suboptimality can happen in typical server cluster and service deployment. Each service is provided at one or multiple virtual IP addresses (VIPs), running over a cluster of servers. Each server in the cluster can be identified by a unique direct IP address (DIP). Traffic and queries from the clients destined to a VIP are load balanced among the DIPs of the service. When deploying the service in the data center network, suboptimal VIP and DIP placement (figure 1.4a [58]) can incur substantial throughput usage overhead and break the full-bisection bandwidth because of traffic redirection, which can be largely reduced by exploiting locality – placing DIPs so that the traffic of a given VIP stays within the same rack (figure 1.4b [59]).

As described in this section, data center architectures have evolved to achieve scalable desploymenet. However, to efficiently manage and operate these large distributed networking architectures, it requires complex overlay technologies and dedicated protocols, which will be introduced in the following section.

1.1.2 Network Protocols

To provide optimized network performance, traffic and network management technologies play an important role. These technologies are build on top of conventional layered network stacks – *e.g.,* "Open Systems Interconnection (OSI) reference model" [73], which consists of 7 layers, *i.e.,*

[4]For the purpose of this section, a tenant is defined as any entity using some of the resources of a data center for a single purpose, be it an individual, a company, or a logical entity such as a project to which resources are assigned, *etc.*

(i) physical, (ii) data link, (iii) network, (iv) transport, (v) session, (vi) presentation, and (vii) application layers. These layered architectures, where layer-n provides service to layer-$n+1$ and uses services from layer $n-1$, allows for building large systems with complex functionalities with high modularity and scalability, since the change of one layer's design and implementation has no impact on the remainder of the system.

From among the 7 layers of the OSI reference model, the most commonly used in the world is the 4-layer TCP/IP model [74], summarized below:

1. The **application layer (Layer-7)**, where various applications reside, allows different endpoints to communicate using corresponding application-layer protocols (*e.g.,* HTTP [75], HTTPS [76], SMTP [77], FTP [78] and DNS [79]), by transparently using the layers below. A packet of information at the application layer is called a *message*.

2. The **transport layer (Layer-4)** transmits application-layer messages between application endpoints via logical channels. The Transmission Control Protocol (TCP) [80] and the User Datagram Protocol (UDP) [81] are two widely deployed transport protocols [1, 82]. TCP provides a connection-oriented service to applications and uses acknowledgement of data transmission to achieve reliable and ordered data transmission. UDP, on the other hand, provides connectionless service to its applications, which guarantees no reliability for data transmission. A transport-layer packet is called a *segment*.

3. The **network layer (Layer-3)**, allows for moving messages across inter-connected networks, and provides the service of delivering transport-layer segments from a source host to the transport layer in the destination host. A network-layer protocol data unit is called a *datagram*. In the Internet, the Layer-3 protocol used is Internet Protocol (IP) [83, 84]. IP defines the fields of the datagram and stipulates the behavior of all Internet components with the network layer. Layer-3 protocols also consists routing protocols, which build routing tables between different devicess and determine the routes of datagrams between source and destination hosts.

4. The **data-link layer (Layer-2)** provides, to the network layer, the service of moving datagrams from one device to the next device. Layer-2 takes care of (wired or wireless) message modulation/demodulation to/from physical mediums and controls access to these mediums by way of collision avoidance, modulation selection, retransmissions, and auhentication. A link-layer protocol data unit is called a *frame*. This manuscript focus on wired data centers, where devices on the same data-link (*e.g.,* physical machines in the same rack) can exchange frames and handle datagrams by Ethernet.

This protocol stack is widely deployed over both the Internet and within data center networks. It embraces the *end-to-end* [85] design, where the networking properties (*e.g.,* reliability, security) between two communicating processes rely on the endpoints of the logical transmission. In the context of data center networks, the end-to-end communications follow the consumer-producer paradigm [86] – where servers in data center networks provide contents to end users – and the end hosts on two sides can be highly asymmetric. With (i) the ever-growing scale of data center networks, (ii) elastically deployed resources, and (iii) dynamic and unpredictable traffic, various protocols are proposed based on the protocol stack and networking topology abstraction to improve scalability, handle system dynamics, and guarantee QoS with constraints over provisioned resources.

In terms of scalability, the exhaustion of IPv4 addresses[5] has triggered the use of Network Address Translation (NAT) devices, and the development of IPv6 [84]. In multi-tenant networks, Virtual Local Area Networks (VLANs) [87] enable each Ethernet packet to carry a 12-bit identifier, which allow switches to forward packets only among per-VLAN-specified physical ports, so as to hide the complexity of physical network architectures and provide consistent and isolated services [67]. The limited amount of tenants ($2^{12} = 4096$) that are supported by VLAN further triggers the development of Virtual eXtensible Local Area Networks (VXLANs) [88] and Network Virtualization using Generic Routing Encapsulation (NVGRE) [89], both of which support up to 2^{24} tenants. Though requiring additional encapsulations – by way of UDP/IP and GRE packets, respectively, which incur additional communication overhead, VXLAN and NVGRE achieve improved scalability, flexibility, and can be applied for inter-data-center communications. Other

[5] https://www.icann.org/en/system/files/press-materials/release-03feb11-en.pdf

protocols based on Virtual Private Network (VPN) – *e.g.,* Virtual Private LAN Service (VPLS) [90] and Ethernet VPN (EVPN) [91] – also enable inter-data-center communications. These protocols use Multi-Protocol Label Switching (MPLS) [92] labels to identify and transmit – among different tenants – Layer-2 frames across MPLS capable networks between data centers.

To handle bursty traffic (*e.g.,* with a median flow inter-arrival time < 250 μs [93]) in distributed systems, different traffic control techniques have been developed. For instance, Equal Cost Multi Path (ECMP) [94] splits traffic across multiple equal cost paths and routes each flow through one selected path using the hash of five tuples of the flow[6]. ECMP is a fundamental path selection mechanism, and used in many multipath routing protocols *e.g.,* Shortest Path Bridging (SPB) [95] and Transparent Interconnection of Lots of Links (TRILL) [96]. As straightforward as ECMP is, it is agnostic to both existing network utilization and the "size" of a new incoming flow, which can cause overloaded or starved resource utilization and degraded QoS. To resolve this issue, protocols are proposed that enable more fine grained contro, *e.g.,* Hedera [97], which periodically monitors edge switches and ToR switches to detect large flows based on the throughput occupation threshold (*i.e.,* $> 10\%$ link capacity). These large flows are then scheduled and allocated to different paths, using simulated annealing by a centralized controller, to achieve max-min fairness sharing of the Network Interface Controller (NIC) throughput, by calculating the number of large flows between source-destination pairs. Unlike Hedera, which allocates path for each flow, Conga [98] splits traffic to "flowlets" [99] – bursts of packets separated by spacing larger than the delay difference between the parallel paths – so that sending two bursts of packets over different paths does not cause packet reordering. Relying on hardware-modified devices, Conga encodes local link load estimation in VXLAN headers at intermediate switches and switches, and decodes congestion information at ToR switches, which make corresponding traffic placement decisions.

Extensions have also been made on top of TCP to reduce packet loss and timeout. Multipath TCP (MPTCP) [100] stripes data to subflows at the sender, and reorders and reconstructs the received data at the receiver. Similar to incast congestion TCP (ICTCP) [101] and data center TCP (DCTCP) [102], MPTCP also enables congestion control by adjusting the subflow congestion window size to improve network utilization and fairness. Traffic control techniques are also proposed for inter-data-center applications [103, 104]. B4 [103] utilizes centralized traffic engineering, multipath routing, and rate restriction at the network edge to achieve high throughput utilization. B4 detects and avoids using bottleneck links by iteratively and fairly allocating throughput for traffic on their corresponding paths that exclude previously detected bottleneck links, until all traffic allocation demands are fulfilled or every path contains bottleneck links. SWAN [104] classifies traffic into 3 types – *i.e.,* background, interactive, and elastic traffic – and assigns different priorities accordingly. Background traffic has large volumne of data to transfer yet it is not latency-critical. It benefits from a small reserved amount of link capacity (10%) to improve throughput utilization. Interactive traffic is sensitive to packet loss and latency yet it transmits low volumne of data. Elastic traffic falls a the spectrum between the previous two types of traffic. SWAN calculates the traffic rate and path allocations based on estimated interactive traffic and reported background and elastic traffic before pushing configurations down to switches and hosts.

As described in this section, network protocols have been developed along with various data center architectures to address scalability issues and to facilitate traffic management. However, the applications running in data center networks grow more and more heterogeneous [44, 69], where latency-critical applications (*e.g.,* Web), computational-intensive tasks (*e.g.,* big-data), throughput-intensive services (*e.g.,* database) are co-located in the same data center network. This gives rise to – besides the high-performance requirements for data center networks – the demands for high programmability in the data plane.

1.1.3 Towards High Performance and Programmability

As discussed, data center networks have become more and more scalable to support the growth in networking traffic, and to provide high-throughput and low-latency services. In addition, the increasing complexity of provided services require data center networks to quickly react to dynamic requests and efficiently maintain appropriate levels of QoS with available resources. Therefore, with diverse management and optimization objectives, the data plane in data center networks seeks higher flexibility and programmability, in addition to high throughput and low latency. Though

[6] The five tuples of a flow typically refers to the source IP address, destation IP address, source port, destination port, and protocal number.

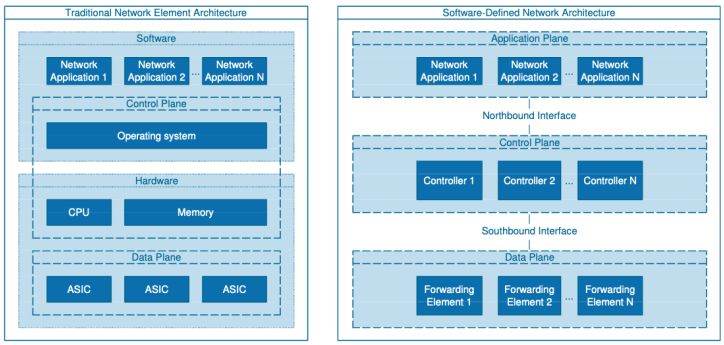


Figure 1.5: Transition from traditional network to SDN architecture

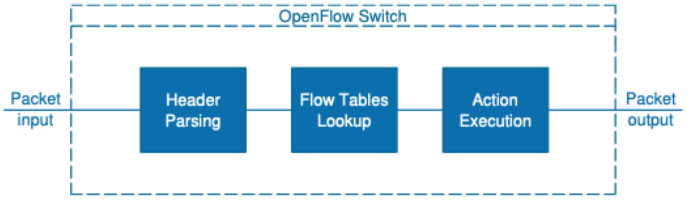


Figure 1.6: OpenFlow switch

networking systems have been conventionally developed and deployed statically in hardware networking devices – *e.g.,* through application-specific integrated circuits (ASICs) [105] – technologies and ideas have been proposed to increase network programmability, discussed in the following.

Software-defined networking (SDN) dissociates the routing and decision-making process (*control plane*) from the network packets forwarding process (*data plane*) [106, 107]. As depicted in figure 1.5, in the traditional network architecture, networking devices conduct both control plane and data plane operations in integrated software-hardware systems, whereas in SDN architectures, based on different objectives, management logics are encapsulated in different centralized controllers in the control plane to bridge the requirements from various applications with simple and straightforward packet forwarding functions in the data plane. Applications can generate rules – which decide how network traffic is handled – based on application-specific demands (*e.g.,* for QoS, routing constraints, resource utilization and fairness), and push these rules to the control plane via the northbound interface[7], which communicates with the layer above. These rules are then compiled and translated to corresponding forwariding rules by the controllers, which then configures via the southbound interface – which communicates with the layer below – network devices (*e.g.,* ToRs and switches), which forward packets based on the configured forwarding table. Communications between the control plane and the data plane is achieved using OpenFlow [108], a protocol with (i) internal flow tables, (ii) communication channel connected to the OpenFlow controller, and (iii) standardized communication interfaces for adding/removing entries to/from the flow tables. As depicted in figure 1.6, on receipt of network flows, an OpenFlow switch parses packet headers to identify to which action the flow corresponds – from among 3 types of actions, *i.e.,* forwarding to one or multiple egress ports, forwarding to the controller, and dropping – defined in the flow table. Though SDN provides programmable APIs to gather per-flow or application-level statistics in a centralised way, to adaptively update configurations, it requires using network equipments that supports the OpenFlow protocol [109, 110].

Network Function Virtualization (NFV) [111] is a different technology that brings programmability to networking systems. NFV extracts and abstracts different network functions (*e.g.,* firewalls, load balancers, and VPN gateways [14, 63]) to provide reliable service management, and transparent operations. These on-demand virtualized network functions (VNFs) are deployed on commodity computing platforms, which increase not only network programmability, but also

[7] In the field of computer networking, a northbound interface is an interface that allows a lower-level network component to communicate with a higher-level or central component. Contrariwise, a southbound interface allows a higher-level component to communicate with a lower-level network components.

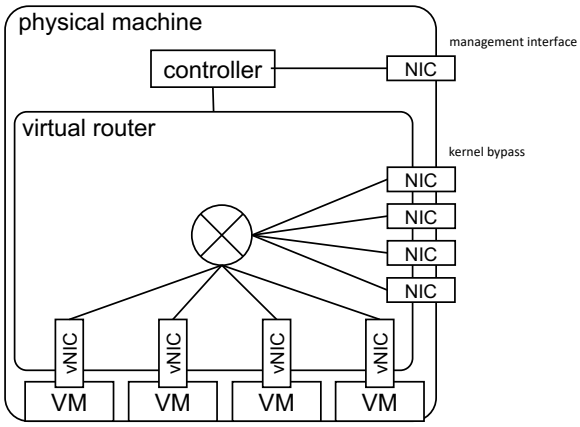


Figure 1.7: Virtual router operation

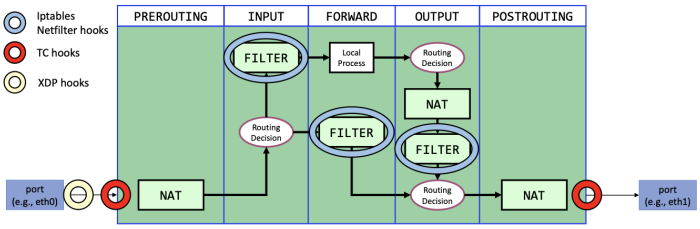


Figure 1.8: eBPF hooks in the data plane

improve scalability and elasticity, therefore they help balance the trade-off between capital expenditures and QoS in data center networks. Unlike OpenFlow, VNFs – running on commodity computing platforms – replace or augment dedicated hardware devices and play a significant role in large-scale data center networks. For instance, as depicted in figure 1.7, a VNF can be instantiated by way of a virtual router, which runs as an encapsulated service in a physical machine. In multi-tenant setups, the VNFs configure (i) virtual interfaces that connect to each tenant (in the form of containers or VMs) hosted in the same machine, and (ii) physical interfaces that connect to physical NICs for inter-machine communications. Interconnecting different devices in the data center networks, VNFs can be deployed – either as independent processes running in the host OS or as containers or VMs themselves – to achieve more advanced functionalities, *e.g.,* firewalling, NAT-ing, encrypting, and load balancing.

SDN and NFV complement each other with their own technical benefits. NFV allows deploying the key idea of SDN by way of virtualizing different network services along with the SDN controller in the cloud, using commodity machines. In return, SDN allows for effectuating network management decisions and configurations generated by VNFs [14].

To implement SDN and NFV, different systems and languages have been proposed [112–116]. The extended Berkeley Packet Filter (eBPF) [112] framework offers a set of instructions and a Linux-kernel-based execution environment, serving as a universal in-kernel virtual machine [117]. It allows developers to program packet processing logic in C language, which is then compiled into eBPF code. The compiled instructions in eBPF can then be executed in programmable hardware devices (*e.g.,* SmartNICs [118]) or be processed in the Linux kernel by attaching to an interface called *hook* (figure 1.8). Hooks permit the registration of custom programs given specific events. The eXpress Data Path (XDP) [119] is one type of the hooks that allow eBPF programs to attach. Operating at the lowest layer of the network stack in the Linux kernel – only on the RX path of the network driver, it enables fast packet processing applications – *e.g.,* mitigating distributed denial-of-service (DDoS) attacks and tunneling – with high programmability. XDP runs the attached eBPF programs before socket memory buffer allocation by the kernel and it does not require kernel recompilation when modifying the eBPF programs. Traffic controller (TC) [120] is another hook for attaching eBPF programs. Though TC is not as performant as XDP, it can make use of more

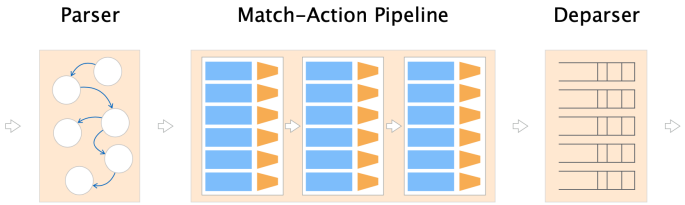


Figure 1.9: Basic building blocks of P4

statistics and data parsed from network packets, than can XDP, and it operates at both ingress and egress path. Similar to eBPF and XDP, Click [114] also allows for plugging in custom networking programs and applications for packet processing in the kernel. Click applications consist of multiple components (called *elements*) implemented in C++. Defined in application-specific configuration files, elements are organized and interconnected as packet processing graphs, which can be compiled and executed in either the user space or the Linux kernel.

In high performance networks, where the link throughput capacity can easily surpasses 10 Gbps, inter-packet arrival times become sub-microseconds or even tens of nanoseconds [113]. Linux-kernel-based instructions and operations can incur significant per-packet processing latency because of *e.g.,* context switches. The Data Plane Development Kit (DPDK) [115] bypasses the kernel and keeps polling packets from NICs directly within user space. The application running in the user space will then process the fetched packets according to different objectives. Based on DPDK library, the Vector Packet Processor (VPP) [121] – an open-source high performance programmable data plane – provides a variety of network functions optimized with techniques including aligned memory access, pre-fetching, loop-unrolling, multi-core processing, etc. Similar to Click, VPP is based on packet processing graphs, which makes it modular and extensible. To bring programmability to line-rate capable hardware devices, the P4 language [116] has been proposed for embedding custom data plane programs in programmable switches (*e.g.,* NetFPGA [7,122]). P4 embraces the Protocol Independent Switch Architecture (PISA), which abstracts the packet processing operation into 3 main steps (figure 1.9). On receipt of network packets, the parser inspects packet headers and extracts values in different fields. These parsed values are fed forward to the ingress process, which consists of configurable match-action tables. The match-action pipeline allows developers to implement their core packet processing logics, before steering packets to corresponding egress ports. Finally the deparser reassembles the packets and either drop or forward them to the next device. With this architecture, P4 enables defining and programing various network functions in line-rate physical switches.

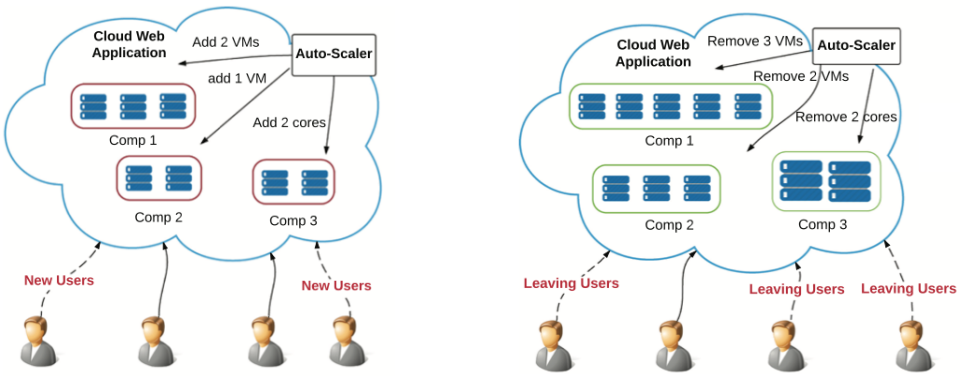
The concept of "programmable networks" has been advanced with the paradigms of SDN and NFV, which help alleviate the challenges emerged together with the evolution of data center network architectures. In support of the role of cloud computing [123, 124], these paradigms have enabled rich network traffic processing services while reduced the granularity of task allocation in data centers. However, the increasing scale, complexity, and hererogeneity of networking infrastructures and protocols, as well as the demand for virtualization and cloud services in terms of efficient resource management, rapid provisioning, and scalability presents a set of new challenges in effective network organization, management and optimization [125–128].

1.2 Bringing Intelligence to Data Center Networks

The abstraction of network functions enhances scalability and elasiticity of network application deployment in data center networks. This section introduces several key network functions and the challenges that they are facing in modern data center networks (section 1.2.1). With the advancement of machine learning algorithms, this section then discusses how data-driven methods can help improve the performance of network functions (section 1.2.2).

1.2.1 Network Functions

By way of NFV, different and distinct network functions can be consolidated on physical machines. They support network applications based on different requirements. Chaining different



(a) Scaling Out/Up **(b)** Scaling In/Down

Figure 1.10: Auto-Scaling

network functions can offer a combination of services – *e.g.,* deep packet inspection (DPI), intrusion detection system (IDS), autoscaling, firewall, load balancing [129–131]. This has increased network operation agility since network services can be updated on the fly, by simply reconfiguring corresponding network function instances. 3 categories of network functions in data center networks will be covered in this manuscript:

- Service assurance [132–134]: service level agreement (SLA) monitoring, auto-scaling;
- Application-level optimization [64, 65, 135]: load balancing, caching;
- Security [136]: firewalls, IDS, DDoS and anomaly detector.

This section introduces network function with one example from each category.

Auto-Scaling

Auto-scaling is a service-aware network function [137,138]. To minimize operational costs, while guaranteeing QoS, cloud operators need to elastically provision server capacities – dynamically and ideally autonomously adjusting computing, networking, storage resources according to the demands of clients. The objective of auto-scaling is to minimize the operational and resource cost without breaking SLAs. For instance, when the incoming requests grow leading to overloaded or congested resources, the auto-scaler can scale out (*e.g.,* spin up more VMs) or scale up (*e.g.,* append resources to existing VMs) for the application to meet an SLA (figure 1.10a). Likewise, when the traffic goes down and the amount of incoming requests decreases, the auto-scaler can scale in (*e.g.,* shutdown some VMs) or scale down (*e.g.,* detach resources from existing VMs). To achieve this, the auto-scaler needs to keep monitoring the system performance (*e.g.,* QoS in terms of latency and throughput, cost of provisioned resources, etc). To this end, performance indicators should be carefully selected so as to reflect the actual system state, in line with the operating and management objectives. Based on measurements, the auto-scaler needs to analyse and determines the necessity of planning scaling events, including *e.g.,* the timing, the scale of impacted VMs, the amount of added or removed resources, etc. The auto-scaler needs to consider the prediction of the workload in the system, and need to avoid oscillating between opposite scaling actions within a short period of time. Once scaling events are planned, the cloud operator will execute these scaling actions. With the complex optimization goal of minimizing provisioned resources while providing guaranteed SLAs, there are growing interests in intelligent server capacity configuration systems [29, 137, 139, 140].

Traffic Classification

As one of the key network functions in data center networks, traffic classification allows distinguishing between different types of traffic [19, 22], to allocate appropriate resources and to achieve service level agreements. It also helps detect anomalies and security threats to prevent potential damages and losses [22]. Traffic classifiers therefore, are deployed as network functions on the

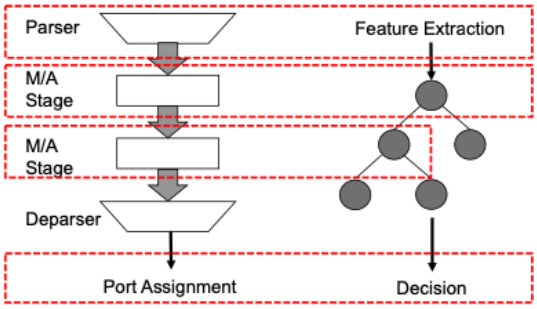


Figure 1.11: Traffic classification procedure mapped to a switch pipeline, where M/A indicates match-action [19].

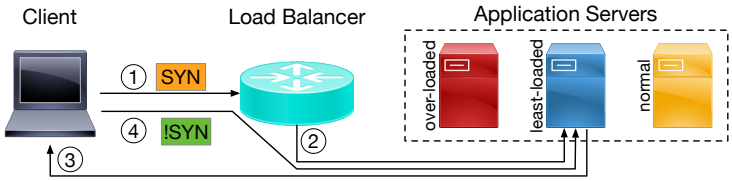


Figure 1.12: Workflow of Layer-4 load balancers in data center networks.

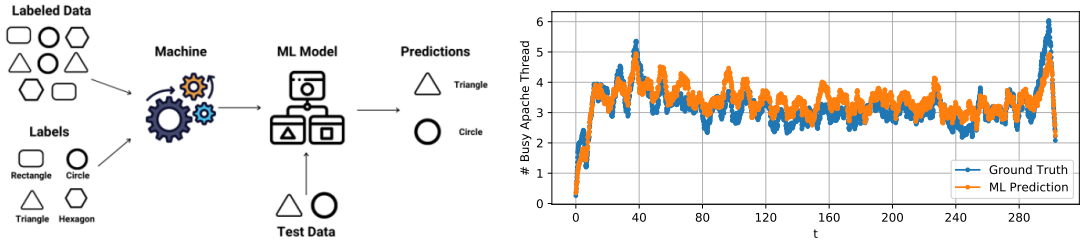
edge of data center networks, to prevent security issues and assign priorities to different sources of network traffic. Port-based traffic classification mechanisms can identify packets associated with specific applications, based on port numbers used. However, dynamic port-negotiation mechanisms might be used to bypass firewalls and security applications. This motivates more traffic classification techniques to be proposed. As depicted in figure 1.11, the first step of traffic classification is to extract and select relevant and accurate features *e.g.,* from packet headers when the payloads are encrypted [141]. Based on the collected features, a decision making model – which can take form of *e.g.,* a decision tree – will be applied to decide whether the packets will be dropped or forwarded to specific egress ports.

Load Balancing

In multi-tenant data center networks, in order to optimize cost and energy, network applications and services are replicated among multiple instances running *e.g.,* in containers or VMs, so as to provide an optimized trade-off between cost and user QoS [142]. With constrained capacity and huge volume of traffic, load balancers are indispensable "middleboxes" in data center networks for transparent operation and optimal resource utilization. As the network traffic keeps growing rapidly and network traffic becomes more dynamic and heterogenious [1,69], Layer-4 load-balancers become a key component for efficient resource utilization in data center networks, distributing network traffic addressed to a given cloud service *evenly* on all associated servers, while *consistently* maintaining established connections [64, 65, 143, 144]. The workflow of network LBs is depicted in figure 1.12. On receipt of a new connection request ① (*e.g.,* a TCP SYN), LBs ② determine to which server the new connection is to be dispatched. Servers ③ respond to the request using direct-source-return (DSR) mode[8]; LBs thus have no access to the server-to-client side of communication. Finally, ④ the load balancing decision made upon the new connections is preserved until connection terminates. There are two requirements for LBs:

- *Per-Connection-Consistency* (PCC): Packets from the same connection need to be forwarded to, and handled by, the same server. This is a strong request, so as to gurantee per-connection consistency and provide high service availability;
- *Fairness*: Workloads on all servers need to be balanced, and overloading and starvation of provisioned resources must be avoided, so as to improve resource utilization and QoS.

[8]DSR is enabled for response packets from servers to clients to bypass LBs. It relieves LBs of handling 2-way traffic, improving network throughput [64].



(a) Overview **(b)** Example: server load prediction (regression)

Figure 1.13: Supervised learning

Network LBs have limited observations of the system availability and they are agnostic to application-level information including task sizes and actual server load states. With different server VMs, running on top of heterogeneous hardware and elastic infrastructures [72], it is challenging to assign correct weights to servers according to their actual processing capacities. This process conventionally requires human intervention, which can lead to error-prone configurations [65, 145].

1.2.2 Demands for Learning in the Cloud

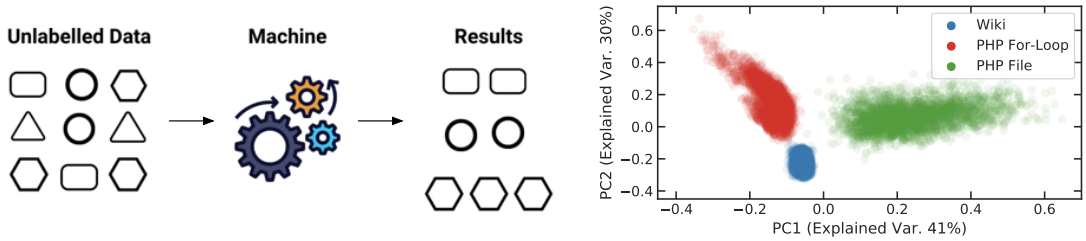
This chapter so far has introduced 3 key network functions in data center networks. They play different roles yet they share common challenges that need to be addressed. In the context of cloud computing, growing elasticity, scalability, and QoS requirements in data center networks demand for heterogeneous yet adaptive and autonomous rules and decision-making policies to be installed in network functions. These requirements attract a rising trend of applications of machine learning (ML) algorithms to networking problems [22].

Data-driven mechanisms based on machine learning (ML) [27, 146] and reinforcement learning (RL) algorithms [26, 28] can be applied and can show performance gains in various network applications. With large amount of data flowing in the networking systems, these learning algorithms effectively helps network operators to extract information from the data and gain insights in data center networks, so as to make informed management decisions and optimize networking performance. For instance, auto-scaling systems and load balancers can achieve improved QoS with reduced cost based on periodically polled resource utilisation of distributed network devices (*e.g.,* application servers) [139, 143]. Traffic classification and anomaly detection can help detect security threats with increased accuracy, based on network traffic characteristics extracted from network traces [17, 147]. This section introduces the 3 main families of data-driven mechanisms applied to networking problems, *i.e.,* supervised learning, unsupervised learning, and reinforcement learning.

Supervised Learning

As depicted in figure 1.13a, supervised learning takes labeled input data together with their associated labels to train ML models, which can make predictions given new sets of input data. Applications aiming at assigning input data to one of a finite amount of discrete categories are called *classification* tasks. For instance, support vector machine (SVM) [148] and Bayesian neural networks [149] are used to classify traffic using statistical characteristics of packet payload as application signatures. More supervised classification algorithms – *e.g.,* naive Bayes with discretization, naive Bayes with kernel density estimation, C4.5 decision tree, Bayesian network, and naive Bayes tree – are evaluated for conducting traffic classifcation [150] and intrusion detection [151]. Another type of applications whose output consists of continuous variables is called regression tasks. For instance, as depicted in figure 1.13b, Layer-4 per-flow-based networking features can be extracted to infer server load states [152], which will be described in detail in chapter 5 in part III. Anomaly detection and traffic classification tasks also attracts the application of supervised learning algorithms, in the context of both SDN [153] and cloud data centers [154]. In wireless networks, supervised learning algorithms also helps autonomously configure networking policies and optimize communication latency and resource utilization [155, 156].

Supervised learning algorithms have shown promising results in networking applications, yet they require high quality labeled datasets. However, unlike in other fields *e.g.,* computer vision [157–165] and speech recognition [166–171], few labeled datasets are available and considered



(a) Overview **(b)** Example: traffic classification (PCA)

Figure 1.14: Unsupervised learning

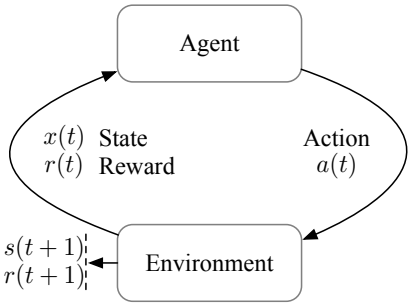
as a benchmark for ML applications in networking systems (*e.g.,* traffic analysis and anomaly detection) [172–175]. The available open-sourced datasets have been accused of being inconsistent and impractical [176, 177].

Unsupervised Learning

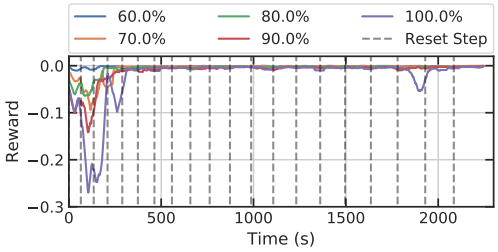
Unsupervised learning algotihms do no rely on labeled datasets and take unlabeled input data with no corresponding target values as training data (figure 1.14a). The goal of unsupervised learning algorithms is to recognize patterns in the dataset. One type of unserpervised learning tasks is to group input data with high similarity into different clusters. For instance, statistical characteristics of flow and payload can be combined to cluster network traffic and detecting unidentified traffic [148, 178]. The authors of [148] combined flow statistical feature-based clustering and payload statistical feature-based clustering for mining unidentified traffic. On top of clustering network flows into a small set of clusters – *e.g.,* using k-means or expectation maximization (EM) algorithms – efforts have been made to manually label each cluster to a network application. In [178], the authors proposed to group traffic flows into a small number of clusters using the expectation maximization (EM) algorithm and manually label each cluster to an application. However, with little information about the real applications, this does not effectively resolve the problem of clustering algorithms – mapping from clusters to real applications. Another type of unsupervised learning tasks is to project the input data from a high-dimensional space down to low-dimensional space – *e.g.,* to 2 dimensional space so as to visualize the distribution of input data as depicted in figure 1.14b [179], which will be described in detail in chapter 3 in part II. This type of tasks are called dimensionality reduction, where networking feature selections can be achieved by unserpervised learning algortihms [180]. It is also possible to infer functions that indicates the analyzed data structure and help cloud operators investigate into correlations between measurements and incidences. For instance, applications have been conducted to infer QoS in wireless networks [181] and study the causation between weather and networking device utilization [182]. Though no labeled dataset is required for unsupervised learning, the performance of an unsupervised learning model may degrade when the fitted models are brought back encountered with unseen datasets from a different distribution.

Reinforcement Learning

Reinforcement learning (RL) algorithms also do not rely on labeled datasets, yet their performance can be quantitatively assessed by rewards generated during the interactions between the learning agents and the environments [25, 183] (figure 1.15a). The learning process is based on trial-and-error with learning agents keeping interacting with the environment to explore the state space and action space with immediate reward for each action as feedback. The goal of the learning agent is to maximize the reward over a sequence of interactions with the environment, where each action not only has an impact on the immediate reward, but also influences the rewards at subsequent time steps. For instance, in the context of SDN, network operators can create a target policy for maintaining the communication latency of a given set of connections below a threshold. A RL agent can then take actions on the SDN controller and update configurations. For each configuration update (an action), the RL agent receives a reward, which indicates the performance of the previous action and guides the RL agent to bring the actual policy closer to, and finally achieve the target policy. RL has shown performance gains in various system and networking problems – *e.g.,*



(a) Overview



(b) Example: training a reinforcement-learning-based load balancer when subjected to different expected resource utilization

Figure 1.15: Reinforcement learning

task scheduling [26], resource allocation [27], congestion control [25] and load balancing [183–185]. An example of the converged training process of a RL model for network load balancer is depicted in figure 1.15b [28]. When subjected to traffic rates – thus different expected resource utilization on the server side, the convergence rates are also different. With heavier traffic, load balancing becomes more challenging, which makes the RL agent take longer to converge. This topic will be discussed further in chapter 7 in part III in this manuscript.

RL helps avoid error-prone manual configurations. However, this learning paradigm has an intrinsic trade-off to balance between exploration (learning agents try out various actions to evaluate how effective these actions are) and exploitation (learning agents take advantage of the collecte information and select actions that yield high rewards). In addition, RL algorithms have been developed and evaluated heavily relying on simulated environments [25], which does not guarantee their performance in real-world systems.

Challenges

This chapter has so far introduced 3 learning paradigms that utilize fine-grained observations of network and system states [25]. Periodically polling resource utilisation and system performance allows for obtaining timely and dedicated observations to make data-driven management decisions [30, 135, 139, 143, 186, 187]. However this incurs additional control messages and reduces system scalability, especially for large-scale distributed systems. Another way to collect a wide range of fine-grained networking features is to parse and extract from offline collected network traces or in simulated environments, which is employed for developing clustering algorithms and RL algorithms [25, 147, 188]. However, this scheme assumes a minimal gap between real-time systems and offline or simulated systems, which does not necessarily hold in networking systems [30, 183]. The data plane is constrained by low-latency and high-throughput requirements [114], which makes it challenging to apply off-the-shelf ML algorithms on networking problems.

Applying advanced ML techniques alongside the data plane on the fly is computationally intractable [8, 189]. Extracting fine-grained (sub-flow-level) observations (*e.g.,* sketches [188]) from data planes can incur additional per-packet processing overhead and memory consumption. Minimizing such performance overhead leads to either (i) heuristic algorithms in the data plane [98, 190, 191], which may not be adaptive to dynamic environments, or (ii) control planes that rely on reactive polling mechanisms, which incur additional control messages [139, 143, 145, 187]. Therefore, in real-world high-performance and large-scale networking systems, heuristics – which may not be adaptive to dynamic environments – prevail over advanced learning alrogithms [30, 65, 66, 98, 135, 186, 190–193].

1.3 Thesis Statement: Autonomous Data Centers

This chapter so far has introduced and discussed the development of data center architectures and network protocols along with the evolution of challenges when operating large-scale, elastic, high-performance data center networks. Section 1.1.1 has shown that in-production data center network topologies can have elevated complexity, intended to improve system scalability. A large variety of network protocols and frameworks to support the communication architectures have been

demonstrated in section 1.1.2, in favor of multi-tenant setups in large-scale data center networks. Section 1.1.3 has discussed the demands for high programmability in the data plane to enable flexible and "intelligent" service management in the cloud and subsequent challenges. Then, the applications of ML techniques to networking problems is discussed in section 1.2, enhancing network functions for making informed decisions. Section 1.2.1 has introduced the responsibility and limitations of network functions with 3 examples. Section 1.2.2 has described 3 main categories of ML algorithms and their applications to enable autonomous and informed decision making in different network functions.

There is a rising trend of embedding intelligence and applying ML techniques in the cloud and distributed systems to dynamically monitor and adaptively configure system parameters and characteristics (*e.g.,* server configurations, forwarding rules) [22, 25, 30, 147, 184, 189]. This can be achieved by enriching Layer-3 (with the help of data-driven algorithms), embedding feature collection mechanism and decision making process in the network stack by *transparently connecting* it to the applications. Large amount of data flows in data center networks and useful networking features can be extracted for making informed decisions to improve networking performance. Correlations can be identified between networking features and system states, which can provide insights for better operating the cloud data center networks in an autonomous manner.

The challenges and trade-offs required to collect features efficiently so as to make informed decisions in the context of different network functions in the cloud is the overall topic of this thesis. Specifically, this manuscript:

- discusses the challenges of collecting measurement and deploying data-driven networking policies in real-world (part II);
- builds generic tools to extract networking features from the data plane and deploy ML algorithms for various networking functions in real-world networking systems (part II);
- proposes and showcases a methodological framework that allows developing algorithms of different learning paradigms for networking problems (part II);
- presents a survey on network load balancing problems in data center networks (part III);
- proposes a hardware-based load balancing mechanism that achieve line-rate load-aware workload distribution by exploiting server load information embedded in packet headers as feedback signals (part III);
- investigates the challenges and potential performance gains using ML-based load balancing algorithms and propose both an open-loop and a closed-loop learning load balancing algorithms (part III).

In sum, the augmented programmable data plane allows offering autonomous data center network orchestration, management, and load-balancing with autoscaling as a network service, while using unmodified applications. As will be studied throughout this manuscript, this offers significant benefits in terms of network traffic overhead, optimization of resource utilization, quality and fairness of service, and energy reduction.

Chapter 2

Thesis Contributions

This chapter concludes this introductory part of the manuscript by summarizing the thesis contribution in section 2.1 and, by presenting a list of publications in section 2.2.

2.1 Thesis Summary and Outline

This thesis studies the usage of data-driven methods to optimize network functions – specifically load balancing – performance in data center networks. It is comprised of 4 parts and 8 chapters, structured as follows.

Part I provides an introductory discussion. In chapter 1, background on data-center architectures and associated network protocols as well as management frameworks are introduced. Then, the increasing requirements for embedding more programmability in the data plane are introduced, which gives the potential of making more informed decisions and applying data-driven networking policies. Four interesting network functions are introduced – VPN gateway, auto-scaling, load balancing, and traffic classification – to demonstrate the roles of different network functions, along with their expectations in modern data center networks. Finally, a discussion is presented about how using different types of data-driven methods can help augment data centers by providing adaptivity primitives directly at the network layer. Chapter 2 then summarizes the contributions of applying this concept throughout this thesis and describes the outline of this thesis.

Valuable insights can be obtained from networking measurements. However, multiple challenges and limitations of current in-production measurement mechanisms exist in networking systems – (i) limited available features, (ii) additional management and communication overhead thus constrained scalability, and (iii) difficulties in deploying data-driven networking policies in real-time. To tackle these challenges, Part II presents Aquarius, a **tool to collect and exploit networking features** in data centers. In chapter 3 (published in a conference and a journal [179, 194]), a framework that enables **efficient data-driven network functions** is introduced. To dynamically manage and update networking policies in cloud data centers, Virtual Network Functions (VNFs) use, and therefore actively collect, networking state information – and in the process, incur additional control signaling and management overhead, especially in larger data centers. To avoid intractable additional processing latency under high-performance and low-latency networking constraints, VNFs in production adopt distributed and straightforward heuristics instead of advanced learning algorithms. Chapter 3 identifies the challenges of deploying learning algorithms in the context of cloud data centers, and proposes Aquarius to bridge the application of machine learning (ML) techniques on distributed systems and service management. Aquarius passively gathers reliable observations without introducing noticeable overhead, and enables the use of ML techniques to collect, infer, and supply accurate networking state information — without incurring additional signaling and management overhead. It offers fine-grained and improved visibility of a flexible and configurable set of networking features to distributed VNFs, and enables both open- and close-loop control over networking systems. Chapter 3 illustrates the use of Aquarius with a traffic classifier, an auto-scaling system, and a load balancer -– and demonstrates the use of three different ML paradigms — unsupervised, supervised, and reinforcement learning, within Aquarius, for network state inference and service management. Testbed evaluations show that Aquarius suitably improves network state visibility and brings notable performance gains for various scenarios

with low overhead.

Part III studies the **network load balancing** problem in data center networks. Load-Balancers play an important role in data centers as they distribute network flows across application servers and guarantee per-connection consistency. It is hard, however, to make fair load balancing decisions so that all resources are efficiently occupied yet not overloaded.

In chapter 4 (published in a workshop [195]), a **NetFPGA-based load balancing mechanism** is proposed. Tracking connection states allows load balancers to infer server load states and make informed decisions, but at the cost of additional memory space consumption. This makes it hard to implement on programmable hardware, which has constrained memory but offers line-rate performance. Specifically, Charon is designed, a stateless load-aware load balancer that has line-rate performance implemented in P4-NetFPGA. Charon passively collects load states from application servers and employs the power-of-2-choices scheme to make load-aware load-balancing decisions and improve resource utilization. Per-connection consistency is guaranteed statelessly by encoding the server ID in a covert channel. The prototype design and implementation details are described in this chapter. Simulation results show performance gains in terms of load distribution fairness, QoS, throughput, and processing latency.

Chapter 5 (published in a workshop [152]) **applies ML algorithms to optimize network load balancers**. Workload distribution algorithms are based on heuristics, *e.g.,* Equal-Cost Multi-Path (ECMP), Weighted-Cost Multi-Path (WCMP) or naive ML algorithms, *e.g.,* ridge regression. Advanced ML-based approaches help achieve performance gains in different networking and system problems. However, it is challenging to apply ML algorithms on networking problems in real-life systems: it requires domain knowledge to collect features from low-latency, high-throughput, and scalable networking systems, which are dynamic and heterogenous. This chapter conducts both offline data analysis and model training, and online model deployment in real-world systems based on Aquarius. The results show that the ML models improve load balancing performance yet they also reveals more challenges to be resolved to apply ML for networking systems, including the lack of generalization in dynamic environments.

In chapter 6 (published in a journal [196]), an **open-loop load balancing algorithm** based on Kalman filter is proposed, to achieve load-aware workload distribution with increased generalization. This chapter proposes a distributed, application-agnostic, Hybrid Load Balancer (HLB) that – without explicit monitoring or signaling – infers server occupancies and processing speeds, which allows making optimized workload placement decisions. This approach is evaluated both through simulations and extensive experiments, including synthetic workloads and Wikipedia replays on a real-world testbed. This work is, again, based on Aquarius. Results show significant performance gains, in terms of both response time and system utilization, when compared to existing load-balancing algorithms. With only passively observed networking features, HLB can achieve similar performance as the shortest expected delay (SED) algorithm, which requires manual configurations to make load balancers aware of server processing capacities.

In chapter 7 (published in a workshop and in 2 conferences [28, 183, 185]), a **closed-loop load balancing algorithm** based on reinforcement learning (RL) is proposed. As central components in data centers, load balancers operate in dynamic environments with limited monitoring of application server loads and provide scalable services. Though HLB can achieve similar performances with no prior knowledge about the system configuration, the state-of-the-art load balancers still rely on heuristic algorithms that require manual configurations for fairness and performance. To alleviate that, a distributed, asynchronous, reinforcement learning mechanism is proposed to – with no active load balancer state monitoring and limited network observations – improve the fairness of the workload distribution achieved by a load balancer. Since multiple load balancers are deployed in data centers to avoid single point of failure, the multi-agent reinforcement learning (MARL) framework is used. The challenges of this problem consist of the heterogeneous processing architecture and dynamic environments, as well as limited and partial observability of each LB agent in distributed networking systems, which can largely degrade the performance of in-production load balancing algorithms in real-world setups. Centralised training and distributed execution (CTDE) RL scheme has been proposed to improve MARL performance, yet it incurs – especially in distributed networking systems, which prefer distributed and plug-and-play design schemes – additional communication and management overhead among agents. We formulate the multi-agent load balancing problem as a Markov potential game, with a carefully and properly designed workload distribution fairness as the potential function. A fully distributed MARL algorithm is proposed to approximate the Nash equilibrium of the game. Experimental evaluations

involve both an event-driven simulator and a real-world system, where the proposed MARL load balancing algorithm shows close-to-optimal performance in simulations and superior results over in-production LBs in the real-world system.

Finally, part IV concludes this manuscript.

2.2 List of Publications

The following publications have been published, during the course of this Ph.D.

Journal Publications

- Zhiyuan Yao, Yoann Desmouceaux, Juan-Antonio Cordero-Fuertes, Mark Townsley, Thomas Clausen. *Aquarius-Enable Fast, Scalable, Data-Driven Service Management in the Cloud*, IEEE Transactions on Network and Service Management, August 2022 (chapter 3).
- Zhiyuan Yao, Yoann Desmouceaux, Juan-Antonio Cordero-Fuertes, Mark Townsley, Thomas Clausen. *HLB: Toward Load-Aware Load Balancing*, IEEE/ACM Transactions on Networking, June 2022 (chapter 6).

Conference or Workshop Publications

- Zhiyuan Yao, Zihan Ding. *Learning Distributed and Fair Policies for Network Load Balancing as Markov Potentia Game*, Proc. 36th Conference on Neural Information Processing Systems (NeurIPS'22), November 2022 (chapter 7).
- Zhiyuan Yao, Yoann Desmouceaux, Juan-Antonio Cordero-Fuertes, Mark Townsley, Thomas Heide Clausen. *Efficient Data-Driven Network Functions*, Proc. 30th International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'22 **Best Paper Award Recipient**), October 2022 (chapter 3)
- Zhiyuan Yao, Zihan Ding, Thomas Clausen. *Multi-agent reinforcement learning for network load balancing in data center*, Proc. 31st ACM International Conference on Information and Knowledge Management (CIKM'22), October 2022 (chapter 7).
- Zhiyuan Yao, Yoann Desmouceaux, Mark Townsley, Thomas Heide Clausen. *Towards Intelligent Load Balancing in Data Centers*, 5th Workshop on Machine Learning for Systems at 35th Conference on Neural Information Processing Systems (NeurIPS'21), December 2021 (chapter 5)
- Zhiyuan Yao, Zihan Ding, Thomas Heide Clausen. *Reinforced Workload Distribution Fairness*, 5th Workshop on Machine Learning for Systems at 35th Conference on Neural Information Processing Systems (NeurIPS'21), December 2021 (chapter 7)
- Carmine Rizzi, Zhiyuan Yao, Yoann Desmouceaux, Mark Townsley, Thomas Clausen. *Charon: Load-Aware Load-Balancing in P4*, Proc. 1st Joint International Workshop on Network Programmability and Automation (NetPA) at 17th International Conference on Network and Service Management (CNSM'21), October 2021 (chapter 4).

Data Availability

All the data and scripts necessary to reproduce the graphs included in this thesis can be publicly accessed, and can be reused by anyone interested in doing so[1].

[1] https://github.com/zhiyuanyaoj/phd-thesis-data

Part II

Data-Driven Network Functions

Chapter 3

Aquarius - Enabling Fast, Scalable, Data-Driven Service Management in the Cloud

Demands for responsive, high-available, low-latency cloud services require content providers and cloud operators to efficiently manage cloud data centers (DCs) [1, 69]. To increase network programmability, and balance the trade-off between capital expenditures and quality of service (QoS), Virtual Network Functions (VNFs) (*e.g.,* firewalls, load balancers, and VPN gateways [14, 63]) are deployed in cloud DCs to provide flexible and reliable service management and transparent operations. Running on commodity computing platforms, VNFs replace or augment dedicated hardware devices, and play a significant role in large-scale DCs. To dynamically monitor and configure VNFs, software-defined networking (SDN) can be deployed, dissociating the routing and decision-making process (*control plane*) from the network packets forwarding process (*data plane*) [5, 107].

The control plane adaptively manages and updates networking policies in dynamic cloud DC environments to offer high service availability and QoS. Data-driven mechanisms based on machine learning (ML) [27, 146] and reinforcement learning (RL) algorithms [26, 28] can be applied, and show performance gains in various network applications. For instance, auto-scaling systems and load balancers can achieve improved QoS with reduced cost based on periodically polled resource utilization of distributed network devices (*e.g.,* application servers) [139, 143]. Traffic classification and anomaly detection help detect security threats with increased accuracy based on network traffic characteristics extracted from offline-collected network traces [17, 147]. However, it is challenging to harness these algorithms to drive management decisions in networking systems in real-time, for multiple reasons, detailed in the following.

ML and RL algorithms require fine-grained observations of network and system states [25]. Periodically polling resource utilization and system performance allows for obtaining timely and dedicated observations to make data-driven management decisions [30, 135, 139, 143, 186, 187]. However, this incurs additional control messages and reduces system scalability, especially for large-scale distributed systems. Another way to gather a wide range of fine-grained networking features is to offline parse collected network traces or in simulated environments and extract traffic characteristics, which then are employed for developing clustering algorithms and RL algorithms [25, 147, 188]. However, this scheme assumes a minimal gap between real-time systems and offline or simulated systems, which does not necessarily hold in networking systems [30, 183].

The data plane is constrained by low-latency and high-throughput requirements [114], which makes it challenging to apply off-the-shelf ML algorithms on networking problems. Applying advanced ML techniques on-line in the data plane is computationally intractable [8, 197]. Therefore, in real-world high-performance and large-scale networking systems, heuristics – which may not be adaptive to dynamic environments – prevail over advanced learning algorithms in this environment [30, 65, 66, 98, 135, 186, 190–193].

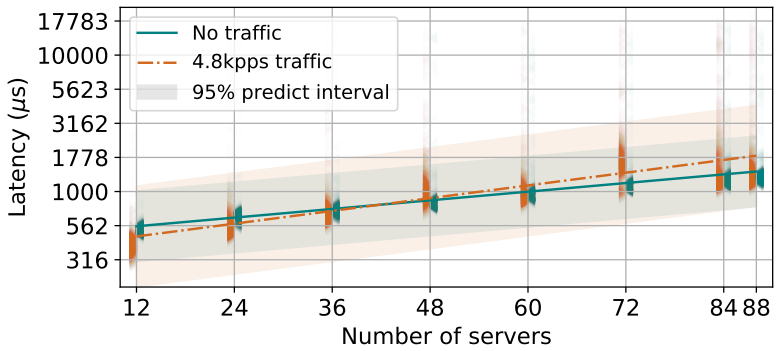


Figure 3.1: Linear regression on probing latencies (with and without background network traffic) and additional response time collected on clusters of different numbers of servers.

Statement of Purpose

This chapter proposes Aquarius, a fast and scalable data collection and exploitation mechanism that bridges different requirements for data planes (low-latency and high-throughput) and control planes (making informed decisions). It enables learning algorithms to make inferences and open- or closed-loop control decisions based on fine-grained observations, and it allows the deployment of distributed and intelligent VNFs, which harness ML algorithms to make data-driven operational decisions. This chapter makes the following contributions.

First, this chapter identifies the challenges of gathering networking features to make valuable inferences and informed operational decisions in high-performance and large-scale cloud DCs Experimental evaluations demonstrate that traditional mechanisms for feature collection (*e.g.,* active probing [143, 187, 198–202] and trace capture [172–177]) cause substantial overhead.

Second, this chapter proposes a fast and configurable mechanism – *Aquarius* – that allows collecting a wide range of fine-grained networking features in a scalable layout, which is suitable for applying various ML techniques to networking problems Aquarius embeds programmable and flexible feature collection state machines in the data plane. These state machines are used to extract user-defined networking features. To efficiently gather observations, Aquarius collects 2 types of features – *counters* and *samples* – using *multi-buffering* [203] and *reservoir sampling* [204], respectively. Networking features are grouped separately corresponding to different network applications and types of equipment (*e.g.,* links, servers). Features are made available in a scalable layout, which offers high flexibility when aggregating and processing data under various requirements (*e.g.,* by single equipment or by groups of equipment).

Third, this chapter provides an extensive performance and overhead evaluation of Aquarius, using experiments in a realistic testbed Within the context of (i) an unsupervised-ML-powered network traffic classifier, (ii) a supervised-ML-powered auto-scaling system, and (iii) an RL-powered Layer-4 load balancer, this chapter shows that the collected features enable:

- **unsupervised learning + *offline* data analysis**: creating benchmark datasets to gain insight into different networking problems with minimal data collection overhead;
- **supervised learning + VNF *management***: embedding ML techniques to achieve self-aware monitoring and self-adaptive orchestration in an elastic compute cloud;
- **reinforcement learning + *online* policy updates**: enabling closed-loop control, where collected networking features help optimise routing policies and improve QoS.

Chapter Outline

The remainder of this chapter is organized as follows. Section 3.1 describes the challenges of harnessing data-driven algorithms for networking problems and compares this chapter with

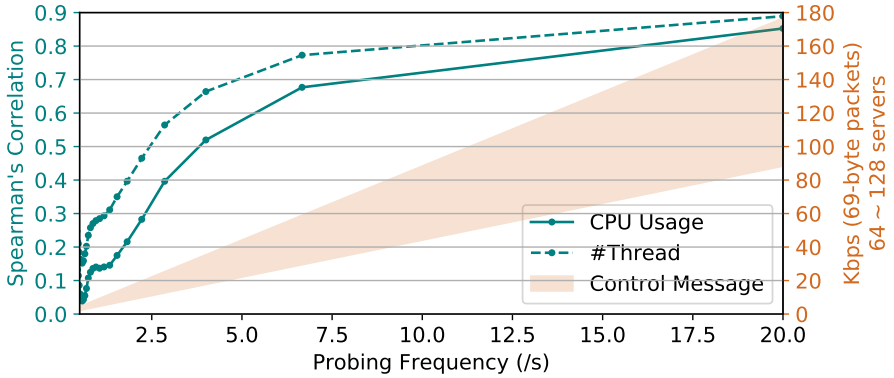


Figure 3.2: Correlation (Spearman) increases when the probing frequency grows, yet, so do additional control messages.

related work. Section 3.2 presents both the rationale and the design of the feature collection and exploitation mechanism of Aquarius. Section 3.3 demonstrates the use of Aquarius in the context of 3 different VNFs on a realistic testbed. Section 3.4 summarizes the results of this chapter.

3.1 Background

This section presents the challenges of efficient feature collection and data-driven VNFs in cloud DCs, and, with a comparison of related work, motivates the design of Aquarius.

Efforts have been made to embed intelligence and apply ML techniques in the cloud, and distributed systems, to dynamically monitor, and adaptively configure, system parameters and characteristics (*e.g.,* server configurations, forwarding rules) [22,25,30,147,184,189]. However, this raises many challenges and trade-offs that require to be handled, to efficiently collect features and make data-driven decisions. These challegnes are summarized in the following.

Online Feature Collection Datasets with high quality are essential to ML studies. However, few datasets are available as a benchmark for ML applications in networking systems (*e.g.,* traffic analysis and anomaly detection) [172–177]. To collect a wide range of features (*e.g.,* traffic rates, packet sizes, TCP congestion window sizes), these datasets are collected based on logged network traces (*e.g.,* by TCPdump). Though log-based feature collection provides abundant information for various types of applications, it does not scale in terms of log file size [205]. Log-based feature collection also incurs performance overhead when subjected to heavy traffic, which leads to inaccurate and irrelevant measurements and makes it hard to bring ML algorithms "online" (*i.e.,* making inference and management decisions in real-time) [189].

Scalability vs. Visibility Active probing is another feature collection approach to monitor the system state and make informed decisions [143,187,198–202]. However, this requires modifications on each networking device to maintain management and communication channels. There is also a trade-off between the communications overhead via probing channels, and the visibility of the state of VNFs. As depicted in figure 3.1, when a controller VM periodically (every 50ms) probes a cluster of servers[1] via TCP sockets, the latency overhead increases with the number of servers, which diminishes the QoS. As depicted in figure 3.2, the visibility of the state of VNFs, unsurprisingly, correlates with the probing frequency. Additional management traffic can exceed the 90-th percentile of per-destination-rack flow rate (100kbps) in production [1].

Flexibility vs. Performance Developing, prototyping, and benchmarking ML applications on different networking problems is hard in high performance networks because of the low-latency and high-throughput expectations in the data plane. A general data processing framework has been proposed [8] to accelerate data-driven network functions on reconfigurable hardware, which provides line-rate performance. However, in dynamic and elastic networking environments, where

[1] In the 69-byte control packet emitted by the server, the 24-byte payload consists of the server ID, CPU and memory usage, and the number of busy application threads.

Property	[198, 199]	[109] [110]	[209–211]	[29, 139] [137, 140]	[25, 135, 143] [27, 186, 187]	[26, 152] [212]	[22]	[200] [201]	[19] [213]	[8]	*Aquarius*
No Control Message	✗	✗	✗	✗	✗	✓	✓	✗	✓	✓	✓
Distributed	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
Commodity Device	✓	✗	✓	✓	✓	✓	✓	✓	✗	✗	✓
Use Case	Generic	Management Protocol		Autoscaling	Traffic Optimisation		Traffic Classification			Generic	Generic

Table 3.1: Comparison of data-driven VNF systems.

additions and removals of networking devices and services happen frequently [190], hardware devices are scalable in terms of performance (*e.g.,* throughput) but not in terms of network topology (*e.g.,* number of services and networking devices). Hardware programming and verification procedures can also be difficult and time-consuming [206], which thus relies more on simulations for interdisciplinary research [25, 28, 207, 208]. Yet the flexibility offered by simulations hinders the real-world deployment of ML algorithms because simulators fail to capture the complexity of high performance networking systems [30].

3.1.1 Requirements

The challenges give rise to the following requirements for the forwarding plane enabling data-driven VNFs in the cloud:

Universality – the feature collection mechanism should cover a wide range of features and be application-agnostic (section 3.3.1);

Relevance – the collected features should be representative, providing useful information to address real-world applications in different circumstances (section 3.3.3);

Scalability – the feature collection and exploitation mechanism should incur minimal performance overhead and support large-scale and dynamically changing network topology (section 3.3.2);

Flexibility – the mechanism should be configurable and easy to be tailored for various use cases and learning algorithms (section 3.2.1);

Deployability – the mechanism should be plug-and-play and require no additional installation or configuration (section 3.2.1).

3.1.2 Related Work

Various mechanisms (summarised in table 3.1) dynamically configure and manage VNFs, making data-driven decisions.

ML benefits various networking applications and network functions, *e.g.,* congestion control [23, 24], intrusion detection systems [21, 22], traffic classification [18, 214], and traffic optimization by way of task scheduling [26, 27]. It allows inferring system states from networking features. To obtain networking features, these ML applications operate at the Application Layer. However, they are not application-agnostic and do not generalize to different use cases. Acting as proxies, they also terminate networking connections, increasing processing latency [215, 216]. Aquarius collects a wide range of features at the Transport Layer and enables generic data-driven network functions with minimal overhead.

Management and Orchestration (MANO) frameworks [198, 199] and autoscaling systems [139, 140] use centralized controllers to monitor and update VNF and topology configurations. Based on active monitoring, MANO and autoscaling systems help provision computing, storage, and networking resources. Software-Defined Network (SDN) provides programmable APIs to gather per-flow or application-level features in a centralized way, to adaptively update configurations, using network equipment that supports the OpenFlow protocol [109, 110]. Other management protocols [209–211] collect and send data from network equipment to a centralized controller via

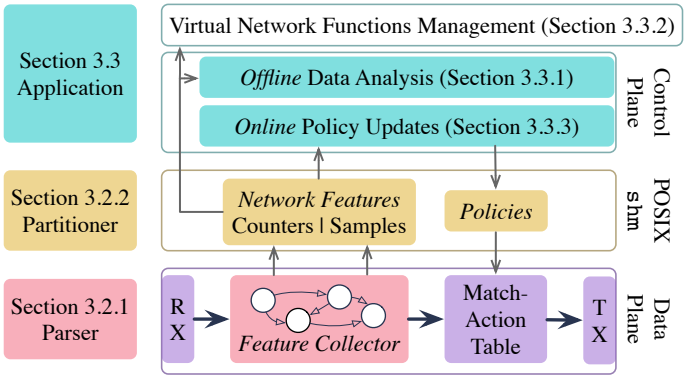


Figure 3.3: Aquarius architecture overview.

active probing, but with high communication overhead. Aquarius passively extracts networking features from the data plane and lets VNFs make decisions in a distributed and scalable way.

Distributed VNFs also use periodically polled network states (*e.g.,* packet arrival rates, CPU and memory usage), to ensure service availability, and improve QoS [143, 187], or classify networking traffic [200, 201]. Additional control messages and communication latency limit the system scalability [17, 202]. In [186], controllers are notified of the occurrence of malfunctioning network devices to avoid periodic probing. Some network functions gain more visibility via in-network telemetry (INT) [212] and covert-channels [217]. However, these require either deploying agents or modifying the protocol stack on network devices, which limit their deployability on generic hardware or systems. Aquarius is designed to employ plug-and-play design, incurring no additional modifications in the network stacks.

Learning algorithms incur additional inference and processing latencies. To reduce latency, dedicated hardware, *e.g.,* NPU [218], and NetFPGA [19], help improve data processing efficiency for in-network ML applications [219]. Taurus [8] enables in-network distributed data plane intelligence using a map-reduce abstraction for generic ML algorithms on a coarse-grained reconfigurable array (CGRA) [220]. These hardware solutions boost performance, yet they lack flexibility when developing ML algorithms for different use cases in elastic networking systems. MVFST-RL [25] proposes to asynchronously update networking configurations to benefit from learning algorithms without inducing additional latency in the data plane. However, performance gains are shown only in simulators with a single use case. Aquarius is designed to incorporate intelligence in a variety of VNFs, requiring no dedicated device, yet it is ready to be deployed in real-world systems.

3.2 Design

To meet the 5 requirements, summarised in section 3.1.1, Aquarius is designed as a 3-layer architecture (figure 3.3). Aquarius embeds a feature collector at the Transport Layer in the data plane (*deployability*), to efficiently and passively extract a wide range of features (*universality*) with high quality (*relevance*), and low latency and performance overhead. It makes the features available and easily accessible via shared memory (*scalability*), for applications of ML algorithms on various use cases in the control plane (*flexibility*).

3.2.1 Parser Layer

To balance the tradeoff between scalability and visibility, networking features which indicate system states can be passively collected from the data plane to avoid active probing and additional installations and configurations. However, located within network function chains, VNFs in modern DCs may observe only 1-way traffic (*i.e.,* half of the traffic, in only one direction of each flow) addressing to their egress equipment (*e.g.,* links and servers), to reduce additional processing latency [64]. This requires: (i) *careful design of feature collection mechanisms to offer high scalability and configurability*, and (ii) *domain knowledge to extract valuable and representative networking features and reason their correlations with system states.* This chapter illustrates the design using

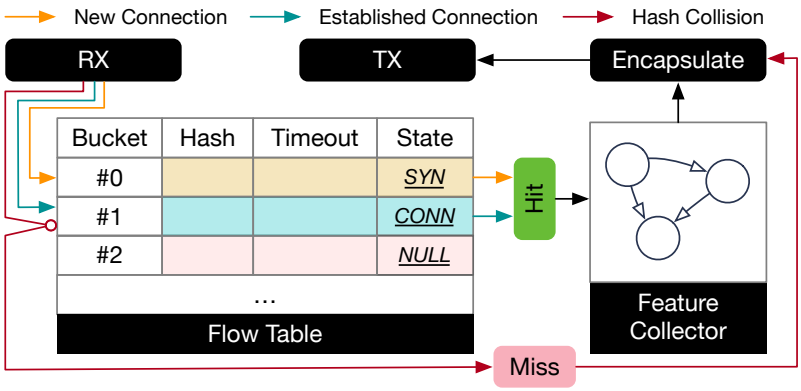


Figure 3.4: Flow table data structure and workflow.

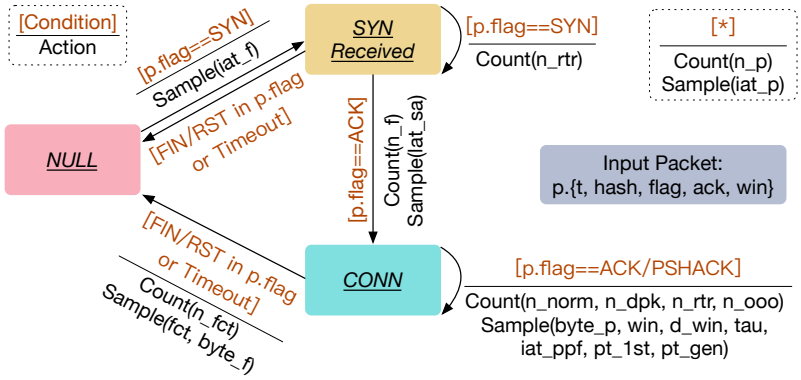


Figure 3.5: A state machine of feature collector for TCP traffic.

TCP traffic, which is a widely used protocol [1, 82, 221]. The same workflow, however, also applies to other network traffic (*e.g.,* UDP).

Stateful Feature Collection

Network traffic consists of flows that traverse different nodes (*e.g.,* edge routers, load balancers, servers) in the system, whose states can be traced and retrieved from the flows – along with traffic characteristics.

Stateless feature collection mechanisms – *e.g.,* sketching [188, 222], a family of streaming algorithms for networking measurement summarisations – do not track the state of network flows, yet they can gather counters as ordinal features for ML algorithms using hashing functions, with little performance overhead. However, ordinal features (counters) contain less information than quantitative features – time-related features (*e.g.,* round-trip time, inter-arrival time, flow duration) and throughput information (*e.g.,* congestion window size, flow size) – all of which are not captured by stateless mechanisms.

Aquarius tracks flow states in bucket entries with a stateful table (figure 3.4), which can be configured to collect a wide range of features using a state machine depicted in figure 3.5. In the flow table, Aquarius stores the information of each flow into a bucket entry indexed by the hash of the flow id modulo the size of the flow table size – $hash(fid) mod M$, where fid is the flow ID[2] and M is the flow table size. An entry in the flow table can be in one of three states – `SYN`, `CONN` and `NULL` (figure 3.5). When a new flow arrives (TCP `SYN`), it is registered in a bucket entry of the flow table with its fid and state (`SYN`). On receipt of its subsequent packets, the state in the entry is retrieved and updated to `CONN` (connected) if the flow starts transmitting data[3]. On receipt of packets which terminate TCP flows (or timeout for UDP flows), the flow is evicted and the

[2]TCP network flows are identified by their 5-tuples: protocol number, source and destination IP addresses and port numbers.

[3]For a TCP flow, if it is well-established (*e.g.,* after 3-way handshakes), and the first data packet is received, its state will be updated to `CONN`.

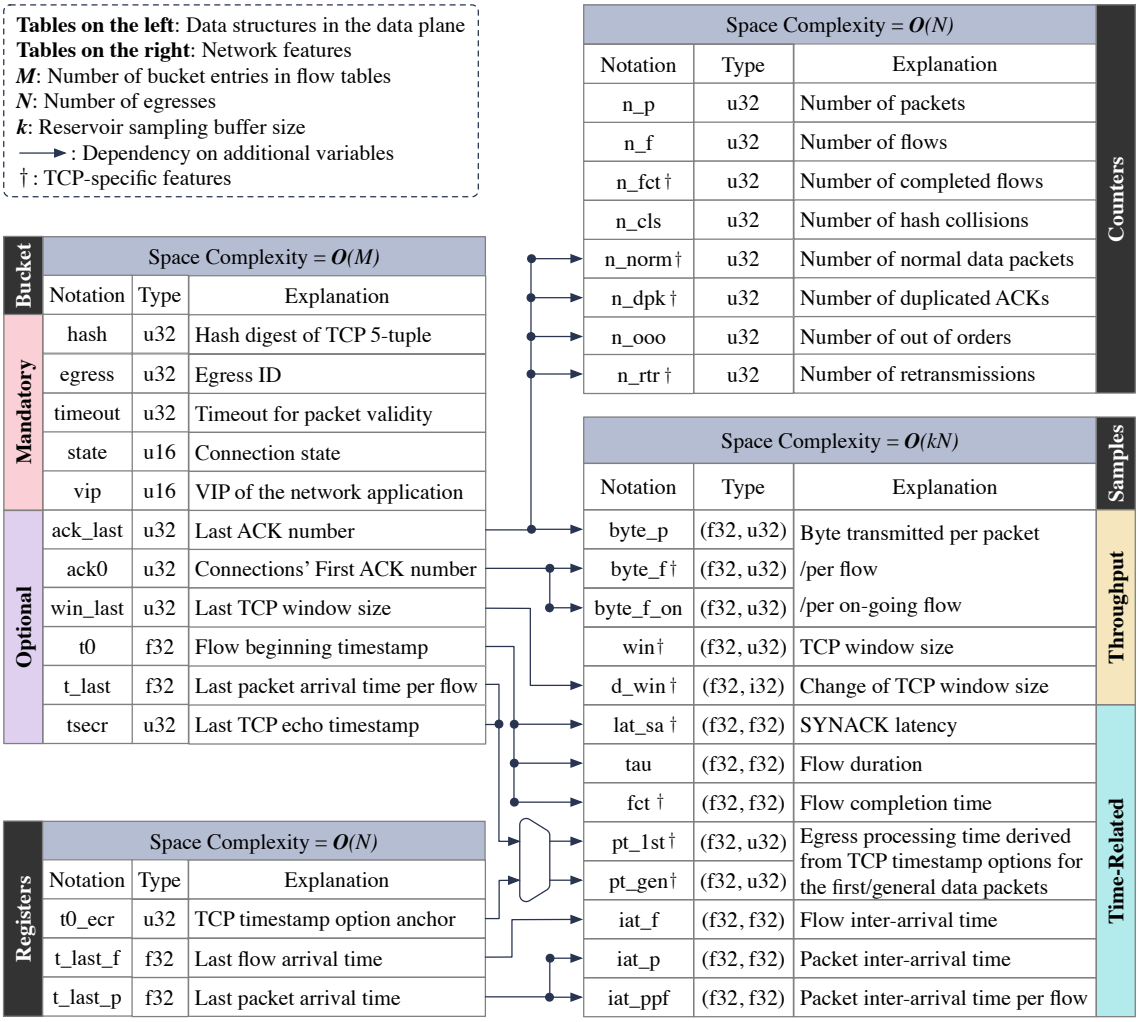


Figure 3.6: Notations, categories, variable dependencies, and space complexity of all network features.

entry state returns to `NULL`, so that the bucket entry is available for new flows. In the case where the bucket entry is not available when a new flow appears, the flow is considered a "miss" and is excluded by the feature collector.

By passively extracting networking features from flow states, Aquarius does not require additional configurations and improves the *deployability*.

Network Features

Various features can gainfully benefit the decision-making process for different use cases. Counting the number of ongoing flows helps track instant server load states, thus helping balance workloads distribution (section 3.3.3). Throughput information helps classify whether the network traffic is IO-intensive (section 3.3.1). Time-related features help understand the QoS on servers, thus helping predict resource utilization and schedule scaling events (section 3.3.2).

As a generic feature collection mechanism, Aquarius should be able to collect as much information as possible with minimal overhead (*e.g.,* memory space consumption). With the flow table, Aquarius allows *flexible* configuration of attributes, to gather the most significant features and optimize the memory usage overhead for different applications. Figure 3.6 lists all configurable features that are implemented in this chapter[4].

Ordinal features are collected as counters, which are incremented either as integer variables or using sketches. For simplicity, this chapter uses accumulative integer variables, *e.g.,* when a flow

[4] All features depend on the state and timeout attributes in the flow table, thus these dependencies are omitted for clarity. More attributes can be potentially added to obtain more features, *e.g.,* to track packet TTL.

Algorithm 1 Reservoir sampling with no rejection

```
k ← reservoir buffer size
buf ← [(0,0),...,(0,0)]   ▷ Size of k
for each observed sample v arriving at t do
    randomId ← rand()
    idx ← randomId%N   ▷ randomly select one index
    buf[idx] ← (t, v)   ▷ register sample in buffer
```

state transits from `SYN` to `CONN`, the total number of received flows `n_f` is incremented[5]. Counters for general network protocols include:

1. *Number of packets and flows* (`n_p`, `n_f`), which quantifies the volume of network traffic addressed to each egress equipment.

2. *Number of hash collisions* (`n_cls`), which evaluates the amount of untracked flows and can be used to estimate the coverage of collected features.

3. *Number of out-of-ordered packets (`n_ooo`)*, which indicates the multiple path existence in networks, where the packet ordering is not necessarily preserved.

For TCP traffic, additional counters can be gathered:

1. *Number of completed flows* (`n_fct`), which is incremented when a flow terminates. The number of ongoing flows (a canonical feature) is derived as `#flows` = `n_f` − `n_fct`, to estimate instant queue lengths of ongoing tasks on the server side.

2. *Number of duplicated ACK packets, retransmissions* (`n_dpk`, `n_rtr`), which can be used for diagnostics, reflecting *e.g.,* the level of congestion on links.

Quantitative features are collected as samples, using reservoir sampling [204] (algorithm 1), which is a statistical mechanism that helps gather a representative group of samples, in fix-sized buffers from a stream, with minimized computational overhead and memory usage for high-performance data planes [114, 121]. To capture the system dynamic, besides feature values, it is also important to trace the timestamps of different events, *e.g.,* for sequential ML algorithms. Reservoir sampling collects an exponentially-distributed number of samples over time and gives more importance to "fresher" observations. For a Poisson stream of events with rate λ, the expectation of the amount of samples that are preserved in buffer after n steps is $E = \lambda\left(\frac{k-1}{k}\right)^{\lambda n}$, where k is the size of reservoir buffer. Based on the characteristics of different system dynamics, *e.g.,* long-term distribution shifts or short-term oscillations, the reservoir sampling mechanism can to be tuned (*e.g.,* number of buffers) to collect representative statistical distributions of the states over time, since both the sampling timestamps and exponentially-distributed numbers of samples are captured over a time window. Periodic queries to the reservoir sampling buffers can generate generic time-series data, suitable for sequential pattern analysis.

For general network flows, the following features can be sampled in reservoir buffers:

1. *Bytes transmitted per packet (`byte_p`) and bytes transmitted per flow (`byte_f_on`)*, which help estimate overall IO occupation in the networks. Bytes transmitted per flow keep increasing as more data packets are received, until the flow ends or times out.

2. *Flow and packet inter-arrival times* (`iat_f`, `iat_p`), which reflect the arrival rates of flows thus the burst of network requests. The values of `iat_f` are updated when new flows arrive while `iat_p` requires no stateful flow tracking.

3. *Flow duration* (`tau`), which contributes to characterizing the type of network traffic, *e.g.,* long-lived flows or short queries. This feature is updated on receipt of each data packet of the flow.

For TCP traffic, additional features can be collected:

1. *Congestion window size* (`win`, `d_win`), which embed the congestion states of networking systems. Their values are updated on receipt of new ACK packets, after 3-way handshakes.

[5]The counter `n_fct` is incremented only if one flow ends with a previous flow state as `CONN`. A similar DDoS mitigation mechanism based on flow tables is proposed in Prism [221], but it is out of the scope of this chapter.

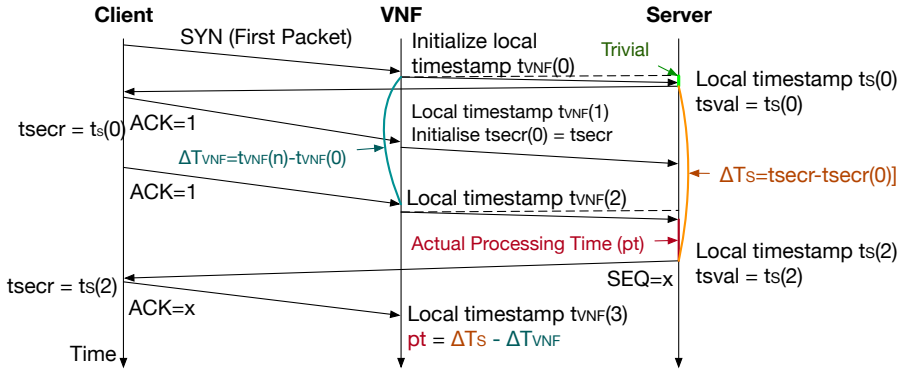


Figure 3.7: Calculation of egress (*e.g.,* application server) processing time with TCP timestamp options.

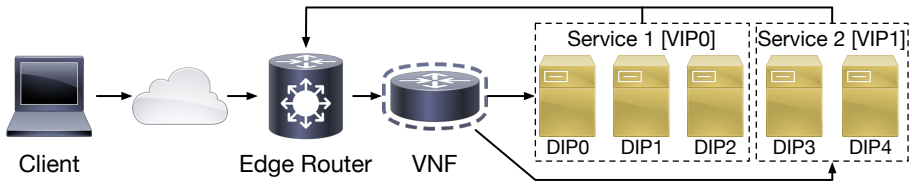


Figure 3.8: Cloud service topology.

2. *Flow completion time* (`fct` and *flow byte size* (`byte_f`), which are collected when flows terminate. Their values indicate the characteristics of the managed services.

3. *`SYN` to first `ACK` latency* (`lat_sa`), which estimates the 3-way handshakes latency for TCP traffic. When a `SYN` packet is received on one host, `SYN` cookies statelessly generate a immediate `SYNACK` response. Their values help estimate and calibrate the baseline RTT between two end hosts of the flows.

4. *Data packet processing time* (`pt_1st` and `pt_gen`), which can be derived from TCP timestamp options `tsecr`. Intuitively, the time difference between the reception and the response of a data packet indicates the processing time and resource usage on the egress network equipment. However, given the constraint of observing only 1-way traffic on VNFs (*e.g.,* DSR mode for layer-4 LBs), this information is hard to obtain. Using the TCP option fields, the timestamp of the egress equipment's response (`tsval`) is recovered from the `ACK` packets sent by the client (`tsecr`). The procedure of rebuilding the processing time on the egress side is illustrated in figure 3.7. With respect to Web applications, the processing time is further distinguished by the first data packet (`pt_1st`) and the subsequent ones (`pt_gen`).

In-Network Telemetry (INT) features can also be collected as samples and stored in reservoir buffers [191, 223]. For simplicity, these features are omitted in this chapter.

3.2.2 Partitioner Layer

As introduced in section 1.1, cloud service is identified by a virtual IP address (VIP) (figure 3.8), which corresponds to clusters of provisioned resources – *e.g.,* servers, each identified by a unique direct IP (DIP). In production, cloud DCs are subject to high traffic rates and their environments and topologies change dynamically. This requires to *organize collected features in a generic, yet scalable format*, and *make features available for ML algorithms without disrupting the data plane.*

Features describing different cloud services should be separated to (i) avoid multimodal distributions in collected features and (ii) allow dynamically adding or removing services. Based on use cases, features collected from a given service should be further partitioned – by ingress or egress equipment, *e.g.,* links or servers – to have finer granularity for learning algorithms. Even under heavy traffic and high access rates, features should be reliable and easy to access.

Aquarius organizes observations of each VIP in independent POSIX shared memory (`shm`) files, to provide scalable and dynamic service management. In each `shm` file, collected features are further

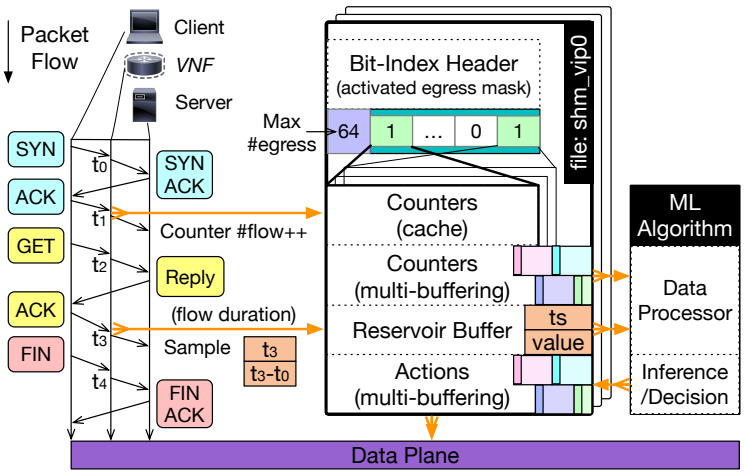


Figure 3.9: Aquarius `shm` layout and data flow pipeline.

Operation / Complexity		Computation	Memory
Add / Remove VIP		$\mathcal{O}(1)$	$\mathcal{O}(kN+mN)$
Add egress node		$\mathcal{O}(1)$	$\mathcal{O}(k+m)$
Remove egress node		$\mathcal{O}(1)$	$\mathcal{O}(1)$
Register reservoir sample Update counter (cache)		$\mathcal{O}(1)$	$\mathcal{O}(1)$
Update counters / actions (multi-buffering)		$\mathcal{O}(1)$	$\mathcal{O}(N)$
Get the latest observation	1 node	$\mathcal{O}(m)$	$\mathcal{O}(k+m)$
	All nodes		$\mathcal{O}(kN+mN)$
Update action in the data plane	1 node	$\mathcal{O}(m)$	$\mathcal{O}(1)$
	All nodes		$\mathcal{O}(N)$

Table 3.2: Computation and memory complexity of different operations, where k is the size of reservoir buffer, N is the number of egress nodes, and m is the level of multi-buffering.

partitioned by egress equipment[6]. Figure 3.10 exemplifies the `shm` layout and workflow (detailed version in figure 3.9).

Bit-Index and Masking

The first byte in the `shm` file of a VIP defines the max number of egress equipment N, which determines the number of "columns" to be reserved for feature collection. It is determined *a priori* by the scale of the cloud service, so that N is large enough to cover all the networking devices that are required in all circumstances. The N-bit *bit-index header* helps quickly identify activated egress and its corresponding "column" – the i-th bit is set to 1 if the i-th egress is active and 0 otherwise. With minimal memory space, this design informs ML algorithms to skip features of inactive equipment, gather features (*e.g.,* also in separated `shm` files) and update policies only for active equipment, reducing processing latency.

Independent Egress Memory Space

Each egress node has its own independent memory space, storing counters, reservoir samples, and data plane policies (actions). As depicted in figure 3.10, on receipt of the first `ACK` from the client to a specific egress node i, VNF increments the number of flows in the counters cache of node i. Quantitative features (*e.g.,* flow duration $t_3 - t_0$ gathered at t_3 in figure 3.10) can be stored in the reservoir buffer of node i using algorithm 1. Gathered features (counters and samples) are made available in a layout where they can be quickly accessed by ML algorithms running in a different process. With the bit-index header, locating features for a given egress node requires

[6]Depending on different applications, observations for each VIP can also be organized in different ways, *e.g.,* by ingress ports.

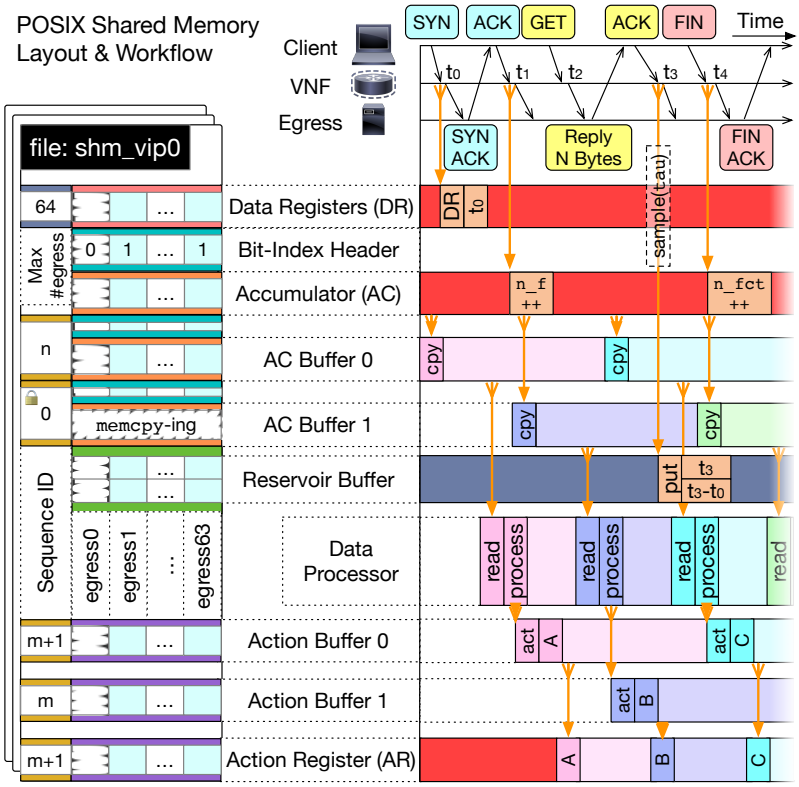


Figure 3.10: Detailed Aquarius `shm` layout and data flow pipeline.

$\mathcal{O}(1)$ computational complexity and $\mathcal{O}(k)$ memory complexity, where k is the reservoir buffer size. Obtained features for all active egress nodes can then be aggregated and processed to make further inferences or data-driven decisions, which can be written back to the memory space of each egress node.

Multi-Buffering and Asynchronous I/O

While quantitative features are collected using reservoir sampling, counters are directly incremented by the data plane in a local cache, and then periodically copied from cache to shared memory buffers with incremental sequence ID. Counters and actions are exchanged between cache and buffer using m-level multi-buffering with incremental sequence ID. The bit-index binary header is copied with the counters, to efficiently identify active equipment. When copying data between cache and buffer, the sequence ID is set to 0 to avoid I/O conflicts. ML algorithms can pull the latest observations from the buffers with no disruption in the data plane. Similarly, new network policies (*e.g.,* forwarding rules) can be updated via action buffers. The level of multi-buffering in this chapter is set to $m = 3$ (as in figure 3.10). This design offers an asynchronous 2-way communication interface to exchange fine-grained features extracted from data planes and data-driven decisions made by control planes with low latency.

Both computation and memory space complexity is presented in table 3.2. The whole data-flow is asynchronous and avoids stalling in the data exchange process in both the data plane and the control plane. This design optimizes the *scalability* of Aquarius.

3.2.3 Implementation

Based on Aquarius (chapter 3), and similar to MLB (chapter 5), Aquarius is implemented as a plugin to the Vector Packet Processor (VPP) [121] as in section 3.3.3. This chapter sets $N = 64$ since it suffices for the typical configuration in production [224] and the 64-bit bit-index header fits in the cache line for modern computer processors. The flow table size is configured as $M = 65536$.

To reduce hash collision probability, each bucket in the flow table is configured with 4 entries[7]. The level of multi-buffering is set to 3 (as in figure 3.10). The buffers copy the latest counters from the cache every 200ms (as the active probing frequency in [143]). Each sampled network feature is a 2-tuple of a 32-bit float timestamp and a 32-bit value – which fits in a single cache line. The reservoir buffer size is set to $k = 128$ for each feature per egress equipment.

[7] When a new flow is mapped into a bucket, an available entry can be found using `Timeout` attribute with $\mathcal{O}(1)$ computational complexity.

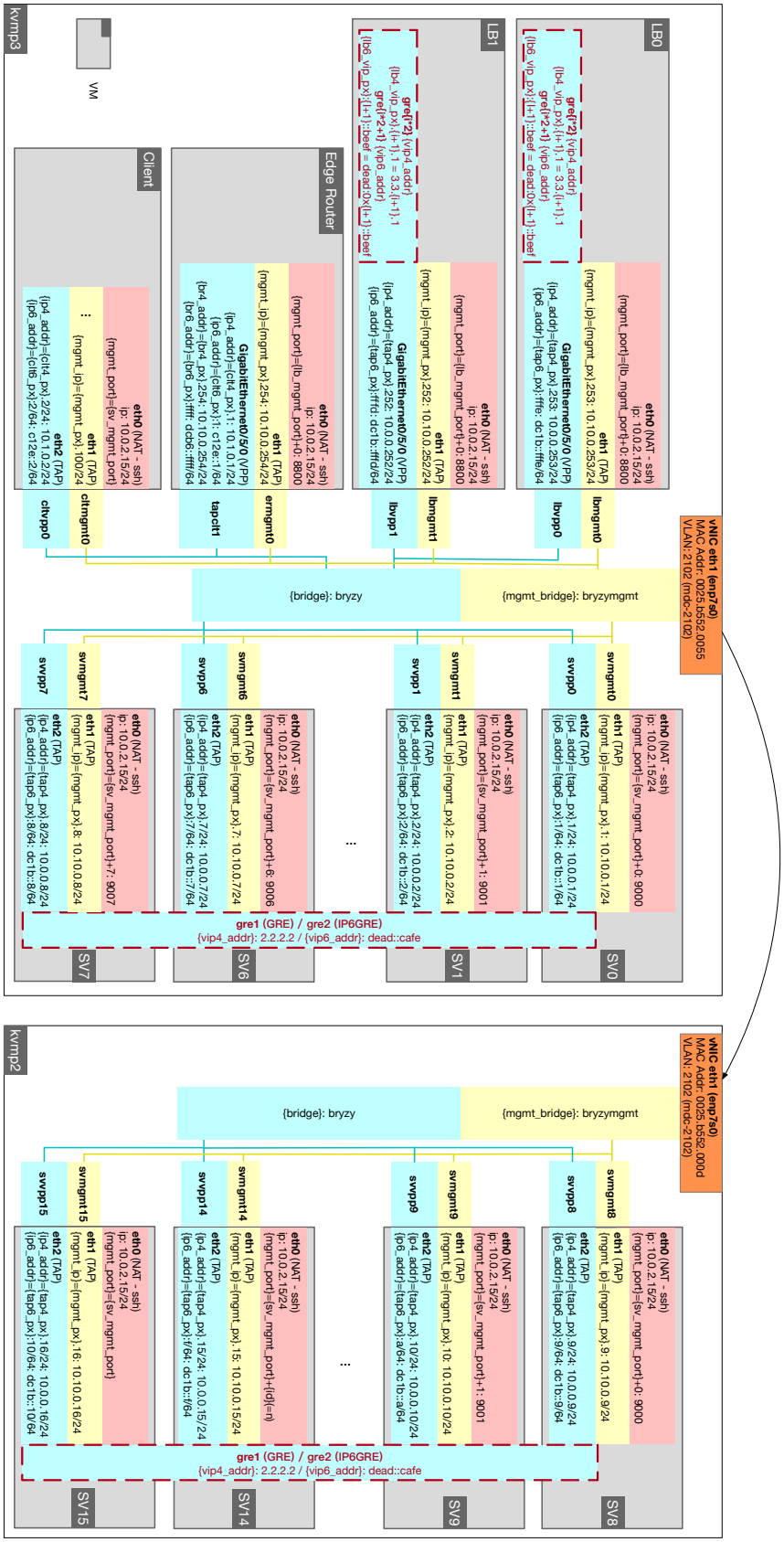


Figure 3.11: Network topology of the testbed across 2 physical servers.

All observed samples are gathered with probability $p = 1$ to further reduce the performance overhead. In these conditions, to collect all features listed in figure 3.6, the flow table takes 10.24MB of memory. The `shm` file of each VIP consists 6KB 3-level multi-buffering counters and 832KB reservoir sampling buffers.

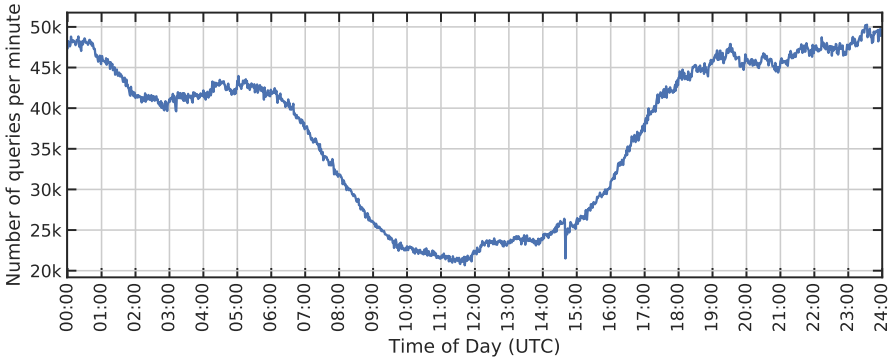


Figure 3.12: Wikipedia 24 hour replay network trace.

System Platform

Kernel-based Virtual Machines (KVMs) are run on 4 UCS B200 M4 servers, each with one Intel Xeon E5-2690 v3 processor (12 physical cores and 48 logical cores), interconnected by a UCS 6332 16UP fabric. The operating systems are `Ubuntu 18.04.3 LTS` (`GNU/Linux 4.15.0-128-generic x86_64`), and applications are compiled using `gcc` version `7.5.0` (`Ubuntu 7.5.0-3ubuntu1 18.04`). Applications used for the experiments provided in this chapter are: `Apache 2.4.29`, `VPP v20.05`, `MySQL 5.7.25-0ubuntu0.18.04.2`, and `MediaWiki v1.30`. The VMs are deployed on the same layer-2 link, with statically configured routing tables. An example network topology configuration deployed across 2 physical servers is depicted in figure 3.11.

Apache HTTP Servers

All Apache HTTP servers share the same VIP address on one end of the GRE tunnel – with the load balancer on the other end of this GRE tunnel. They use the `mpm_prefork` module to boost performance. Each server has maximum of 32 worker threads, and the TCP backlog is set to 128. In the Linux kernel, the `tcp_abort_on_overflow` parameter is enabled, so that a TCP RST will be triggered when the queue capacity of TCP flow backlog is exceeded, instead of silently dropping the packet and waiting for a SYN retransmit. With this configuration, the flow completion time (FCT) measures application response delays rather than potential TCP SYN retransmit delays (same as in [135]). Two metrics are gathered as ground truth server load state on the servers: CPU utilization and instant number of Apache busy threads. CPU utilization is calculated as the ratio of non-idle CPU time to total CPU time measured from the file `/proc/stat` and the number of Apache busy threads is assessed via Apache's *scoreboard* shared memory.

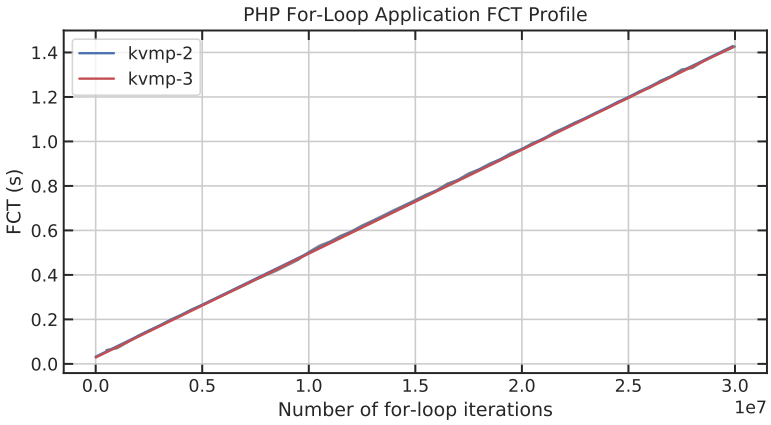
24-Hour Wikipedia Replay Trace

To create Wikipedia server replicas, an instance of MediaWiki[8] of version 1.30, a MySQL server and the `memcached` cache daemon are installed on each of the application server instance. The *WikiLoader* tool [225] and a copy of the English version of Wikipedia database [226], are used to populate MySQL databases. The 24-hour trace is obtained from the authors of [226] and for privacy reasons, the trace does not contain any information that exposes user identities. Figure 3.12 depicts the traffic rates of the network trace.

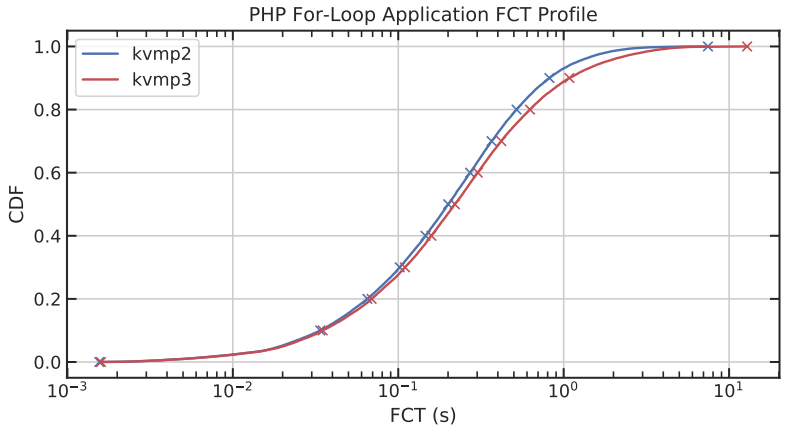
PHP `for`-Loop Trace

To study CPU-bound applications, a PHP `for`-loop script is used, whose requested number of iterations `#iter` follows an exponential distribution. The sizes of the query's replies are proportional to the number of iterations. This allows generating a heavy-tail distribution of flow durations, and of transmitted bytes, as in [1]. Figure 3.13a depicts the correlation between the number of iterations and FCT. Figure 3.13b shows the CDF of the trace being replayed on 2 different physical servers with moderate traffic rate (50% CPU usage on application servers), which demonstrates the server capacity heterogeneity and the existence of artifact.

[8] https://www.mediawiki.org/wiki/Download



(a) Correlation between `#iter` and FCT.



(b) FCT CDF comparison between 2 physical servers.

Figure 3.13: PHP `for`-loop trace profile.

PHP File Trace

To simulate IO-bound applications, PHP queries for static files of different sizes are used as in [187]. The sizes of files are 100KB, 200KB, 500KB, 750KB, 1MB, 2MB, and 5MB. 50 files are generated for each size. Figure 3.14 depicts the corresponding FCT for files with different sizes with sufficient resource provisioned.

3.3 Applications

This section shows 3 applications of Aquarius in cloud DCs in the context of 3 key VNFs – traffic classification, resource prediction, and auto-scaling, and Layer-4 load balancing, along with 3 different ML paradigms.

3.3.1 Traffic Classification

As one of the key VNFs in the cloud, traffic classification allows distinguishing different types of traffic [19, 22, 200, 201, 213], to allocate appropriate resources and achieve service level agreements [200, 213]. It also helps detect anomalies and security threats to prevent potential damages or losses [22].

Task Description and Testbed Configuration

This section shows the capability of Aquarius to collect reliable features and conduct traffic classification with unsupervised ML algorithms. A testbed is implemented using KVM, where a virtual router embedded with Aquarius forwards different types of traffic to 4 VIPs (figure 3.15). In

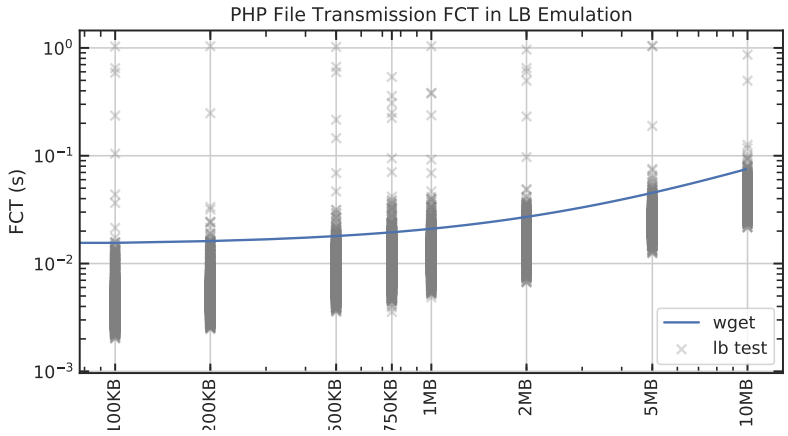


Figure 3.14: Transmission FCT for files with different sizes.

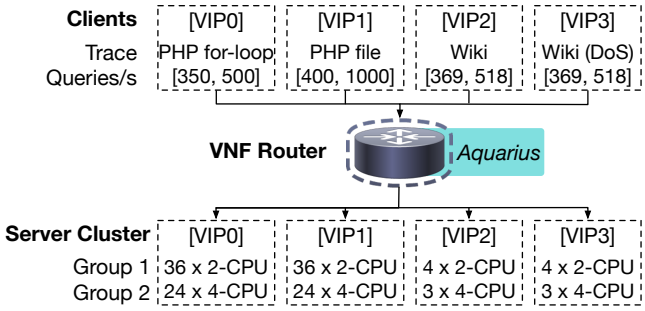


Figure 3.15: Network topology for traffic classification.

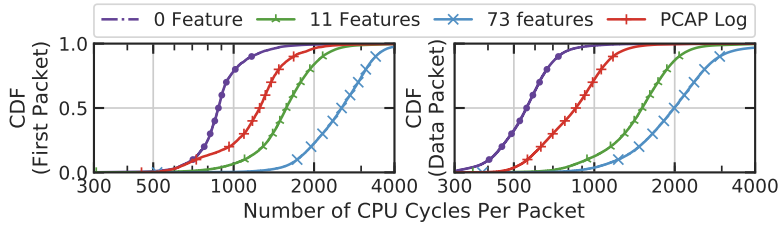
VIP0, a simple PHP `for`-loop script on each server takes requests for a given number of iterations (`#iter`) and replies with proportional sizes. The flow duration (200ms on average) and the number of transmitted bytes follow an exponential distribution as in [1]. In VIP1, static files of different sizes are served on each server[9] as in [187], to represent IO-bound applications. In VIP2 and VIP3, each application server is an independent replica of an Apache HTTP server [227] that serves Wikipedia databases. Two samples of 600s duration are extracted and replayed from a real-world 24-hour replay [226]. In VIP3, an additional 5000 queries per second SYN flooding traffic is applied to simulate a DoS attack. Server clusters are scaled to be able to serve all the queries when subjected to heavy traffic rates – when no attack happens – with reasonable FCT (under 400ms) as in [135].

Feature Engineering

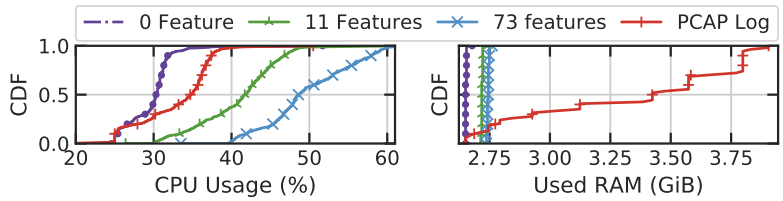
The features are fetched every 250ms from counter buffers and reservoir buffers. This chapter demonstrates the flexibility of feature engineering offered by samples collected in reservoir buffers, by reducing each feature channel to 5 scalars, *i.e.,* average, standard deviation, 90-percentile, and exponential moving average (decay) of average and 90-percentile. The moving average is a sequential feature calculated, whose weight is computed as, $0.9^{t'-t}$, where t is the timestamp of each sample and t' is the moment when the reduced sample is calculated. This yields in total 8 ordinal features (counters) and 13×5 quantitative features[10].

[9] The sizes of files are 100KB, 200KB, 500KB, 750KB, 1MB, 2MB, and 5MB. 50 files are generated for each size.

[10] The collected dataset is preprocessed and converted to have zero mean and unit standard deviation. Outlier data points (value beyond 99th-percentile) are dropped. The data preparation procedure is done using `scikit-learn` [228] and it is the same throughout the whole chapter.



(a) Per-packet processing latency comparison.



(b) System resource consumption.

Figure 3.16: Aquarius feature collection overhead.

Configuration		0 Feature	11 Features	73 Features	PCAP
First Packet	CPU Cycles	938.232	1635.838	2609.019	1295.284
	Delay (μs)	0.361	0.629	1.003	0.498
	Difference	1.000×	1.744×	2.781×	1.381×
Data Packet	CPU Cycles	576.357	1583.798	2602.684	885.041
	Delay (μs)	0.222	0.609	1.001	0.340
	Difference	1.000×	2.748×	4.516×	1.536×
CPU Usage (%)		26.687	40.858	49.716	31.480
CPU Difference		1.000×	1.376×	1.675×	1.060×
RAM Usage (GiB)		2.652	2.719	2.744	3.305
RAM Difference		1.000×	1.025×	1.034×	1.246×

Table 3.3: Per-packet processing overhead (on 2.6GHz CPU) and system resource consumptions (avg.) comparison.

Algorithm	KMeans	MeanShift	Spectral Clustering	Ward	Agglomerative Clustering	DBSCAN	OPTICS	BIRCH	Gaussian Mixture
Adjusted Rand Index	0.674	**0.863**	0.715	0.689	−0.002	0.696	0.757	0.742	0.687
Mutual Info Score	0.941	**1.160**	0.948	0.992	0.031	0.914	0.968	0.953	0.965
Adjusted Mutual Info Score	0.709	**0.820**	0.718	0.731	0.023	0.692	0.733	0.721	0.731
Homogeneity	0.713	**0.878**	0.718	0.752	0.024	0.692	0.733	0.722	0.731
Completeness	0.709	0.820	0.811	0.732	0.241	0.736	**0.914**	0.811	0.798
Fowlkes-Mallows Score	0.765	**0.901**	0.805	0.774	0.513	0.785	0.840	0.824	0.786
Fit Time (ms)	75.689	541.594	991.382	4806.554	2785.787	45.249	2769.734	52.959	**18.505**
Require Cluster Number	✓	✓	✓	✓	✓	✗	✗	✓	✓

Table 3.4: Comparison of (unsupervised) clustering algorithms for traffic classification.

Overhead Analysis

To study the feature collection overhead, Aquarius is compared with a "vanilla" router which collects 0 features, and with a router logging packet information in the memory using `pcap`. When subjected to 500 queries/s PHP loop traffic towards a 176-CPU server cluster, when collecting 11 features[11] or collecting all 73 features[12], Aquarius incurs different overheads depicted in table 3.3 and figure 3.16a. On a 2.6GHz CPU, the additional per-packet processing delays are trivial compared with the typical round trip time between two directly connected network devices (higher than 200μs [229]). The mean CPU usage of Aquarius is 1.376× and 1.675× higher than the "vanilla" router when collecting 11 and 73 features, respectively (figure 3.16b). As expected, the log-based

[11] 1 counter (n_flow_on) and 2×5 sampled features (flow duration, FCT).
[12] 8 counters and 13×5 quantitative features.

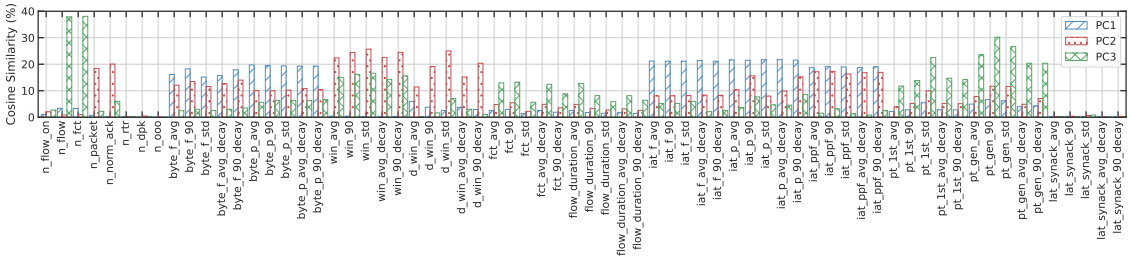
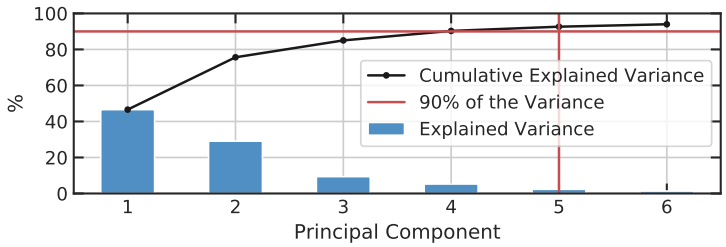
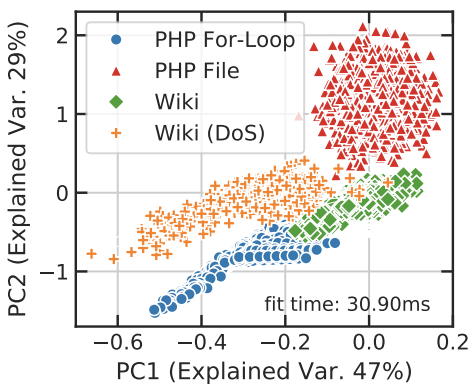


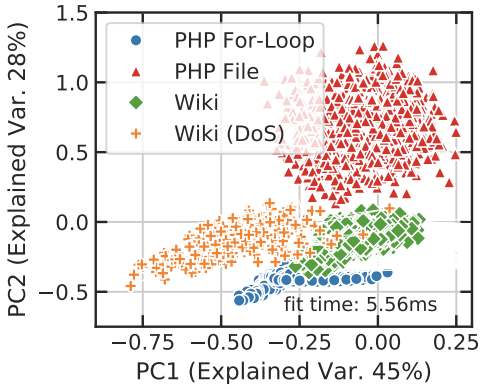
Figure 3.17: Variance contribution of each feature in top-3 principal components (PCs).



(a) Percentage of explained variance.



(b) PCA clusters (all features).



(c) PCA clusters (25 features).

Figure 3.18: PCA analysis and 2D visualization.

feature collection mechanism does not scale in terms of memory consumption[13].

Feature Selection with PCA

More features provide multi-dimensional observations, yet at the cost of higher computation and memory overhead. Principal Component Analysis (PCA) is thus conducted to understand the relative importance of the feature and reduce dimensionality while preserving data representation. As depicted in figure 3.18a, 90% of the data variance can be explained with 4 principal components (PCs). Figure 3.17 shows that multiple features share similar contributions (cosine similarity) to top-3 PCs, especially features reduced from the same reservoir buffer. Therefore, the number of features can be decreased by using only 2 (standard deviation and `decay`-ed average) out of the 5 reduced scalars. Also by removing sampled data that has low contribution to the top-4 PCs (*i.e.*, `iat_synack`), 25 features are selected out of all 73 features.

As depicted in figure 3.18b, 4 clusters for the 4 traces are visualized in a 2D representation. Among the 4 traces, PHP `for`-loop is pure CPU-bound and PHP file is pure IO-bound. The `Wiki`

[13]The results can be machine-dependent. This chapter aims at showing the order of magnitudes, rather than providing a precise quantification.

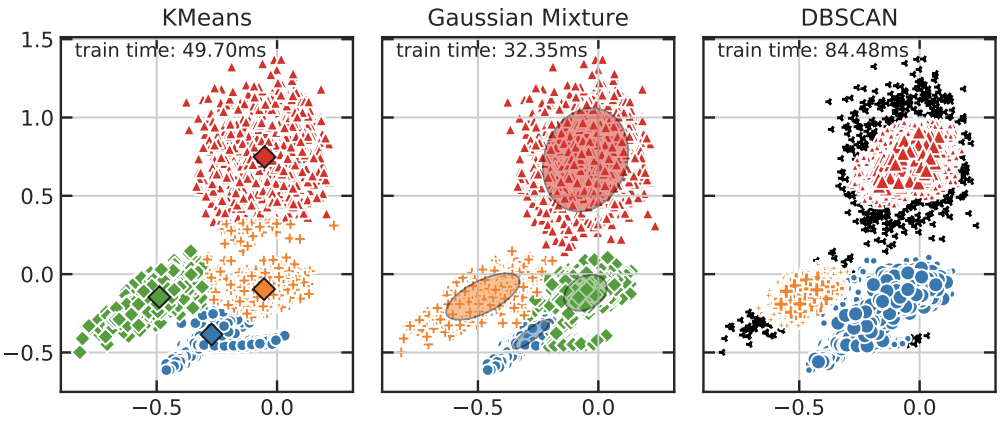


Figure 3.19: Unsupervised clustering using 25 features.

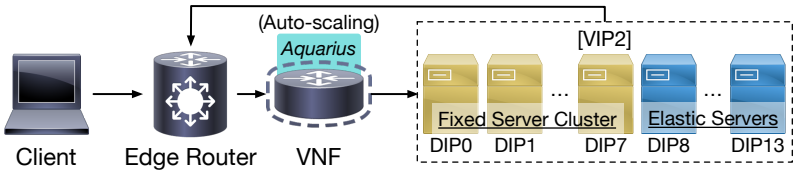


Figure 3.20: Network topology for autoscaling system.

Algorithm	Linear Regression	Ridge Regression	Decision Tree	Random Forest	SVR (Linear)	SVR (RBF)	XGBoost	RNN	LSTM	GRU	GRU+ 1dConv	WaveNet	Active Probing
First Step MAE	9.216	9.232	12.135	8.717	9.255	9.040	8.544	7.629	7.433	7.488	7.492	7.830	**3.504**
First Step RMSE	11.779	11.763	15.558	11.125	11.876	11.395	10.871	9.591	9.403	9.478	9.481	9.770	**4.593**
Last Step MAE	10.640	10.638	15.043	11.009	10.683	11.206	10.797	9.774	9.855	9.798	10.001	**9.652**	11.892
Last Step RMSE	13.266	13.261	18.965	13.794	13.327	13.994	13.557	12.285	12.496	12.427	12.653	**12.194**	14.935
All Step Avg. MAE	10.038	10.044	14.109	10.090	10.063	10.335	9.891	8.986	9.046	9.022	9.123	9.010	**8.334**
All Step Avg. RMSE	12.575	12.575	17.866	12.742	12.616	12.963	12.512	11.331	11.505	11.464	11.594	11.390	**11.057**
Avg. Predict Time (ms)	1.429	1.375	1.765	152.558	629.212	1462.446	5.315	115.179	117.146	113.117	87.831	106.475	**0.026**
Predict Time Stdev. (ms)	0.380	0.294	0.013	0.305	0.093	2.015	0.288	1.961	4.100	3.592	1.680	1.968	**0.001**

Table 3.5: Comparison of supervised ML algorithms for resource prediction (using selected *non-sequential* features to predict 8 steps ahead).

trace consists of both queries for SQL database (CPU-bound) and static files (IO-bound), thus its cluster is located between the former 2 traces. The `Wiki` trace under DoS attack, however, can be clearly noticed as an independent cluster. As depicted in figure 3.18c, using the 25 selected features [14] still gives clear clustering results, yet it reduces data processing time from 30.90ms to 5.56ms.

Unsupervised Learning

Nine clustering algorithms are applied and compared over the obtained dataset. As in table 3.4, MeanShift shift has the best overall performance, yet at the cost of relatively high fit time. As depicted in figure 3.19, when applying unsupervised learning algorithms, K-Means [230] and Gaussian Mixture [231] are able to generate clusters similar to the ground truth, while they require the number of expected clusters (4) as input. Gaussian Mixture model has the shortest fit time and can be an interesting candidate for online traffic classification. In case where the number of

[14] The input 25 networking features are: byte_f_std, byte_f_avg_decay, byte_p_std, byte_p_avg_decay, win_std, win_avg_decay, d_win_std, d_win_avg_decay, fct_std, fct_avg_decay, flow_duration_std, flow_duration_avg_decay, iat_f_std, iat_f_avg_decay, iat_p_std, iat_p_avg_decay, iat_ppf_std, iat_ppf_avg_decay, n_flow_on, n_flow, n_fct, n_packet, n_rtr, n_dpk, n_ooo.

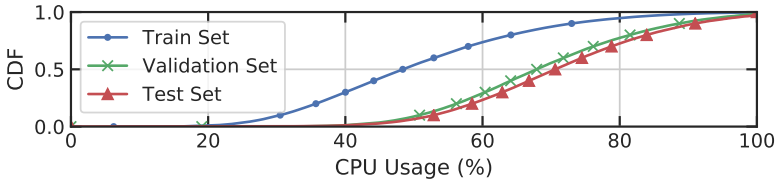


Figure 3.21: Comparison of ground truth distributions.

Algorithm	Linear Regression	Ridge Regression	Decision Tree	Random Forest	SVR (Linear)	SVR (RBF)	XGBoost	RNN	LSTM	GRU	GRU+ 1dConv	WaveNet	Active Probing
First Step MAE	9.186	9.206	12.134	8.697	9.232	8.999	8.486	7.577	7.379	7.458	7.496	7.527	**3.505**
First Step RMSE	11.715	11.723	15.538	10.991	11.838	11.302	10.716	9.515	9.414	9.559	9.578	9.595	**4.593**
Last Step MAE	10.858	10.861	14.976	11.129	10.897	11.322	10.964	9.794	9.935	9.579	9.786	**9.504**	12.238
Last Step RMSE	13.666	13.673	19.000	13.995	13.716	14.266	13.810	12.476	12.657	12.231	12.435	**12.130**	15.470
All Step Avg. MAE	10.399	10.404	14.587	10.555	10.424	10.776	10.362	9.342	9.478	9.165	9.374	**9.146**	10.216
All Step Avg. RMSE	13.039	13.046	18.393	13.262	13.077	13.525	13.035	11.852	12.069	11.700	11.933	**11.625**	13.335
Avg. Predict Time (ms)	2.689	2.659	3.557	304.490	1281.118	3026.451	10.445	96.102	120.756	111.489	89.297	105.831	**0.033**
Predict Time Stdev. (ms)	0.285	0.292	0.016	0.458	1.617	0.871	0.093	1.882	5.700	5.829	3.774	3.756	**0.014**

Table 3.6: Comparison of supervised ML algorithms for resource prediction (using selected *non-sequential* features to predict 16 steps ahead).

clusters is not known *a priori*, DBSCAN [232] can distinguish the potential security threat, based only on a predefined distance (0.1). With a training latency lower than 100ms, these algorithms can be interesting candidates for online traffic classification and anomaly detection systems. OPTICS [233] also achieves the highest completeness – all members of a given trace type are assigned to the same cluster, though with a much higher processing latency than DBSCAN.

Take-Away This section effectively has demonstrated the high *universality* of Aquarius. Aquarius enables gathering fine-grained and reliable datasets – for different types of network traffic – which allows feature engineering and in-depth data analysis. Aquarius' fast and configurable design help achieve the right balance between visibility and performance.

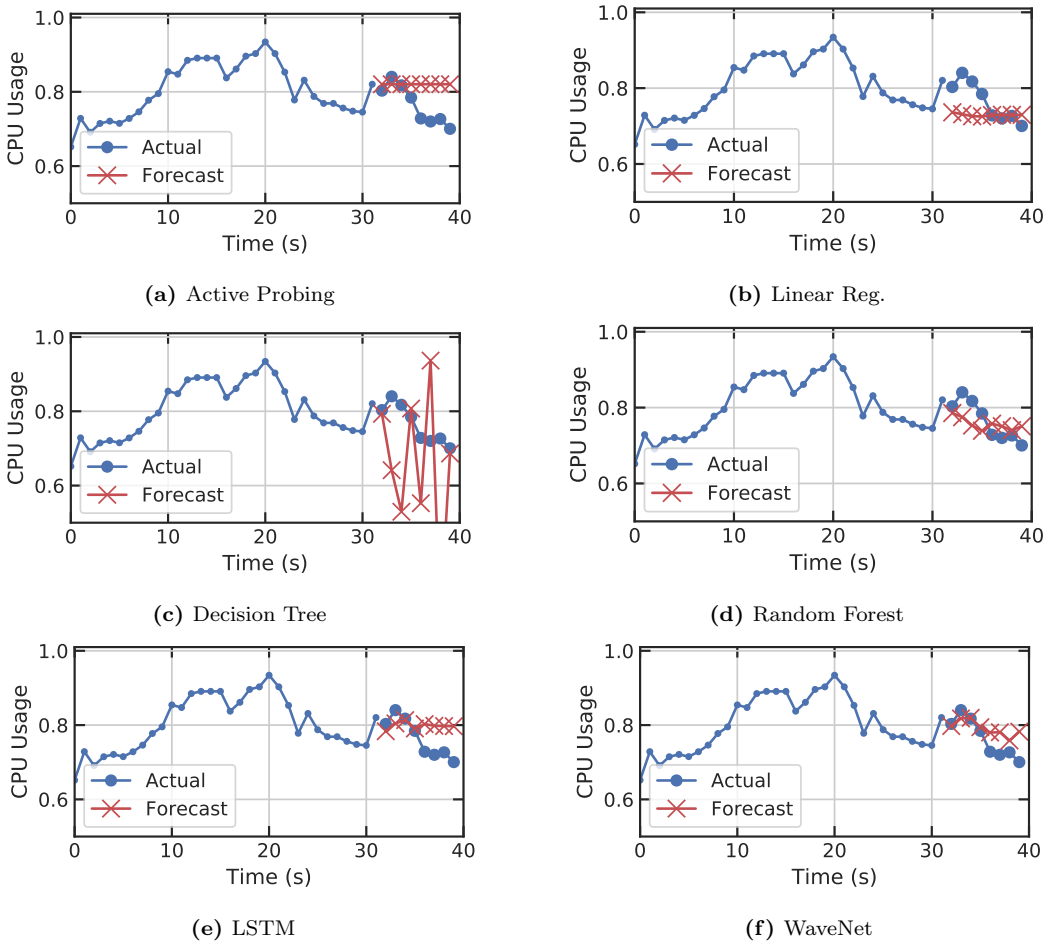
3.3.2 Resource Prediction and Auto-Scaling

To minimize operational costs while guaranteeing QoS, cloud operators need to elastically and intelligently provision server capacity and configurations [29, 137, 139, 140]. This section explores the capability of Aquarius as a platform to enable the development of supervised ML algorithms, to infer resource utilization, and performance – thus, the need for up-scale or down-scale actions – with no active signaling.

Task Description and Testbed Configuration

This section studies networking features and system utilization when subjected to different levels of workloads, to avoid additional control messages in existing auto-scaling mechanisms [137]. 600s samples, extracted from each hour of the real-world 24-hour Wikipedia trace [226], are replayed on the network topology depicted in figure 3.20. Workloads are randomly distributed among running servers (2-CPU each) by way of Equal-Cost Multi-Path (ECMP). The server cluster requires $8 \sim 14$ servers to provide "reasonable" QoS (median FCT $\leq$ 400ms [135]). A learning task can be framed as predicting server load states (CPU usage[15]) on each server with the same set of features as in section 3.3.1. The predicted utilization can be then used to plan and re-scale server clusters to guarantee QoS with reduced operational cost. This task consists of 2 steps – offline model training and online prediction. The first 23-hour samples are applied on 10 2-CPU servers to gather datasets for offline model training. The last-hour sample, which is not seen by any trained

[15]This chapter uses CPU usage as the metric to evaluate and plan server cluster capacity for demonstration. The same methodology can be applied to problems using multi-variate metrics.



(a) Active Probing **(b)** Linear Reg.

(c) Decision Tree **(d)** Random Forest

(e) LSTM **(f)** WaveNet

Figure 3.22: Prediction results of 7 selected models using sequential features to predict 8 steps ahead.

Algorithm	Linear Regression	Ridge Regression	Decision Tree	Random Forest	SVR (Linear)	SVR (RBF)	XGBoost	RNN	LSTM	GRU	GRU+ 1dConv	WaveNet	Active Probing
First Step MAE	9.219	9.223	12.419	8.892	9.241	8.953	8.758	8.205	7.842	7.622	7.785	8.040	**3.504**
First Step RMSE	11.543	11.553	15.745	11.328	11.582	11.288	11.141	10.370	10.012	9.716	10.005	10.207	**4.593**
Last Step MAE	10.658	10.662	15.023	10.840	10.689	10.855	10.704	9.787	9.627	**9.442**	9.566	9.550	11.892
Last Step RMSE	13.240	13.244	18.829	13.639	13.277	13.600	13.449	12.253	12.239	**11.933**	12.110	12.061	14.935
All Step Avg. MAE	10.059	10.063	14.077	10.073	10.074	10.056	9.950	9.272	9.091	8.855	8.949	9.027	**8.334**
All Step Avg. RMSE	12.528	12.534	17.772	12.743	12.552	12.647	12.585	11.646	11.564	11.225	11.379	11.433	**11.057**
Avg. Predict Time (ms)	1.394	1.390	1.727	150.036	604.762	1377.609	5.579	115.887	118.995	109.009	89.465	104.772	**0.022**
Predict Time Stdev. (ms)	0.301	0.316	0.009	0.186	0.023	0.056	0.243	3.537	2.673	4.954	3.413	2.584	**0.008**

Table 3.7: Comparison of supervised ML algorithms for resource prediction (using selected *sequential* features to predict 8 steps ahead).

model, is synthesized to have 5 different levels of traffic rates for online prediction and real-time auto-scaling.

Offline Model Training

To predict the resource utilization of server clusters using networking features, 12 widely used ML algorithms are selected to cover different families of ML algorithms, *e.g.,* sequential and non-sequential, parametric and non-parametric, linear and non-linear [30]. The dataset is pre-processed in the same way as described in section 3.3.1. To adapt the dataset for sequential models, the sequence length (time steps) of input features is set as 32 and the stride as 16, which gives 50k data points in total. These data points are sequentially split 70 : 20 : 10 into training, validation, and testing sets. The distribution of the ground truth CPU usage in the training set covers the other two datasets so that the prediction task is feasible, yet the ML models have not seen the datasets for evaluations (figure 3.21). Sequential models are created and trained using Keras with

Algorithm	Linear Regression	Ridge Regression	Decision Tree	Random Forest	SVR (Linear)	SVR (RBF)	XGBoost	RNN	LSTM	GRU	GRU+ 1dConv	WaveNet	Active Probing
First Step MAE	9.205	9.210	12.514	8.841	9.229	8.911	8.712	8.339	7.889	7.863	7.855	8.149	**3.505**
First Step RMSE	11.527	11.537	15.799	11.158	11.567	11.162	10.964	10.375	9.990	10.115	10.014	10.353	**4.593**
Last Step MAE	10.823	10.828	15.019	11.064	10.856	11.054	10.895	9.993	9.699	**9.394**	9.564	9.447	12.238
Last Step RMSE	13.605	13.612	18.834	13.884	13.645	13.916	13.706	12.634	12.303	12.020	12.200	**12.002**	15.470
All Step Avg. MAE	10.412	10.417	14.576	10.478	10.431	10.444	10.337	9.711	9.220	**9.143**	9.316	9.148	10.216
All Step Avg. RMSE	13.003	13.010	18.328	13.177	13.029	13.103	12.998	12.181	11.661	11.668	11.852	**11.576**	13.335
Avg. Predict Time (ms)	2.547	2.522	3.464	298.758	1228.016	2832.406	10.699	95.248	114.952	111.944	89.810	105.492	**0.028**
Predict Time Stdev. (ms)	0.310	0.164	0.011	0.319	0.051	70.800	0.419	1.787	4.793	2.463	3.843	3.036	**0.007**

Table 3.8: Comparison of supervised ML algorithms for resource prediction (using selected *sequential* features to predict 16 steps ahead).

TensorFlow as backend [234]. Non-sequential models (built using scikit-learn [228]) use the last time step features as input data. Each model is trained to predict the CPU usage multiple steps ahead.

ML Models Six non-sequential models are implemented using scikit-learn with their default hyperparameters, *i.e.,* linear regression, ridge regression, decision tree, random forest, SVM regression (SVR) with both linear and RBF kernel, and XGBoost. 5 sequential models are implemented using Keras, *i.e.,* RNN, LSTM, GRU, GRU with a 1-dimensional convolutional layer, and WaveNet. RNN has 2 20-hidden-unit SimpleRNN layers (first layer with return_sequence=True) and 1 output layer. LSTM replaces the SimpleRNN in the RNN model with LSTM layers and GRU replaces with GRU layers. GRU with 1d convolutional layer adds one 1-dimentional CNN (as in textCNN) before the GRU model. Wavenet stacks 4 stacked dilated 1D convolutional layers with 1 layer of 20-hidden-unit GRU and 1 fully connected layers (output layer). As a benchmark, a naive model is implemented to simulate active probing by using the last observed CPU usage as prediction.

Feature Selection To reduce input size, features are selected using feature_selection.f_regression in scikit-learn [228], in two different procedures, namely in a non-sequential and a sequential manner. In a non-sequential manner, the top 20-percentile features with the highest correlation with the CPU usage are selected[16]. In a sequential manner, networking features are first re-arranged by 32 time steps, then the features that appear more than 3 time steps in the top 20-percentile features with the highest correlation with the CPU usage, are selected[17].

Different Prediction Steps Ahead The further in the future that one can predict, the better configuration plans can be made. Therefore, tasks are created to predict the different number of time steps ahead, namely 8 or 16 steps, to study the capabilities of predicting the future among different ML models.

Results Instances of the prediction results from a subset of ML models are visualized as in figure 3.22. The scores achieved by each predicting model using the test set are shown in table 3.5-3.8. The prediction time for each model is evaluated using 256 datapoints (as predicting resource utilization on 256 servers). The results show that sequential models achieve better performance when using sequential features as input data than using non-sequential features.

Simple and non-sequential ML models perform worse that sequential models, especially when predicting 16 steps ahead as sequential models has more visibility on the history. WaveNet has the best overall performance and robustness across all 4 different tasks among all ML models, therefore it is chosen in this chapter to be applied online. Linear regression, on the other hand, is the simplest ML model and has the shortest processing latency overhead when making prediction, therefore it is chosen to be applied online as well.

[16]The 21 "non-sequential features" consist of: fct_90_decay, fct_avg_decay, fct_std, flow_duration_90, flow_duration_90_decay, flow_duration_avg, flow_duration_avg_decay, flow_duration_std, iat_f_avg, iat_f_avg_decay, iat_p_std, iat_ppf_90_decay, iat_ppf_avg, iat_ppf_avg_decay, iat_ppf_std, n_flow_on, pt_1st_90, pt_1st_90_decay, pt_1st_avg_decay, pt_1st_std, pt_gen_90_decay.

[17]The 15 "sequential features" consist of: n_flow, n_packet, iat_f_avg, iat_f_90, iat_f_std, iat_f_avg_decay, iat_f_90_decay, iat_p_avg, iat_p_std, iat_p_avg_decay, pt_1st_std, lat_synack_avg, lat_synack_90, lat_synack_90_decay, flow_duration_std.

Algorithm 2 Auto-scaling Rule

$n\_servers\_min, n\_servers\_max \leftarrow 8, 14$ ▷ *Server number range*
$\mathbb{S} \leftarrow$ Initial set of running servers
$cpu\_lo, cpu\_hi \leftarrow 0.7, 0.8$ ▷ *Desired CPU usage range*
for each time step **do** ▷ $\Delta t = 250ms$
  $\delta \leftarrow 0$ ▷ *Initialize server state counter*
  $y(\mathbb{S}) \leftarrow$ CPU usage prediction of 16 steps ahead
  $threshold \leftarrow \lceil \frac{|\mathbb{S}|}{3} \rceil$ ▷ *Threshold that triggers scaling actions*
  for $s \in \mathbb{S}$ **do**
    if $y(s) < cpu\_lo$ **then**
      $\delta++$ ▷ *Increment δ if s is under-loaded*
    else if $y(s) > cpu\_hi$ **then**
      $\delta--$ ▷ *Decrement δ if s is over-loaded*
  if $\delta > threshold$ **and** $|\mathbb{S}| > n\_servers\_min$ **then**
    $\mathbb{S} \leftarrow downscale(\mathbb{S})$
    skip 8 time steps ▷ *Cool-down period*
  else if $\delta < -threshold$ **and** $|\mathbb{S}| < n\_servers\_max$ **then**
    $\mathbb{S} \leftarrow upscale(\mathbb{S})$
    skip 8 time steps ▷ *Cool-down period*

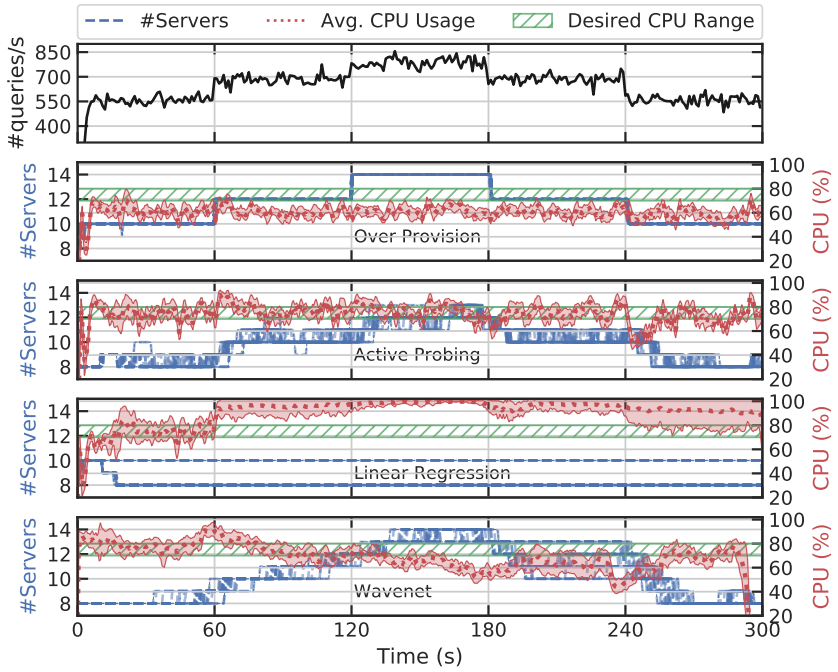
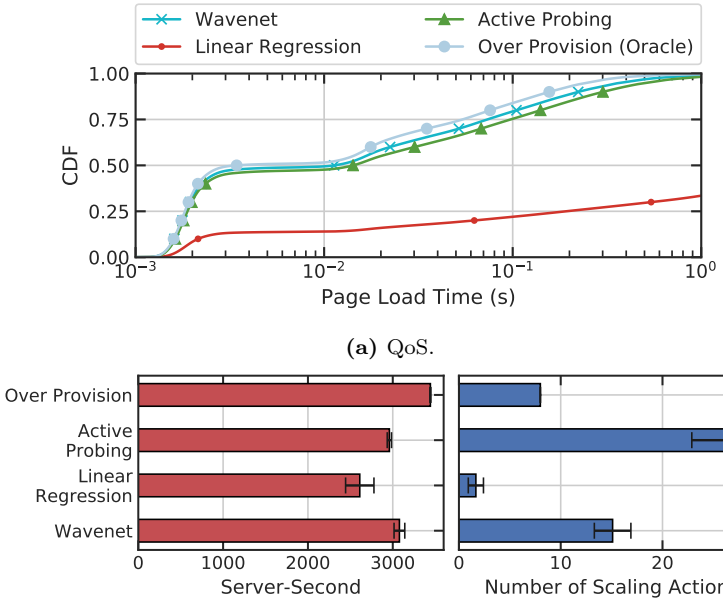


Figure 3.23: Comparison of online auto-scaling performance using different algorithms. The (discrete) numbers of running servers are plotted for each run in dashed lines, while CPU usage is summarised as avg. ± stddev across 30 runs.

Online Auto-Scaling

To test the online performance of offline-trained ML models, a 300s Wikipedia replay trace sample of the last hour (unseen by the ML models) is synthesized to have scheduled changing traffic rates every 60s. Based on the 16-step-ahead CPU usage predictions of running servers $y(\mathbb{S})$, a simple heuristic is proposed (algorithm 2) to keep the CPU usage of $\frac{2}{3}$ servers within the desired range ($70 \sim 80\%$). Using the same counter Δ for over- and under-loaded servers reduces the variance induced by imbalanced workload distributions. As a reference, an active probing mechanism is implemented, whose predicted CPU usage for running servers $y(\mathbb{S})$ comes from periodic polling (every 250ms, same as the prediction interval of ML methods). An "oracle" benchmark is implemented to over-provision the number of servers proportional to the scheduled traffic rates.



(a) QoS.

(b) Operational cost and complexity.

Figure 3.24: Trade-off between QoS and cost using different autoscaling mechanisms.

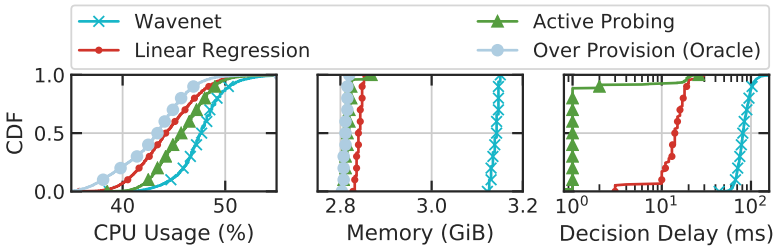


Figure 3.25: Comparison of system overhead using different autoscaling mechanisms.

Results

As depicted in figure 3.23, active probing keeps the average CPU usage within the desired range, however, it requires frequent scaling events and leads to oscillating CPU usage with high variance. Linear regression is simple yet not robust when applied to an online auto-scaling system. Its under-estimated server load states lead to over-loaded servers. WaveNet takes sequential features as input and is more robust when applied online. It keeps the average CPU usage close to the desired range with fewer oscillations.

As depicted in figure 3.24, WaveNet is able to provide better QoS than active probing – 78.37ms less page load time (26.04%) at 90th percentile and 35.70ms less (30.45%) on average – with 3.99% additional server-second cost, and 42.44% less scaling events. When over-provisioning the server cluster, the page load time is shorter than using WaveNet by 67.13ms at 90th percentile and 28.55ms average, though it requires 11.86% more server-second operational cost than WaveNet.

Overhead Analysis

As depicted in figure 3.25, ML models incur additional memory usage and predicting delay. Active probing also incurs additional CPU usage. WaveNet, as a more sophisticated ML model, incurs 1.844% additional CPU usage and 322.61MiB additional memory usage than active probing. However, Aquarius parses features stored in the local shared memory with no control messages, achieving more than 94.18μs less median latency than typical VM- and container-based probing mechanisms (figure 3.26). As dedicated hardware, NetFPGA achieves better performance by

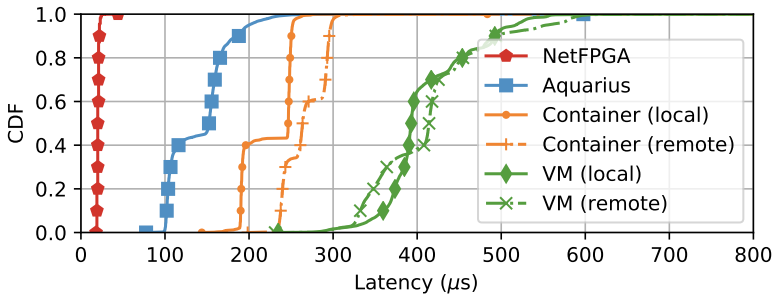


Figure 3.26: Feature collection latency comparison between Aquarius and active probing techniques.

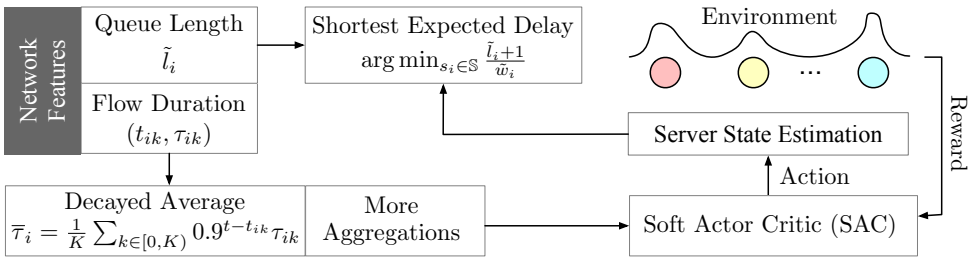


Figure 3.27: Overview of the RLB algorithm [28].

sending and receiving packets at line rate, yet it lacks adaptivity and flexibility in developing data-driven algorithms when compared with Aquarius.

Take-Away In addition to feature *universality*, this section has shown the high *scalability* of Aquarius. Aquarius enables agile development, offline model selection, and online deployment of learning algorithms to improve network performance. It makes features quickly accessible – even if the networking topology (*e.g.,* number of servers) changes over time – while saving management bandwidth for data transmission.

3.3.3 Traffic Optimisation and Load Balancing

As a key component in cloud DCs, Layer-4 load balancers (LBs) distribute workloads across servers to provide scalable services. This section shows that Aquarius can enable applications of RL algorithms to optimise load balancing performance.

Task Description and Testbed Configuration

In cloud DCs, servers can be virtualized on infrastructures with different processing speeds [72]. This section inherits the configuration of VIP2 (figure 3.15) – replaying the `Wiki` trace and load balancing on 2 groups of servers of different processing capacities. The task is to extract and infer server processing capacity information from networking features and make informed load-balancing decisions. 3 benchmark LB algorithms are implemented – (i) ECMP [190] randomly distributes workloads regardless of server processing speed differences; (ii) WCMP [64, 235] statically assigns weights to servers based on their provisioned capacities; (iii) active WCMP [143, 186, 187] polls server job queue lengths and updates weights every 200ms based on probed utilization information.

RL Algorithm

RLB [28] is an RL-based LB algorithm implemented and evaluated in simulators, similar to many RL algorithms for networking [25, 184]. In this chapter, RLB is implemented and evaluated

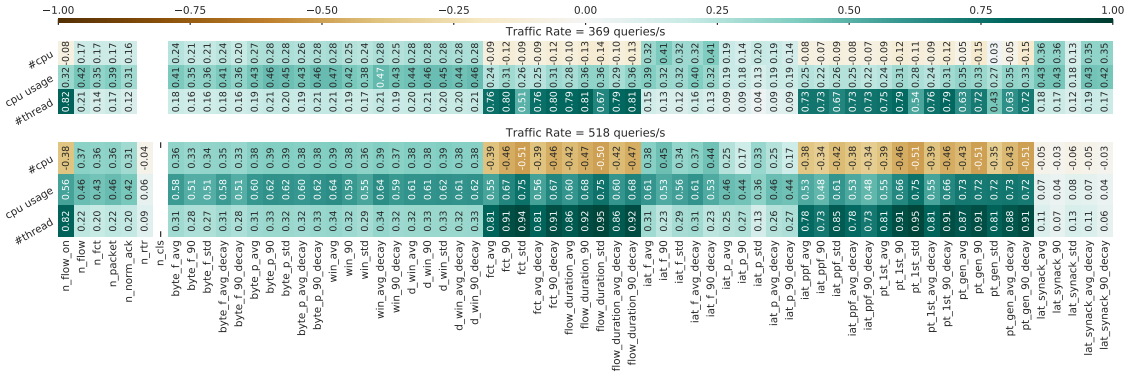


Figure 3.28: Correlation between networking features and server states - VIP2 (Wikipedia trace).

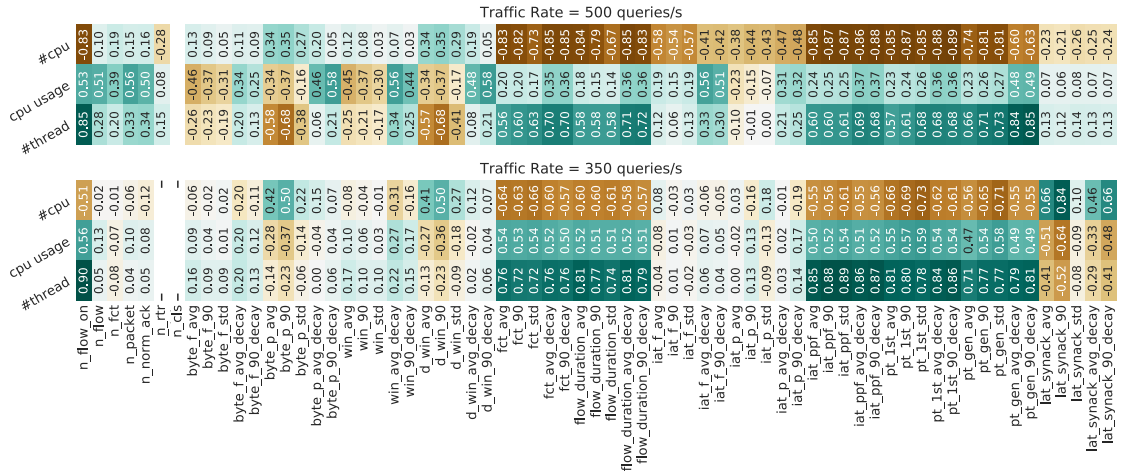


Figure 3.29: Correlation between networking features and server states - VIP0 (PHP for-loop).

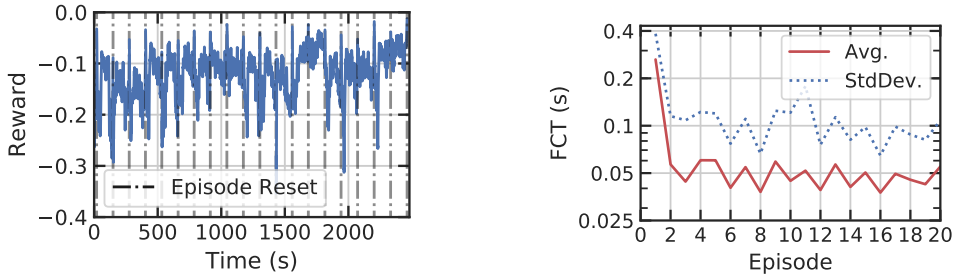
in a realistic testbed using Aquarius. As depicted in figure 3.27, with Aquarius, RLB (i) counts ongoing flows $\tilde{l}_i$ on servers and (ii) asynchronously updates (every 250ms) server weights $\tilde{w}_i$ (actions) for each application server as server load state estimations, derived from flow durations τ_i sampled in reservoir buffers as input features. The same architecture with a Soft Actor-Critic model as in [28] is implemented. However, the actor and critic networks take the batch-normalised features only based on locally observed per-server states. On receipt of new requests, RLB assigns servers based on scores that estimate the time to finish all the workloads for each server using the shortest expected delay algorithm [236], *i.e.,* $\arg\min_i \frac{\tilde{l}_i+1}{\tilde{w}_i}$, which prioritizes servers with higher processing speed and shorter queue lengths. Different from [28], which uses actively probed ground truth information, this chapter derives the reward from features collected by Aquarius. The reward is chosen as $\frac{\bar{\tilde{\tau}}^2}{\overline{\tilde{\tau}^2}} - 1$, where $\tilde{\tau}$ is a list of discounted average of flow duration on each server, which is also collected by Aquarius.

Feature Validation

RLB uses the number of ongoing flows to indicate server queue occupation and, it uses flow duration as an input feature to infer server processing capacity. To verify the feature selection of RLB and study the correlation of network features with server load states in real-world networking systems, moderate and heavy network traces of both Wikipedia replay (VIP2 in figure 3.15)[18] and PHP for-loop (VIP0 in figure 3.15) are applied on the testbed. The correlation between all the features and server states when subjected to different traffic rates is depicted in figure 3.28 and 3.29.

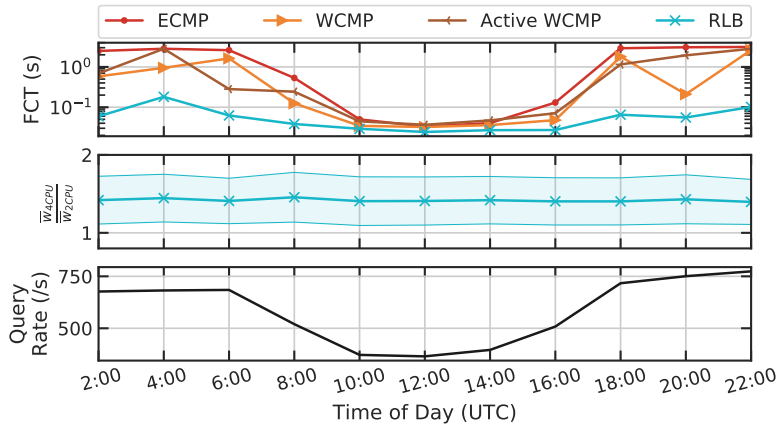
[18] Using ECMP, which does not distinguish the server processing capacity difference, an average FCT of 45ms and 836ms is achieved respectively when subjected to light and heavy traffic.

As expected, one intuitive feature among the counters that helps infer server load state is the number of ongoing flows (n_flow_on). For VIP2, since the replayed trace is not IO-intensive – SQL queries with small file sizes whose average and standard deviation are both 12KiB, throughput-related features are indicative of the different provisioned server processing capacities (the number of CPUs `#cpu`).

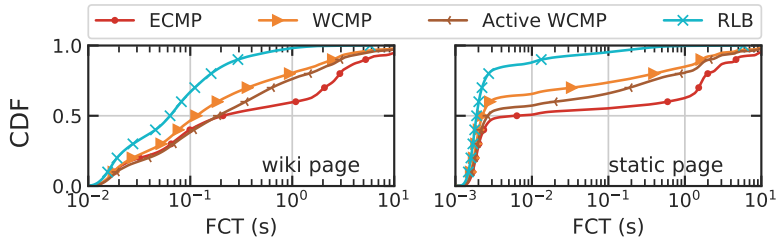


(a) RLB reward. **(b)** FCT.

Figure 3.30: Evaluation during 20-episode training.



(a) Mean FCTs (top), the ratio between weights assigned to the 2 groups of servers by RLB (middle), and traffic rates (bottom).



(b) Peak-hour (query rate higher than 500/s) FCT distribution.

Figure 3.31: Wikipedia trace replayed using different LBs.

However, for both VIP0 (requests are CPU-intensive) and VIP2, latency-related features (*e.g.*, FCT, flow duration) show a higher correlation than achieved using active probing (figure 3.2), since they capture the fact that heavily loaded or less powerful servers have slow processing speeds. This effectively shows that networking features passively gathered by Aquarius are reliable and the selected input features of RLB are representative.

Results

RLB is trained using the first hour of `Wiki` trace sample for 20 runs (episodes). As depicted in figure 3.30, RLB learns server capacity differences. The rewards of RLB during training grow

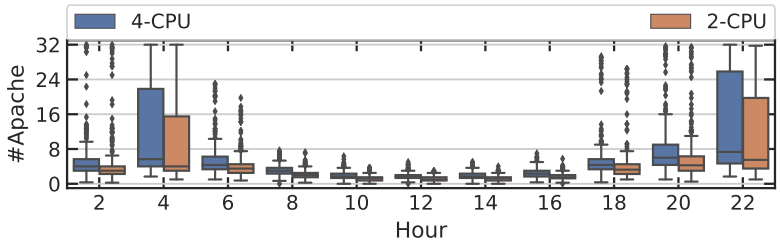
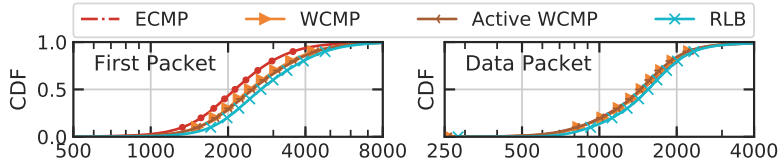
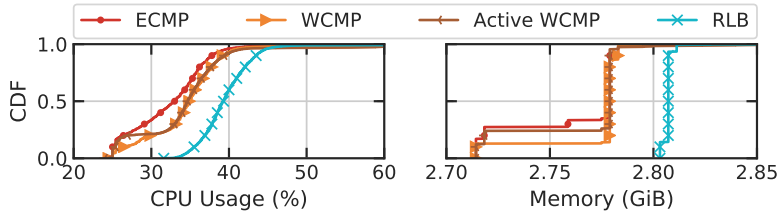


Figure 3.32: Query distribution (number of busy Apache threads) on 2 groups of application servers.



(a) Per-packet processing latency comparison.



(b) System resource consumption.

Figure 3.33: Overhead comparisons.

higher and less variant, and the FCT becomes lower. The trained RLB model is then tested on unseen traffic and compared with other LB algorithms (figure 3.31a). During off-peak hours, servers are under-utilized and all algorithms show similar performance. As traffic rates grow, RLB achieves lower FCT for both static pages and Wikipedia pages when compared with other LB algorithms (figure 3.31b).

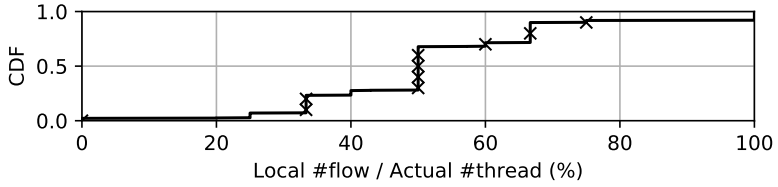


Figure 3.34: Partial observations happen when traffic is split across 2 VNFs.

RLB is trained to learn server processing speed differences and assigns higher weights, thus more queries, to more powerful servers (figure 3.32). When using RLB, 4-CPU servers handle respectively 1.258× and 1.523× more tasks than 2-CPU servers when subjected to 676.92 and 372.01 queries/s traffic.

Overhead Analysis

As depicted in figure 3.33a, throughout all test runs, RLB consumes on average 692.89 more CPU cycles (0.26μs on 2.6GHz CPU) than ECMP, as it computes and compares the server scores when making load balancing decisions. Figure 3.33b depicts CPU and memory consumptions of all LBs. RLB incurs 0.22× additional CPU usage and 45.99MiB memory usage on average.

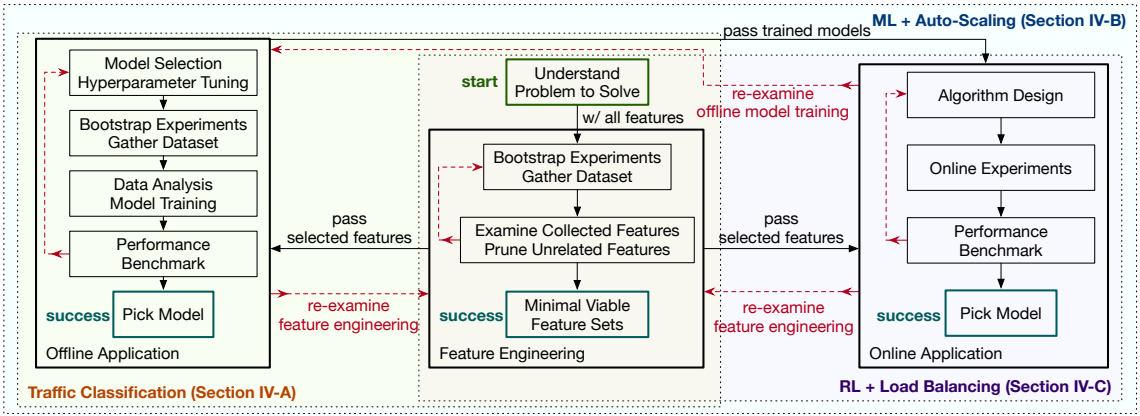


Figure 3.35: Methodology blueprint. The application of ML techniques starts from understanding the problem to solve, including *e.g.,* the objectives and constraints. Aquarius provides high configurability and programmability to conduct extensive and iterative feature engineering and selection process to prune unrelated features (reduce additional feature processing overhead) and to pick a minimally viable set of features that can be gainfully used for solving the target problem. The selected features can be passed to both offline application (*e.g.,* clustering algorithms + traffic classification in section 3.3.1) and online application (*e.g.,* RL + load balancing in section 3.3.3). Offline trained ML models can also be brought online to evaluate their performance in real-time (*e.g.,* supervised learning + autoscaling in section 3.3.2). As a platform that helps harness reliable networking features and learning algorithms, Aquarius allows iteratively investigating networking features, developing models, and designing algorithms.

Partial Observations

Though RLB achieves better performance than heuristic load balancing methods, it relies on ordinal features, which, collected in a distributed system, risk reflecting only partial system states. For instance, when traffic is split across multiple load balancers, the locally counted number of ongoing flows does not reflect the actual queue length on the application server. As is depicted in figure 3.34, in presence of 2 load balancers, the ratio of the locally observed number of flows over actual queue length #thread has a standard deviation of 22.49%. However, RLB relies also on latency-related features (flow duration), which can be gainfully used to infer server load states and compensate for the impacts of partially observed ordinal features.

Take-Away This section effectively demonstrated the high feature *relavance* of Aquarius, even under changing environments. Aquarius enables closed-loop control (RL) to dynamically adapt to networking systems and optimize performance. It empowers real-world deployment and evaluation of learning algorithms developed in simulated environments.

3.4 Summary

Networking features and system state information help Virtual Network Functions (VNFs) make informed decisions, and intelligently manage and update networking policies in cloud data centers. Actively collecting features and system state information entails substantial control signaling and management overhead, in particular in large-scale data center networks.

This chapter has proposed Aquarius, a framework that collects, infers, and supplies accurate networking state information with little additional processing latency, in a scalable buffer layout. By using multi-buffering and reservoir sampling, Aquarius extracts representative features from network traffic, and allows VNFs – in particular ML-based VNFs – to exploit these features. Aquarius can be deployed in the network on commodity CPU, empowering real-world learning algorithm deployments and evaluations. Following the methodology blueprint summarised in figure 3.35, this chapter has illustrated the use of Aquarius for various ML-based VNFs: traffic classification (offline, unsupervised learning), autoscaling (online, supervised learning), and load-balancing (reinforcement learning) purposes, and evaluates experimentally the impact of Aquarius in the system performance.

The application of ML techniques to networking problems starts from understanding the target problem to solve. Aquarius improves the visibility on the data plane and allows collecting a wide range of networking features for feature engineering, which iteratively prunes unrelated features to reduce additional feature collection processing latencies and selects the minimal set of viable features that can be gainfully used for the task. The selected features can be passed to both offline and online applications for data analysis, model training, and benchmark evaluations. Aquarius provides a reliable feature collection and experimenting platform in real-world systems that allows iteratively studying model selection, parameter tuning, and algorithm design for various use cases. Both open-loop (*e.g.,* supervised learning + autoscaling system) and close-loop (*e.g.,* reinforcement learning + load balancer) control can be achieved based on Aquarius to improve resource orchestration and utilisation. Extensive evaluations show that Aquarius helps bring significant performance gains – reduced flow completion time, improved resource utilisation – in the three considered cases of data-driven VNFs.

The results from this chapter have been published in [179, 194]. The source code and data of the Aquarius framework, and both simulation and experimental evaluations are open-sourced at https://github.com/ZhiyuanYaoJ/Aquarius. This chapter has shown that Aquarius is a key enabler for **data-driven network functions** that are ready to be deployed in real-world data plane.

Part III

Load-Balancing

Chapter 4

Charon: Load-Aware Load-Balancing in P4

As discussed in section 1.1, data centers are expected to manage significant volume of flows [64, 224], and users expect an elevated server responsiveness [58]. To provide this degree of responsiveness, applications are replicated into multiple virtualized instances in data centers to provide scalable services [63,237]. A given service provided in a data center is identified by virtual IP (VIP). Each application instance behind the VIP is identified by direct IP (DIP). In this architecture, load balancers (LBs) play an important role. They distribute requests from clients among application servers and maintains per-connection consistency (PCC) for each flow [64, 65], as introduced in section 1.2.1 in chapter 1.

This chapter exemplifies the challenges that LBs aim to handle by way of a simple heuristic load-balancing mechanism based on Equal-Cost Multi-Path (ECMP) used for load-balancing TCP requests. As is depicted in figure 4.1, on receipt of a new request (step ①), ECMP LBs randomly select a server from among the server pool to which the request is forwarded (step ②). This selection is based on the hash over the 5-tuple of the flow[1]. The replies for the service are sent back directly to the client instead of traversing the LBs (step ③) in what is called "direct source return" (DSR) mode. DSR mode was proposed in [64] so that LBs avoid handling 2-way traffic, and avoid becoming a throughput bottleneck between servers and clients. Though easy to implement, ECMP is agnostic to the server load states: ECMP randomly distributes workloads, thus new requests may be forwarded to overloaded servers, reducing load-balancing fairness. ECMP is also not able to guarantee PCC since server pool updates may cause changes to the DIP entries in the hash table, potentially forwards subsequent packets of connected flows to be sent to different servers, and breaks connected flows.

Statement of Purpose

This chapter proposes Charon, a stateless, load-aware, hardware load balancer, which addresses 3 key aspects of load-balancing performance as follows:

- *Availability*: encapsulates the chosen server id in a "covert channel" within packet headers. Different "covert channels" are available (*e.g.,* `flow-id` of QUIC flows and the least significant bits of IPv6 addresses) [144]. This chapter uses the higher bits of TCP timestamp options.
- *Fairness*: Charon makes load-balancing decisions on *predicted* server load states based on passive feedback from the application servers with actual load states encoded in `SYNACK` packets. Two factors are integrated at the same time, *i.e.,* queue lengths and processing speed.
- *Performance*: Charon implements all functionalities on programmable hardware, so as to maximize performance and achieve low latency and high throughput.

Virtual simulations show promising results and performance gain using Charon. Physical testing also demonstrates the high throughput of the solution when implemented.

[1] The 5-tuple corresponds to IP source, IP destination, protocol number, TCP source port, and TCP destination port.

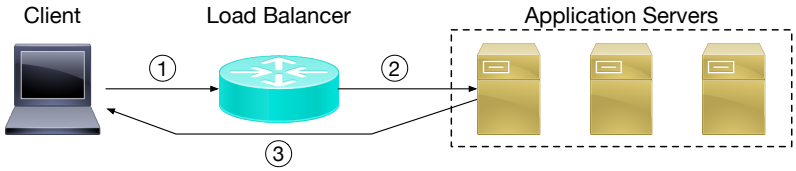


Figure 4.1: Network load balancer in data centers.

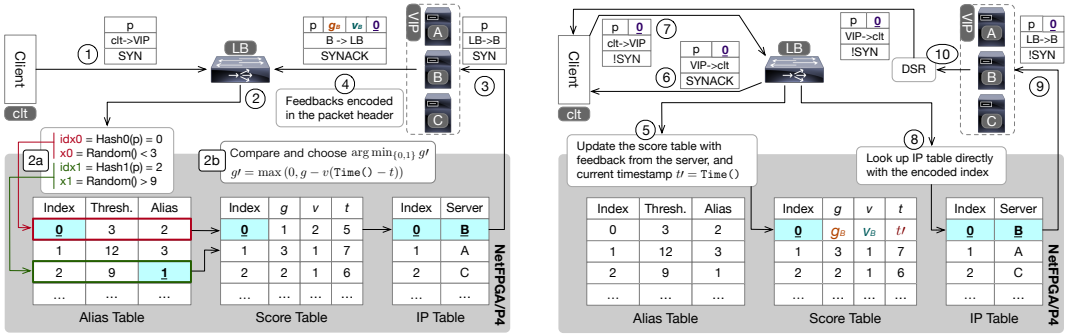


Figure 4.2: Charon overview.

Related Work

PCC needs to be ensured by LBs. The alternative is that, flows will break, and re-establishments may occur, which take time and degrade the quality of service (QoS) [238, 239], thus potentially causing revenue loss for cloud providers [240]. PCC can be achieved by LB algorithms from two categories: *stateful* and *stateless*.

Stateful LBs [59, 64, 65, 190, 235, 241] use flow tables to store mappings between flow IDs (*e.g.,* 5-tuple hashes[2]) and servers. To guarantee PCC, stateful LBs track the state of the flows they forward [64, 65, 135, 143]. Advanced hashing mechanism (*e.g.,* consistent hashing [65, 135]) permits that server pool updates have little impact on the hashing table – and therefore the amount of disrupted flows when servers are added or removed is minimized. However, stateful LBs require additional memory, for flow tables to store flow states. When encountering DoS attacks, flow tables risk of overflowing, and as a result they no longer track all valid flows. Also, in case of LB failures, the flow states are lost, and all flows via the failed LBs need to be re-established. This degrades the QoS. Stateless LBs [192, 193, 217] use alternative techniques to recover the right server destinations, without monitoring flow states. They encode flow-server mapping information in "covert channels" (*e.g.,* TCP timestamps) [144], or delegate the task of redirecting misrouted packets to servers [192, 193, 242, 243]. SHELL [217], Faild [193], and Beamer [192] daisy-chain two possible server candidates, to retrieve a potentially changed flow-server mappings.

Charon adopts stateless load-balancing scheme [144, 193, 217] by encapsulating the server id inside the packet. In particular, the TCP timestamp option [244] is used in Charon to transport this information.

To improve load-balancing fairness, different mechanisms are proposed to evaluate server load states before making load-balancing decisions. Segment Routing (SR) [245] and power-of-2-choice [246] are used in [135, 217] to daisy chain 2 servers, and let each decide independently, based on their actual load states, to which server a new flow is assigned. Another approach is to periodically poll servers' instant "available capacities" [143]. Ridge Regression is used in [187] to predict server load states and compute the relative "weights." In [186], the servers are clustered based on their load states. Clusters with less workload are prioritized. The servers notify the LBs about load state changes if their resource consumption surpasses pre-defined thresholds. LVS [236] presents a heuristic that combines the queue lengths of active flows and provisioned server capacity to determine server load states.

Unlike these approaches, Charon passively polls and retrieves the server load from a server

[2]TCP 5-tuple consists of source IP address, destation IP address, source port, destination port, and protocal number.

when a new flow is assigned to it. This is used to predict the future server load states and make informed – and fair – load-balancing decisions, to improve resource utilization and QoS.

To optimize performance in terms of throughput and latency, hardware solutions have also been proposed. SilkRoad implements LB functions on a dedicated hardware device [190], while other designs implement a hybrid solution combining software and hardware LBs [58, 59]. As a hardware solution, Charon is realized on a NetFPGA board using P4-NetFPGA tool-chain [247] and achieves low jitter and delay.

Chapter Outline

The rest of this chapter is organized as follows. In section 4.1 the overview of Charon is described. Section 4.2 presents the design choices of Charon. Section 4.3 presents the implementation of Charon, and section 4.4 shows the results obtained. Finally, section 4.5 summarizes the contribution of this chapter.

4.1 Overview

Charon relies on 3 tables and 1 server agent to achieve stateless load-aware load balancing on NetFPGA. 2 tables are constructed and managed by the control plane. The *Alias Table* allows selecting servers, based on various weights, with low computational complexity and low memory space consumption. The *IP Table* is used to map the server id to an actual IP address. The *Score Table*, is updated in the data plane on a per-flow basis.

The workflow is depicted in figure 4.2. When a `SYN` packet arrives at the LB (step ①), Charon employs *power-of-2-choices* and applies 2 hash functions to the 5-tuple of the packet. The 2 hashes are then used as indexes in the "Alias Method" [248] (step 2a) to generate 2 random server candidates, based on their relative weights. This is further detailed in section 4.2. Referring to the Score Table, Charon calculates and compares the load states of the 2 candidate servers (step 2b). The server with a lower score is assigned to the new flow. In the example of figure 4.2a, the DIP of the selected server is retrieved from the IP Table as server B (step ③). At step ④, along with the reply to the flow request, the agent on server B encapsulates its load state information and its server id in the packet header. In this chapter, the server load state is encoded inside the key option field of the GRE header [249], which encapsulates the original IP packet[3]. This "passive feedback" design differs from other LBs and reduces communication overhead concerning periodic polling mechanisms yet keeps LBs informed before application servers reach a critical load level. On reception of the `SYNACK` packet from the server (DSR is disabled for step ④), Charon updates the load state information in the Score Table. The packet is decapsulated and the response is forwarded back to the client (step ⑥). The server id (0 in the example) is preserved in the higher bits of the TCP timestamp option. In this way, the subsequent packets from the same flow (step ⑦) contain the server id, which helps Charon retrieve the server's IP address (step ⑧) from the IP Table and redirect immediately to the right server (step ⑨). The server can directly answer the client using DSR mode (step ⑩) till the end of the flow.

4.2 Design

The first building block of the design of Charon is the *Alias Method*. It is a probabilistic algorithm that, given initial weights, generates a table of probabilities and "aliases." The role of the Alias Method is to distribute – with higher probability – the flows to servers with higher weights. The weights are derived from the instant load states of the servers and are updated periodically. The update time interval is 1s, and this design choice is explained in section 4.4.

As shown in figure 4.2a, each entry of the Alias Table has a *threshold* and an *alias*. The former determines which value is chosen, while the latter is the alternative index with respect to the entry initially selected. Generating a server candidate requires 2 input values, *i.e.,* an entry index and a random number. If the random number is bigger than the threshold, the output of the Alias Method is the alias, otherwise the entry index. In the given example, four values are taken into considerations when generating 2 server candidates. The `idx0=0` and `idx1=2` are the two initial

[3]IPv6's flow-id field can also be exploited to store server load information. Charon uses the key option field of the GRE header to achieve better compatibility between IPv4 and IPv6.

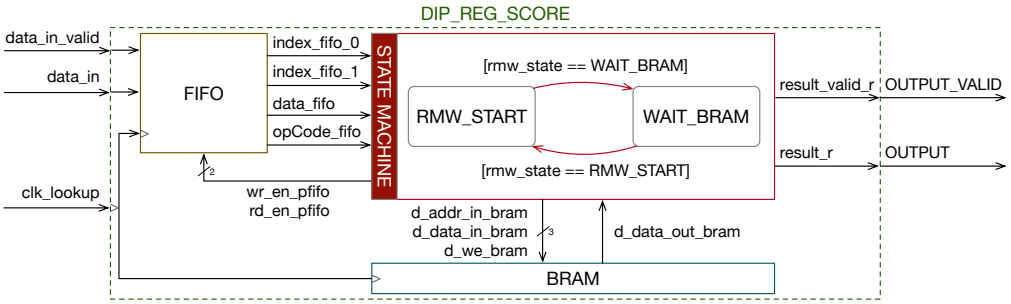


Figure 4.3: Schematic of `dip_reg_score` module.

entry indexes of the table. Given that the random value is smaller than the threshold `x0` < 3, the first output is the entry index 0. Similarly, since `x1` ≥ 9, the second output is the alias 1.

The 2 values obtained from the Alias Method are then used as the ids of the 2 server candidates. Their associated scores are computed with the function $g' = max(0, g - v * (Time() - t))$, where g' is the new score, g is the previous score of the server, v is the "velocity" or the server processing speed, $Time()$ is a function that returns current timestamp and t is the previous timestamp. The 3 variables, g, v, and t, are saved in the Score Table. The score g is the amount of work remaining or the number of active flows on the server to execute. The processing speed v is derived from the average flow completion time (FCT) on the server side. The timestamp t corresponds to the last time the score was updated. The time difference `Time()` $- t$ measures the elapsed time since the last update. The intuition of this function is to predict the remaining amount of tasks or queue length that a server needs to accomplish. A higher score translates into a busier server. The $max()$ function guarantees that the score stays non-negative. Once the scores of the servers are computed, the server with a lower score is assigned to the flow. In the example in figure 4.2a, supposing that $Time() = 8$ then the scores of index 0 and 1 are $g'_0 = max(0, 1 - 2 * (8 - 5)) = 0$ and $g'_1 = max(0, 3 - 1 * (8 - 7)) = 2$ respectively. The selected server is the server with index 0, which is then mapped to server B in the IP Table. The power-of-2-choices is applied as it has lower computational complexity than calculating the minimum yet it offers recognizable performance gains [135]. This increases the applicability of Charon for load balancing in large-scale data centers.

4.3 Implementation

The main functions of Charon are implemented in a single Verilog module called `dip_reg_score`. This design choice depends mainly on the read and write actions that should be executed on multiple indexes. Figure 4.3 shows the architecture of this module. It takes as input `data_in_valid`, `data_in` and `clk_lookup` and as output `OUTPUT_VALID` and `OUTPUT`. The core logic of `dip_reg_score` locates in the `STATE_MACHINE` block. It interacts with `FIFO` and `BRAM` (Block RAM). The former receives and stores the inputs of the block, while the latter is used to save the server load states information. Figure 4.4 depicts in detail the workflow of `STATE_MACHINE`. It consists of two states `RMW_START` and `WAIT_BRAM`. Each state can be further decomposed into several states. The initial state in `RMW_START` is `READ_FIFO`. In this state, the input saved in the FIFO is extracted. Depending on the `opCode` value, different operations can be executed. Three operations are available: `UPDATE_OP`, `COPY_OP` and `GET_INDEX_OP`. The code `UPDATE_OP` is used to update the server load states in the Score Table given the feedback extracted from the `SYNACK` packets sent by the application servers. The code `COPY_OP` is a buffering operation. It copies the data received from the input into the output. The code `GET_INDEX_OP` is executed when a `SYN` packet reaches the LB. It extracts the server load states with the 2 given server indexes. The reading operations require 2 clocks for each value, which yields 4 clocks in total for reading 2 blobs from `BRAM`. In the `WAIT_BRAM` state, with the fetched blobs, the two scores are then computed, and stored in the Score Table. The server with a lower score is selected and its corresponding score is increased by 1 unit task so that the new flow can be taken into account immediately. The two scores are then stored back in the Score Table before Charon forwards the flow to the chosen server.

Charon is implemented using a P4-NetFPGA framework. P4 is a programming language in the family of Software-Defined Networking (SDN) [4]. It is used to program network device data planes and has been used for implementing other LBs [187, 190, 192]. P4 being flexible, it can be

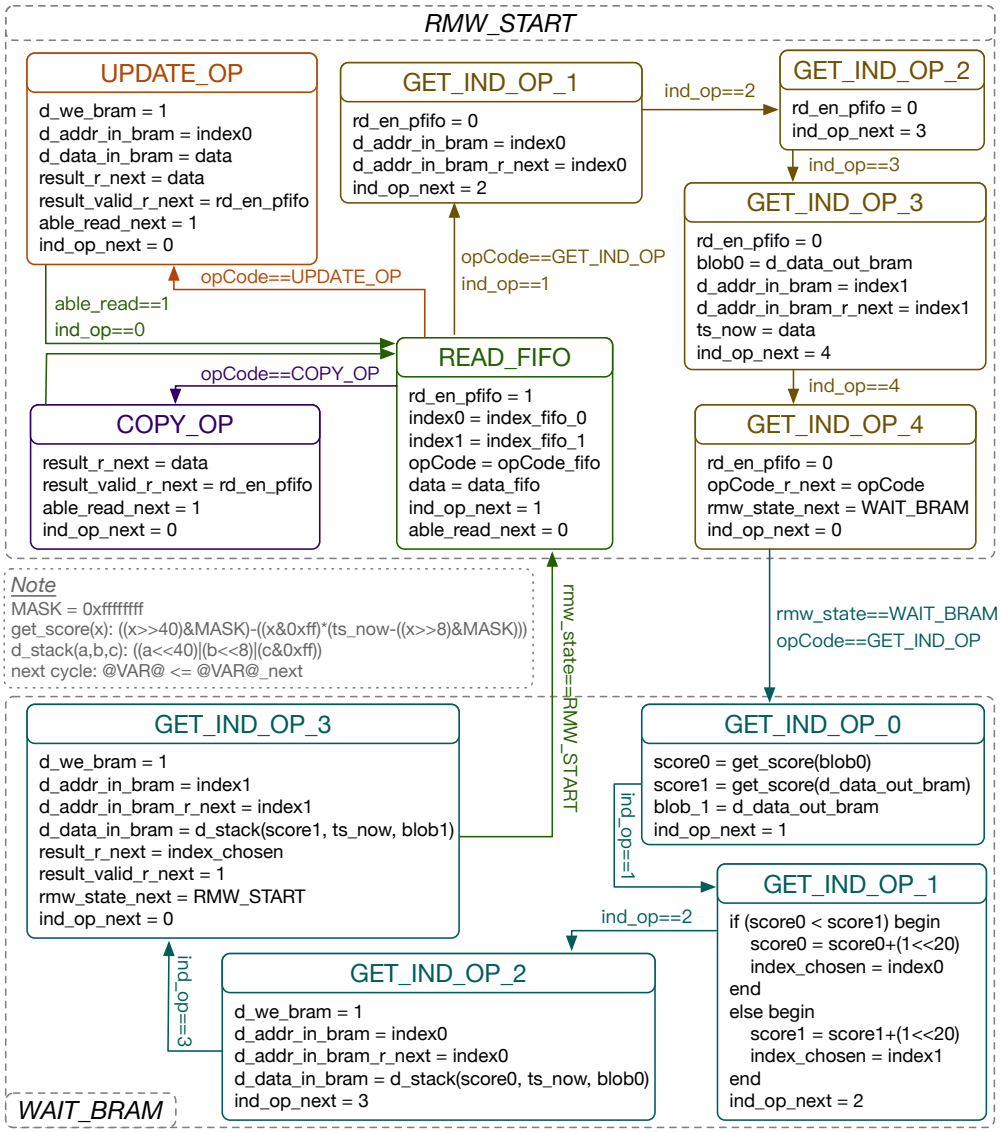


Figure 4.4: State machine in the `dip_reg_score` module.

translated into Verilog – and the created module can be compiled into bitstream files using Vivado toolkit [250].

Figure 4.5 shows the workflow of Charon in the PISA model of P4. It can be split into 3 parts: *Parser*, *Match-Action Pipeline* (MAP) and *Deparser*. The Parser separates different packet headers, depending on the values of different fields. Packets with unexpected packet headers are dropped while the others will be forwarded to the MAP. In MAP, the main logic of P4 takes place using tables, which are a collection of keys and values. A possible input could be an IP address. Using the longest-prefix match, one key is matched, associated with which an action can be executed as, for instance, setting the egress port value. Multiple tables can be applied during one packet processing. Charon uses 2 tables, one for each of the two server indexes obtained from the Alias Method. The last step is the Deparser, which assembles all the header information and sends a new packet out of the egress port.

Despite the flexibility of P4, it presents several limitations. For instance, an external function is necessary to create memory cells to store additional information. Depending on the hardware targets, different languages can be used to describe these external modules. Verilog is used for the NetFPGA target, which is the reason why the external module `dip_reg_score` is implemented in Verilog. Its complexity cannot be expressed using the P4 language. This block is mainly accessed for `SYN` and `SYNACK` packets. For other types of packets, it is simply used as a buffer. The other external modules of Charon implemented in Verilog are the IP table, current timestamp calculation, and server id extraction from the TCP timestamp option. The IP table is a fundamental component

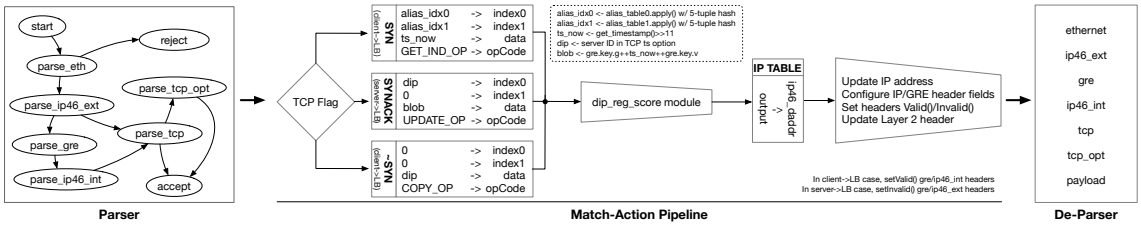


Figure 4.5: P4 workflow.

used to redirect packets coming from the client. The timestamp calculation takes place when a score computation or update happens. Server id extraction is used for any packet traversing the LB with the presence of TCP timestamp option, except for `SYN` packets because in this case the LB has not yet assigned any server to the flow.

Some design choices were made for these external blocks. First, the target number of servers is set to 16, which yields $\mathcal{O}(16)$ memory space complexity with the 3 tables[4]. Next, the FIFO memory inside the `dip_reg_score` module has a queue length of 64. As a small-scale prototype implementation, the Vivado simulations have been applied only to 1 of the 4 possible Ethernet interfaces. Another assumption of this chapter concerns the server id encapsulation in the TCP timestamp option. To be able to encode up to 16 servers, the server id uses the 4 higher-bit of the total 32 bits timestamp value. However, it is possible to increase the amount of servers to 256[5]. To simplify the P4 code, the only option considered in the TCP header is the timestamp option.

Finally, the server agent is then implemented as a VPP plugin [121] on each application server, which uses an Apache HTTP. This VPP plugin corresponds to a modified version of GRE, which encodes the instant server load state in the key option field and encapsulates the original IP packet. The number of Apache busy threads, which can be retrieved from Apache scoreboard is used in this chapter as server load state and the score of the server. Other metrics for the score could be applied for different applications.

4.4 Evaluation

This section evaluates Charon on, (i) the acceptance rate of covert channel modification in packet headers, (ii) the performance gain in terms of load balancing fairness and QoS, and (iii) the throughput and additional processing latency using P4-NetFPGA implementation.

4.4.1 Covert Channel Acceptance

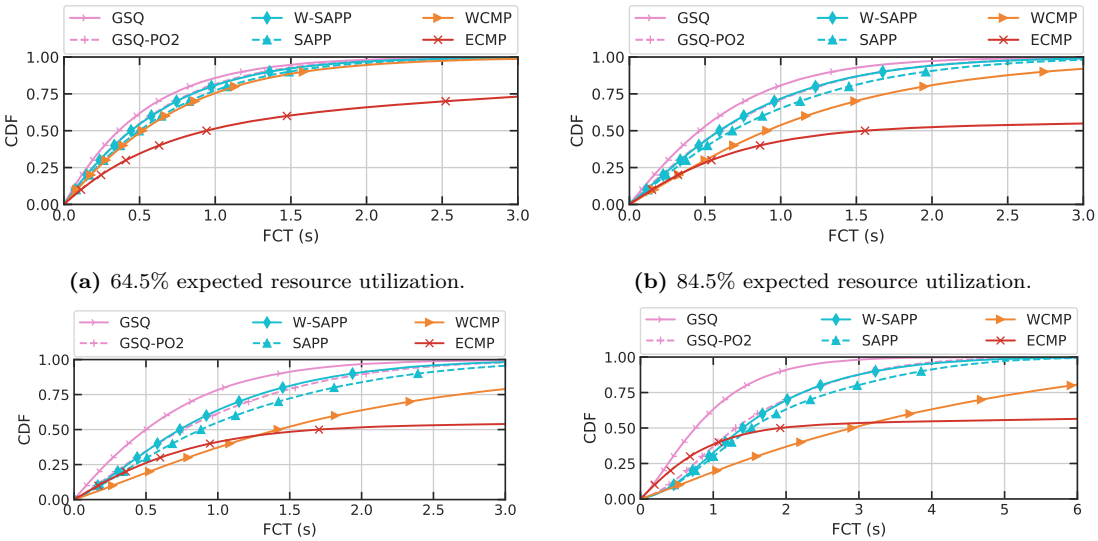
To understand how the Internet would react to the presence of the timestamp option in the TCP header, an initial experiment is conducted, where requests are sent from Paris to random sets of over 60k distinct IP addresses, and the connection success rates were recorded. The results obtained read as follows:

- `NO CONNECTION` = 45019
- `SUCCESS` = 12876
- `FAILURE` = 5787
- `TOTAL` = 63682

`NO CONNECTION` indicates the number of flows which have not received any response regardless of the presence of the timestamp option – likely because no host with the destination IP address is present on the Internet. `SUCCESS` indicates the number of flows that have answered to a packet with

[4] 16 is small yet can be updated at ease.

[5] The assumed maximum number of bits used for encoding server id is 8, *i.e.,* 256 servers in total, which is sufficient for modern data centers [224]. Any change in the timestamp value that modifies more than 24 bits is ignored.



(a) 64.5% expected resource utilization.

(b) 84.5% expected resource utilization.

(c) 92.5% expected resource utilization.

(d) 100.0% expected resource utilization.

Figure 4.6: Flow completion time (FCT) when subjected to different expected resource utilization.

the timestamp option. `FAILURE` indicates the number of flows that have not answered to a packet with the timestamp option, but which answered to packets without timestamp option. Excluding the `NO CONNECTION` cases, and analyzing only `SUCCESS` and `FAILURE` gives an acceptance rate of the timestamp option of 68.99%. This experiment does not study the different geographic locations of the clients and servers, or other factors, yet it validates that the stateless design of Charon works for most end hosts. It is also confirmed by a high acceptance rate (over 86%) obtained by experimenting on a larger scale testbed in [251].

4.4.2 Load Balancing Fairness

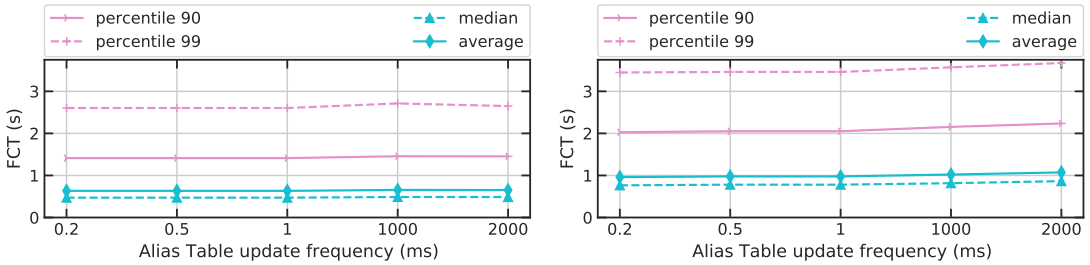
Charon was tested using a virtual simulator with 2 LBs and 64 application servers with different processing capacities[6], to study the load balancing performance in terms of workload distribution fairness. 3 episodes of 50k requests of clients for flows that last 500ms, on average, are simulated with Poisson traffic at different variances. The traffic rates are normalized by the total server cluster processing capacities. Figure 4.6a- 4.6d depicts the cumulative distribution functions (CDFs) of flow completion time (FCT), which is the time necessary to complete a flow. Different LB designs are compared. Global shortest queue (GSQ), as the name suggests, chooses the application server with the shortest queue. It is an oracle solution that can be achieved assuming that the LBs are aware of the actual queue lengths on each server and no computational overhead is incurred when computing the minimum queue length. It represents the best performance an LB can achieve. GSQ-PO2 applies power-of-2-choices over GSQ. Similar to GSQ, the LBs are assumed to be aware of the exact instant queue lengths on each server. Unlike GSQ, GSQ-PO2 selects 2 random server candidates and then pick the one with a shorter queue. This represents the theoretical best performance that Charon can achieve.

ECMP randomly selects the application servers and is widely employed as a load-balancing mechanism [64, 65, 144, 190, 235, 243]. Weighted Cost Multi-Path (WCMP) selects the application servers according to their statically configured weights which are proportional to the server processing capacities. W-SAPP denotes the implementation of Charon. SAPP corresponds to a simplified version of Charon, where the 2 server candidates are chosen using a uniform distribution instead of using weighted sampling with the Alias method. The main difference between SAPP and W-SAPP is therefore the probabilistic method that W-SAPP applies to obtain the 2 choices. The weights, which are used later as probabilities, are defined using the relative processing capacities of application servers. As depicted in figures 4.6a-4.6d, the performance of both SAPP and W-SAPP is similar to the performance of GSQ, which is considered as the method that takes the perfect

[6]Half of the application servers have 2 times higher processing capacities than the other half.

Utilization	GSQ	GSQ2	W-SAPP	SAPP	WCMP	ECMP
64.5%	**0.59**	0.54	0.52	0.51	0.43	0.34
76.5%	**0.64**	0.59	0.56	0.57	0.47	0.47
84.5%	**0.68**	0.63	0.61	0.60	0.49	0.49
92.5%	**0.69**	0.67	0.66	0.66	0.54	0.51
100%	0.75	0.74	**0.76**	0.74	0.63	0.52

Table 4.1: Jain's fairness indexes of different LBs at different traffic rates.



(a) 64.5% expected resource utilization.

(b) 92.5% expected resource utilization.

Figure 4.7: FCT of different Alias Table update interval LB designs at various traffic rate.

choice. Improvements can be observed for W-SAPP over SAPP, which is the reason why the Alias Method is used in Charon. Despite its limited additional complexity, W-SAPP would be able to capture the different capacities of servers.

Another interesting metric for evaluating load balancing fairness is Jain's fairness index [252], which computes the fairness of workload distribution. Considering n servers each one with a particular amount of flows x_i, the fairness index is computed as $\left(\frac{\sum_{i=1}^{n} x_i}{n}\right)^2 \cdot \left(\frac{\sum_{i=1}^{n} x_i^2}{n^2}\right)^{-1}$. The maximum and minimum values that the index can reach are respectively 1 and $\frac{1}{n}$ with n as the number of servers. If the index reaches the value 1, it means that the load has been fairly distributed. The worst case is when the index is equal to $\frac{1}{n}$ which proves that only one server has taken all the flows.

Using the same configuration as in previous simulations, the fairness indexes of different LB designs are computed. These values have been obtained by periodically computing the fairness over the remaining workload of each of the application server during the simulation execution. These results also take the different capacities of the servers into consideration. As shown in table 4.1, ECMP and WCMP have the worst performance. Random choices do not guarantee a fair distribution of flows. In addition, GSQ and GSQ-PO2 get the best fairness: they have perfect knowledge of the flows, and they always choose the server with the shortest queue length. W-SAPP and SAPP achieve similar performance to GSQ and GSQ-PO2. Although the fairness indexes of SAPP and W-SAPP are not so different, W-SAPP takes into account the processing capacities of the servers which can be useful when their capacities are different inside the same server cluster. The Alias Method in Charon, however, uses dynamic weights to select a subset of candidates.

Another important parameter to analyze is the update time intervals of the Alias Table. If the update time interval is too high, the LB choice would not reflect the real-time load states of the application servers. For this reason, different update time intervals are simulated.

The results of the simulations are depicted in figure 4.7. 4 values have been considered: percentile 90, percentile 99, median and average of FCT at 2 different rates. Five different time intervals of Alias Table updates have been used: 0.2 ms, 0.5 ms, 1 ms, 1 s, and 2 s. The plots show a slight improvement in FCT when the update time interval is lower (higher update frequency). This difference is too small to justify a shorter time interval update. The LBs are not significantly influenced by a real-time update of the weights. However, it has to be considered that these simulations are virtual. Physical experiments are necessary to further justify this assumption.

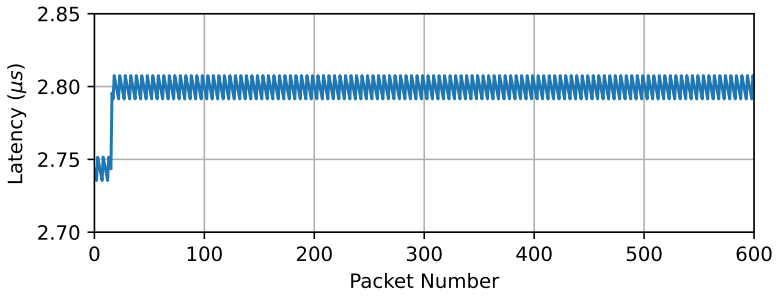


Figure 4.8: Delay in packet departure with respect to the number of packets sent.

4.4.3 P4-NetFPGA Implementation Performance

The per-packet processing performance of NetFPGA has been illustrated in figure 3.26. This chapter describes in detail how the evaluation is conducted, especially when comparing with software solutions. On the NetFPGA, the `reference_NIC` bitstream file is loaded. This program returns the packets that the NetFPGA receives from the Ethernet interface to the PCI interface on the motherboard. Similarly, a packet received on the PCI interface is then sent onto the Ethernet interface – *i.e.,* the NetFPGA behaves as a NIC. Using this, it is possible to estimate at which time the packet sent through the PCI traverses one of the four Ethernet interfaces and it reaches the host machine. Figure 3.26 shows the CDF of the latency. The Round-Trip Time (RTT) of ping packets of other software solutions are also depicted. In particular, four cases are considered: when there are two containers or two VMs on the same machine, or on different machines. The performance of the NetFPGA implementation largely outperforms the software solutions, which makes NetFPGA a preferred solution in terms of performance.

To demonstrate the performance of the Verilog module `dip_reg_score`, Vivado behavioural simulations have been executed. A burst of 600 packets is sent to the NetFPGA board. The delay between packet arrival and departure is shown in figure 4.8.

The difference between the first 16 packets and the remaining packets is due to the nature of the packets sent. The first batch of 16 packets traverses a shorter path as they upload the IP addresses in the IP table. The remaining packets are TCP `SYN` packets. They traverse the longest path in which the destination application server of the flow is chosen. The delay is almost linear with the number of cycles required for packet processing and stays constant. The sinusoidal shape is because of the jitter, which is low. This plot shows the high performance of the designed module and its efficiency in executing the proposed LB algorithm.

4.5 Summary

This chapter proposes Charon a stateless, load-aware, hardware load balancer in data centers, which (i) fairly distributes flow requests, (ii) avoids flow disruptions, and (iii) avoids additional latency due to its presence. The design choices of Charon make it suitable for implementation on programmable hardware. Simulation results show that Charon improves load balancing fairness and helps achieve a better quality of service than other load balancing mechanisms. Evaluations of throughput and processing latency demonstrate the advantage of hardware implementations. However, with limited and delayed observations, Charon does not outperform heuristic load balancing algorithms in terms of workload distribution fairness.

The results from this chapter have been published in [195]. The source code and data of simulation results are available under an open-source license at: https://github.com/ZhiyuanYaoJ/SimLB/tree/charon.

Chapter 5

MLB: Load Balancing with "Intelligence"

In data centers (DCs), cloud services and network applications are associated with server clusters to provide high scalability, availability, and quality of service (QoS) [63, 142]. As a key component for efficient resource utilization in DCs, Layer-4 *load-balancers* (LBs) distribute network traffic addressed to a given cloud service *evenly* on all associated servers, while *consistently* maintaining established flows [64, 65, 135, 143, 144]. As already demonstrated in Chapter 3, as well as in [17, 147], Machine Learning (ML) can help achieve performance gains in different networking problems. This chapter investigates challenges and potentials when applying ML algorithms to improve network load balancing performance, with various networking features extracted based on the Aquarius framework.

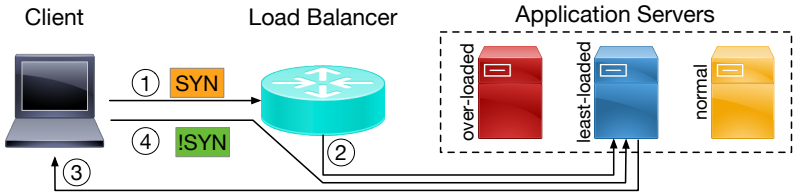


Figure 5.1: Workflow of Layer-4 load balancers in data center networks.

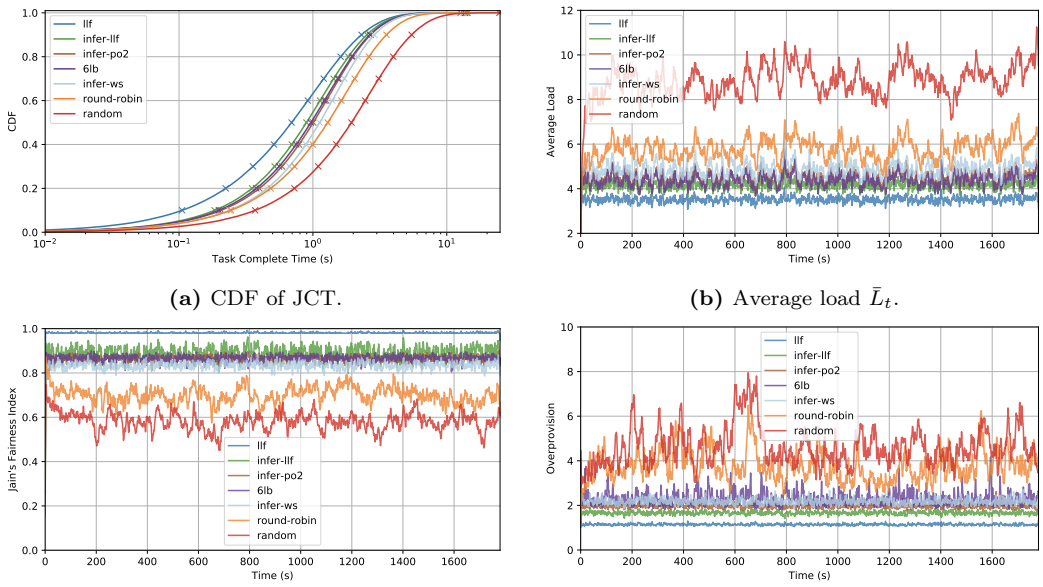
As a reminder, the workflow of network LBs is depicted in figure 5.1. On receipt of a new flow request ① (*e.g.,* a TCP SYN), LBs ② determine to which server the new flow is to be dispatched. Servers ③ respond to the request using direct-source-return (DSR) mode[1]; LBs thus have no access to the server-to-client side of communication. Finally, ④ the load-balancing decision – once made – is preserved until flow terminates.

Statement of Purpose

As discussed already in the related work section in chapter 4, per-connection consistency (PCC) of network flows has been explored in the literature [59, 64, 65, 144, 190, 192, 193, 221]. Yet many Layer-4 LBs in data centers [58, 64, 65, 190, 192, 193] do not consider application server load when distributing workloads, which can lead to suboptimal load-balancing fairness and resource utilization. Without considering real-time server utilization, heterogeneous traffic will potentially incur collisions of elephant flows on the same application server and lead to uneven load distribution. Hence, this chapter stresses the significance of LB load distribution fairness and investigates an ML-based scheme to help address this issue.

This chapter explores how the features extracted by Aquarius (chapter 3) can be exploited for network load balancing in data centers, and discusses the constraints as well as the tradeoffs between performance gain and additional complexity when applying ML algorithms. This chapter

[1] DSR is enabled for response packets from servers to clients to bypass LBs. It relieves LBs of handling 2-way traffic, improving network throughput [64].



(a) CDF of JCT.

(b) Average load $\bar{L}_t$.

(c) Jain's fairness index.

(d) Overprovision.

Figure 5.2: Simulation result with a sample setup.

also studies the potential benefits and challenges of the application to help improve load-balancing fairness by learning and extracting correlations between networking observations and available server resources.

Chapter Outline

The remainder of this chapter is organized as follows. Section 5.1 describes the network load-balancing problem in data centers, and presents a quantitative study based on simulations that demonstrate the challenges of improving load-balancing fairness. Section 5.2 describes the design of MLB and its load-balancing decision-making process. Section 5.3 describes the implementation details of the experimental setups. Section 5.4 presents quantitative evaluations, comparing and contrasting different ML algorithms load-balancing algorithms, by way of both offline and online applications. Section 5.5 finally concludes this chapter.

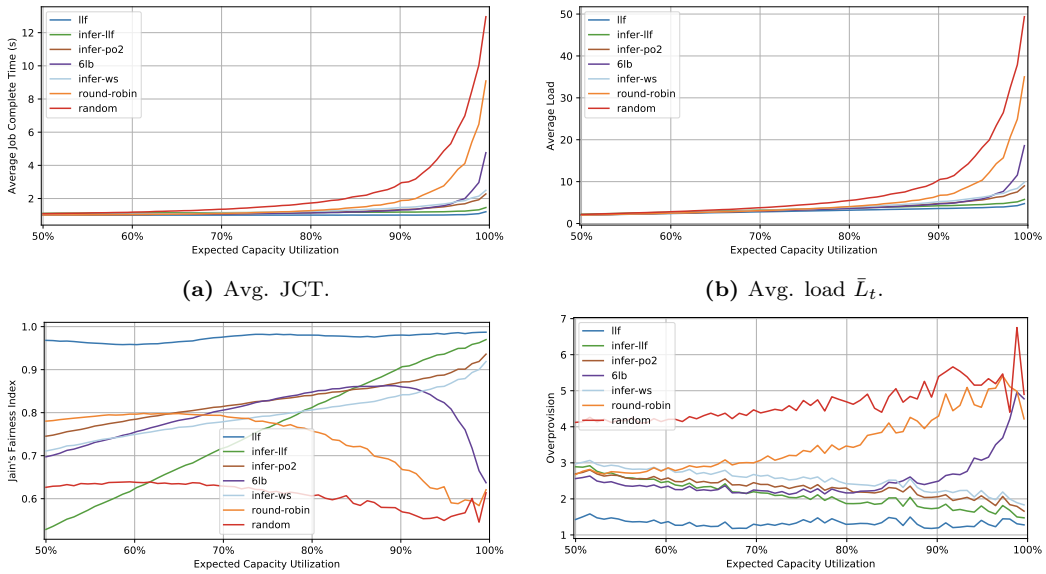
5.1 Background

An abstract way of understanding the process of load balancing is as a stream of tasks (each requiring a certain amount of processing time) dispatched towards a cluster of workers or servers, each with constrained capacities. The key motivation of this chapter is that making dispatching decisions based on the state of real-time resource utilization can not only improve quality of service (QoS), but also reduce cost and energy required. This can be demonstrated by a simple simulation of a service provided by a pool of servers with constrained capacity below.

Given a cluster of N servers $\mathbb{S} = \{s_i | i \in \{1, \ldots, N\}\}$, each with finite processing capacities $\mathcal{C}(S_i, t), i \in [1, N]$, and a Poisson stream of M jobs $\mathbb{J} = \{j_k | k \in \{1, \ldots, M\}\}$ with different estimated time to complete $T(j_k) \in (0, +\infty)$, as well as a Poisson arrival rate λ (*i.e.,* the number of arriving jobs per second), and constrained processing capacities of each server as $C(s_i) \in (0, +\infty)$, and an actual load of each server at timestamp t as $L_t(s_i)$. To simulate the nature of a DC network, where traffic follows heavy-tail distribution [1], the estimated time to complete of a job is assumed to follow an exponential distribution, with a standard deviation equal to its mean μ. With the arrival of a new job, j_k, the task of the LB is to dispatch j_k to a server s_i, which will enqueue j_k locally and keep processing jobs in its queue with its capacity $C(s_i)$.

Different task-dispatching strategies can be employed. Whenever a new task arrives, LB can:

- *randomly* select a server,



(a) Avg. JCT.

(b) Avg. load $\bar{L}_t$.

(c) Jain's fairness index.

(d) Avg. Overprovision.

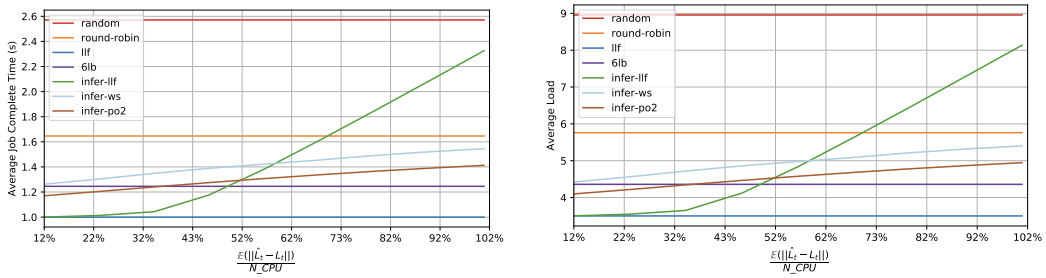
Figure 5.3: Simulation result with various expected capacity utilization.

- select a server in a *round-robin* fashion,
- "guess" the current load of each server $\hat{L}_t(s_i)$ and, use a dispatching method such as:
  - *least-loaded-first* (LLF): select a server with least load $(\arg\min_{s_i \in \mathbf{S}} \hat{L}_t(s_i))$,
  - *power-of-two* (Po2): randomly select two servers $(s_p, s_q \in \mathbf{S})$ and choose the less loaded one $(\arg\min_{s_i \in \{s_p, s_q\}} \hat{L}_t(s_i))$,
  - *weighted sampling* (WS): select a server in a weighted sampling scheme.

Among these, the last two are respectively implemented as the power of two [246] and the alias method [248]. When $\hat{L}_t(s_i) = L_t(s_i)$, optimal performance is expected to be achieved by adopting LLF. To evaluate and compare the performance of each dispatching strategy, four metrics can be employed to evaluate:

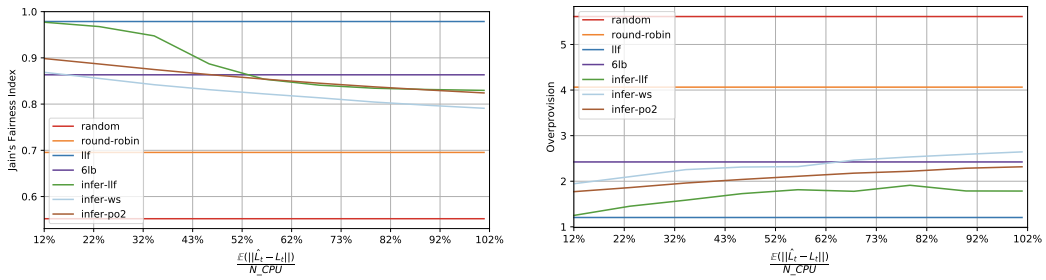
- quality of service: *job complete time* (JCT),
- cost and energy: *average load* $\bar{L_t} = \left(\frac{\sum L_t(s_i)}{N}\right)$,
- distribution fairness:
  - *Jain's fairness* $\left(\frac{\mathbb{E}(L_t(s_i))^2}{\mathbb{E}(L_t^2(s_i))} \in [0,1]\right)$ [252],
  - *overprovision* $\left(\frac{\max L_t(s_i)}{\bar{L_t}}\right)$.

The simulation results presented in figure 5.2 are of a sample setup, where $N = 128$ identical servers with capacity $C(s_i) = 4$ are subjected to a Poisson stream (traffic rate $\lambda = 450$) of $M = 80k$ jobs with average estimated time to complete $\bar{T}(j_k) = 1$. This setup challenges the server cluster with jobs that consumes potentially $\frac{\bar{T}(j_k)\lambda}{C(s_i)N} \approx 87.89\%$ of their total theoretical capacity. The optimal dispatching strategy `llf-optimal` assumes that the LB knows the exact current load of each server, such that all new queries are forwarded to the least-loaded server in the cluster. The "guessing" process is then simulated by adding Gaussian noise with a standard deviation (an error $\mathbb{E}(\|\hat{L}_t - L_t\|)$ which equals to $\frac{C(s_i)}{2} = 2$) to the actual ground truth. Figure 5.2 shows that, if LB can make load-balancing decisions based on the information of server load state, better performance can be achieved in terms of quality of service, average load, and distribution fairness. Even if



(a) Avg. JCT.

(b) Avg. load $\bar{L}_t$.



(c) Jain's fairness index.

(d) Avg. Overprovision.

Figure 5.4: Simulation result with various "guessing" error $\mathbb{E}(\|\hat{L}_t - L_t\|)$.

the information of server load state is inferred ("guessed" with bounded error), load-balancing performance can still be improved when compared with load-agnostic load-balancing methods.

As depicted in figure 5.3, with an increased load (or, expected capacity utilization, by varying the Poisson traffic rate λ), the performance gain becomes significant when load balancers are aware of server load states, even if the information is inferred with a bounded error. The average JCT and server load increase drastically when LB is load-agnostic. However, regardless of the "guessing" error $\mathbb{E}(\|\hat{L}_t - L_t\|) = 2$, the 3 load-balancing strategies, based on inference, reduce the JCT, and decrease the average server load, achieving performance similar to the optimal solution `llf`.

Adjusting the accuracy of the "guessing" process, by adding Gaussian noise with different standard deviation ($\mathbb{E}(\|\hat{L}_t - L_t\|)$), influences the performance of each load-balancing strategy, as depicted in figure 5.4. The performance of each of the 3 dispatching strategies (LLF, WS, Po2) degrades when the accuracy of inference decreases. Especially with LLF, when making *wrong* load-balancing decisions when undertake high server load inference error (*e.g.,* dispatching more tasks to servers which, according to the load balancer inference, have a lower load, however in reality they are already overloaded), a new task is potentially dispatched to a server which is actually more loaded than average, resulting in a worse performance as is shown in figure 5.4. The impact of wrong server load inference is somewhat neutralized by randomization with Po2 or WS. However, the benefit of load-aware load balancing using a lower server load inference error is evident.

5.1.1 From Simulator to Testbed

To implement, deploy, and compare the performance of different methods (Maglev [65], active-probing, and a simple heuristic counting number of ongoing flows), a set of experiments are conducted with a testbed[2], whose topology is shown in figure 5.5. A simple Poisson stream of queries with expected job complete time (JCT) following a heavy-tail distribution is employed. The following load balancing algorithms are then tested:

- *maglev*: Randomly dispatch new-coming flows to server cluster [65].
- *maglev-ws* (Maglev with weighted-sampling): Randomly dispatch new flows to server cluster, using a statically configured weighted-sampling based on the server resource allocation.

[2]The testbed configuration details can be found in section 3.2.3.

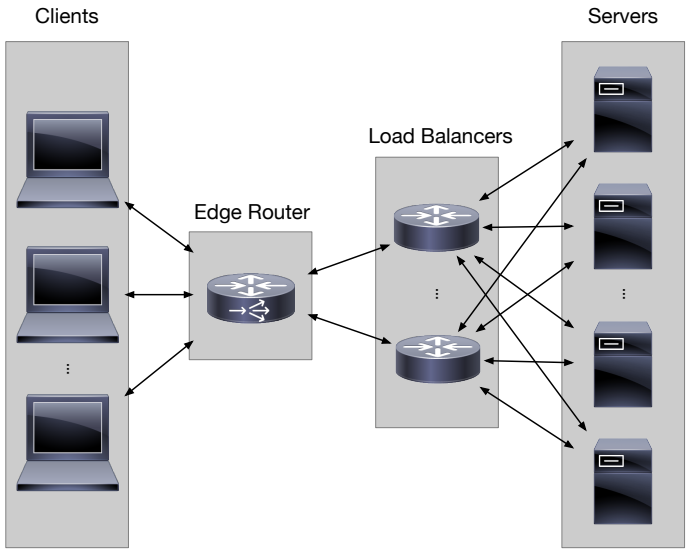
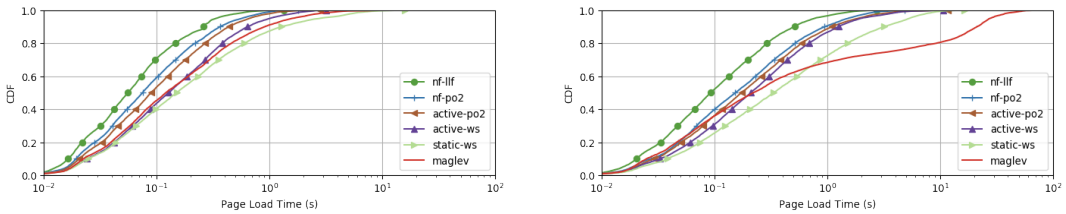


Figure 5.5: A representative network topology of DC networks with LBs.



(a) Without colocated workload

(b) With colocated workload

Figure 5.6: A sample experiment result.

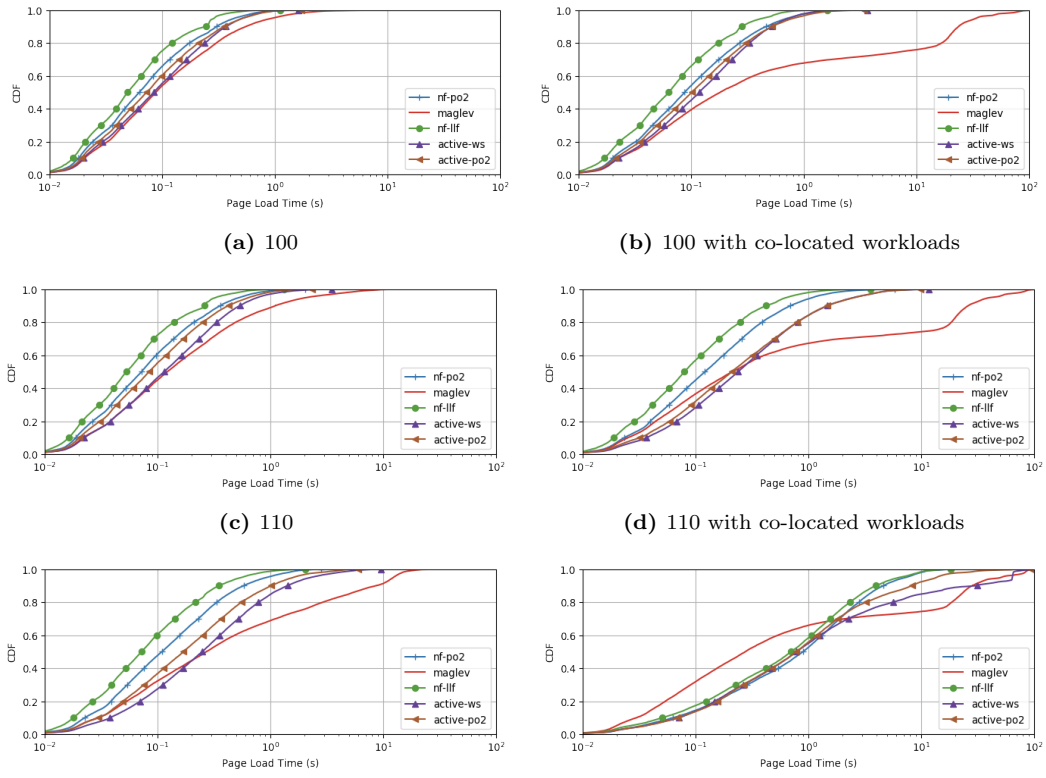
- *active-ws* (active probing with weighted-sampling): Actively probe server load (*i.e.,* the number of busy Apache threads), based on which a list of weights are generated and new flows are dispatched with weighted-sampling.

- *active-po2* (Active probing with power-of-2): Actively probe server load (*i.e.,* the number of busy Apache threads) and randomly (with consistent hashing) select 2 servers, then choose the least-loaded from among these. Presumably, this method outperforms 6LB [135] as long as the probing frequency is high enough, and as long as there are no extreme bursts of new flows. This is because 6LB daisy-chains 2 servers and offloads the decision-making process to the servers, which can compare the predefined "overload" threshold with their actual load states. 6LB improves load-balancing performance only when the second server in the chain is less loaded than the first server. Otherwise, the request can not be sent back to the first server, therefore it ends up with the heavier-loaded server.

- *nf-po2* (Counting number of ongoing flows with power-of-2): Keep recording the number of ongoing flows to each server and randomly (with consistent hashing) select 2 servers, and choose the one that currently serves the least ongoing flows.

- *nf-llf* (Counting number of ongoing flows with least-loaded-first): Keep recording the number of ongoing flows to each server and choose the one with the least ongoing flows.

The first experiment is conducted with single LB and Poisson traffic with heavy-tail distributed tasks. Co-located workloads are applied in one scenario on half of the servers to emulate, a data center environment where servers can have heterogeneous architectures, or multiple different services run on shared hardware resources.



(a) 100 **(b)** 100 with co-located workloads

(c) 110 **(d)** 110 with co-located workloads

(e) 120 **(f)** 120 with co-located workloads

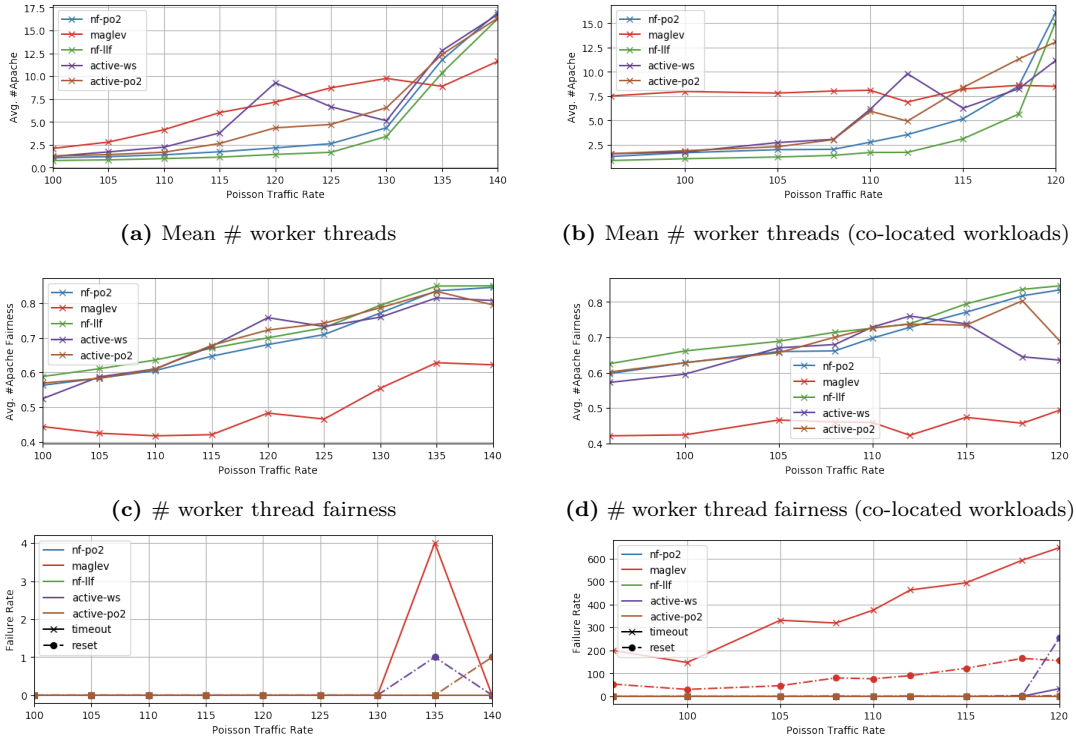
Figure 5.7: Page load time CDF with different traffic rate without (upper) or with (lower) co-located workload

As depicted in figure 5.6, Maglev (with weighted sampling) is very sensitive to co-located workloads, when resources are not even. Both active probing (labeled as `active` in the plot) and counting the number of established flows (labeled `nf` in the plot) can largely prevent performance degradation in terms of page load time. Between the two, `nf` always outperforms `active` since with a single LB, the metric that `nf` uses to evaluate the current server load state (number of established flows) accurately reflects the actual number of threads on each server, whereas `active` requires at least 1 round-trip time between LB and server for accurate and timely observation of server load.

Experiments with different traffic rates are are depicted in figure 5.7. With a higher traffic rate, the advantage of counting the number of flows and of active probing becomes more significant – up to a point where the rate is so high that server capacity is saturated. With co-located workloads (constrained capacity), the pattern is similar. However, the performance improvement is limited when the traffic rate grows high: neither `nf` nor `active` can capture the actual server capacity difference (8 servers with 2 CPUs and 4 servers with 1 CPUs).

Figure 5.8 shows that Maglev is not able to allocate resources according to the actual workload, especially when co-located workloads are present on the servers. However, both `active` and `nf` help LB minimize average load when traffic rate is low, and take fully advantage of all server capacities when traffic rate becomes high. The lower average number of (Apache) worker threads when using Maglev is because some servers are overloaded while others are starving. Both `nf` and `active` approaches can effectively reduce timeouts or connection resets caused by heavy traffic.

Since multiple LBs are deployed in data center networks to avoid a single point of failure, another point to investigate is the load-balancing performance when the LB has a global or local observation. Figure 5.9 demonstrates that (i) `nf-llf` is more sensitive to local observations – biased server state measurements lead to its performance degradation; (ii) `nf-po2` is more resilient to local observations, however, it is less resilient comparing with active approach, which can obtain global observations.



(a) Mean # worker threads

(b) Mean # worker threads (co-located workloads)

(c) # worker thread fairness

(d) # worker thread fairness (co-located workloads)

(e) Failure cases

(f) Failure cases (co-located workloads)

Figure 5.8: Comparison with different traffic rate on other metrics

5.1.2 Challenges

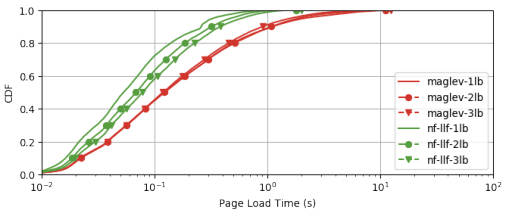
The simulation and experimental testbed results in section 5.1.2 demonstrate that load-balancing performance (*i.e.,* the average flow complete time, resource utilization, load distribution fairness, and overprovision) can be improved when the load balancer is aware of its corresponding servers.

To this end, Machine Learning (ML) is increasingly applied in the field of networking [147], to infer information about the global system and network states from local observations. Recurrent neural networks (RNN), built for analyzing time-series datasets, have shown feasible for processing and analyzing networking observations, within the field of computer networking [21, 22]. However, the high computational complexity required to run advanced and complicated ML models hinders their application in the field of high-performance network [26], where a high degree of reactivity, and "line speed" communications are expected.

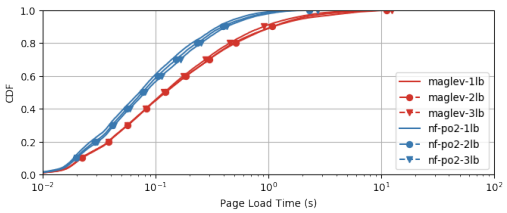
For instance, consider a simple 3 layer neural network with 4865 parameters, built Keras/Tensorflow [234]. If input datapoints have 32 features, to obtain an output from the neural network requires around $100ms$ processing time ($10Hz$), using single CPU core (Intel Xeon CPU E5-2690 v3 at 2.60GHz), as depicted in figure 5.10[3]. This result shows the difficulty of applying off-the-shelf ML techniques to per-flow based L4 LB for DC in production, where flows arrive with much higher frequency than $10Hz$ (as shown in [253] where packets can arrive at "line speed" – 1Gbps – in data center networks). As a result, the capacities of ML models (*e.g.,* number of parameters) applicable off-the-shelf are limited.

Apart from the computational complexity of ML, server load inference is expected to be adaptive to incoming traffic. In a DC network setup, LBs are expected to handle requests with various distribution, for different applications running on the servers, as well as diurnal and nocturnal patterns of user activities [254]. Hence, ML schemes deploy offline-trained ML models for online tasks, "hoping" that training and testing datasets share similar distributions. This however does not apply for the load-balancing problem in DC networks. Continual lifelong learning (CLL) [255] is one technique for adapting to a stream of various incoming tasks. However, this requires not

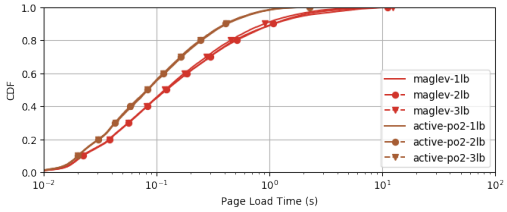
[3]The measurement is conducted on CPU machines since no Cisco device is equipped with GPU.



(a) nf-llf with different # LB



(b) nf-po2 with different # LB



(c) active-po2 with different # LB

Figure 5.9: Comparison with different number of LB devices

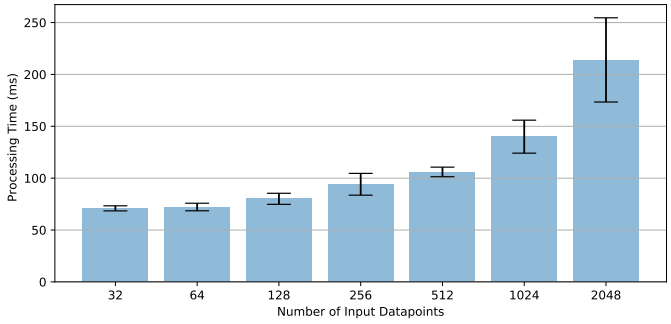


Figure 5.10: Inference latency using 3-layer neural networks.

only identifying and classifying different tasks, but also incrementing the capacity of ML models (*e.g.,* adding hidden neurons or layers to the original models) as more types of tasks are introduced. Another option to achieve adaptivity is to use reinforcement learning (RL) [256]. RL model learns a policy based on previously observed states and rewards and has achieved great performance in complicated environments together with deep neural networks (DNN) [257]. However, RL requires careful design of reward function and an optimization algorithm to achieve fast convergence. This will be studied further in chapter 7.

Last but not least, it is not a trivial task to integrate the server load inference engine into LB, without disrupting its intrinsic properties, *e.g.,* PCC, availability, robustness, etc. For instance, an LB should guarantee that the same flow will always be directed to the same application server once a decision has been made, regardless of the latest server-load circumstance. When adopting weighted hashing techniques [241, 258] for making load-aware dispatching decisions, it requires reconstructing the hash table, which may cause inconsistent mapping between network flows and servers when the weights need to be updated frequently.

To summarize, MLB needs to resolve the following key challenges:

- minimizing inference error to avoid making wrongful server-load dispatching decisions (regardless of local observations),
- minimizing potential performance and management overhead of the inference engine (tradeoff between sampling frequency and performance),
- adapting to different and/or time-variant incoming traffic (stream of tasks),
- retaining the properties of LB – PCC, availability, robustness, etc.

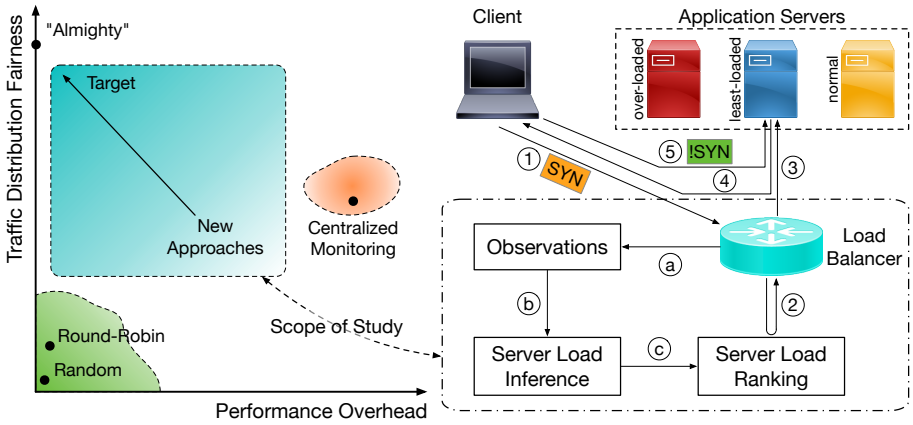


Figure 5.11: Overview: the scope of study is to find new approaches to load-aware load-balancing so as to improve traffic distribution fairness meanwhile restrict performance overhead.

5.2 Design

The proposed architecture of MLB is depicted in figure 5.11: ① whenever receiving new flow request (*e.g.* TCP SYNs), ② the load-balancer looks up the table of predicted server load, updated concurrently by steps ⓐ-ⓒ. Based on this, the load-balancer is able to make a load-balancing decision – selecting predicted least-loaded application server based on local observations, and determining to which application server the new flow is directed in step ③. Then ④ the load-balancer constructs a mapping between the server and network flow, hence ⑤ subsequent packets from the client can directly be forwarded to the corresponding server without further decisions. Based on Aquarius, this workflow follows the principles laid out in section 3.3.3.

While purely passive (*i.e.,* random and round-robin) load-balancing strategies and active approaches sit at the two ends of the trade-off between traffic distribution fairness and performance overhead, MLB strives for an optimal solution by exploring networking features that could be observed by LB and be used to infer server load (section 5.2.1).

5.2.1 Networking Features: From LB's Perspective

As a "middlebox", in between clients and servers, an LB could observe 2-way traffic, which allows LB to estimate processing time for each server, and therefore infer server load and make load-aware load-balancing decisions. However, as discussed in chapter 4, the Direct Source Return (DSR) mode can be enabled in most LB to improve system throughput. The flows from servers to clients can be (and often are) up to several times bigger than the incoming requests from clients to servers. In this case, the risk of the load balancer becoming a bottleneck increases considerably. Hence the role of DSR, which modifies the traffic flow by permitting the server to respond directly to the client, is to relieve the network load balancer of the need to handle the heavy traffic load. A consequence of this is, that the LB limits the insights that can be obtained by observation of the traffic by the LB since only traffic from the client can be observed. Therefore, the first challenge is to extract features that can embed server load information, from the network traffic that the LB sees.

One feature to infer server load for TCP traffic is to count the number of SYNs and FINs (or RST) that the LB has seen, to estimate the number of established flows (`#flows`). As is shown in figure 5.12, when there is only one LB, `#flows` reflects the number of busy threads on the server. However, as is discussed in section 5.1.2, when there are multiple LBs, it is not necessarily the case. It is visible in figure 5.12 at the bottom. Even though the summarized `#flows` correlates with the actual number of busy threads on the server, the observation on the second LB (green) loses track of the actual server load (especially during the period from 40 to 60 seconds). Also, this feature may not be robust when faced SYN flood attacks. Therefore, it is necessary to explore various types of networking features, to augment the information that can be extracted from the data plane for making inferences about the system and network states in different scenarios.

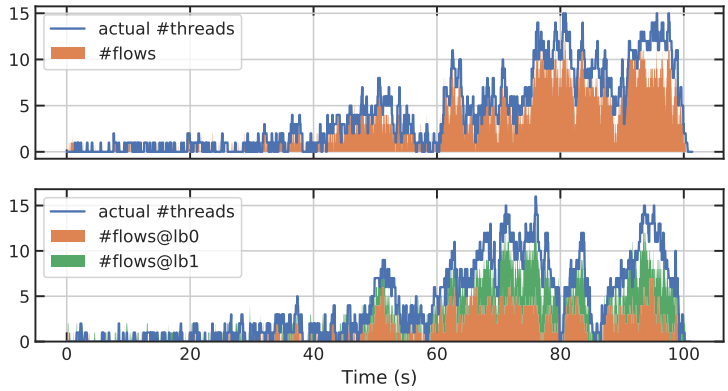


Figure 5.12: A simple heuristics counting number of SYNs and FINs to infer the current load on the server.

Type	Notation	Observation
Temporal	iat_f	flow inter-arrival time
	iat_p	flow-agnostic packet inter-arrival time
	iat_{ppf}	per-flow packet inter-arrival time
	fct	flow complete time
	lat_{synack}	SYN to first ACK latency
	pt_{1st}	first data packet processing time derived from TCP `tsecr`
	pt_{gen}	general data packet processing time derived from TCP `tsecr`
Counter	n_p	number of packets
	n_f	number of flows
	n_{fc}	number of completed flows
	n_{ooo}	number of out of ordered ACK packets
	n_{rtr}	number of packets retransmission
	n_{dpk}	number of duplicated ACK
Other	win	congestion window size
	δ_{win}	1^{st} direvative of congestion window
	$byte_p$	bytes transmitted per packet
	$byte_f$	bytes transmitted per flow
	$byte_{ff}$	bytes transmitted per flow on the fly

Table 5.1: Features extracted from the data plane at the load-balancer.

For simplicity, this chapter considers TCP traffic and revisits all the networking features that can be extracted using Aquarius, as introduced in section 3.2.1. However, many of these features are also available for other Transport-layer protocols (*e.g.,* UDP and QUIC). The notations and descriptions of all collected features that are available in TCP and to the LB, to infer server load are shown in table 5.1, and explained in the following.

Temporal Features: Temporal features are significant metrics for networking applications. For example, round-trip-time (RTT) is employed to evaluate QoS for software-defined network (SDN) [259]. As for heterogeneous and complex data center network setups, more such temporal features are candidates for server load inference.

1. *Flow inter-arrival time* (iat_f): The arrival rates of flows reflect the density of requests from clients. This feature captures all flow inter-arrival times to one application server, representing general flow-based traffic load going through one LB to its application servers. It is updated whenever a new flow is observed targeting each application server, *e.g.,* for TCP traffic whenever a first SYN packet is seen for a new 5-tuple digest (flow ID).

2. *Packet inter-arrival time* (iat_p): The arrival rates of packets have similar meanings but with

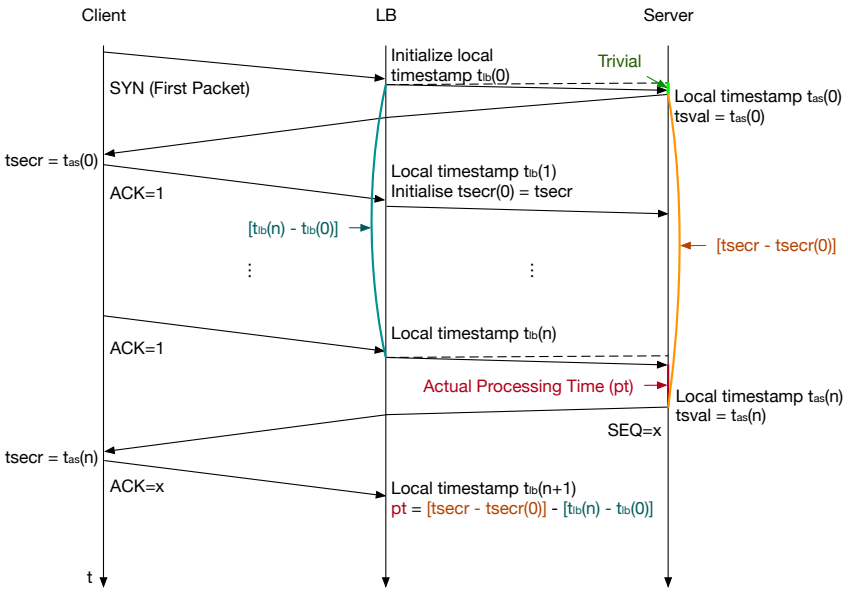


Figure 5.13: Calculation of server processing time with TCP timestamp options in the context of network load balancers.

finer granularity. Instead of being flow-based, it captures all the packet inter-arrival times to one application server regardless of flow ID. It can be used together with iat_f as canonical features, indicating the distribution of flow duration, packet arrival density, etc.

3. *Per-flow packet inter-arrival time* (iat_{ppf}): Unlike previous inter-arrival time, iat_{ppf} is specific to each flow (each request from a client), estimating all the packet inter-arrival times to one application server for each network flow. It is then the accumulation of processing time on the server, on routers, and on clients – as well as the communication latency along the path – which potentially can be used to learn the distribution of clients (*e.g.,* their distance from the servers).

4. *Flow complete time* (fct): Capturing the distribution of flow duration can potentially indicate server load. With the same type of application, when fct is observed to increase, it can be used as an indicator that the corresponding application server is overloaded[4]. This feature encodes not only the duration of a flow, but also the timestamp when a flow is finished. It is updated whenever a flow finishes, *e.g.,* for TCP traffics, whenever received the first FIN/RST for an established flow. Since there might be more than one flow that finishes at the same time, the number of finished flows is also estimated as one of the counter features introduced below.

5. *SYN to first ACK latency* (lat_{synack}): This feature is constructed to calculate the time difference between the first SYN and the first ACK packet from the client. For TCP traffics, the initialization process (the 3-way handshake) allows detecting the round trip time (RTT) between the server and the client. Whenever a SYN is received, SYN cookies statelessly generate a SYN+ACK response immediately, hence the processing time on the server is negligible. This allows estimating the baseline RTT between the client and the server, for each flow. It is updated whenever the first ACK of an established flow is received.

6. *Data packet processing time derived from TCP timestamp option* `tsecr` (pt_{1st} and pt_{gen}): As already discussed in section 3.2.1, this feature exploits TCP options (`tsval` and `tsecr` w/ 1ms granularity) and tries to capture the variation of processing time on the server. The way this feature is calculated in the context of network LBs in data center networks is shown in figure 5.13. Intuitively, the time difference between when the server receives a packet, and

[4]This shares the same principle as TCP congestion control algorithms (*e.g.,* Reno and Vegas) [260] – an "early" ML application in networking – which detects congestions based on increasing round trip time (RTT).

when the server replies, indicates the processing time of a query on the server. However, this information is hard to observe by LB because of DSR. By exploiting the TCP option fields, the timestamp of when the server processes the request and replies to the client can be obtained from `tsecr`: `tsecr` in a TCP segment from the client is equivalent to the `tsval` recorded in the previous TCP segment from the server. The difference between `tsecr` and previously stored timestamp when LB received last packet is then calculated, which serves as an estimation of processing time on the server. As the clocks on LB and server are not synchronized, all timestamps need to be initialized with the first network flow. An example procedure of calculating pt is demonstrated in figure 5.13. This calculated value might be corrupted by the time spent on the wire from LB to server, however, in DC network, this is trivial when compared to processing time on the application server. This feature is updated for each received ACK to data packets. To make it more favorable to Web applications, distinctions are made between the first data packet and the following data packets. One limitation of this feature is that it requires clients to enable the TCP option field.

Counter Features: As a virtual network function located on the path between clients and servers, LB can accumulate a set of counters for each application server it is connected to, which could potentially help infer traffic throughput and server load. However, the non-stationary counters could induce noises when there is a scaling event or failover of LB.

1. *Number of packets* (n_p): This feature captures the amount of all newly arrived packets to each application server via an LB. It is a local observation of how many packets are directed to each application server, which serves as a primitive baseline of the current traffic load.

2. *Number of flows* (n_f): Similarly, the amount of newly arrived flows to each application server via an LB can be collected. It is a local observation of how many flows are directed to a specific application server. Together with fct, it helps indicate the lifecycle of flows.

3. *Number of completed flows* (n_{fc}): In addition to counting the arrival of new flows, the amount of all terminated network flows to each application server also sketches the lifecycle of flows. Together with n_f, it gives a locally observed number of ongoing flows.

4. *Number of out-of-ordered and duplicated ACK packets, and retransmissions* (n_{ooo}, n_{dpk}, and n_{rtr}): These features are specific to TCP. They reflect the congestion level on the path between servers and clients.

Other Features: other than temporal and counter features, more observations can be obtained to reflect the congestion level and throughput, which may allow decoupling server-load information.

1. *Congestion window size* (win): An estimation of the current TCP congestion window size embeds various information, including the congestion state for each flow as perceived by the clients, distribution of elephant and mouse flow, etc. It is updated whenever a new ACK packet is observed targeting each application server from an LB's perspective.

2. 1^{st} *derivative of congestion window size* (δ_{win}): The congestion state is encoded in the varieties of TCP congestion window size during a flow lifetime, *i.e.,* 1^{st} derivative of window size for each network flow. A negative value indicates that TCP fast retransmission is taking place, hence it allows the LB to know the congestion level for the corresponding flow. It is updated whenever a new ACK packet (after the first ACK packet) is received.

3. *Bytes transmitted per packet* ($byte_p$): To take a glimpse of the traffic from the server, ACK numbers in packets from the clients' side can be utilized to calculate the amount of data that are successfully transmitted. It is determined largely by the application running on the server and the type of query. In addition, it indirectly suggests the congestion level from the server's point of view. Along with the number of packets, local observation on throughput can be estimated, which serves as an IO metric for server load after scaling by the number of LBs.

4. *Bytes transmitted per flow* ($byte_f$): This feature captures information concerning only completed flows. It's a feature that helps look back into the history, which gives hints on the throughput utilization in the past few time steps.

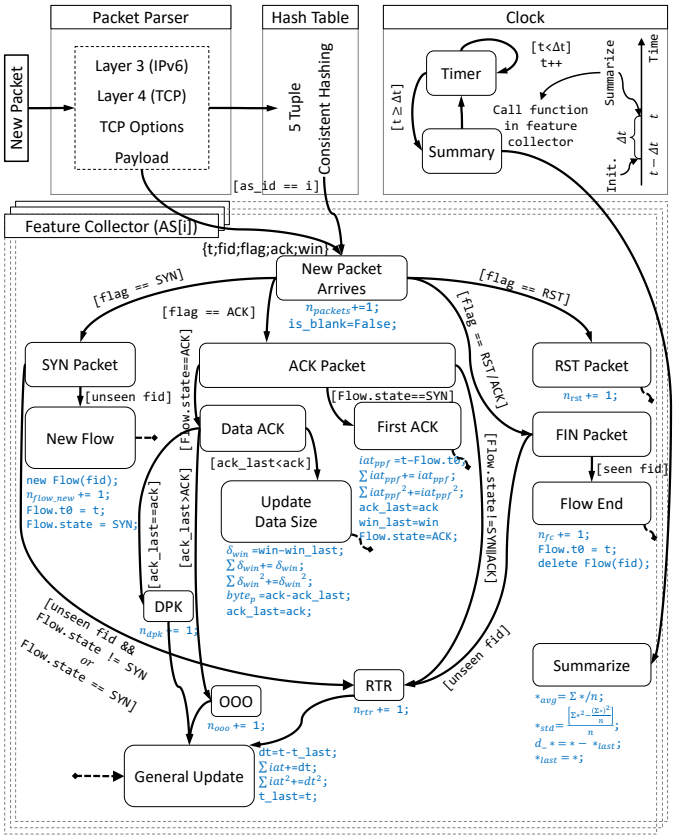


Figure 5.14: Simplified flow chart of TCP traffic feature collector: when receiving a new packet, load-balancer parses the 5-tuple (source & destination addresses, L4 protocol, source & destination port) digest as flow ID. Parsed header information is fed to an embedded state machine which collects and calculate various features according to the flow state and current packet for the corresponding application server. The features are summarized every δt. The arguments in square brackets denote conditions while the ones in blue are actions. For clarity, some transitions are broken into dashed lines pointing out and into corresponding boxes.

5. *Bytes per flow on the fly w/ reservoir sampling* ($byte_{ff}$): Feature $byte_f$ cannot be easily obtained for some transport layer protocol which has no indicator of flow completion (*e.g.,* FIN/RST flag for TCP traffics). Instead, capturing the number of bytes on the fly can be generalized to many protocols (*e.g.,* UDP, QUIC). With reservoir sampling, whenever an ACK packet is received, the transmitted byte can be calculated and put in the buckets with a fixed probability.

To collect all the features described above, a feature collector is designed and embedded, using Aquarius in the LB. Other than the network 5-tuple, TCP flags $flag$, acknowledgment number ack, congestion window size win, and TCP timestamp options $tsecr$ are taken as inputs. Additionally, local timestamp t, flow-id fid constructed from the 5-tuple digest, and assigned application server id as_id from the LB are used to keep tracking per server and per-flow observations.

A simplified feature collection workflow for TCP traffic is depicted in figure 5.14. For each application server, there is an independent table of variables that is updated with the state machine whenever a new packet shall be directed to the corresponding application server. For each flow, its state is tracked according to the flags of the packets (for TCP traffic). Whenever a new flow arrives, a state is initialized and stored in the flow table. Whenever the first FIN/RST packet is seen, the state is evicted from the flow table to create space for future flows. The list of features is collected with incremental counter and accumulated sum, with which average and standard deviation can be calculated as $\bar{x} = \frac{\sum x}{n}, stddev(x) = \sqrt{\frac{\sum x^2}{n} - \bar{x}^2}$. Periodically, the LB takes a "snapshot" of the current set of features collected, from which proper statistical features (*e.g.,* average, standard

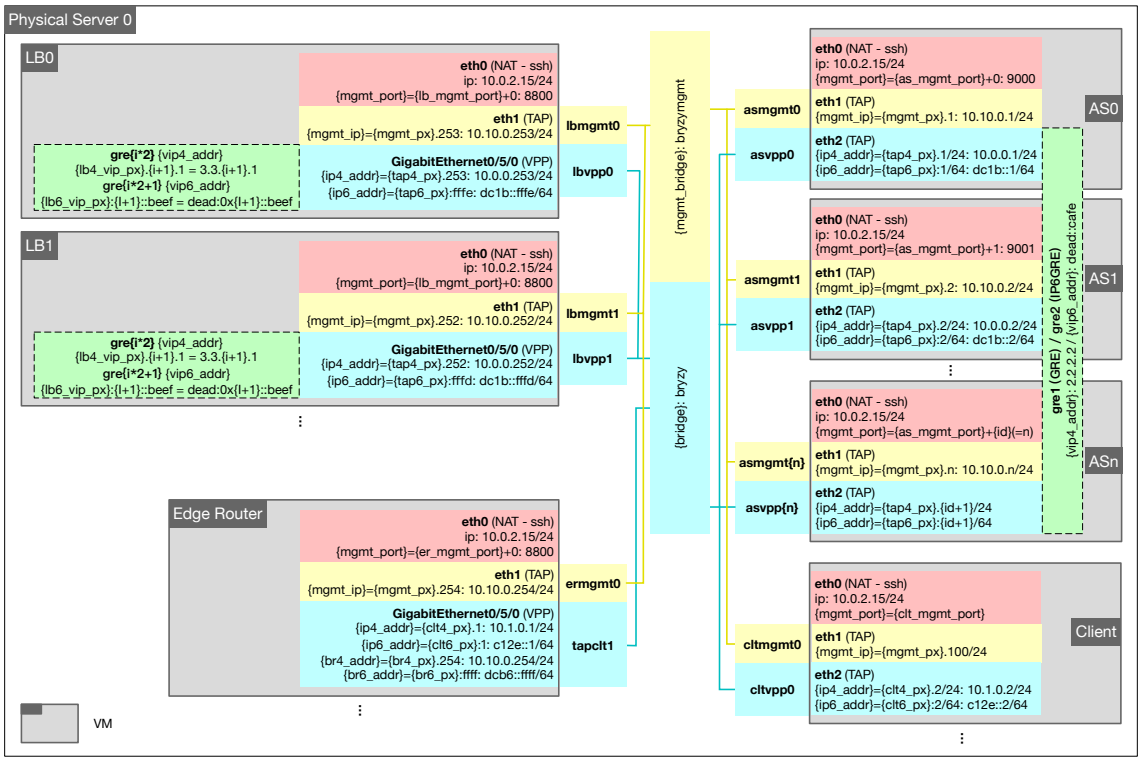


Figure 5.15: An instance of testbed network configuration with one layer of load balancers.

deviation, percentiles) for server load inference can be generated.

5.3 Experimental Setup

Similar to the setup used in chapter 3, the experimental platform consists of VMs representing clients, edge routers, load balancers and server agents. Its topology is depicted in figure 5.15. As a reminder of what has been described in section 3.2.3, several explications are made as follows:

1) Load-Balancer: A simple load-balancer, that randomly maps flows to servers, is implemented as a VPP plugin [121]. With high modularity and extensibility, as well as kernel-bypass capabilities, VPP provides a platform to build a prototype predictive application load balancer. Based on the load-balancing plugin in VPP, a variety of load-balancing schemes are implemented, including active-probing-based WCMP, simple heuristics counting the number of ongoing flows, and ML-based load-balancing algorithms.

2) Apache HTTP Server Agent: An Apache HTTP server agent [227] is running on each application server. Three metrics are periodically ($10Hz$) gathered as ground truth for the load state: CPU utilization, memory usage, and the number of Apache busy worker threads. To be more precise, the CPU utilization is calculated as the ratio of non-idle CPU time to total CPU time measured from the file `/proc/stat` that keeps track of statistics about the system; the memory status is obtained from `/proc/meminfo`; and the number of Apache busy threads is accessed via Apache's *scoreboard* shared memory.

3) System Platform: Experiments are conducted on one physical machine with a 24-physical-core (48-logical-core) Intel Xeon E5-2690 CPU. Load-balancer instances (VPP) are, by default, deployed as a 2-core (logical) VM. Application instances of an Apache HTTP server reside each also as a 2-core (logical) VM. All the VMs are on the same link (Layer 2), with routing tables statically configured. All Apache HTTP application servers share the same virtual IP address on one end of GRE tunnels with the load-balancer on the other end, as is shown in figure 5.15.

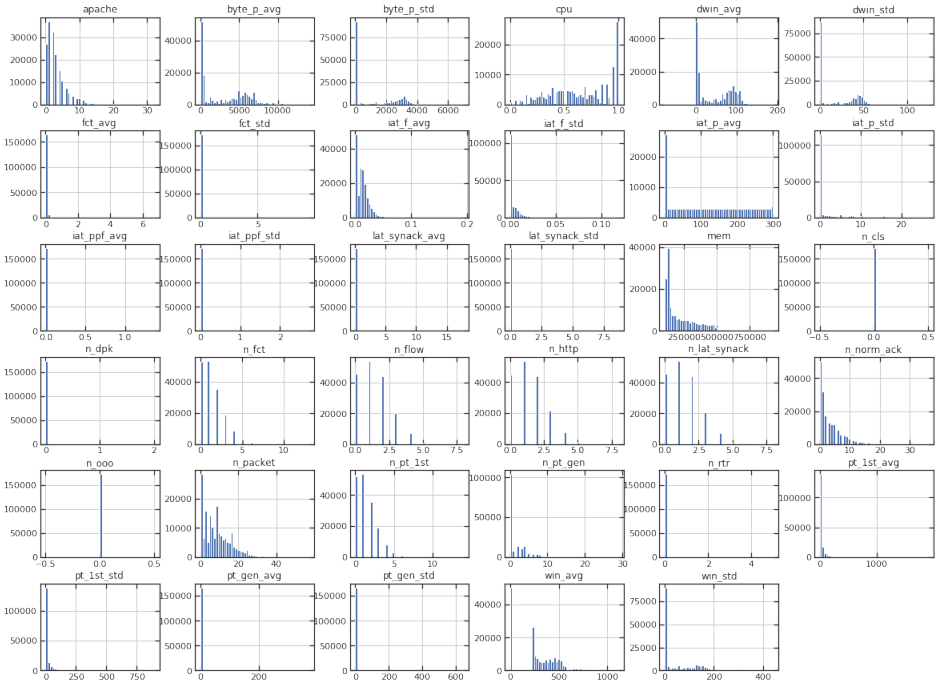


Figure 5.16: Networking feature distribution collected on the LB.

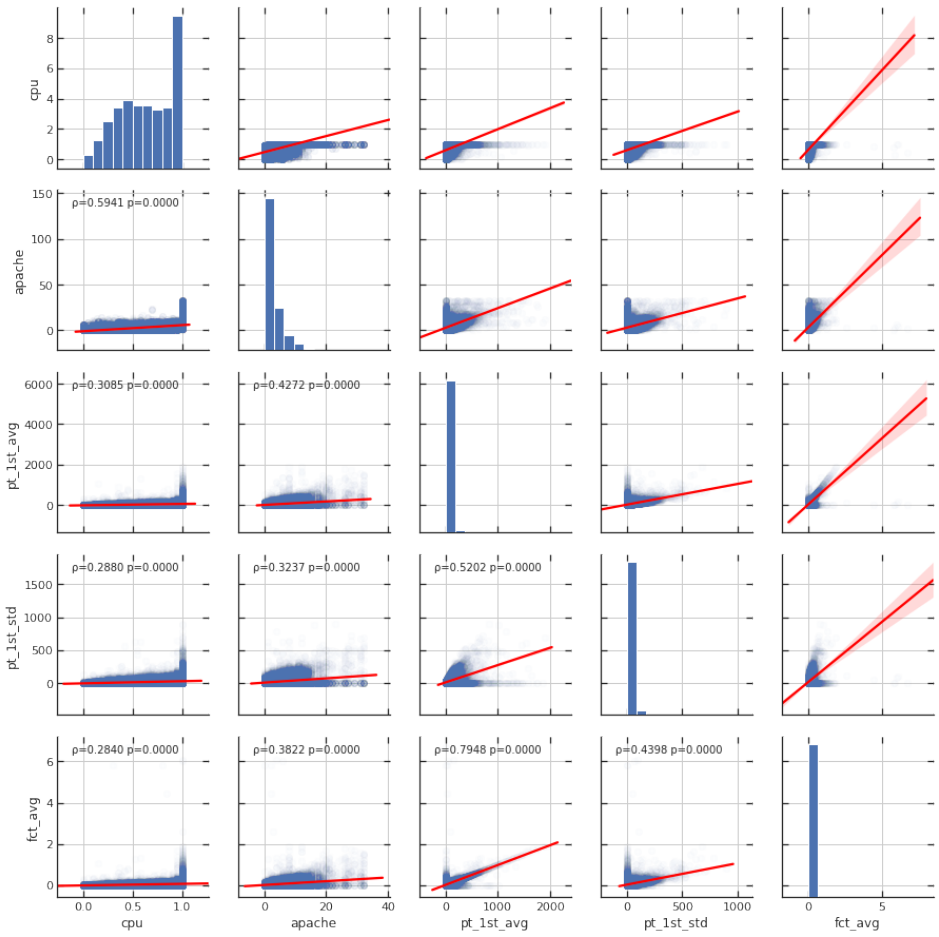
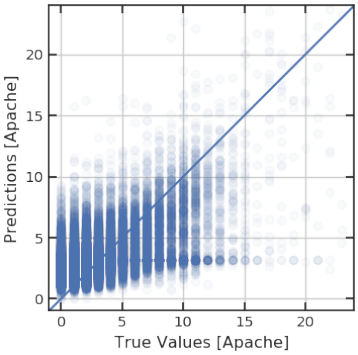
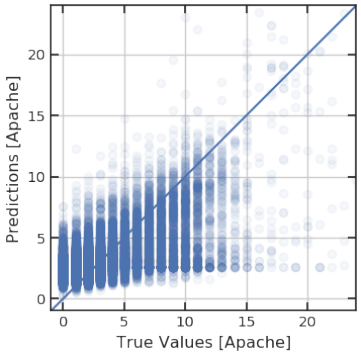


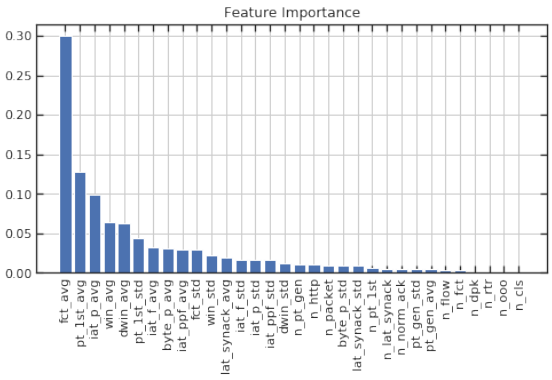
Figure 5.17: Networking feature correlation with p-value.



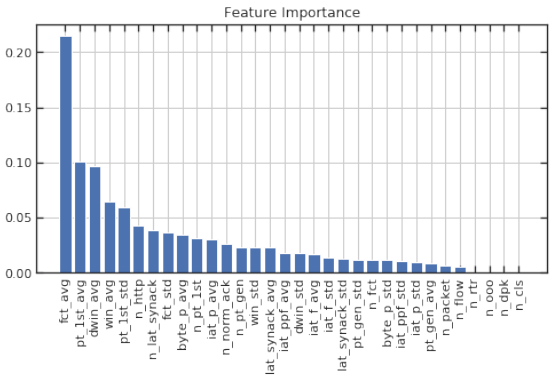
(a) Visualization of prediction error (GBM).



(b) Visualization of prediction error (XGB).



(c) Feature importance (GBM).



(d) Feature importance (XGBoost).

Figure 5.18: Feature engineering and preliminary evaluation using GBM and XGBoost after fine-tuning.

4) Network Traces: Similar to in section 3.3.1, 3 types of network traces are used for conducting experiments in this chapter:

- Poisson stream of PHP for loop, both CPU-intensive (two subcategories are exponential and lognormal distributions), as described in section 3.2.3;
- Poisson stream of PHP file transmission, IO-intensive, as described in section 3.2.3[5];
- Wikipedia 24 hour replay, as described in section 3.2.3.

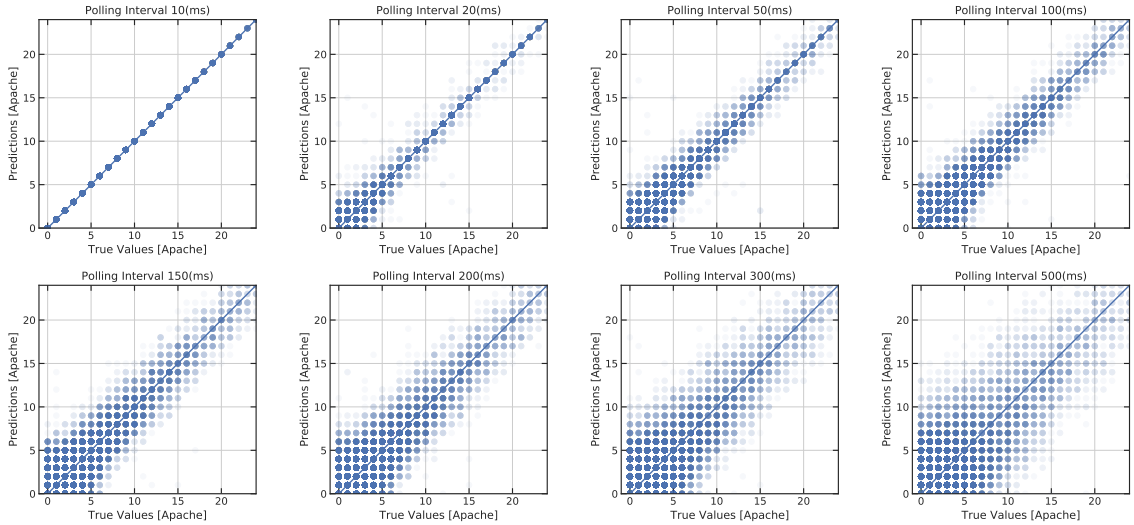
5.4 Experiment

The first experiment is carried out offline, by collecting networking features, as detailed in section 5.2.1, as well as instantaneous server load as ground truth on a 12-server cluster with 1 LB. This is designed to study whether ML models can infer server load from networking observations. The feature distribution is shown in figure 5.16. Correlations between some networking features and ground truth label data (*i.e.,* CPU usage and the number of busy Apache threads) are depicted in figure 5.17.

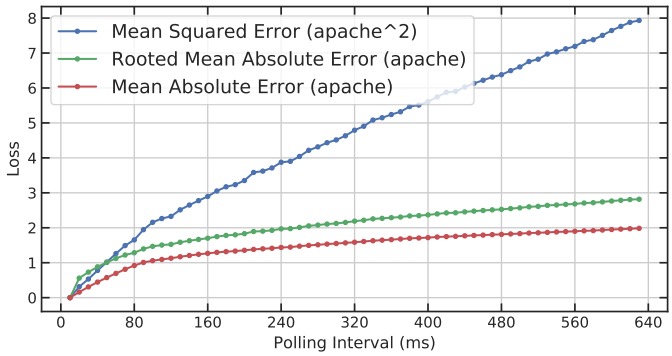
To further understand the correlation and feature importance in terms of server load inference, a Gradient Boost Regressor (GBM) and an XGBoost model are trained and fine-tuned using the collected dataset, whose results are shown in figure 5.18. It is shown that temporal features especially flow complete time and first packet processing time are the two major contributors to the inference process. The performance of the two Boosting models is not ideal because they are not designed for sequential datasets.

To select the best sequential ML model, a list of models with different structures are implemented with TensorFlow. They are compared with a simple benchmark with candidates including active probing and a naive neural network which is not sequential.

[5]Traffic rates are generated from 24 to 128 requests per second for a 12 server cluster.

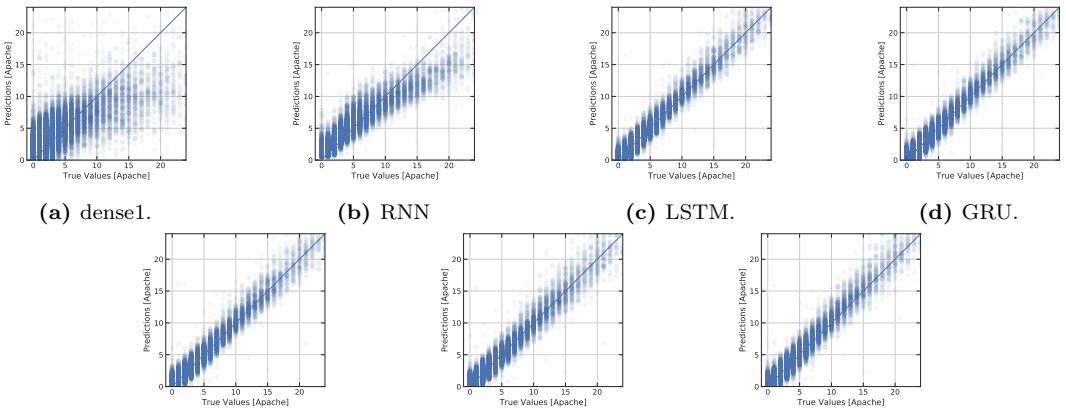


(a) Active probing performance with different polling intervals (10, 20, 50, 100 ms interval in the 1st row and 150, 200, 300, 500 in the 2nd row).



(b) Regression metrics evaluation using active probing with different polling intervals.

Figure 5.19: Active probing performance evaluation and comparison.



(a) dense1. **(b)** RNN **(c)** LSTM. **(d)** GRU.

(e) 1dconv. **(f)** wavenet. **(g)** wave-rec.

Figure 5.20: Validation results comparing predicted and ground truth #apache.

The results using synthesized Poisson traffics are depicted in figure 5.19b and figure 5.20. And the full compare and contrast table for both Wiki and Poisson traces is shown in table 5.2.

Active probing is implemented by inferring server load, based on previously observed server load. The ML benchmark is built as a neural network with one flattened layer and one dense layer, which takes into account features collected across all of the sequences. For ML models, only a

Task	Metrics	Dense1	RNN2	LSTM2	GRU2	1DConv-GRU1	Wavenet-GRU1	Wavenet-Reconst.
Wiki Replay	**MSE**	253.203	2.557	**1.553**	1.660	1.878	1.923	2.421
	RMSE	15.912	1.599	**1.245**	1.288	1.371	1.387	1.556
	MAE	1.804	1.099	**0.916**	0.931	0.988	0.996	1.117
	Avg. Latency (Train) (ms)	116	116	194	162	**78.8**	137	136
	Latency std. (Train) (ms)	2.53	2.53	6.91	2.33	2.48	1.48	**0.89**
	Avg. Latency (Infer) (ms)	50.5	52.1	53.4	53.7	**44.8**	54.9	54.6
	Latency std. (Infer) (ms)	1.03	1.74	3.01	1.3	2.8	0.80	**0.52**
Poisson Traffic	**MSE**	1520.888	2.804	0.801	**0.774**	0.874	0.965	0.946
	RMSE	38.999	1.675	0.895	**0.880**	0.935	0.982	0.973
	MAE	3.176	1.162	0.602	**0.600**	0.635	0.648	0.649
	Avg. Latency (Train) (ms)	**60.5**	305	214	215	110	162	195
	Latency std. (Train) (ms)	**1.66**	29.5	13.8	14.4	6.57	7.16	27.4
	Avg. Latency (Infer) (ms)	**55.4**	91.4	69.3	70.9	61.5	65.6	70.2
	Latency std. (Infer) (ms)	0.714	6.41	2.23	1.32	2.51	1.67	**0.429**

Table 5.2: Accumulated score board for different models and different jobs executed with 1 CPU core.

part of the dataset is selected as the training dataset while the rest are kept as the test set[6]. The configuration of sequence length as 64 and stride as 32 gives around 160k datapoints. Validation results are depicted in figure 5.20, and summarized in table 5.2. The results show that neural networks with simple structures (*e.g.,* Dense1) have lower latency in terms of both the training and the inference process. However, Dense1 is not able to predict server load states with high accuracy. Sequential models (*e.g.,* RNN, LSTM, and GRU) achieve the best prediction accuracy at the cost of higher training and inference overhead. Simplified sequential NN structures including 1D convolutional networks and Wavenet help balance the trade-off between accuracy and learning overhead.

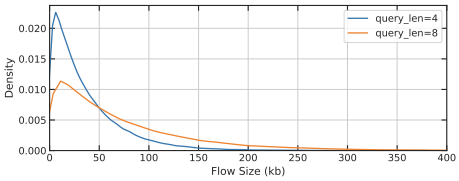
5.4.1 Applying Poisson-Trained LSTM on Poisson Traffic

Based on the offline training results using Poisson traffic, ML models can predict server load states with reduced errors. The trained models are brought online to verify their performance when subjected to real traffic. The set of traffic rates is extended from $[100, 160]$ to $[100, 175]$, which is based on the scale of the configured testbed. The testbed consists of 12 2-core (logical) servers. Therefore the theoretical processing capability "per second" of the cluster is 24 for 1-second-duration jobs. Given the average process time of the generated queries (simple for-loop with iterations of exponential distribution whose mean equals 400k) is around 140ms, the expected 100% loaded traffic rate is, therefore, 170. With different traffic rates, one round of experiment takes from 800s to 1000s with 100k queries. The set of traffic rates tested is $[100, 115, 125, 135, 142, 146, 150, 154, 158, 162, 165]$.

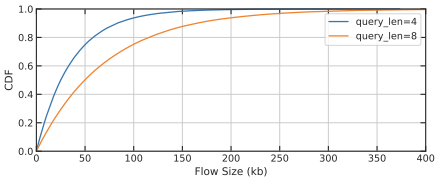
In addition to varying the traffic rate, it is also interesting to vary the query length (*i.e.,* the mean of the exponential distribution of number of iterations μ_{query_len}) to verify if ML model is able to generalize to different queries. Since the ML model is trained with $\mu_{query\_len} = 4$, another set of queries is generated with $\mu_{query\_len} = 8$, thus increasing the average process time from 140ms to 160ms, while also pushing the saturating traffic rate to around 150. The characteristic of the two types of queries of an exponential distribution is depicted in figure 5.21. Since the PHP for loop application running on servers print the current number of iteration every 1000 iterations, the flow size for each query can be calculated based on their number of iterations. For instance, a query for $4e6$ iterations will generate around 36kb flow.

The results of these experiments are shown in figure 5.21. As the LSTM network is trained on Poisson traffic, it can help select the less loaded server hence has great improvement on average page load time, for both short queries and long queries, as is shown in figure 5.21c. It's also interesting to see that with a higher average expected job duration ($\mu_{query\_len} = 8$), the page load time with ML is close to Maglev when the traffic rate is around 150 to 155. Another observation from figure 5.21c is that the difference between the average page load time using Maglev and ML becomes less significant when the traffic rate becomes high enough to saturate all servers, and when the ML model performance degrades as is shown in figure 5.21g. Similarly, improvements can be found in terms of the number of busy Apache threads as well as CPU usage.

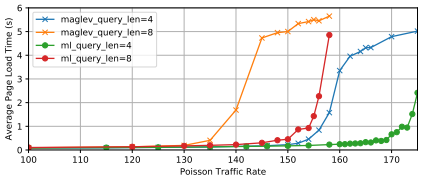
[6]For instance, for Poisson traffic, traces with traffic rates of $105, 120, 130, 140, 144, 148, 150, 152, 155, 157$ are used as the training dataset



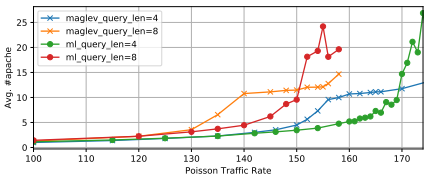
(a) PDF of flow size.



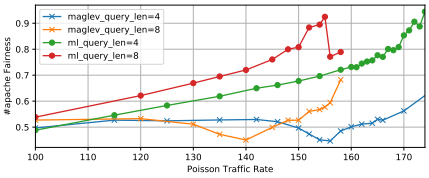
(b) CDF of flow size.



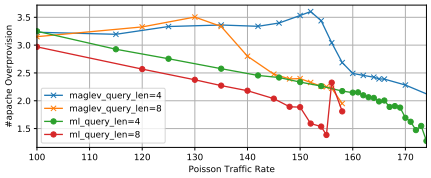
(c) Page load time



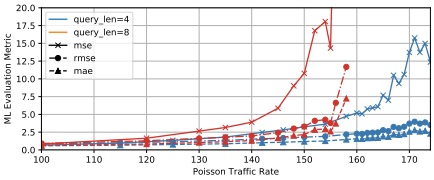
(d) Avg. #apache.



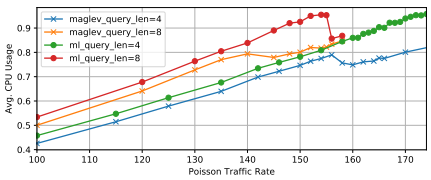
(e) #apache fairness.



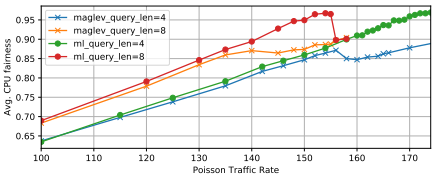
(f) #apache over.



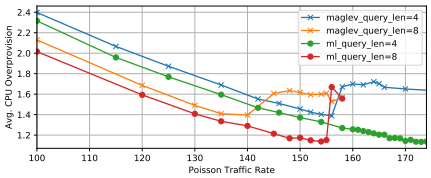
(g) ML score.



(h) Avg. CPU.



(i) CPU fairness.



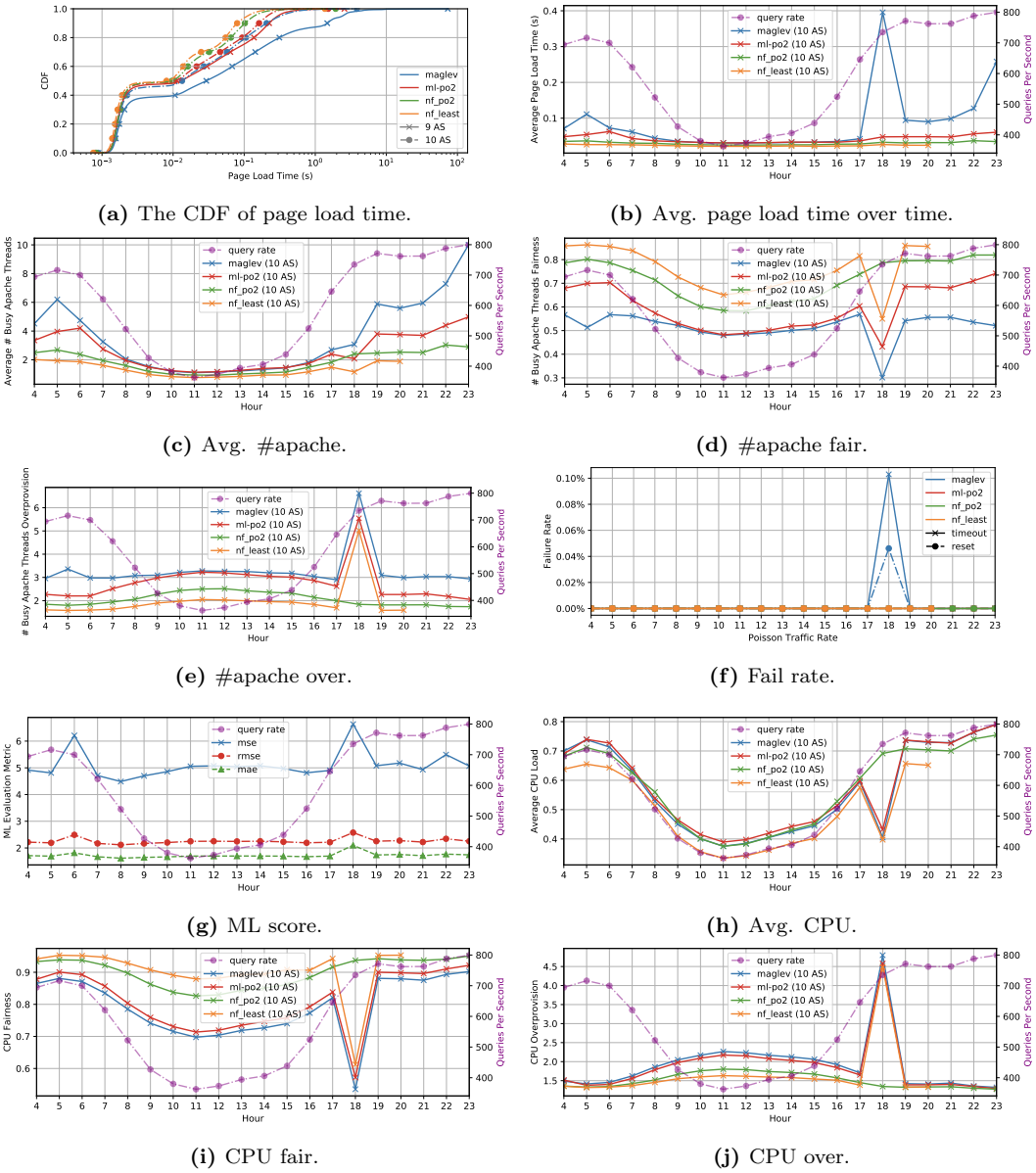
(j) CPU over.

Figure 5.21: Results of online scaling experiments on synthesized Poisson traffic using LSTM model trained with Poisson traffic.

In figures 5.21d, 5.21e, and 5.21f, it may appear confusing at the first glance that the average number of Apache threads using ML-based LB can surpass Maglev when traffic rate grows to a certain level. However, this is an instance where the averaged networking measurements fail to reflect the system states. With higher fairness, ML-based LB is able to spread flows across all servers instead of having skewed flow distribution (*e.g.,* two or more elephant flows with super high number of iterations on the same server).

In figures 5.21h, 5.21i, and 5.21j, it is more clearly visible that the improvement with ML-based LB increases as loads get higher[7]. As depicted in figures 5.21e, 5.21f, 5.21i, and 5.21j, the performance of Maglev starts to degrade when the server cluster is subjected to lower traffic rates than the ML-based LB. The ceiling of the Apache thread pool (32), as well as the CPU usage (100%), implies an absolute capacity limit. When subjected to lower traffic rates, Maglev tends to assign new flows to the server which has already reached its capacity limit. Therefore, the fairness of #apache decreases while the overprovision increases. When subjected to heavier loads, more servers reach their capacity limits and therefore the load-balancing fairness is improved and the

[7]The general level of overprovisioning/utilization rate in a DC setup would be 75% to 90% [65].



(a) The CDF of page load time.

(b) Avg. page load time over time.

(c) Avg. #apache.

(d) #apache fair.

(e) #apache over.

(f) Fail rate.

(g) ML score.

(h) Avg. CPU.

(i) CPU fair.

(j) CPU over.

Figure 5.22: Results of online experiments on real-world Wiki replay traces using LSTM model trained with Wiki replay traces.

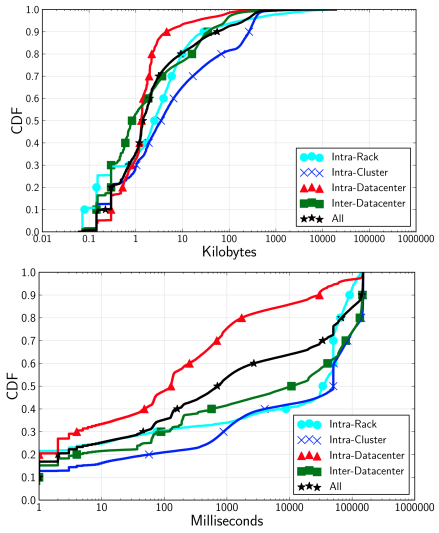
overprovisioning factor becomes lower. However, the ML-based LB is more aware of server load, and thus can make "smarter" load-balancing decisions. This increases the traffic rate that starts saturating the server cluster, when the load-balancing performance degrades. In simpler words, ML-based LB helps handles not only more intensive traffic but also more computational-intensive queries with fewer servers.

5.4.2 Applying Wiki-Trained LSTM on Traffic with Wiki Replay

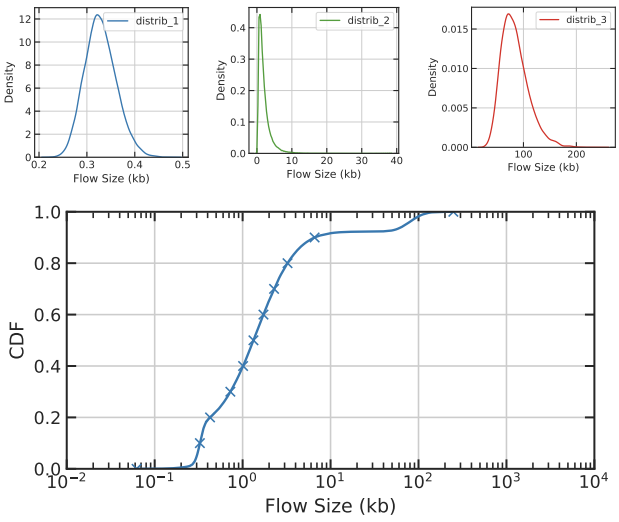
The LSTM networks with the same structure as is described above trained with Wiki replay traces are applied online with 2 different configurations: 12 servers (when servers are not heavily loaded) and 9 servers (when servers are supposed to be overloadeded). The first 3 hours of the 24-hour network trace are used to train and validate the LSTM networks using both configurations – which achieves 1.800 MSE, 1.342 RMSE, and 0.874 MAE in the validation set.

Next, deploying the trained LSTM networks online allows the LB to make dispatching decisions for the remaining traces of hours $4-24$. The results are shown in figure 5.22[8]. It can be observed

[8]The data point of the hour 18 is corrupted by an unexpected co-located workload running on the physical



(a) Traffic distribution shown in a Facebook paper [1].



(b) The composition of our heavy-tail traffic

Figure 5.23: Visualization of active probing performance with different intervals

that – especially when the server cluster is subjected to heavy traffic rates during peak hours ($4-7$ and $19-23$) – ML-based LB can help improve the load-balancing performance in terms of not only page load time (figure 5.22b), but also work load distribution fairness (figure 5.22d). However, the ML-based load-balancing mechanism is marginally outperformed by heuristic load-balancing algorithms based only on the number of ongoing flows (nf-po2 and nf-llf). This indicates that more networking features do not necessarily provide improved server load inferences.

5.4.3 Applying Poisson-Trained LSTM on Traffic with Realistic Distribution

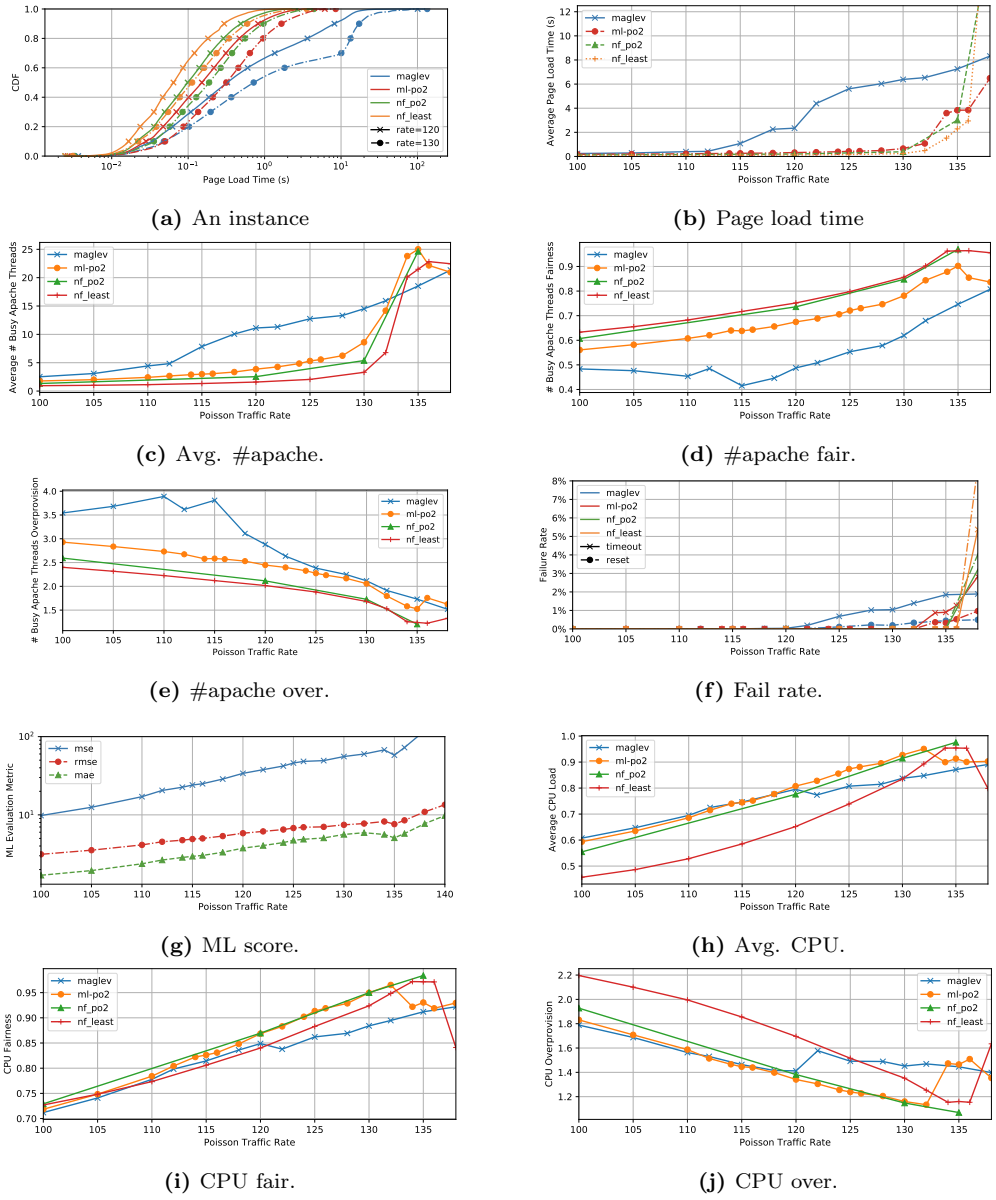
Performance gains have been shown with Poisson traffic and Wikipedia replay. It is also interesting to apply it on network traffic with multi-modal distributions. As discussed in [1], web server traffic is not a simple exponential distribution, but, the distribution is a combination of several heavy-tail queries, as is illustrated in figure 5.23a. Hence, the query length – as determined by the number of iterations (same PHP application) with log-normal distribution instead of exponential distribution – is applied as heavy-tail distribution traffic.

Three distributions of queries are generated, to construct a CDF similar to the one depicted in figure 5.23b. The traffic of distribution $1 \sim Lognormal(10.5, 0.1^2)$ has $2e4$ queries, representing some short queries that terminate within milli-seconds *e.g.,* 404 error; distribution $2 \sim Lognormal(12, 0.8^2)$ has $1e5$ queries, representing the majority of mice flows; and distribution $3 \sim Lognormal(16, 0.3^2)$ has $1e4$ queries, representing elephant flows[9]. The average iterations $(\exp\left(\mu + \frac{\sigma^2}{2}\right))$ for these sets of queries are respectively 36497.535, 224134.142, and 9295119.171, which yield a weighted average of 893035.051. Even though the weighted average of iterations is smaller than the traces we used for previous experiments (in section 5.4.1), the elephant flows dominate the servers' load. Hence the expected saturating traffic rate is estimated with traffic of distribution 3, which is around 140. To infer server load from this combined distribution is a challenging task for ML-based LB trained with only Poisson traffic with exponential flow duration or size.

Results are shown in figure 5.24. The CDFs of page load time for two Poisson traffic rates (120 and 130 queries/s) as depicted in figure 5.24a, demonstrate improvements using ML-based LB over Maglev. It can also be observed that the shape of CDF for Maglev, when subjected to Poisson traffic rate with 130 queries/s, has similar shape as in figure 5.23a, which means the composition of three log-normal distribution traffics is close to reality.

server which hosts the experimental testbed.

[9] Elephant flows conventionally refers to continuous TCP flows with a large amount of data to transmit – over a certain period. Similarly, mice flows refer to the TCP flows that transmit little data during a short period.



(a) An instance

(b) Page load time

(c) Avg. #apache.

(d) #apache fair.

(e) #apache over.

(f) Fail rate.

(g) ML score.

(h) Avg. CPU.

(i) CPU fair.

(j) CPU over.

Figure 5.24: Experiment results using Poisson-trained LSTM model on composed heavy-tail traffic.

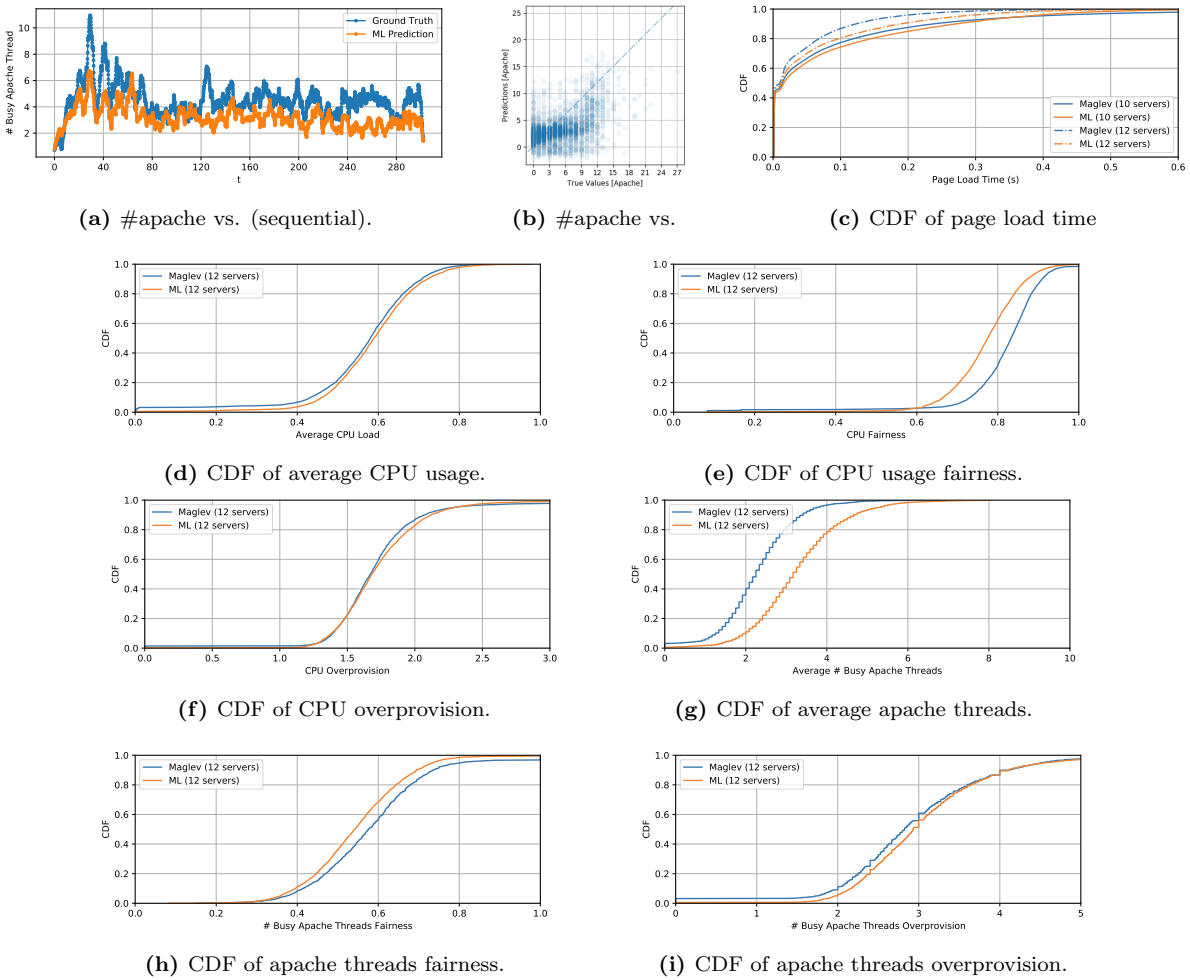
In figure 5.24f, it can be seen that the Maglev LB saturates, when the traffic rate surpasses 120 queries/s while the ML-based LB can extend this limit to 140 – the estimated theoretical limitation for this configuration. It is another proof that with ML-based LB, in simple words, more queries could be served with fewer servers.

Compared against figure 5.21, the improvement with ML-based LB is significant in terms of page load time, fairness, and CPU overprovision. It can be observed that the "gap" of page load time between Maglev and ML-based LB is larger and wider, since fairness in this experiment is more sensitive given the existence of some elephant flows (traffic of distribution 3).

All of the improvements are at the expense of one additional CPU core (logical) running with 50% $\sim$ 70% usage alongside LB, parsing observations, and inferring server load with LSTM networks with 10Hz update frequency.

5.4.4 Limitation: Generalization

To study whether the trained ML models can generalize, the LSTM model is trained only using Poisson `for`-loop traffic and brought online to work with both `for`-loop and Wiki traces. By comparing the performance of all models in table 5.2, for Poisson traffic GRU outperforms



(a) #apache vs. (sequential). **(b)** #apache vs. **(c)** CDF of page load time

(d) CDF of average CPU usage. **(e)** CDF of CPU usage fairness.

(f) CDF of CPU overprovision. **(g)** CDF of average apache threads.

(h) CDF of apache threads fairness. **(i)** CDF of apache threads overprovision.

Figure 5.25: Experiment results using Poisson-trained LSTM model on 300s Wiki replay trace.

the rest and LSTM is the closest match. However, since LSTM has better performance dealing with more realistic traffic (Wiki replay), the LSTM model trained with Poisson traffic is reused to cross-validate if it can be generalized to Wiki replay. With a sample of 300s Wiki replay trace, the LSTM model achieves an MSE of 15.119, an RMSE of 3.888, and an MAE of 2.846, which is much worse than on Poisson traffic, which means online training is required to adjust a pre-trained model to the new traffic pattern. Figure 5.25 shows that the trained model generalizes poorly if the applied traffic is not seen by the model before, which is consistent to [30].

5.5 Summary

Machine Learning (ML) algorithms show promising results on different problems yet it is challenging to apply on realistic networking problems and in real-life systems. Aquarius bridges ML and distributed networking systems and takes a preliminary step to integrate ML approaches in networking field. Based on Aquarius, this chapter investigates a wide range of networking features that can be extracted from the data plane for optimizing load balancing performance. All the collected features are used as input data for offline data analysis pipelines first, and the trained ML models are then brought online to make decisions on the fly.

In addition to the evaluation results demonstrated in section 3.3, this chapter further illustrated demonstrates the potential of MLBto conduct feature engineering, model selection, and online model deployment. The models applied in this chapter have shown the ability to learn and infer server load states with networking features. It also shows that networking problems are dynamic and heterogenous, thus it is challenging to train a model that generalizes well. To that end, the

following two chapters will explore two different approaches to improve the model generality. As demonstrated in section 5.4, more features do not necessarily lead to improved performance than heuristic load balancers, in-depth analysis of the key factors that indicate server load states will be further provided in this thesis. More extensive evaluations will be conducted, including *e.g.,* the impact of asynchronous and delayed decision making process.

The work and results from this chapter have been published in [152].

Chapter 6

HLB: Towards Load-Aware Load Balancing

As is discussed in chapter 5, networking features can help achieve optimized load-balancing performance, with the help of Machine Learning (ML) algorithms. However, the ML algorithms do not generalize well when facing a various spectrum of networking traffic. The purpose of this chapter is to investigate the different factors that influence workload distribution fairness of network load balancers (LBs). To this end, this chapter proposes Hybrid LB (HLB), a distributed, load-aware, open-loop control load-balancing algorithm that infers server occupancy and processing speeds for making optimized load-balancing decisions, with no need for offline training. HLB requires no explicit monitoring or signaling, thus generating no additional management traffic that would grow with the size of server pools and probing frequencies[1], and thus would reduce the effective capacity available in core links [59].

This chapter also argues that to improve workload distribution fairness and quality of service (QoS), the server load information needs to be taken into consideration, including:

- server occupancy, which indicates queuing delays,
- processing speed, determined by available resources.

Doing so allows HLB to make per-flow-level load-balancing decisions and offer each server subject to a fair share of workloads. HLB estimates these factors with no additional overhead for coordination among LBs, or with servers. HLB works out of the box and requires no network or application modification, nor additional control message.

Statement of Purpose

The contributions of this chapter are three-fold: (i) a study of the dominating factors in load-balancing performance, with a taxonomy of existing approaches, (ii) specification of a "fair" LB algorithm, HLB, that requires no manual configuration, or additional interaction with servers or other LBs, (iii) evaluations, by way of simulations and testbed experiments, that compare HLB with existing LB algorithms, in various data center configurations, and under realistic network traffic.

Related Work

LVS (Linux Virtual Server) [236] implements a wide range of load-balancing algorithms to improve fairness, however, without attaining throughput and latency characteristics meeting production requirements for data centers. Using statically configured *match action tables* or hash tables [64, 65, 190] increases throughput and reduces packet processing latencies. However, these tables do not support advanced load-balancing algorithms, *e.g.,* weighted round-robin [262] or least loaded server [261], which requires dynamically managing flow-server mappings. Cheetah [144] allows dynamically registering and recovering mappings of flows and servers, by encoding mappings

[1] With 50-byte packets, active probing 128 servers at 10Hz generates 64kbps traffic, while the 90-th percentile of per-destination-rack flow rate is 100kbps in production [1].

LB Algorithms	Description	Aware of Server Capacities	Aware of Server Occupancy	No Error-Prone Configurations
ECMP [58, 59, 193]	Randomly assigns a server.	✗	✗	✓
WCMP [64, 65], [190, 192, 221, 241]	Assigns servers based on weights defined by provisioned resources.	✓	✗	✗
AWCMP [143, 186, 187]	Assigns servers based on weights defined by polled resource utilization.	✗	✓	✗
LSQ [144, 261], GSQ2 [135, 217]	Assigns servers with the shortest/shorter queue occupancy based on local/global observations.	✗	✓	✓
SED [236]	Assigns servers with the lowest delay derived from static server weights defined by provisioned resources.	✓	✓	✗
HLB (This chapter)	Assigns servers with the lowest delay derived from *adaptive* server weights based on passive observations.	✓	✓	✓

Table 6.1: Taxonomy of Related Work.

as cookies in covert channels in packet headers. This allows retrieving the server handling a given flow if it is lost, *e.g.,* when an LB fails. Prism [221] *statelessly* maps flows to their hash buckets and *statefully* registers flow-server mapping information in a table of migrated flows when facing potential risks of flow disruptions, *e.g.,* during server pool updates. When servers are added or removed, it creates an independent table to track migrated flows, and updates server weights for balanced workloads distribution[2]. Integrating these algorithms [144, 221] will allow HLB to build load-aware algorithms while guaranteeing PCC for large-scale data centers.

The load-aware load-balancing decision-making process uses the estimation of server occupancy and processing speeds, as well on use of the application of different rules (probabilistic or minimisation rules). Table 6.1 summarises the taxonomy of network LB algorithms, based on their awareness of server occupancies and processing speeds.

1. Equal-Cost Multi-Path (**ECMP**) treats all servers as equal, and is agnostic to server load state differences. It is applied in many LB mechanisms [58, 59, 190, 193, 243] that aim at minimizing performance overhead.

2. Weighted-Cost Multi-Path (**WCMP**) assigns weights to servers proportional to their provisioned resources [64, 65, 221, 235, 241], which may not correspond to their actual processing capacity. However, as available server capacities change with time in elastic data centers [63] or when workloads are co-located in a shared infrastructure [69,72], these quantified capacities may not correspond to the actual processing capacities of servers.

3. Active WCMP (**AWCMP**) is a variant of WCMP. It periodically updates server weights, based on probed resource utilization information (CPU/memory/IO usage) [143, 186, 264, 265]. AWCMP requires server modifications to manage communication channels and collect observations. Higher probing frequencies help achieve more accurate server load estimation, yet lead to an increased volume of control messages and reduced throughput [59, 143].

4. Local Shortest Queue (**LSQ**) tracks for each server the number of connected flows [144, 261]. On arrival of new flow requests, LBs assign the corresponding flows to the server with the shortest queue based on observed traffic. Global Shortest Queue with Power-of-2-Choices (**GSQ2**) is an LSQ variant that leverages (i) the actual server queue occupancy, and (ii) the power of choices [246, 266].

5. Shortest Expected Delay (**SED**) derives the "expected delay" as server occupancy divided by statically configured server processing speed [236]. New flows are then assigned to the server with the minimal "expected delay."

Among load-aware LBs, TWF [261] obtains the actual queue lengths on each server via periodic out-of-band communications. It uses statistical models to reduce the impact of outdated

[2]This approach of adding a table to track migrated flows is also employed in Yoda [263], a Layer-7 LB.

observations. However, TWF assumes that all servers have the same processing speed, which is not the case with servers instantiated on heterogeneous architectures [72], malfunctioning servers, or servers running colocated workloads [69]. SED [236] statically configures server processing speeds (based on provisioned server CPU numbers) which neither reflect the actual processing speed for a given application (*e.g.,* IO-intensive) nor adapt to server health or operational status. HLB considers server occupancies and adaptively updates server processing speeds based on passive observations to improve load-balancing performance with little additional overhead.

6LB [135] and SHELL [217] offload the fine-grained load-balancing decision-making processes to servers, and allow them to hand off requests to another server using SRv6 [267], if they are already overloaded. Spotlight [143] and LBAS [187] periodically poll each server for their resource utilization information, and either classify servers into several priority classes, or predict server load states using Ridge Regression [268], to dynamically update server weights. INCAB [186] tunes server weights on receipt of notifications from overloaded servers, which are defined by manually configured thresholds.

Compared to all these LB algorithms, HLB is an out-of-the-box LB – requiring no modification in the networking systems – that passively collects networking features. It requires no monitoring or signaling among networking devices, and avoids error-prone manual configuration.

Chapter Outline

The remainder of this chapter is organized as follows. Section 6.1 studies 2 dominating factors in LB performance, and discusses the challenges of implementing load-aware LBs. Section 6.3 describes the design of HLB and its load-balancing decision-making process. Section 6.4 describes the implementation details of the testbed and the simulator for conducting evaluations. Section 6.5 presents quantitative evaluations, comparing and contrasting different load-balancing algorithms under various scenarios. Section 6.6 concludes this chapter.

6.1 Problem Formalization

A load balancing system consists of one or several LBs, connected to a set of servers in a data center. The set of servers is denoted $\mathbb{S} = \{s_1, ..., s_N\}$, with $|\mathbb{S}| = N$. Given a server, $s_i \in \mathbb{S}$, its processing speed is denoted μ_i, and its total number of jobs or flows in the queue is denoted l_i.

All LBs implement the same LB algorithm. When there are multiple LBs, the traffic (a stream of jobs) injected into the system is randomly distributed among the LBs, *e.g.,* by the edge router of the data center. Each LB thus is exposed only to a fraction of the flows in the system - those traversing that LB. The occupancy estimator of server s_i on LB j, is denoted $\tilde{l}_{ij}$ ($\tilde{l}_{ij} \leq l_i$). The observed and unobserved traffic rates on server s_i are denoted λ_i and γ_i respectively, with the total observed traffic rates as $\lambda = \sum_{s_i \in \mathbb{S}} \lambda_i$, and total unobserved traffic rates as $\gamma = \sum_{s_i \in \mathbb{S}} \gamma_i$, that subject to the system.

The following hypotheses are made for the remainder of the chapter:

- **TCP** still is the most widely used protocol in content delivery networks (CDNs) [1, 82]. Further, this assumption allows experimenting using existing flow traces [226], and does not limit the generality of the results over any other connection-oriented transport protocols (*e.g.,* QUIC [269]).

- **Finite-duration flows** are assumed for a flow q with a flow-completion time (FCT) of $T(q) < \infty$. $T(q)$ is modeled as a random variable with a uni-modal distribution (*e.g.,* a long-tail distribution as in [1, 69]).

- **Non-communicating LBs**, *i.e.,* LBs do not communicate with each other. LBs implementation complexity is reduced [65, 192], especially for high-performance LBs deployed on dedicated hardware [58, 190].

- Unless otherwise specified, modeling, simulations, and experiments rely on the hypothesis that traffic follows the Poisson distribution.

Algorithm	λ_i
ECMP	$\frac{1}{2}$
WCMP (AWCMP)	$\lambda \cdot \frac{\mu_1}{\mu_1+\mu_2}$
LSQ (GSQ2)	$\lambda \cdot Pr\{i = \arg\min_{j=1,2} \tilde{l}_j\}$
SED	$\lambda \cdot Pr\{i = \arg\min_{j=1,2} \frac{\tilde{l}_j+1}{\tilde{w}_j}\}$

Table 6.2: Traffic distribution in 2-servers LB system.

6.2 Analysis of Existing LB Algorithms

Given the defined problem space, this section provides an analytic examination of the performance of different LB algorithms in a simple setup, and analyze the impact of inaccuracies in their input parameters. The trade-off between performance and overhead of different design choices is discussed, and the remaining challenges, that motivate the design of HLB, are presented.

6.2.1 Stochastic Modeling and Simulation

The performance and operation of the LB algorithms described in section 6, as well as their sensibility to inaccuracies in their input parameters, is analyzed stochastically, on a basic load balancing setup, with 2 servers with a processing speed ratio $\frac{\mu_1}{\mu_2} = 2$ (*i.e.,* server 1 is 2x faster than server 2).

Each server has a queue of size Q, such that $0 \leq l_1, l_2 \leq Q$. Traffic arrivals and departures are modeled as Poisson processes with rates λ (observed traffic), γ (unobserved traffic), and μ_1, μ_2. With sufficiently short timeslots, it can be assumed that only one arrival or departure (at most) happens at a given timeslot (*i.e.,* $\sum_{i=1}^{2}(\lambda_i + \gamma_i + \mu_i) \leq 1$); the system is then Markovian with the state (l_1, l_2), departure rates (μ_1, μ_2), and arrival rates $(\lambda_1, \lambda_2, \gamma_1, \gamma_2)$. For simplicity, the system works at *nominal* capacity (*i.e.,* $\lambda + \gamma = \mu$). In these conditions, table 6.2 describes the traffic arrival rate λ_i assigned to server i using different LB algorithms. Note that this section studies LB algorithms (ECMP, WCMP, LSQ, SED) that correspond to fundamentally different design choices, while AWCMP and GSQ2 are variants of WCMP and LSQ respectively.

With $s_i(n)_{l_i}$ denoting the probability (or probability density function), of server s_i to have a queue length of l_i at time-step n, the transition of server occupancies between two time-steps can be described as:

$$\begin{aligned} s_i(n)_{l_i} - s_i(n-1)_{l_i} \quad = \quad & (\lambda_i + \gamma_i) \cdot s_i(n-1)_{l_i-1} + \\ & +\mu_i \cdot s_i(n-1)_{l_i+1} - \\ & -(\lambda_i + \gamma_i + \mu_i) \cdot s_i(n-1)_{l_i} \end{aligned}$$

for $0 < l_i < Q$ (corner cases are treated accordingly).

Figure 6.1 depicts the LB performance of each LB algorithm, measured as the weighted service duration of a flow ($\sum_{i\in\{1,2\}} \frac{l_i}{l_1+l_2} \frac{l_i}{\mu_i}$), for different configurations.

When the LB observes 100% traffic (*i.e.,* $\gamma = 0$) and assigns server weights based on actual processing speeds $\frac{w_1}{w_2} = \frac{\mu_1}{\mu_2} = 2$, the WCMP and SED have the best performance. By considering the state of the queues, LSQ can largely outperform ECMP. When the LB observes only 50% of traffic (*i.e.,* $\gamma = \lambda$) and the other 50% of traffic is uniformly split between the two servers ($\gamma_1 = \gamma_2$), LSQ and SED outperform WCMP, which is agnostic to instant server occupancy. However, partial traffic observation substantially degrades the performance of LSQ and SED. As an LSQ variant, GSQ2 gets global observations thus it is not subject to any impact from this source of inaccurate observation.

When LBs have inaccurate server weights (*e.g.,* in case of misconfiguration, $\frac{w_1}{w_2} = \frac{1}{2}$, while $\frac{\mu_1}{\mu_2} = 2$), WCMP and SED exhibit degraded performance even when the LB sees all the traffic ($\gamma = 0$). As a WCMP variant, AWCMP derives server weights from servers and may avoid the negative impact of misconfigurations, though with additional communication overhead. Taking both server occupancies and processing speeds into account, SED makes more informed load balancing decisions. However, while LSQ is only sensitive to partial observation, the performance of SED can be degraded by both inaccuracy sources: (i) partial observations on server occupancies, and (ii) inaccurate server weights.

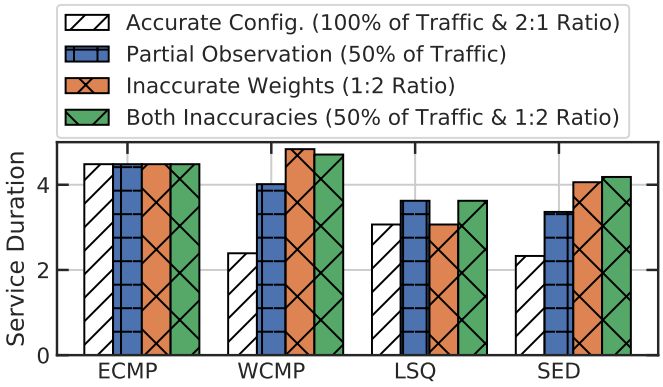


Figure 6.1: Load balancing performance for a cluster of 2 servers with different processing speeds ($\frac{\mu_1}{\mu_2} = 2$) in different scenarios for algorithms that consider different factors under system steady state ($\lambda + \gamma = \mu_1 + \mu_2$).

6.2.2 Challenges

Section 6.2.1 shows that performance degrades with 2 sources of inaccuracies (partially observed traffic, and misconfigured server weights), which are found to be present in production data centers [65, 135], which are challenging to resolve.

A single LB allows observing all network traffic, but also constitutes a single point of failure [224]. Multiple LBs are thus deployed for reliability, leading to partial observations.

Existing load-aware LBs gather observations of server occupancies either by actively probing or passively observing networking traffic, to update weights allocated to different servers and to improve workload distribution fairness [143, 144, 187, 261]. 6LB and Shell hand the load balancing decisions to the end host who knows better whether it is overloaded [135, 217]. As such, these mechanisms are exposed to the trade-offs between performance and overhead:

- *Estimating occupancies* based on passive traffic observation at the LBs requires tracking flow states – whereas incurs substantial underestimation of server occupancies, if multiple LBs exist in the system [261].

- *Actively probing server load information* allows an LB to obtain accurate but delayed server occupancies. Higher probing frequencies may increase load balancing fairness – but maintaining additional communications incurs management traffic and complexity [143].

While it is possible to avoid inaccuracies due to partial or delayed observations (as in [135, 217]), this requires server modifications to accommodate further feedback mechanisms.

Apart from partial observations and delayed updates, it is hard to determine the optimal weights for different applications that rely on different resources [270]. In addition to this, explicit weights configuration cannot capture or adapt to the dynamic networking environment. "Correctly" assigning weights to servers is therefore challenging in cloud data centers because:

- servers may have different provisioned resources;

- colocated workloads not captured by the LB may reduce available resources [69] on shared infrastructures;

- applications may have different profiles (*e.g.,* CPU-intensive, IO-intensive) and consume provisioned resources differently, whose impact is hard to quantify [26].

6.3 HLB Design

HLB dynamically distributes workloads on servers using estimations of both server occupancies and processing speeds. HLB estimates server weights and queue occupation from passive observations on network flows, by sampling flow durations and counting ongoing flows, respectively. HLB

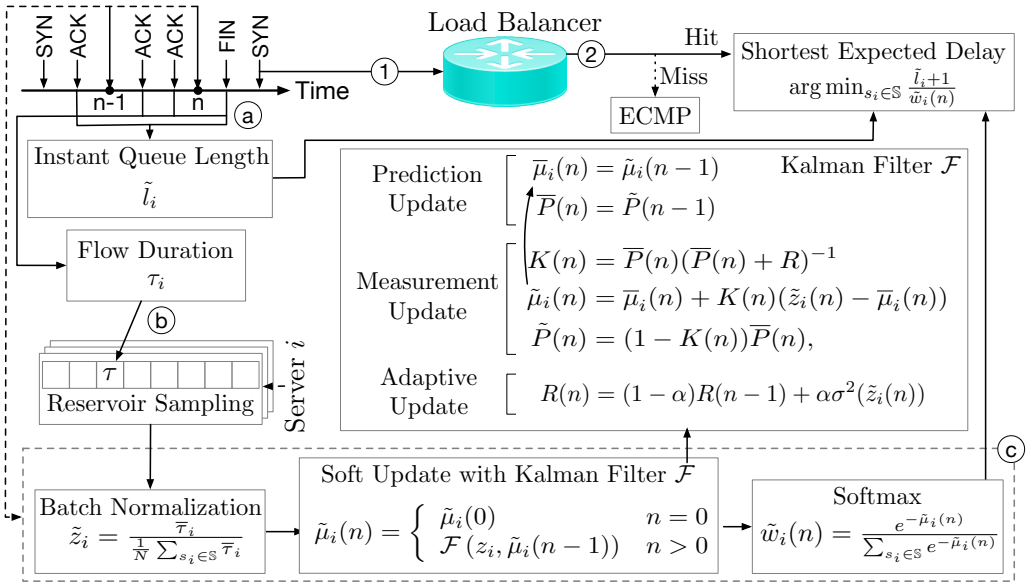


Figure 6.2: HLB workflow overview. Step ① and ② represent the decision making process on arrival of new flows. In step ⓐ–ⓒ, HLB collects networking observations and periodically learns server load states.

minimizes: (i) instant server load state estimation errors due to inaccurate observations, (ii) mismatches between assigned server weights and actual server processing speeds, and (iii) performance and management overhead, to improve load balancing performance.

HLB consists of two components: (i) a server state observation mechanism, and (ii) an algorithm that uses observed server states to place the incoming flow onto a server.

The first component tracks flow states using reserved memory locations (*buckets*) in flow tables, and extracts server state observations, without additional control or signaling. As depicted in figure 6.2, on receipt of new packets, HLB:

ⓐ inspects headers, and passively gathers observations (numbers of ongoing flows $\tilde{l}_i$ and flow durations τ),

ⓑ gathers statistical flow duration distributions on each server, using reservoir sampling,

ⓒ at each time-step n, periodically learns from gathered flow durations and updates estimated processing speeds of each server, using Kalman filters.

For the second component, when receiving ① a new flow request, HLB ② computes the hash digest of the flow ID and, maps the flow to a corresponding bucket in its flow table. HLB then integrates estimated server occupancies and processing speeds using the SED rule, to generate server state estimations, for all servers. The server with the lowest estimated load, then, receives the flow. HLB uses an adaptive approach based on passively collected observations of network flows, with no additional monitoring or management overhead – in contrast to SED, which relies on manual server weight configurations.

6.3.1 Observation Extraction from The Data Plane

As the LB sees only traffic from clients and to servers within the data center network, and not the return traffic, HLB statefully maintains flow states using flow tables, and estimates (i) server occupancies ($\tilde{l}_i$) by counting the number of ongoing flows per server, and (ii) server processing speeds by collecting flow durations on each server.

Stateful Observation Extraction

TCP flows are identified by their 5-tuples – as discussed in chapter 4 – and are statefully tracked in flow tables. As depicted in figure 6.3a, for LB, a flow exists in one of the three states. On receipt of the first TCP `SYN` packet from the client, the LB selects a server s_i to which the new flow is

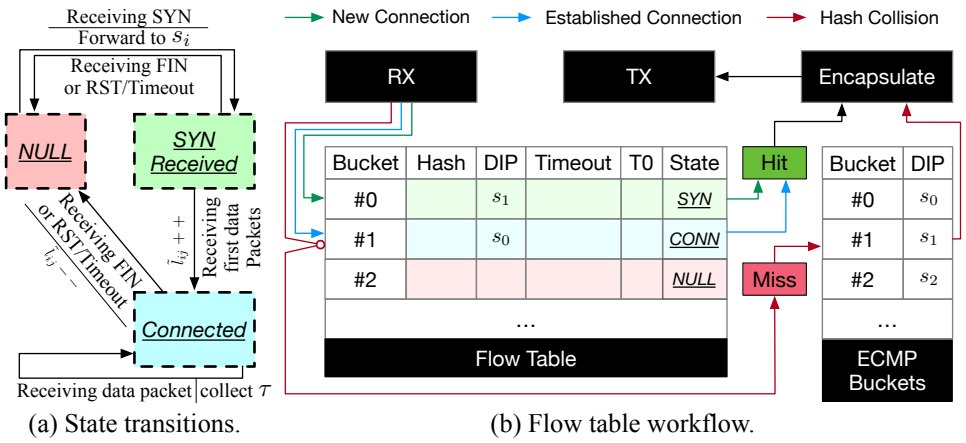


Figure 6.3: HLB's observation collection mechanism.

assigned. The LB also registers the flow state `SYN`, along with the flow ID and other corresponding information, in a bucket in its flow table. Once the flow is established, its state is updated to `CONN` (connected) on receipt of the first data packet. On flow termination (reception of `FIN` or `RST` packets), or in the case of flow timeout, the flow state is reset to `NULL` to evict the registered flow from the flow table, and the bucket is available for a new flow.

The state machine in figure 6.3a allows dynamically (i) tracking the number of ongoing flows $\tilde{l}_i$ when flows transit from `SYN` to `CONN` or from `CONN` to `NULL`, and (ii) collecting samples of flow durations from flows in state `CONN`, without interrupting the data plane.

Flow Table Workflow

HLB stores flow states in a flow table (figure 6.3b). Each bucket in this table comprises: (i) the hash digest of the TCP 5-tuple (`hash`) as flow ID, (ii) the target server ID assigned by the LB (`DIP`), (iii) flow "liveness", renewed on receipt of new packets (`timeout`), (iv) the first data packet arrival time (`T0`), for computing flow durations, and (v) the state of the flow (`state`).

A new flow is hashed and registered in corresponding bucket in the flow table, along with its assigned application server, which is identified with a direct IP address (`DIP`). Subsequent packets of the established flow are encapsulated with the target `DIP` as destination and forwarded to the corresponding server. When a bucket is not available on receipt of a new flow request, the flow gets a "miss" and is excluded by both the observation extraction and the load-aware load balancing process. In that case (hash collision[3]), HLB falls back to ECMP for the "miss"-ed flow yet guarantees PCC. A "miss" can happen in 2 cases:

- if there is no available entry for new flows;
- if no matched entry is found for connected flows;

where available buckets have `NULL` state. For flows registered in buckets that have `SYN`/`CONN` states, the counter of ongoing flows $\tilde{l}_i$ is not incremented until the first data packet is received, so that the counter is not corrupted when subjected to `SYN` flooding attacks. $\tilde{l}_i$ is decremented only if the flow ends and its state transits from `CONN` to `NULL`[4].

6.3.2 Load-Aware Load Balancing Algorithm

HLB estimates server occupancies, and server processing speeds, with l_i and τ_i, respectively, extracted from the data plane as described in section 6.3.1. To minimize the additional processing and memory overhead, HLB uses reservoir sampling to collect flow durations for each server. HLB

[3] To reduce hash collision probability, each bucket can have multiple entries. The implementation detail is omitted since it is out of the scope of this chapter.

[4] Similar DDoS mitigation mechanism using flow tables is proposed in [221], but it is out of the scope of this chapter.

Algorithm 3 Collect flow durations with reservoir sampling

```
N ← number of servers
k ← reservoir buffer size
L ← flow table size
buf ← [(0,0),...,(0,0)]   ▷ Size of k
tauBuf ← [buf,...,buf]   ▷ Size of N
t0Table ← [0,...,0]   ▷ Size of L
stateTable ← [0,...,0]   ▷ Size of L
for each packet (towards s_i from query q arriving at t) do
    qid ← Hash5Tuple(q)
    i ← s_i.id
    if t0Table[qid] == 0 & SYN packet then
        t0Table[qid] ← t   ▷ store t_0
        stateTable[qid] ← SYN
    else if t0Table[qid] != 0 & not SYN packet then
        lastState ← stateTable[qid]
        if lastState == CONN then
            randomId ← rand()
            idx ← randomId%N   ▷ randomly select one index
            τ ← t − t0Table[qid]   ▷ calculate duration
            tauBuf[i][idx] ← (t, τ)   ▷ register τ in buffer
        else
            stateTable[qid] ← CONN
        if FIN packet then
            t0Table[qid] ← 0   ▷ evict finished query
            if lastState == CONN then
                stateTable[qid] ← NULL
```

then processes collected flow duration samples using Kalman filters, to smoothly adapt server weights $\tilde{w}_i(n)$ at each time-step n, and uses both parameters, $\tilde{l}_i$ and $\tilde{w}_i$, for making load-aware load balancing decisions.

Sampling Flow Durations to Estimate Processing Speeds

Based on Aquarius, which is described in chapter 3, HLB collects flow duration information on each server using *reservoir sampling*.

The procedure of reservoir sampling that implements the state machine from figure 6.3a, which collects flow durations in flow state `CONN`, is shown in algorithm 3. The arrival times t of data packets received from a connected flow with server s_i are compared with the arrival time of the first packet $t0$ stored in $t0Table$ to compute flow durations τ. These flow durations τ, along with their corresponding packet arrival times t, are stored in a fix-sized buffer $tauBuf$ of the corresponding server s_i. These buffers thus serve as a snapshot that captures a statistical distribution of flow durations on each server, and are made available for data processing.

HLB uses flow durations to estimate and infer server processing speeds for the following reasons. First, since flows addressed to the same network application are expected to terminate with FCTs of a certain distribution $T(q)$ given sufficient privisioned resources, observed flow durations are correlated to server processing speeds depending on available resources in each server (*e.g.,* overloaded CPUs, drained memory space, congested IO). Second, flow duration is collected on receipt of each new packet of the flow in state `CONN`, thus provides measurements with finer granularity (higher update frequency) and reduced delay. Third, as an estimator of server processing speeds, the flow duration can be generalized for connection-less transport protocols (*e.g.,* UDP).

Periodic Processing Speed Inference with Kalman Filter

With flow durations gathered in reservoir buffers, HLB computes the average flow duration $\overline{\tau}_i$ on server i as observed by the LB, and then derives the normalized server processing duration measurement $\tilde{z}_i(n) = \frac{\overline{\tau}_i}{\frac{1}{N}\sum_{s_i \in \mathbb{S}} \overline{\tau}_i}$ at each time-step n. Between different time steps, the samples of $\tilde{z}_i$ may have high variance, and their values can change significantly. As a function of flow duration τ_i, $\tilde{z}_i$ is correlated to server processing speeds. In addition, $\tilde{z}_i$ also depends on the distribution of $T(q)$, which may vary in time. To decouple the possibly abrupt variations of $T(q)$, and adapt

to actual server states, HLB uses Kalman filters $\mathcal{F}$ to smoothly update server processing duration estimations $\tilde{\mu}_i(n)$ at step n:

$$\tilde{\mu}_i(n) = \begin{cases} \tilde{\mu}_i(0) & n = 0 \\ \mathcal{F}\left(\tilde{z}_i, \tilde{\mu}_i(n-1)\right) & n > 0 \end{cases},$$

where $\tilde{\mu}_i(0)$ is initialised as 0.5 on all servers.

The Kalman filter takes streams of measurements observed over time and tracks the estimated system state as well as the level of uncertainty. It works in the following 2-step process:

- Prediction update:

$$\begin{aligned} \overline{\mu}_i(n) &= \tilde{\mu}_i(n-1) \\ \overline{P}(n) &= \tilde{P}(n-1), \end{aligned}$$

- Measurement update:

$$\begin{aligned} K(n) &= \overline{P}(n)(\overline{P}(n) + R)^{-1} \\ \tilde{\mu}_i(n) &= \overline{\mu}_i(n) + K(n)(z_i(n) - \overline{\mu}_i(n)) \\ \tilde{P}(n) &= (1 - K(n))\overline{P}(n), \end{aligned}$$

where $\overline{\mu}_i$ is the predicted processing duration, $\overline{P}$ is the expected prediction error, R is the measurement variance, $\tilde{P}$ is the expected estimation error, and K is the Kalman gain.

The only parameter to be tuned is the measurement variance R, which can be configured based on the expected noise in measurements z_i. The value of R can be increased if the flow durations of input traffic vary a lot. To avoid manual configuration, R can be adaptively estimated using the variance of measurements $\sigma^2(z_i)$, and can be smoothly updated as $R(n) = (1-\alpha)R(n-1) + \alpha\sigma^2(z_i(n))$, where $\alpha = 0.01$ helps regularize the variation in z_i.

Merging Occupancy and Processing Speed

The server processing duration estimation from the latest step $\tilde{\mu}_i(n)$ is used to calculate the weight $\tilde{w}_i$ assigned to each server in the following form:

$$\tilde{w}_i(n) = \frac{e^{-\tilde{\mu}_i(n)}}{\sum_{s_i \in \mathbb{S}} e^{-\tilde{\mu}_i(n)}} \in (0, 1),$$

which normalizes the negation of server processing duration estimations, and creates a probability distribution centered around the servers with higher estimated processing speeds.

After obtaining both measurements, *i.e.,* server occupancy $\tilde{l}_i$ and inferred processing speed $\tilde{w}_i(n)$, a score is computed from these two factors using SED. During the time interval of a step n, the target server s_i is selected by:

$$\arg\min_{s_i \in \mathbb{S}} \frac{\tilde{l}_i + 1}{\tilde{w}_i(n)},$$

where the added 1 on the numerator takes the new incoming flow into account. This form gives priority to servers with high estimated processing speeds and low occupancies.

6.4 Experimental Setups

To evaluate load balancing performance in different realistic setups, subject to both partial traffic observations and potential server weights misconfigurations as described in section 6.2, a physical testbed similar to the one used in chapter 3 is configured and deployed, and an event-based simulator is implemented. This allows testing with realistic network traces – when available – and large-scale simulated scenarios.

6.4.1 Testbed

Experiments are conducted on a testbed running network traces on physical servers. The experimental platform consists of VMs representing clients, an edge router, load-balancers, and Apache HTTP server agents as depicted in figure 6.4.

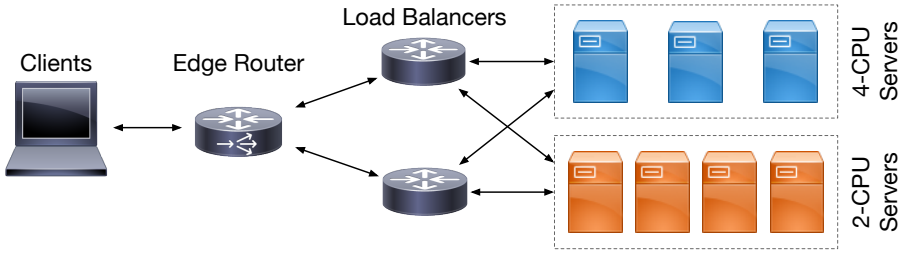


Figure 6.4: An example of network topology with two groups of 7 servers.

Load-Balancers

HLB, along with other LB algorithms, is implemented as a plugin to VPP [121], a performant packet-processing stack that runs on commodity CPUs. The number of buckets (160 bits/entry) in the flow table for stateful LB algorithms is set to 65536. Each load balancer is connected to all application servers.

Apache HTTP Servers

Running on each server VM, an Apache HTTP servers gather two metrics every 200ms as "ground truths" for the occupancies: CPU utilisation, and the number of busy Apache worker threads[5]. The server configurations are the same as in section 3.2.3. Servers are organized into 2 groups, where server capacities within a group are identical, yet may be different between the 2 groups.

System Platform

The system platform is similar to the configuration described in section 3.2.3. Depending on the scale of experiments, the testbed resides on 2 to 4 physical machines, each with a 48-CPU Intel Xeon E5-2690 CPU. An 8-CPU traffic generator, representing the clients, and a 4-CPU edge router device, run on one machine. The other machine hosts 4-CPU VMs running LB instances on VPP. The number of CPUs of Apache HTTP servers may vary from 2 to 8 using different configurations. All VMs are on the same Layer-2 link, with statically configured routing tables. The mean round-trip-time (RTT) between 2 network devices is 0.322ms and the standard deviation of RTT is 0.037ms.

Wikipedia Replay

In order to evaluate LB performance using realistic workloads, LB algorithms are evaluated in a data center setup that provides typical Web services. To emulate Wikipedia server clusters, the same Wikipedia 24 hour replay trace as in section 3.2.3 is applied. The sizes of Wiki pages follow a long-tail distribution, whose average and standard deviation are both 12KiB. The traffic generator is used to generate a MediaWiki access trace and to record page response times.

6.4.2 Simulator

To compare and contrast the performance of load balancing algorithms in various scenarios, in particular those difficult or where no network trace exists to evaluate in testbeds (*e.g.,* large-scale data center networks), an event-driven simulator is implemented, based on hypotheses described in section 6.1. The simulator implements the network topology as in figure 6.4, where each load balancer is connected to all servers.

Real-world network applications can be CPU-bound or IO-bound [271, 272]. The simulator allows configuring applications that require multi-stage processes switching between CPU/IO queues (figure 6.5). For instance, a flow request for a 2-stage application is first processed in the CPU queue, then in the IO queue, before being sent back to the client.

[5]CPU utilization is calculated from the file `/proc/stat` and the amount of Apache busy threads is assessed via Apache's *scoreboard* shared memory.

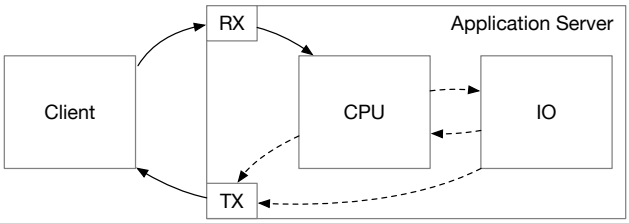


Figure 6.5: Illustration of the processing states of flow requests. Solid and dashed arrows represent deterministic and non-deterministic procedures respectively.

Two different processing models are used for CPU and IO queues, respectively. A FIFO model is defined for CPU queues, and flows that arrive when no CPU is available will be blocked in a backlog queue until there is an available CPU. IO is simulated as a simple processor sharing model, in which the instantaneous processing speed is the inverse of the number of flows in the IO queue. The backlog queue length of each server is configured as 64. Connections that arrive when the backlog queues are full will be rejected, with 40s timeout. Communication latency between 2 devices is uniformly distributed between 0.1ms and 1ms.

6.4.3 Benchmark LB Algorithms

All the 6 LB algorithms described in section 6 are implemented to be evaluated in the simulator. Similarly to SED, AWCMP is implemented to be aware of the server speed difference ratio, and with the server occupancies (*i.e.,* queue lengths) on each server polled periodically. The default update frequencies of server weights are the same for AWCMP and HLB (every 0.5s).

In the simulator, an *Oracle* LB algorithm is implemented, which distributes flows to the server which is expected to finish all its job with the lowest delay (including the new flow). The Oracle LB is aware of the remaining time of each flow, which is otherwise not observable for layer-4 LBs. By adding the Oracle LB, the load balancing performance of HLB and other LB algorithms can be compared to the potential upper bound of performance, corresponding to "perfect" network and server state observations.

For algorithms that consider instant server occupancies, power-of-2-choices can be applied additionally. In addition to GSQ2, the simulator thus also implements LSQ2, SED2, HLB2, and Oracle2, to study the impact of partial observations and suboptimal load balancing decisions.

6.5 Evaluation

Simulations and experiments are conducted ($\geq$ 10 runs for each setup) to answer the following questions:

- How does the performance of HLB, when subjected to different traffic rates, compare with existing LB algorithms (section 6.5.1);
- What is the impact of heterogeneity in server capacities on load-balancing performance (section 6.5.2);
- Are partially observed server occupancies representative of server load states (section 6.5.3);
- How does HLB perform using different configurations of system parameters (section 6.5.4);
- Can HLB adaptively react to dynamic networking environments (section 6.5.5);
- What is the performance overhead (section 6.5.6).

6.5.1 Performance with Different Traffic Rates

This section presents an overall performance evaluation of HLB, compared to other LB algorithms, when subjected to different traffic rates with both a real-world network trace replay and a large-scale simulation.

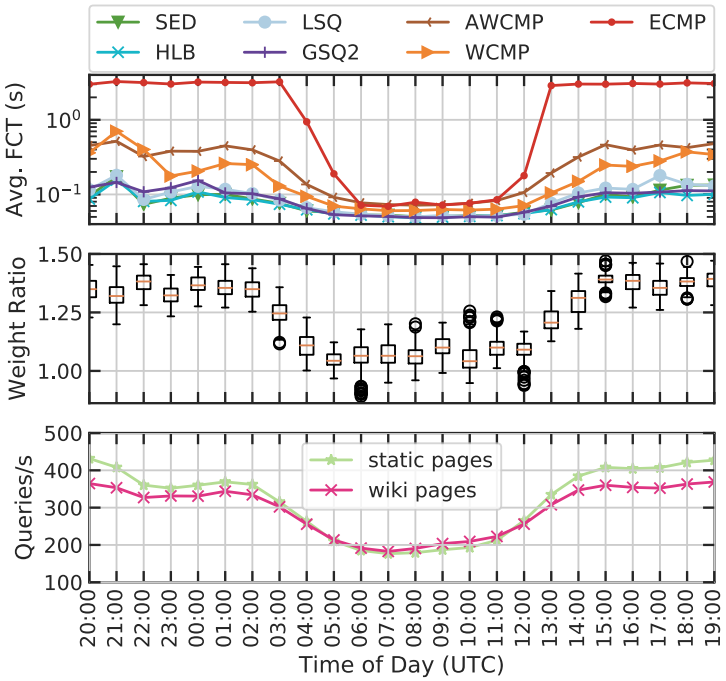


Figure 6.6: [Testbed] 24-hour Wikipedia trace replayed using different LB algorithms. Average FCTs (top), ratio between weights assigned to the 2 groups of servers by HLB (middle), and traffic rate (bottom) are depicted.

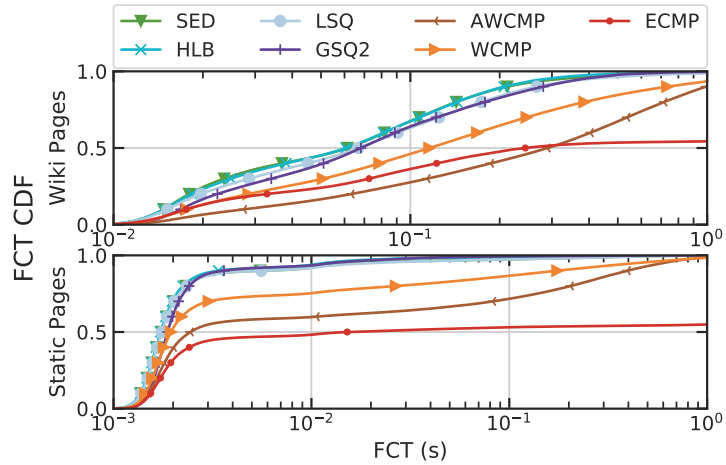


Figure 6.7: [Testbed] FCT CDF comparisons for two types of requests in the 24-hour Wikipedia replay.

24-Hour Trace Evaluation

Samples of 600s duration are extracted from the 24-hour Wikipedia trace and replayed on the testbed. The results are depicted in figure 6.6.

During the off-peak period from `5:00` to `11:00` UTC when servers are under-utilized, all LB algorithms show similar performance. As traffic rates grow, HLB sees less increase in FCT compared with other LB algorithms, which is indicative of improved resource utilization and performance gains achieved by the load-balancing decisions using HLB. LSQ and GSQ2 assume all servers have the same processing capacities, and aim at maintaining equal queue lengths on all servers. When subjected to heavier traffic, servers with less processing capacities receive more workloads than they can process, and their queues grow full. LSQ and GSQ2 thus become less performant than SED and HLB during the peak periods.

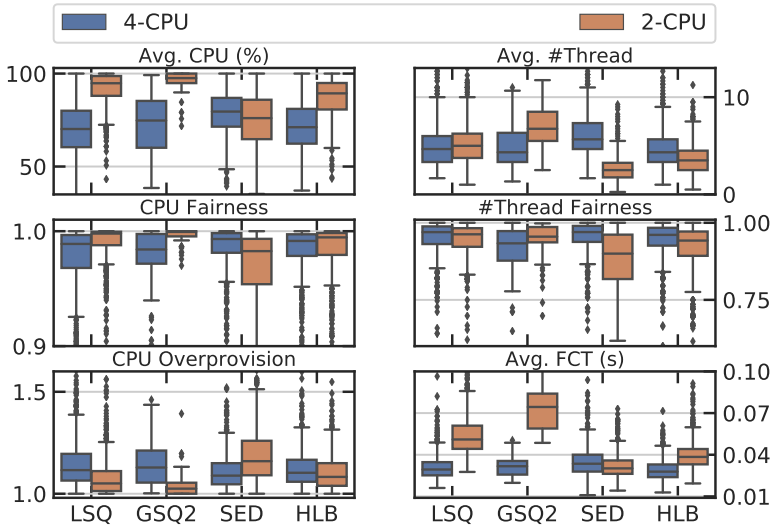


Figure 6.8: [Testbed] Comparison on server resource utilizations using network traces from hour `20:00` (800 queries/s) in the 24-hour Wikipedia replay.

As depicted in the middle plot in figure 6.6, HLB has no *a-priori* knowledge of server capacity differences (the ratio of CPU numbers between the 2 server groups is 2), yet it can passively learn these differences from observations, and achieve similar performance as that of SED. During off-peak hours, as servers have enough processing capacities, thus no additional queuing delay occurs, HLB does not differentiate server processing speeds. When subjected to heavier traffic rates, less powerful servers become "overloaded" and see higher queuing delays, which increase their corresponding flow durations. The increased flow durations thus inform HLB of the server processing speed differences.

Figure 6.7 depicts the FCT CDF of each LB algorithm for two types of requests: (i) static pages, and (ii) Wiki pages[6]. For both types of requests, HLB, SED, LSQ, and GSQ2 show notable performance gains when compared with other LB algorithms. For Wiki pages, which are more computationally expensive (CPU-bound) to load than static pages, HLB achieves 23.66% and 26.43% less 90p FCT than LSQ and GSQ2 respectively. Of particular note is, that HLB achieves the same performance as SED, but does not require manual configurations of server weights. For static pages, which are IO-bound, HLB achieves 38.22% less 90p FCT than SED, which demonstrates that HLB can effectively achieve improved performance.

Workloads Distribution

To understand the workload distribution, this section studies 6 resource utilization metrics: mean CPU usage, fairness, overprovision factor, mean number of busy threads, fairness of number of busy threads, and finally the average FCT. Given a random variable X, the fairness of X is defined as $F = \frac{\mathbb{E}(X)^2}{\mathbb{E}(X^2)} \in [0, 1]$ [252]. The overprovision factor of X is computed as the maximum load over the average load at each time step $\frac{\max(X)}{\bar{X}} \in [1, \infty)$ [65].

The performance of the 4 best performing LB algorithms in the 24-hour trace evaluation, are further analysed – still on a test platform with servers of different capacities. As depicted in figure 6.8, SED achieves balanced average CPU usage between the 2 server groups, thus SED balances the average FCT, since it is aware of both server occupancies and processing speeds, thus assigns `2.3x` flows to 4-CPU servers than to 2-CPU servers. Unlike SED, LSQ and GSQ2 balance queue lengths between the two server groups. They ignore the capacity differences, and overload 2-CPU servers which experience FCTs increased by 71% and 131%, using LSQ and GSQ2 respectively, over FCTs on 4-CPU servers. HLB learns to give less aggressive weights than SED without any *a-priori* knowledge and assigns 35% more requests to 4-CPU servers than to 2-CPU ones. The queue lengths between the 2 groups of servers are less imbalanced than SED yet more proportional to their processing speeds than LSQ and GSQ2.

[6]Wiki pages are identifiable by the string `/wiki/index.php/` in URLs.

Capacity Ratio n	1x	2x	4x
Testbed Group 1	5 × 2-CPU	4 × 2-CPU	2 × 2-CPU
Testbed Group 2	5 × 2-CPU	3 × 4-CPU	2 × 8-CPU
Simulator Group 1	64 × 1-CPU	64 × 1-CPU	64 × 1-CPU
Simulator Group 2	64 × 1-CPU	64 × 2-CPU	64 × 4-CPU

Table 6.3: Configurations with different server capacity ratios.

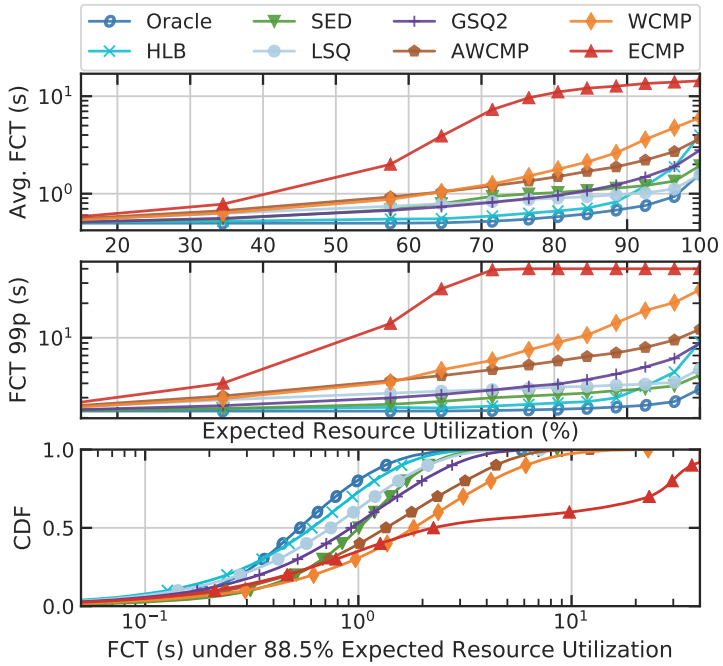


Figure 6.9: [Simulator] FCT comparison using 2x server capacity ratio when subjected to different traffic rates.

Large-Scale Simulation

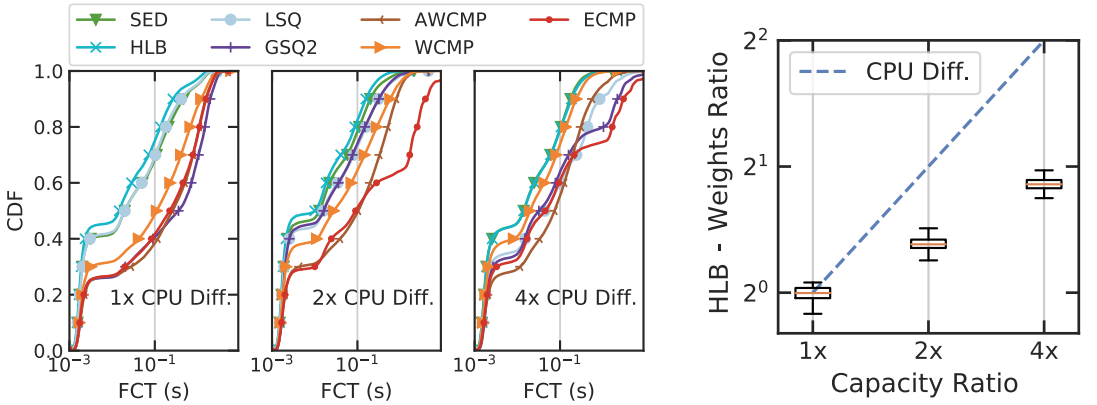
To study LB performance in large-scale data center networks, simulations are conducted in a setup with 4 LBs and 128 servers, half of which has 1 CPU each, while the other half has 2.

The input traffic is a Poisson stream of single-stage CPU-bound application queries. The exponential distribution of FCTs, $T(q) \sim Exp(0.5)$, has an average of 500ms. Traffic rates are normalized concerning the total provisioned resources. Results are obtained from multiple runs, each consisting of 80k network flow requests[7].

As depicted in figure 6.9, server occupancy is the dominant factor of LB performance when traffic rates are heavier, and LB algorithms that are occupancy-aware achieve better performance. Consistent with the testbed experiments in section 6.5.1, HLB yields a lower FCT than other LB algorithms – even though HLB has no *a-priori* knowledge about the server capacity difference. HLB achieves similar performance to the Oracle from moderate traffic rates up to 90% expected resource utilization – when the average FCT becomes more than 5x higher than the expected FCT (200ms) – which covers most cases in data center networks [69]. When subjected to 88.5% expected resource utilization, HLB achieves 24.64% and 25.59% less 90p FCT than LSQ and SED respectively.

The **take-away** for these experiments, and the subsequent simulations, is that, even without

[7]There are 5 runs in total. From each run, only results from the interquartile range of the simulation time are used for analysis, to guarantee that all the metrics are collected using the Poisson stream of input traffic.



(a) (b)

Figure 6.10: [Testbed] Comparison using different server capacity ratios using trace from hour `23:00` (680 queries/s). Figure (a) compares FCT CDF using 3 ratio configurations of CPU capacity differences with different LB algorithms. Figure (b) compares the server weights ratio between the two server groups generated by HLB with the actual provisioned server capacity ratios.

manual *a-priori* configurations, HLB achieves *better load-balancing performance* by learning server capacity differences, which allows fair distribution of workloads to servers. Tracking server occupancies further allows HLB to improve load-balancing performance, when the servers are subjected to heavy traffic rates.

6.5.2 The Impact of Heterogeneity in Server Capacities

Given the results from section 6.5.1, this section will further explore the impact of heterogeneity in server capacity.

Testbed Experiments

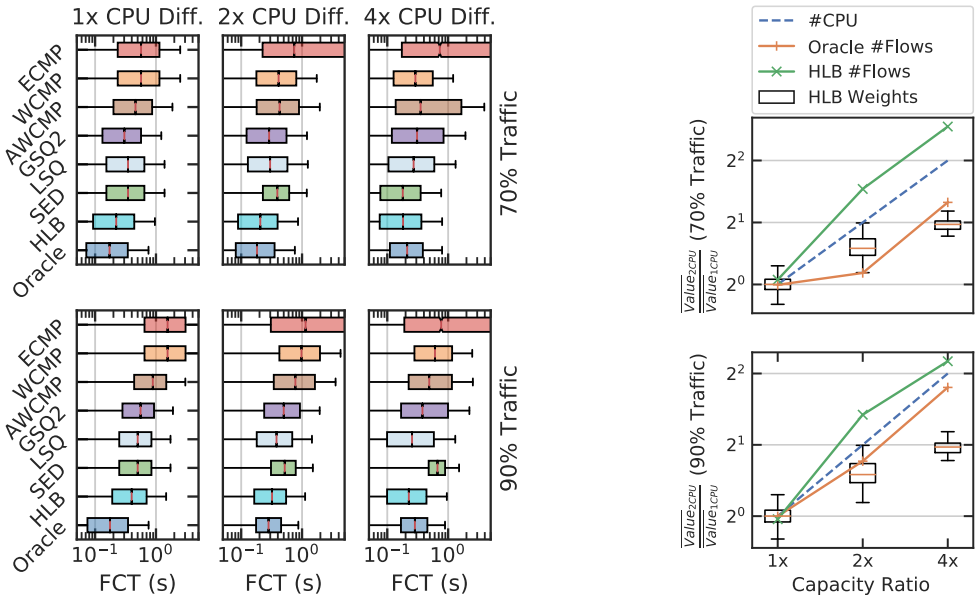
Three different configurations of servers, with a total of 20 CPUs as per table 6.3, are tested with all the different LB algorithms.

As depicted in figure 6.10, for larger differences between server processing capacities, SED and HLB outperform LSQ and GSQ2. The ratio of server weights computed by HLB between the two groups of servers[8] are lower than the ratio of provisioned resources. This is because the estimation of server processing capacities is based on flow durations, which capture not only the server processing time, but also the queuing delays, which are not proportional to server processing capacities. This causes HLB to "under-estimate" the server processing speed differences between the two groups of servers, yet HLB still achieves lower FCT than SED, since the traffic rate does not push the resource utilization to the limit.

Large-Scale Simulation

Two Poisson streams of input traffic with $T(q) \sim Exp(0.5)$ are applied and consume respectively 70% and 90% provisioned resources on average. Three setups are configured as per table 6.3, with the results of the simulation depicted in figure 6.11.

[8] If not specified, the ratios used in this chapter are calculated as the average value of the second group of servers over the average value of the first group of servers.



(a) **(b)**

Figure 6.11: [Simulator] Comparison using different server capacity ratios when subjected to 70% (top) and 90% (bottom) expected resource utilization. Figure (a) compares the FCT distribution and figure (b) compares the ratio of weights and load distribution between two groups of servers.

The results, when the testbed is subjected to moderate traffic rates, show similar trends as the experiments in section 6.5.1. As depicted in figure 6.11a[9], when all servers have the same processing capacity, SED exhibits performance equivalent to LSQ.

With a moderate traffic rate (70% expected resource utilization), HLB and SED are the optimal LB algorithms, especially when the server capacity differences grow. With a heavy traffic rate (90% expected resource utilization), however, LSQ becomes better than SED. This is because the FCT is composed of the network delay, the queuing delay, and the server processing delay. At high resource utilization, the queuing delay becomes dominant, whereas, at low resource utilization, the server processing time becomes significant.

Another observation from figure 6.11a is the performance degradation of SED at high resource utilization. This is because the partial observations on network traffic in presence of 4 LBs, make the server load state evaluation function of SED de-correlated from the actual server load state. For instance, when the provisioned resource difference ratio is $1:4$, SED assigns 12.14 times more workloads to powerful servers, causing them to be overloaded. This will be studied further in section 6.5.3.

As depicted in figure 6.11b, HLB achieves better performance in all the tested scenarios, by dynamically adjusting weights based on the inferred server states: the ratio of server weights calculated by HLB between the two server groups is correlated to, yet lower than, the actual ratio of provisioned resources. When comparing the number of distributed network flows, while HLB uses a different strategy than the Oracle, and prioritizes the servers with higher processing capacities, HLB is adaptive and achieves good performance in different scenarios.

The **take-away** from this set of experiments and simulations is, that when an LB algorithm considers server processing capacities when making decisions, it is important that this information is accurate – at least in as much as the provisioned resource difference ratio is accurate. This can be done either through *a-priori* configurations (as in SED), or through observing and learning (as in HLB). Likewise, as the simulations showed that the impact of the provisioned resources (number of CPUs) on a server on the FCT depends on the overall resource utilization, it is important that the weight for a given server can be adaptive also to the traffic rate; especially when LBs have only partial observations on server load states.

[9]The boxplots used in this chapter are standard boxplots, showing the interquartile ranges and the medians.

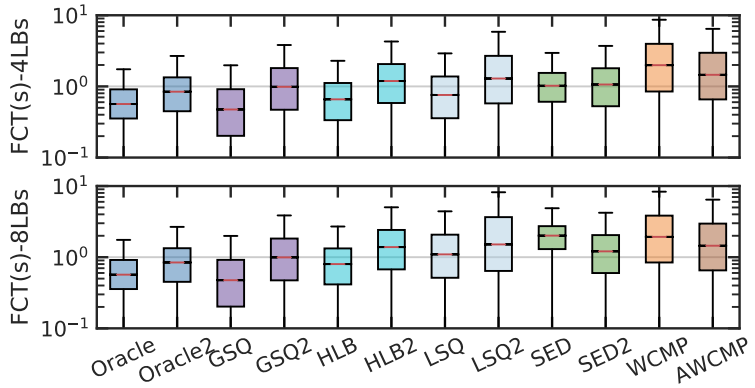


Figure 6.12: [Simulator] The impact of the application of power-of-2-choices on load-balancing performance under 90% expected resource utilization.

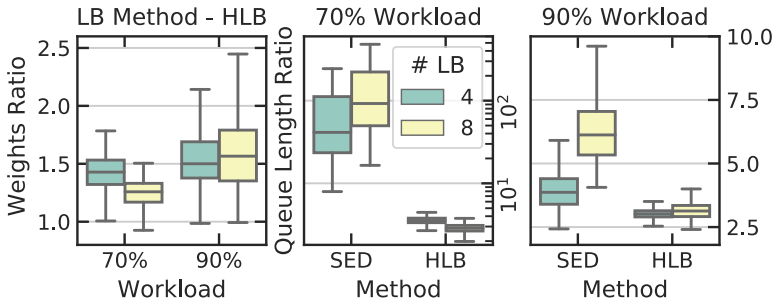


Figure 6.13: [Simulator] Different numbers of LBs give different levels of partial observations, which impact the weights ratio between the two groups of servers computed by HLB (left), and the ratio of queue lengths between the two groups of servers when subjected to different traffic rates (middle and right).

6.5.3 The Representativeness of Partial Observations

As seen in section 6.5.2, when there are multiple LBs, the performance of SED degrades. This section, therefore, studies the representativeness of partial observations of the server occupancies and suboptimality of power-of-2-choices.

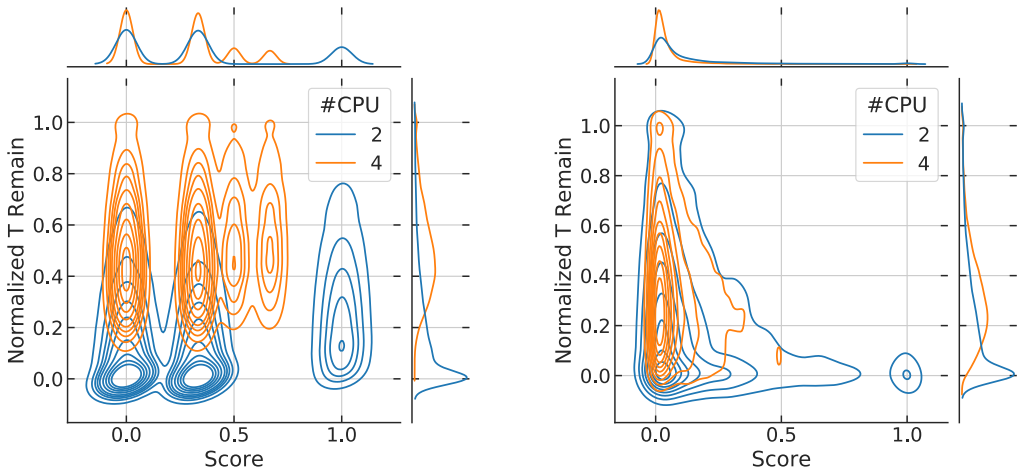
Partial Observations on a Large-Scale

Following section 6.5.2, this section uses the `2x` simulator configuration in table 6.3 to study the impact of partial observations in a large-scale data center network. Two configurations with 4 and 8 LBs are applied to compare different degrees of partial observations. The FCT of the input traffic has the same distribution as in section 6.5.2, *i.e.,* $T(q) \sim Exp(0.5)$. In addition to the studied LB algorithms, this section also studies the power-of-2-choices variants of LB algorithms, that take the server occupancies into consideration: GSQ, HLB, LSQ, SED, and the Oracle, since they may be potentially impacted by partial observations.

Figure 6.12 depicts the FCT distributions using different LB algorithms. As expected, the application of power-of-2-choices degrades the performance of GSQ, LSQ, and HLB for saving compute cycles. SED, however, shows the opposite results and achieves lower FCT with SED2 when there are 8 LBs.

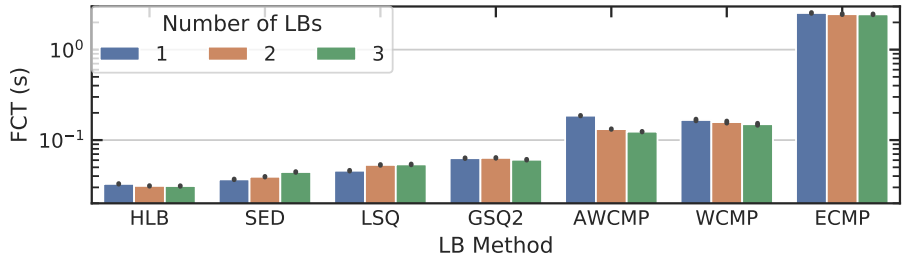
Understanding Workloads Distribution for SED and HLB

Figure 6.13 depicts the queue length ratios between the two groups of servers when using SED and HLB, along with the server weights computed by HLB. SED prioritizes, and steers most workloads to servers with higher weights.

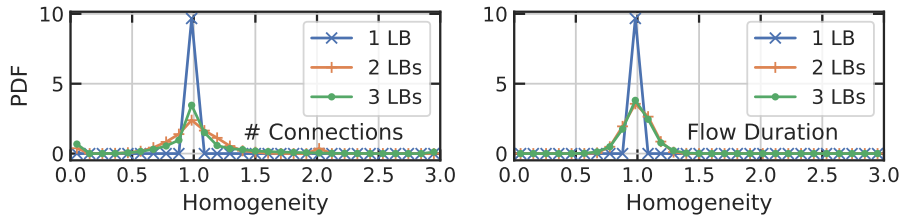


(a) SED score $\frac{\tilde{l}_i+1}{w_i}$. **(b)** HLB score $\frac{\tilde{l}_i+1}{\tilde{w}_i}$.

Figure 6.14: [Simulator] With 8 LBs, when subjected to a traffic rate that consumes 90% resource utilization, the correlation between normalized residual processing time and computed server scores using SED and HLB.



(a) Average FCTs.



(b) HLB observation homogeneity (the closer to 1 the better).

Figure 6.15: [Testbed] Comparison using different number of LB devices.

With more LB devices, the server occupancy observations become more partial and less representative of the actual server occupancies, and the estimations computed by SED are not correlated to the actual workloads on the servers (figure 6.14a). Based on this incorrect estimation, SED assigns 74.2% network traffic to more powerful servers in presence of 8 LBs, leading to worse results than the randomness induced by the power-of-2-choices of SED2. The estimations by HLB, on the other hand, are more accurate and make the two groups of servers subject to similar workloads (figure 6.14b).

Partial Observations on Experimental Testbed

To verify the observations obtained from section 6.5.3 and section 6.5.3, the `2x` testbed configuration in table 6.3 is used with various numbers of LBs. This section applies a 600s Wikipedia trace with an average traffic rate of 680 queries per second. As depicted in figure 6.15a, the performance of SED and LSQ degrades when the presence of more LBs make their observations on server occupancies more partial and less representative of the actual server occupancies.

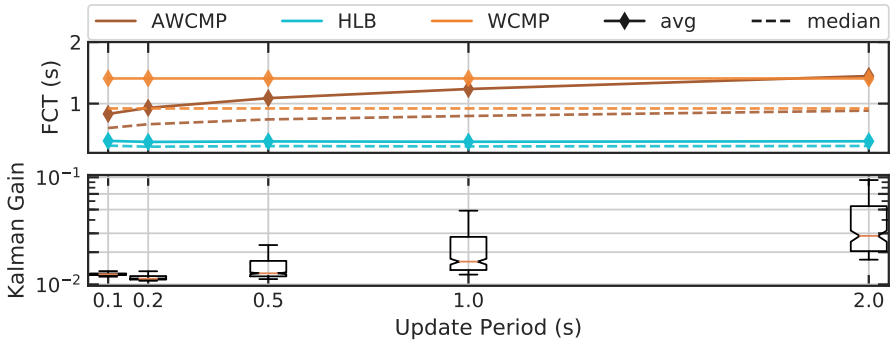
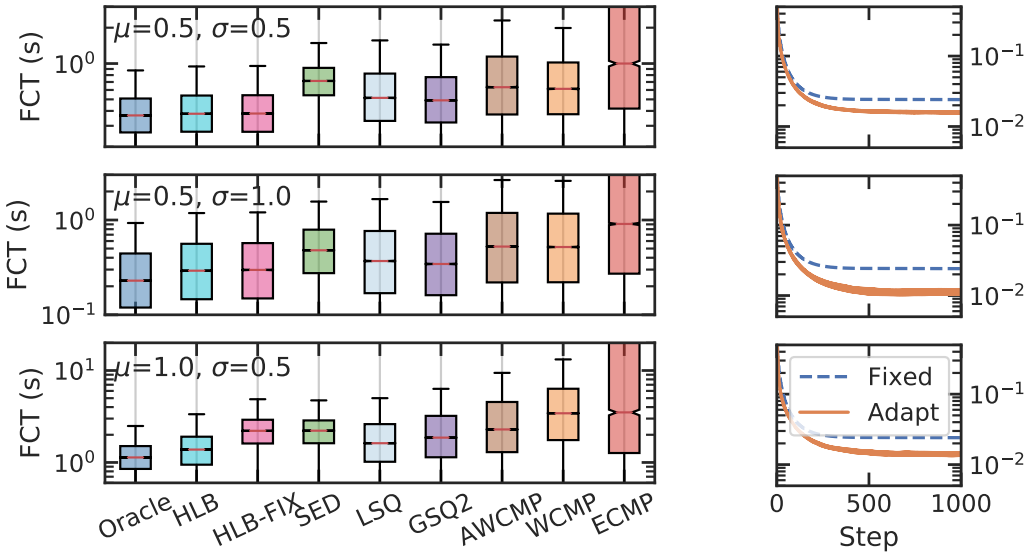


Figure 6.16: [Simulator] Comparison using different weights updating frequency when subjected to 90% expected resource utilization.

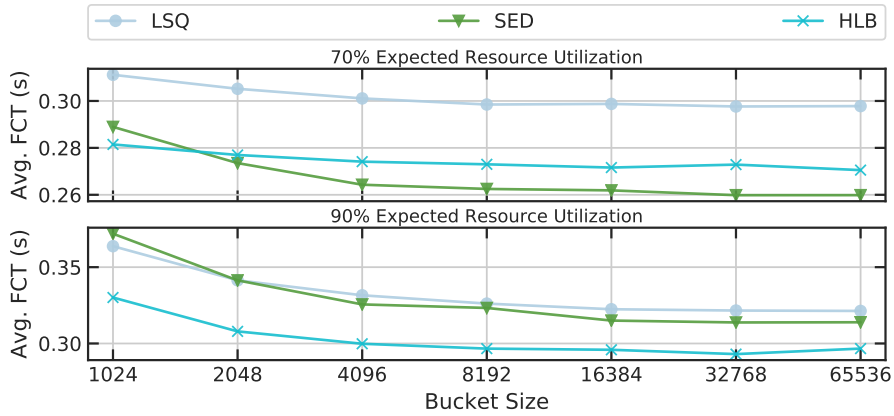


(a) Measured FCT Distribution (s). **(b)** Kalman Gain.

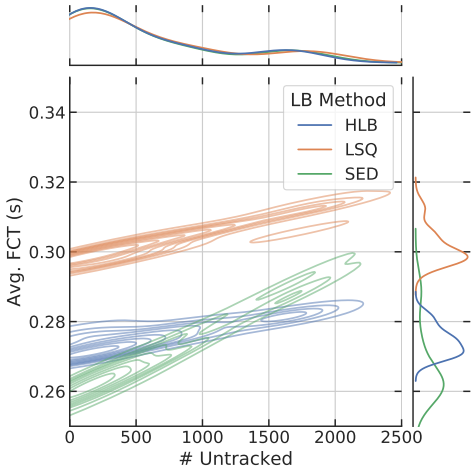
Figure 6.17: [Simulator] Different input traffic FCT distributions when subjected to 90% expected resource utilization.

AWCMP obtains better performance since the presence of more LBs increases the polling frequency of the group of LBs thus making observation granularity finer. Normalizing statistically significant measurements across servers, HLB is less impacted by the factor of partial observations among all LB algorithms. In a setup with M LBs passively observing N servers, denote $z_{ij} = \frac{x_{ij}}{\sum_j x_{ij}}$ where x_{ij} is the measurement of server j made by LB i at a given time step. The distribution of $M\frac{z_{ij}}{\sum_i z_{ij}}$ is compared in figure 6.15b, to quantify the HLB observation homogeneity across LBs *w.r.t.* global measurement distribution. The homogeneity of the number of ongoing flows is less centered and has increased outliers with more LBs, yet the homogeneity of flow durations is less sensitive to the growth of LB numbers, which helps HLB gainfully use observed information for server load ranking.

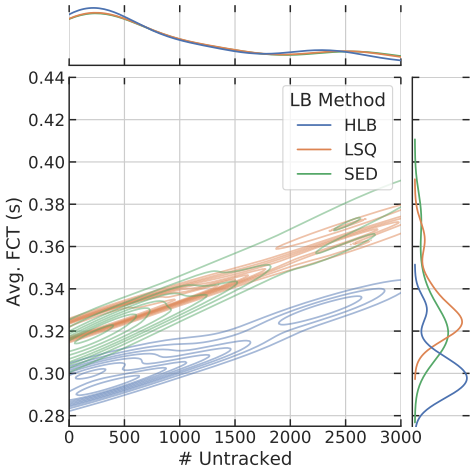
The **take-away** from these experiments and simulations is that, more partial observations can be less representative of the measured system. When having only partial observations on server occupancies available, such as is the case for a multi-LB set-up, the superiority of combined metrics when using HLB emerges. Using Kalman filters, HLB accumulates reliable observations on server processing speeds over time in the history, and predicts the server processing speeds at the next time-step. Integrating both server occupancy and processing speed, HLB is less sensitive to partial observations.



(a) Average FCTs achieved using different bucket sizes in flow tables.



(b) 70% resource utilization.



(c) 90% resource utilization.

Figure 6.18: [Simulator] Comparison using different flow table bucket size.

6.5.4 Sensitivity Analysis

This section studies the potential impacts on LB performance of different conditions and system parameters, namely, (i) weights update frequency, (ii) flow table size, and (iii) RTT between clients and servers.

Weights Update Frequency

AWCMP and HLB periodically update server weights. The time interval between two consecutive updates is a system parameter. In this section, 5 periods are applied to study their impacts. Figure 6.16 shows that higher weight updating periods degrade AWCMP performance. Since AWCMP uses the same set of weights during a complete time interval, the correlation between server weights and server load states decreases, until the next update. HLB on the other hand, is less affected by the update interval because of its adaptive Kalman gain. When the update interval is short, HLB generates higher Kalman gains, and assigns more weights to the newly computed load state estimations to catch up with the dynamics of the environment.

The adaptive Kalman filter also allows adapting server weights accordingly when facing input traffic with different FCT distributions. Three lognormal distributions of FCTs are applied on the `2x` simulator configuration as in table 6.3 with 4 LBs. As depicted in figure 6.17, when facing input traffic with different FCT distributions, HLB can achieve performance close to the Oracle, without manual configuration or additional control messages. The measurement noise R of HLB-FIX is configured as 0.5 while HLB has no hard-coded R. As depicted in figure 6.17b, HLB has different convergence of Kalman gain corresponding to different FCT distributions. It helps HLB achieve better performance than HLB-FIX when the average FCT of input traffic is higher.

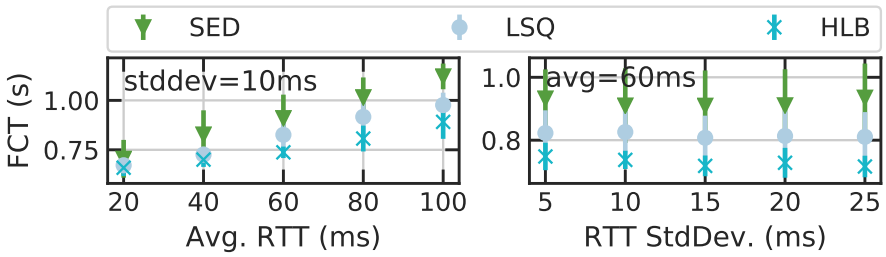


Figure 6.19: [Testbed] FCT (avg. ± stddev) comparison using different RTT distributions between clients and servers.

Pitfalls of Statefulness

LSQ, SED, and HLB track flow states in flow tables, and the number of buckets in flow tables is another system parameter. Flow tables with more buckets can store more flow states, and can therefore enable higher observation accuracy. Untracked flows will be statelessly forwarded to a random server by looking up ECMP bucket tables, without considering server load states. However, managing large flow tables consumes more memory space, which is costly on dedicated hardware [190, 217].

To quantify the performance degradation when memory space is limited, simulations are conducted on a data center network with 4 LBs and 256 servers. Half of the servers have 2 CPUs while the other half have 4 CPUs. Reducing bucket sizes from 65536 to 1024 leads to more untracked flows, and thus degraded load-balancing performance. Figure 6.18 shows that HLB is more robust to traffic rate changes and less sensitive to the flow table size than is LSQ and SED. LSQ and SED use only the observations of server occupancies, while HLB can infer server processing speeds based on measured flow durations, and thus it is less impacted by untracked flows. HLB achieves the best performance with the minimal bucket size, which makes it an interesting candidate for hardware implementations.

RTT Between Clients and Servers

The evaluations above have demonstrated that HLB can improve load-balancing performance for intra-data-center services, where clients locate within the same data center network. To understand whether HLB can benefit the use cases where clients connect to servers through the Internet, this section studies the impact of different distributions of RTT between clients and servers. Intuitively, given a request from a client, the load-balancing decision is not biased by the RTT between this client and the server cluster. HLB makes load-balancing decisions and assigns servers based on its estimations of server load states, which depend on the distribution of sampled flow durations. The flow duration measurements consist of server processing time and RTT between clients and servers. Since requests with different RTTs are indiscriminately distributed across servers yet server processing time varies depending on instant server load states, HLB normalizes flow durations across servers and reserves the variance of server processing speeds. Therefore HLB is not sensitive to different RTT distributions.

To provide an empirical study, using the same setup as in Secion 6.5.1, the RTT between clients and the edge router is shaped using `netem` to follow Pareto normal distributions with different means and standard deviations [273]. Added delays have 25% dependency on their previous values. As depicted in figure 6.19, the FCT grows linearly with the increase of the mean RTT.

With the combination of reservoir sampling and Kalman filters, HLB removes the unimodal RTT distribution so that the processed flow durations still reflect server processing speed differences. In all scenarios, HLB remains superior to LSQ and SED. This shows that HLB is not sensitive to the change of RTT between clients and servers.

The **take-away** in this section is, that HLB is less sensitive to server weights updating frequency than are active LB algorithms. It also requires less memory space than other stateful LB algorithms, which makes it more hardware-friendly. HLB is not sensitive to RTT between clients and servers thus it can potentially benefit more than just intra-data-center applications.

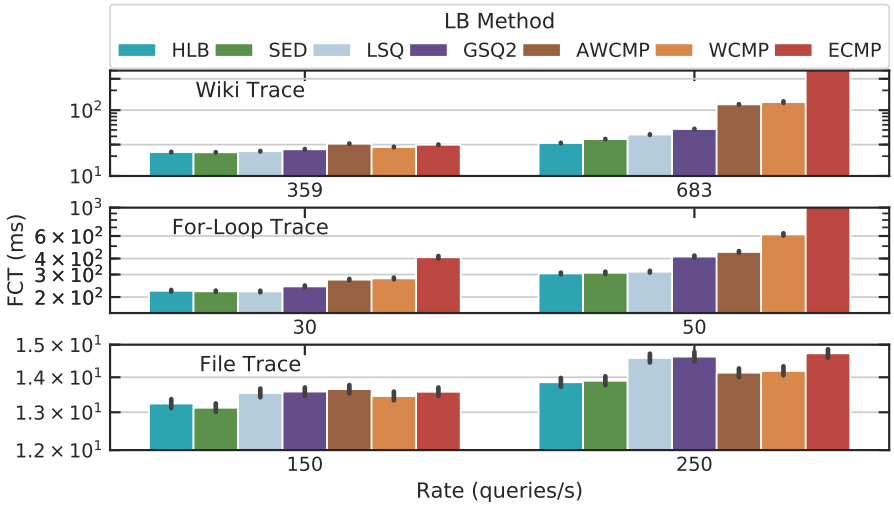
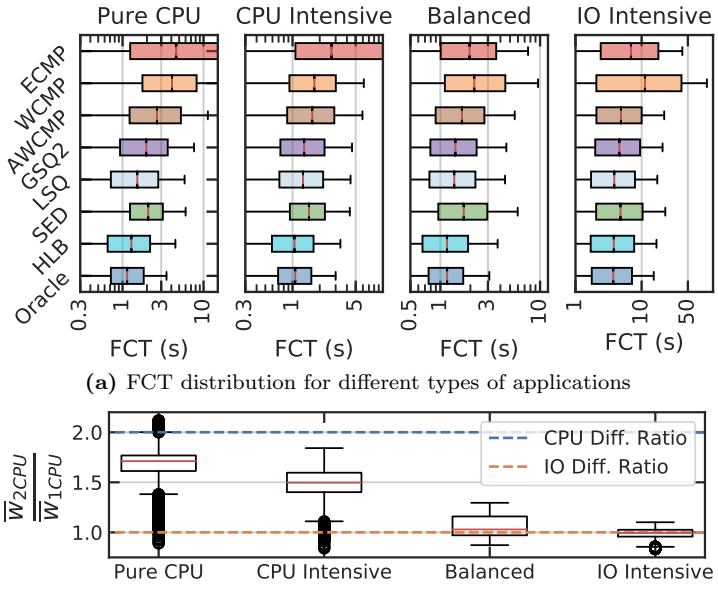


Figure 6.20: [Testbed] Comparison with different types of network applications.



(a) FCT distribution for different types of applications

(b) Server weight ratio between two server groups assigned by HLB.

Figure 6.21: [Simulator] Simulation results with 3-stage application queries when subjected to 90% expected resource utilization.

Application Type	Pure CPU	CPU Intensive	Balanced	IO Intensive
Avg. CPU Time (s)	1.	0.75	0.5	0.25
Avg. IO Time (s)	0.	0.25	0.5	0.75

Table 6.4: Four configurations with different application types.

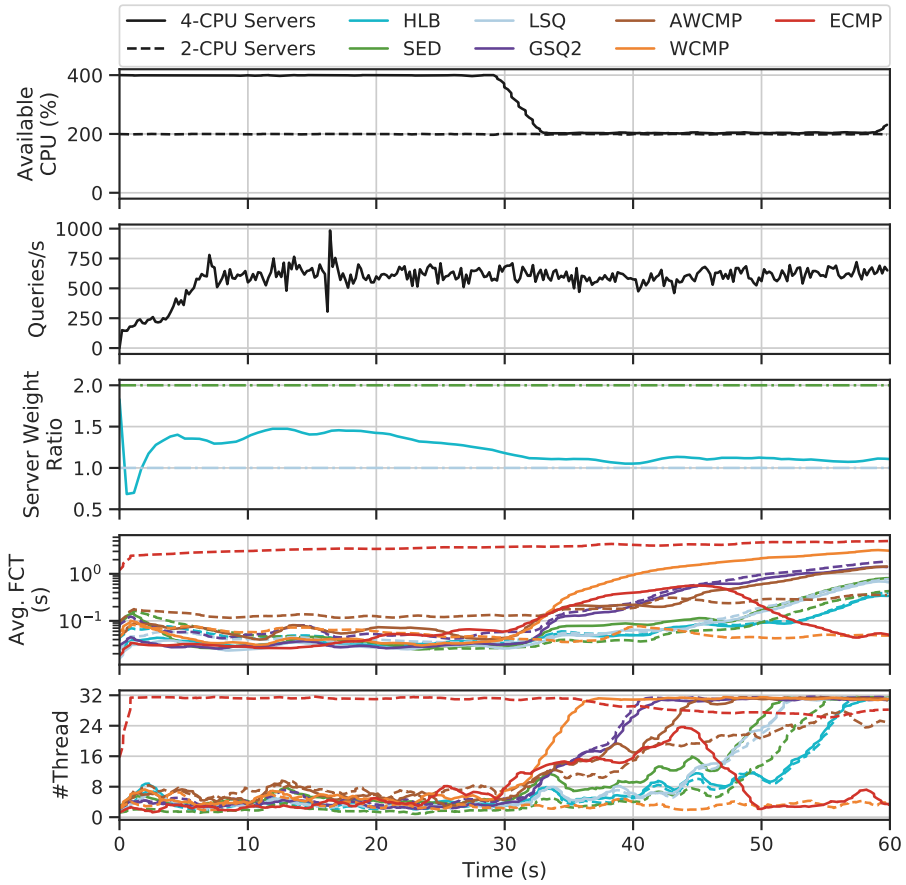
6.5.5 Response to Heterogeneous and Dynamic Environments

This section studies the adaptability of HLB when facing heterogeneous traffic and dynamic data center setups.

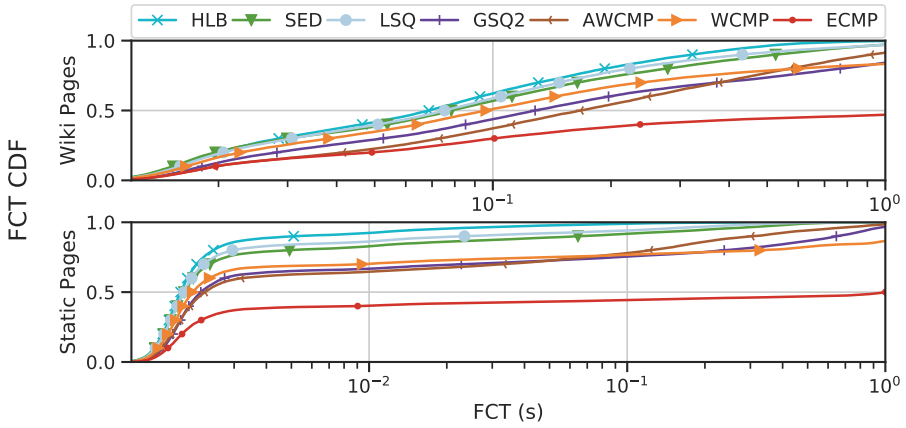
Adaption to Different Types of Input Traffic

In addition to Wikipedia traces, two types of Poisson traffic are injected. A PHP `for`-loop script that runs for a given number of iterations, simulates CPU-bound applications with $T(q) \sim$

$Exp(0.2)$. To simulate IO-bound applications, a farm of static files with different sizes[10] are created and queried. Moderate and high traffic rates are applied for the 3 types of applications in the testbed as in figure 6.4. As depicted in figure 6.20, LSQ achieves similar performance as that of SED and HLB for `for`-loop trace but does not perform better than ECMP for the file trace. AWCMP achieves lower FCT when subjected to CPU-bound traffic, especially when the traffic rate is high. SED and HLB have the best performance for all traces.



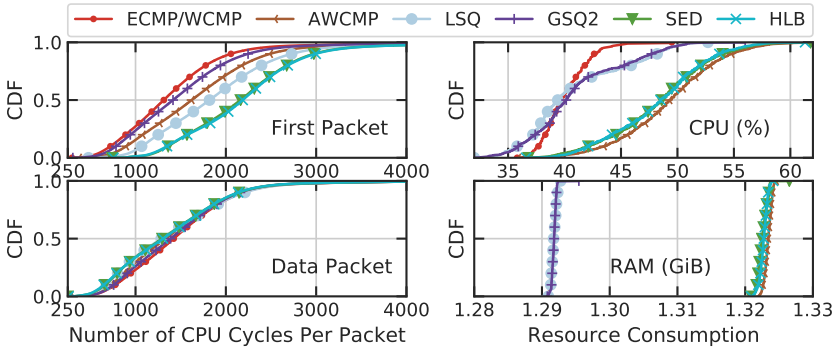
(a) Additional workloads are applied on servers with 4 CPUs after 30s.



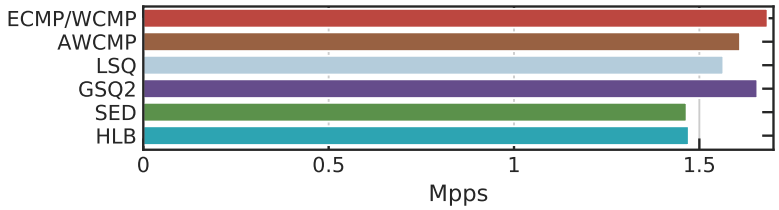
(b) FCT CDF comparisons for two types of requests.

Figure 6.22: [Testbed] HLB is able to adapt to changed environments without manual configurations or additional control messages.

[10]The sizes of files are 100KB, 200KB, 500KB, 750KB, 1MB, 2MB, and 5MB. 50 files are randomly generated for each size.



(a) Processing latency and resource consumption comparison.



(b) Throughput comparison.

Figure 6.23: [Testbed] Overhead analysis.

Though flow duration is affected by many factors including expected workloads, instant server occupancy, and different types of provisioned resources (*e.g.* CPU, IO, networking conditions), by collecting multiple samples (128 per server) of flow durations for the same VIP (and thus for the same application), we obtain a statistical representation of the flow duration distribution on each server. This allows to derive and compare the overall server processing speed for the given application. Using flow duration as an indicator of server load states saves us from profiling different applications (*e.g.* resource dependencies) and allows to generalize to different types of applications.

On a larger scale, simulations are conducted with 4 LBs and 128 servers using the `2x` configuration as in table 6.3. In this section, a 3-stage application whose queries follow CPU-IO-CPU processing stages is compared with a pure CPU application. Both CPU and IO processing time follow exponential distributions and the aggregated average FCT is 1s. The four different types of network applications are configured as in table 6.4. As depicted in figure 6.21, with different provisioned resource ratios for CPU (`2x`) and IO (`1x`) queues, HLB has better performance for all types of applications, with weights adaptive to the requirements of different types of applications.

Adaption to Processing Speed Changes

This section shows the ability of HLB to detect changes in server processing speeds, *e.g.,* when VMs are migrated to a new server. Using the `2x` testbed configuration with 2 LBs, additional CPU-bound workloads are applied on the 4-CPU server group starting from 30s. As depicted in figure 6.22, when subjected to heavy Wikipedia traffic, HLB adapts server weights over time and achieves better performance than other LB algorithms. It can infer that the processing speeds of the two groups of servers become similar to each other after 30s.

6.5.6 Overhead Analysis

To compare the additional processing latency of HLB, one 4-CPU LB, and a 176-CPU server cluster are deployed on 4 physical machines. The number of CPU cycles per packet and resource consumption are compared in figure 6.23a, using 10 runs of 2000 queries/s of Poisson traffic (more than 1150.76 average concurrent flows). The first packets are those that register new-coming flows in the flow table while the data packets are the subsequent packets that are matched in the flow table. As HLB calculates and compares the score of each server when assigning servers to flows,

it consumes on average 871 cycles (0.34μs on 2.6GHz CPU) more than does ECMP for each flow. Compared with ECMP, HLB incurs on average 8% additional CPU usage and 31MiB additional RAM usage. Assuming LBs see one `SYN` packet, one data packet and one `FIN` packet in each flow, the average packet throughput for each LB algorithm on a 2.6GHz CPU are compared as depicted in figure 6.23b. HLB achieves 87.38% throughput of ECMP.

6.6 Summary

Workload distribution fairness, as one of the main expectations for network load balancers in data center networks, requires load balancers to balance incoming traffics to its associated application servers so that its downstream resources can be efficiently utilized. However, as already discussed in chapter 1, the virtualization technologies including virtual machines (VMs) [60] and containers [63] have largely improved the elasticity of data center architectures and make automatic resource orchestration feasible [274–276]. The assumption that all application instances share the same configuration (*e.g.,* number of cores, memory capacities etc.) no longer holds, and replicates of application instances may even run in heterogenous hardwares [72]. Following the study in chapter 5, which aims to improve load-balancing fairness with ML algorithms, this chapter has proposed, and studied the performance of HLB – a load-aware Layer-4 load balancer using classical statistical learning methods with 2 selected networking features.

Implemented based on Aquarius, HLB can estimate both server occupancy and processing speed, which are identified in this chapter as two key factors in load-balancing performance. Using passively gathered networking observations extracted from the data plane, HLB can infer available server processing capacities in real-time, with no *a-priori* knowledge or manual configurations. HLB can be deployed with no modification on the network stack of the target data center networks, since it requires no additional management traffic or active signaling. Therefore, this makes HLB an interesting candidate for autonomous service management and orchestration in the cloud, which helps avoid error-prone manual configurations.

Extensive evaluations and sensitivity analysis have been conducted by way of both simulations and real-world testbed experiments. Evidences have been provided that HLB offers better load-balancing performance than state-of-the-art LB algorithms, not only in terms of load-balancing fairness, but also in terms of performance overhead (no additional communication overhead). HLB outperforms load-balancing algorithms that do not consider both server occupancy and processing speed when making load-balancing decisions in different scenarios. When compared with SED, which considers these 2 key factors using manual configurations, HLB can learn from passive observations autonomously. It is also able to adapt to dynamic data center environments and variant workloads and achieve better load-balancing performance, especially when servers are subject to additional co-located workloads, which make manual configurations fail to capture the actual system states. The limitation of HLB is that, when the server cluster is heavily loaded, it is not able to distribute workloads as effectively as SED, which is aware of the exact server processing speed differences.

The work and results from this chapter have been published in [196]. The source code and data of simulation results are available under an open-source license at: `https://github.com/ZhiyuanYaoJ/SimLB/tree/hlb`.

Chapter 7

Load Balancing with Reinforcement Learning

As discussed in chapter 5, network LBs rely on heuristic mechanisms [65, 135, 186, 236], and are subject to the low-latency and high-throughput constraints of the data plane. However, these heuristics are not adaptive to dynamic environments and require human interventions, and, as such, to misconfigurations, as shown in experiments in section 6.5.5. To improve load-balancing performance under such constraints, HLB has been proposed in chapter 6, and it is shown to be able to achieve similar performance to SED with no prior knowledge about the server configurations. As an "open-loop" control mechanism, HLB infers server load states only based on the measurements and does not evaluate the outcome of the load-balancing actions. It is less computationally expensive, yet it may take longer to stabilize when faced disturbances, as demonstrated in section 6.5.5. As the counterpart of "open-loop" control, "close-loop" control mechanisms consider both measurements as input signals and the output of the system as feedback signals, to update the control policy to the desired condition, with high resilience to disturbances. This chapter explores a "closed-loop" control mechanism – Reinforcement Learning (RL) – that aims at outperforming state-of-the-art load-balancing mechanisms, *e.g.,* SED.

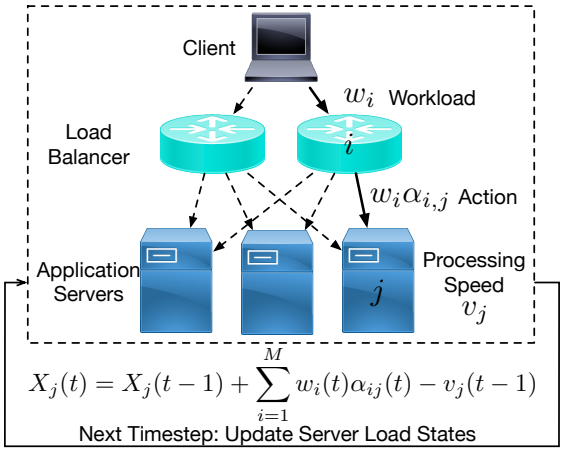


Figure 7.1: Network load balancing in the context of RL.

Related Work

Network Load Balancing Algorithms.

As introduced already in chapter 5, the goal of network LBs is to *fairly* distribute incoming requests across servers. As a reminder, the system transition protocol of the network load-balancing system is described in algorithm 4 and depicted in figure 7.1, in the context of RL.

The results in section 6.2 have shown that, existing load-balancing algorithms are sensitive to partial observations and inaccurate server weights.

Algorithm 4 LB System Transition Protocol

Initialise server load, $X_j(0) \leftarrow 0, \forall j \in [N]$
for each time step t **do**
    for each LB agent $i \in [M]$ **do**
        Choose action $\alpha_{ij}(t)$ for coming tasks $w_i(t)$
    for each server j **do**
        Update workload:
$X_j(t) = X_j(t-1) + \sum_{i=1}^{M} w_i(t)\alpha_{ij}(t) - v_j(t-1)$

Equal-Cost Multi-Path (ECMP) LBs [190, 193, 243] randomly assign servers to new requests, which makes them computationally efficient yet agnostic to server load state differences.

Weighted-Cost Multi-Path (WCMP) LBs [65, 221, 235, 241] assign weights to servers, proportional to their provisioned resources (*e.g.,* CPU power), however, the statically assigned weights may not correspond to the actual server processing capacity.

Active WCMP (AWCMP) is a variant of WCMP and it periodically probes server utilization information (CPU/memory/IO usage) [143, 186]. However, active probing can cause delayed observations and incur additional control messages, which degrades the performance of distributed networking systems.

Local Shortest Queue (LSQ) assigns new requests to the server with the minimal number of ongoing networking flows that are *locally* observed [144, 261]. It does not concern server processing capacity differences.

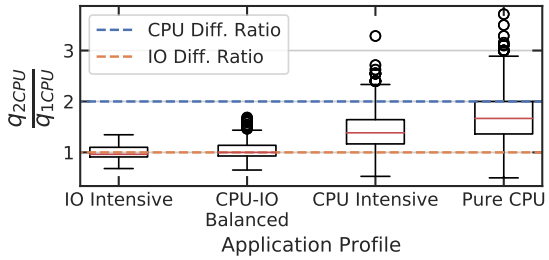
Shortest Expected Delay (SED) derives the "expected delay" as locally observed server queue length divided by statically configured server processing speed [236]. However, LSQ and SED are sensitive to partial observations and misconfigurations, as discussed in section 6.2.

This chapter formulates the network load-balancing problem as a multi-agent RL (MARL) problem and aims at proposing a new algorithm that outperforms the existing LBs mentioned above.

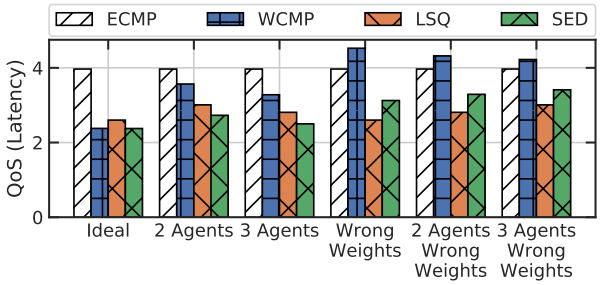
Multi-Agent RL.

MARL [277] is an important avenue for solving different types of games. For cooperative settings, a line of work based on joint-value factorisation have been proposed, involving VDN [278], COMA [279], MADDPG [280], and QMIX [281]. For these works, a global reward is assigned to players within the team, but individual policies are optimized to execute individual actions, known as the CTDE setting. However, deploying CTDE RL models in real-world distributed system incurs additional communication latency and management overhead for synchronising agents and aggregating trajectories. These additional management and communication overhead can cause substantial performance degradation – constrained throughput and increased latency – especially in data center networks. The communication overhead will be quantitatively studied further in this chapter. This chapter proposes an independent MARL framework that outperforms CTDE RL models with no additional control messages. To that end, this chapter applies the characteristics of Markov Potential Games (MPG).

Markov Potential Games. A potential game (PG) [282–285] has a special function called *potential function*, which specifies a property that any individual deviation of the action for one player will change the value of its own and the potential function equivalently. A desirable property of PG is that pure NE always exists and coincides with the maximum of potential function in norm-form setting. Self-play [286] is provably converged for PG. Markov games (MG) is an extension of normal-form game to a multi-step sequential setting. A combination of PG and MG yields the Markov potential games (MPG) [287, 288], where pure NE is also proved to exist. Some algorithms [287, 289, 290] lying in the intersection of game theory and reinforcement learning are proposed for MPG. In particular, independent nature policy gradient is proved to converge to Nash equilibrium (NE) for MPG [287]. By leveraging this characteristic of MPG, an independent learning approach can be more efficient due to the decomposition of the joint state and action spaces, which is leveraged in the proposed methods. Methods like MATRPO [291], IPPO [292] follow a fully decentralized setting, but for general cooperative games. MPG satisfies the assumptions of the value decomposition approach, with the well-specified potential function as the joint rewards.



(a) It is hard to accurately estimate the actual server processing speeds since it depends on both provisioned resources, and application profiles.



(b) The performance of existing network load-balancing algorithms degrades when observation becomes partial with multi-agents and weights are mis-configured.

Figure 7.2: Existing network load-balancing algorithms are sub-optimal in real-world setups.

Application Type	Pure CPU	CPU Intensive	Balanced	IO Intensive
Avg. CPU Time (s)	1.	0.75	0.5	0.25
Avg. IO Time (s)	0.	0.25	0.5	0.75

Table 7.1: Four configurations with different application types.

Motivation

RL has shown performance gains in distributed system and networking problems [25–27, 184], yet applying RL to the network load-balancing problem remains challenging.

Limited Observations

Unlike traditional workload distribution or task scheduling problems [26,27], network LBs have limited observations of the system. In particular, a network LB does not know task sizes, actual server load states, or server processing capacities. Being aware of only the number of tasks an LB has assigned to a server on receipt of new flows, servers with lower processing capacities can be overloaded – by *e.g.,* serving "elephant flows" assigned to the same server – and provide degraded quality of service (QoS).

Partial Observations

Second, to guarantee high service availability, multiple LBs are deployed within the servers in data center. Network flows are split among these LBs. This condition makes thus each LB has only a partial view of the state of the system, which is known as the *multi-agent* setup [293]. This has been investigated both in section 3.3.3 and in section 6.5.3.

Inaccurate Weights

Assigning weights to servers according to server processing speeds has been proposed as an alternative way of making informed load-balancing decisions [65,143,186,221,235,241]. As discussed in chapter 5, however, modern data centers are built, and contain heterogeneous hardware and elastic infrastructures [72], where server capacities vary. That condition makes it challenging to

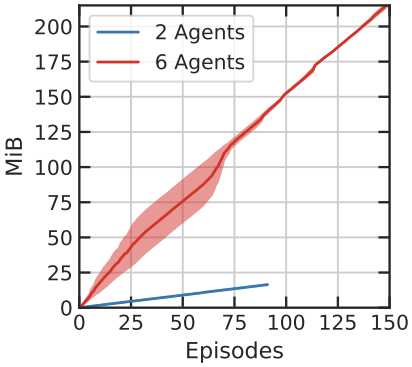


Figure 7.3: Communication overhead for CTDE grows linearly during training.

assign weights to servers according to their actual processing capacities. Not to mention that this process conventionally requires human intervention – which, again, can lead to error-prone configurations [65, 186].

In real-world systems, not only error-prone manual configuration, but also different application profiles can lead to inaccurate server weight assignments. This has been already evaluated extensively in section 6.5.5. As a reminder, 2 clusters of 4 servers were configured with the same IO processing speed but 2x different CPU processing speeds. Four different application profiles (*e.g.,* CPU-bound and IO-bound) are compared to derive the actual server processing capacity differences, as in table 7.1. As depicted in figure 7.2a, with different provisioned resource ratios for CPU (`2x`) and IO (`1x`) queues, to guarantee the optimal workload distribution fairness, and make each server have the minimized maximal remaining time to finish among all servers at all time, the weights to be assigned for servers with different provisioned capacities depend on different application profiles. For instance, servers with the same IO speed yet different CPU capacities have different actual processing speeds when applications have different resource requirements. Therefore, it is a sub-optimal solution for existing load-balancing algorithms to statically configure server weights based on computational resources. In addition, as analyzed in section 6.2, the QoS performance of each load-balancing algorithm degrades from the ideal setup (global observations and accurate server weight configurations, as depicted in figure 7.2b) when network traffic is split across multiple LBs or server weights are mis-configured[1], which prevails in the real-world cloud data center.

Performance Overheads

Last but not least, as already discussed in chapter 4, given the low-latency and high-throughput constraints that network LBs must operate with, the interactive training procedure of RL models and the centralized-training-decentralized-execution (CTDE) scheme [279] can incur additional communication and management overhead. The communication overhead of the CTDE RL scheme in data center networks has been discussed in section 3.1 already in two folds: throughput and latency.

1. *Thoughput:* Active signaling (*e.g.,* periodically probing, or sharing messages) is an intrinsic way to observe and measure system states so that informed decisions can be made to improve performance [293, 294]. Higher communication frequency gives more relevant and timely observations yet there is a trade-off between communication frequency and additionally consumed throughput. Management traffic among different networking devices can cascade and plunder the throughput for data transmission in high-tier links.

   As depicted in figure 7.3, CTDE RL scheme requires agents to communicate and share their trajectories, which include the observed states and actions. This leads to linearly increasing replay buffer size with the growth of number of episodes. The replay buffer size also grows with the number of agents which makes CTDE RL scheme not a scalable mechanism. Transmitting and synchronising replay buffer among agents incur additional communication

[1] The stochastic Markov model of the simulation is detailed in section 6.2

overhead in the networking system, reducing the throughput for data transmission channel – which can break full-bisection bandwidth (an important throughput related performance metric, as discussed in section 1.1) in data center networks [295] – thus decreasing the QoS.

2. *Latency:* When a single controller VM periodically transmit different amount of bytes via TCP sockets towards the agents, the latency overhead increases with the number of servers, which diminishes the QoS. While normal per-packet round trip time (RTT) between two directly connected network devices is $0.099ms \pm 0.014ms$ in the setup, with additional communication overhead, RTT can grow more than 10x (as depicted in figure 3.1 in chapter 3). This is not considered as low additional latency, especially not in high performance networking systems. In elastic and cloud computing context and real-world setups, load balancers can be deployed in different racks [58]. There can be multiple hops between two devices and one flow consists of tens of hundreds of packets, which can lead to cascaded high latency.

Statement of Purpose

This chapter studies the network load-balancing problem using a multi-agent game theoretical approach, by formulating it as a Markov potential game, and by specifying the proper reward function, namely variance-based fairness. This chapter proposes a distributed Multi-Agent RL (MARL) network load-balancing mechanism – Distr-LB , which can exploit asynchronous actions based only on local observations and inferences. Distr-LB considers dynamically changing queue lengths (*e.g.,* sub-ms in modern data center networks [229]), and autonomously adapts to actual server processing capacities, with no additional communications among LB agents or servers. Load balancing performance gains are evaluated based on both event-based simulations and real-world experiments.

Chapter Outline

The remainder of this chapter is organized as follows. Section 7.1 formulates the network load-balancing problem in data centers, and revisits key concepts and constraints in load-balancing problems. Section 7.2 proposes and discusses two different fairness indexes that will be further used as reward functions in this chapter. Section 7.3 introduces the characteristics of the Markov Potential Game (MPG), to which the network load-balancing problem is translated. Then, section 7.4 presents the proposed MARL framework for network load balancers, and section 7.5 describes the implementation details as well as experimental setups. Section 7.6 compares and contrasts different load-balancing algorithms, by way of both simulation and realistic experiments in physical testbeds. Section 5.5 finally concludes this chapter.

7.1 Problem Formalization

The load-balancing problem can be formulated into a discrete-time dynamic game, with strong distributed and concurrent settings, and where no centralized control mechanism exists among agents. M denotes the number of LB agents. $[M]$ denotes the set of LB agents $\{1, \ldots, M\}$. N denotes the number of servers, and $[N]$ denotes the set of servers $\{1, \ldots, N\}$. At each time step (or round) $t \in H$ in a horizon H of the game, each LB agent i receives a workload $w_i(t) \in W$, where W is the workload distribution. Then the LB agent assigns a server to the task using its load-balancing policy $\pi_i \in \Pi$, where Π is the load-balancing policy profile. At each time-step t, a LB agent i takes an action $a_i(t) = \{a_{ij}(t)\}_{j=1}^{N}$, according to which the tasks $w_i(t)$ are assigned with distribution $\alpha_i(t)$. $\alpha_{ij}(t)$ is the probability of assigning tasks to server j, $\sum_{j=1}^{N} \alpha_{ij}(t) = 1$. Therefore, at each time step, the workload assigned to server j by the i-th LB is $w_i(t)\alpha_{ij}(t)$. During each time interval, each server j is capable of processing a certain amount of workload v_j, based on the property of each server (*e.g.,* provisioned resources including CPU, memory, etc.). The server load state (remaining workload to process) can be expressed as $X_j(T) = \sum_{t=0}^{T} \max\{0, \sum_{i=1}^{M} w_i(t)\alpha_{ij}(t) - v_j\} = \max\{0, \sum_{t=0}^{T}\sum_{i=1}^{M} w_i(t)\alpha_{ij}(t) - v_j T\} = \sum_{i=1}^{M} X_{ij}(T)$[2]. l_j denotes the time for a server j to process all remaining workloads, which is also the potential queuing time for new-coming tasks, then, $l_j(t) = \frac{X_j(t-1)+\sum_{i=1}^{M} w_i(t)\alpha_{ij}(t)}{v_j} = \frac{\sum_{i=1}^{M} X_{ij}(t-1)+w_i(t)\alpha_{ij}(t)}{v_j} = \sum_{i=1}^{M} l_{ij}(t)$. The transition from

[2] $X_{ij}(T) = \sum_{t=0}^{T} \max\{0, w_i(t)\alpha_{ij}(t) - \frac{v_j}{M}\}$

time step t to time step $t+1$ is then given in algorithm 4. The reward is $r_i(t) = R(\boldsymbol{l}(t), a_i(t), \delta_i(t))$, where R is the reward function. Finally, $\boldsymbol{l}(t) = \sum_{j=1}^{N} l_j(t) = \sum_{i=1}^{M} l_i(t)$ denotes the estimated remaining time to process on each server, and $\delta_i(t)$ is a random variable that makes the process stochastic.

Definition 7.1. *(Makespan) In the selfish load-balancing problem, the makespan is defined as:*

$$MS = \max_j(l_j), l_j = \sum_i l_{ij} \tag{7.1}$$

The network load-balancing problem can be expressed as a multi-commodity flow problem (NP-hard [296]). This makes it hard to solve using algorithmics within the micro-second timing constraints. The problem can be formulated as a constrained optimization problem for minimizing the makespan over a horizon $t \in [H]$:

$$minimize \sum_{t=h}^{H} \max_j l_j(t) \tag{7.2}$$

$$s.t. \quad l_j(t) = \frac{\sum_{i=1}^{M}(X_{ij}(t-1) + w_i(t)\alpha_{ij}(t))}{v_j}, \quad \sum_{i=1}^{M} w_i(t) \leq \sum_{j=1}^{N} v_j, \quad w_i, v_j \in (0, +\infty) \tag{7.3}$$

$$X_{ij}(T) = \sum_{t=0}^{T} \max\{0, w_i(t)\alpha_{ij}(t) - \frac{v_j}{M}\}, \quad \sum_{j=1}^{N} \alpha_{ij}(t) = 1, \quad \alpha_{ij} \in [0, 1] \tag{7.4}$$

The arrival of network requests is assumed in realistic network load-balancing system [1, 65] to be unpredictable, in both its arriving rate and the expected workload, which introduces non-negligible stochasticity into the problem. Moreover, due to the existence of noisy measurements and partial observations, the estimation of makespan may not indicate the actual server load states or available processing capacities. For instance, collisions of "elephant" flows, or bursts of "mouse" flows, can happen [1, 27]. For simplicity and clarity, this chapter uses the term "task" to designate generically a network flow handled by an application server. Counting the number of ongoing tasks on the server does not indicate server processing capacity. To solve this issue, the *fairness* of task completion time is proposed as a replacement for the original objective makespan. Specifically, makespan is estimated on a per-server level based on the partial observation of each LB, *i.e.,* the number of ongoing tasks observed from each LB on the (locally perceived) most heavily loaded server is derived as the makespan. The estimation of fairness, however, can be decomposed to the LB-level, *i.e.,* overloaded servers yield longer task completion time, which can be reflected in the fairness estimations. This allows evaluating the individual LB performance. This is practical in load-balancing systems due to the partial observability of LBs.

7.2 Distribution Fairness

This chapter introduces two types of load-balancing distribution fairness: (1) variance-based fairness (VBF) and (2) product-based fairness (PBF). This section will show that optimization over either VBF or PBF will be, sufficient but not necessary, for minimizing the *makespan.*

Definition 7.2. *(Variance-based Fairness) For a vector of time to finish all remaining tasks* $\boldsymbol{l} = [l_1, \ldots, l_N]$ *on each server* $j \in [N]$*, let* $\bar{\boldsymbol{l}}(t) = \frac{1}{N}\sum_{j=1}^{N}\sum_{i=1}^{M} l_{ij}(t)$*, the variance-based fairness for workload distribution is just the negative sample variance of the task time, which is defined as:*

$$F(\boldsymbol{l}) = -\frac{1}{N}\sum_{j=1}^{N}\left(l_j(t) - \bar{\boldsymbol{l}}(t)\right)^2 = -\frac{1}{N}\sum_{j=1}^{N} l_j^2(t) + \bar{\boldsymbol{l}}^2(t). \tag{7.5}$$

VBF defined per LB is: $F_i(\boldsymbol{l}_i) = -\frac{1}{N}\sum_{j=1}^{N} l_{ij}^2(t) + \bar{\boldsymbol{l}}_i^2(t)$*, where* $\bar{\boldsymbol{l}}_i(t) = \frac{1}{N}\sum_{j=1}^{N} l_{ij}(t)$*.*

Lemma 7.1. *The VBF for load-balancing system satisfies the following property:*

$$F_i^{\pi_i,-\pi_i}(\boldsymbol{l}_i) - F_i^{\tilde{\pi}_i,-\pi_i}(\tilde{\boldsymbol{l}}_i) = F^{\pi_i,-\pi_i}(\boldsymbol{l}) - F^{\tilde{\pi}_i,-\pi_i}(\tilde{\boldsymbol{l}}) \tag{7.6}$$

Proof. The definition of the variance-based fairness (as definition 7.2) gives the following for $\forall i \in [M], j \in [N]$,

$$F^{\pi_i,-\pi_i}(\boldsymbol{l}) = -\frac{1}{N}\sum_{j=1}^{N}(l_j - \bar{\boldsymbol{l}})^2 \tag{7.7}$$

$$F_i^{\pi_i,-\pi_i}(\boldsymbol{l}_i) = -\frac{1}{N}\sum_{j=1}^{N}(l_{ij} - \overline{l_i})^2 \quad (\overline{l_i} = \frac{1}{N}\sum_{j=1}^{N} l_{ij}) \tag{7.8}$$

By indexing the agent i as the one to change its strategy (and slightly abusing notation), $l_j = l_{ij} + l_{-ij}$, where $l_{-ij} = \sum_{k\neq i} l_{kj}$.

$$F^{\pi_i,-\pi_i}(\boldsymbol{l}) = -\frac{1}{N}\sum_{j=1}^{N}(l_{ij} + l_{-ij} - \overline{(l_i + l_{-i})})^2 \quad (\text{where } \overline{(l_i + l_{-i})} = \frac{1}{N}\sum_{j}(l_{ij} + l_{-ij})) \tag{7.9}$$

$$= -\frac{1}{N}\sum_{j=1}^{N}[l_{ij} + l_{-ij} - (\bar{l}_i + \bar{l}_{-i})]^2 \tag{7.10}$$

$$= -\frac{1}{N}\sum_{j=1}^{N}[(l_{ij} - \bar{l}_i)^2 + (l_{-ij} - \bar{l}_{-i})^2 - 2(l_{ij} - \bar{l}_i)(l_{-ij} - \bar{l}_{-i})] \tag{7.11}$$

$$= -\frac{1}{N}\sum_{j=1}^{N}(l_{ij} - \overline{l_i})^2 - \frac{1}{N}\sum_{j=1}^{N}[(l_{-ij} - \bar{l}_{-i})^2 - \frac{2}{N}\sum_{j=1}^{N}(l_{ij} - \bar{l}_i)(l_{-ij} - \bar{l}_{-i})] \tag{7.12}$$

$$= F_i^{\pi_i,-\pi_i}(\boldsymbol{l}_i) - \frac{1}{N}\sum_{j=1}^{N}(l_{-ij} - \bar{l}_{-i})^2 \quad (\sum_{j=1}^{N}(l_{ij} - \bar{l}_i) = 0) \tag{7.13}$$

where the second term is a common term not depend on the changing policy π_i. Therefore, the second term will be cancelled in $F^{\pi_i,-\pi_i}(\boldsymbol{l}) - F^{\tilde{\pi}_i,-\pi_i}(\tilde{\boldsymbol{l}}) = F_i^{\pi_i,-\pi_i}(\boldsymbol{l}_i) - F_i^{\tilde{\pi}_i,-\pi_i}(\tilde{\boldsymbol{l}}_i)$, thus finishes the proof. □

This property makes VBF a good choice for the reward function in load-balancing tasks, since it satisfies the requirement of being a potential function in the MPG, which will be discussed further in section 7.3.

Proposition 7.1. *Maximizing the VBF is sufficient for minimizing the makespan, subjective to the load-balancing problem constraints (equation (7.3) and (7.4)):*

$$\max F(\boldsymbol{l}) \Rightarrow \min \max_j(l_j) \tag{7.14}$$

this also holds for per-LB VBF as $\max F_i(\boldsymbol{l}_i) \Rightarrow \min \max_j(\boldsymbol{l}_i)$.

Proof. Given the stability constraint in equation (7.3) $\sum_{i=1}^{M} w_i(t) \leq \sum_{j=1}^{N} v_j$, denote the total amount of workload in the system $C = \sum_{j=1}^{N} l_j$, and $l_k = \max_{j\in[N]} l_j$. The constraint in equation (7.4) gives $C \geq 0$, $l_j(t) \geq 0$.

$$\max F(\boldsymbol{l}) \Leftrightarrow \min -F(\boldsymbol{l}) \tag{7.15}$$

$$-F(\boldsymbol{l}) = \frac{1}{N}\sum_{j=1}^{N}((l_j) - \bar{\boldsymbol{l}})^2 \tag{7.16}$$

$$= \frac{1}{N}\sum_{j=1}^{N}(l_j - \frac{C}{N})^2 \tag{7.17}$$

$$= \frac{1}{N}\sum_{j=1}^{N} l_j^2 - \frac{2C}{N^2}\sum_{j=1}^{N} l_j + \frac{C^2}{N^2} \tag{7.18}$$

$$= \frac{1}{N}\sum_{j=1}^{N} l_j^2 - \frac{C^2}{N^2} \tag{7.19}$$

$$\leq [(\max_j l_j)^2 - \frac{C^2}{N^2}] \quad (\text{by means inequality}) \tag{7.20}$$

with the equivalence achieved when $l_j = l_k, \forall j \neq k, j \in [N]$ holds. Therefore,

$$\max F(\boldsymbol{l}) \Rightarrow \min(l_k)^2 - \frac{C^2}{N^2} \tag{7.21}$$

$$\Leftrightarrow \min l_k \tag{7.22}$$

$$\Leftrightarrow \min \max_{j \in [n]} l_j \tag{7.23}$$

and the condition is sufficient but not necessary because $\min(l_k)^2 - \frac{C^2}{N^2}$ is essentially minimizing the upper bound of $-F(\boldsymbol{l})$. □

Definition 7.3. *(Product-based Fairness [28]) For a vector of time to finish all remaining tasks $\boldsymbol{l} = [l_1, \ldots, l_N]$ on each server $j \in [N]$, the product-based fairness for workload distribution is defined as: $F(\boldsymbol{l}) = F([l_1, \ldots, l_N]) = \prod_{j \in [N]} \frac{l_j}{\max(\boldsymbol{l})}$. The PBF defined per LB is: $F_i(\boldsymbol{l}_i) = F([l_{i1}, \ldots, l_{iN}]) = \prod_{j \in [N]} \frac{l_{ij}}{\max(\boldsymbol{l}_i)}$. Similarly, the product-based fairness for each LB agent (e.g., the i-th) $\boldsymbol{l}_i = [l_{i1}, \ldots, l_{iN}]$ can be defined as:*

$$F_i(\boldsymbol{l}_i) = F([l_{i1}, \ldots, l_{iN}]) = \prod_{j \in [N]} \frac{l_{ij}}{\max(\boldsymbol{l}_i)}. \tag{7.24}$$

Proposition 7.2. *Maximizing the product-based fairness is sufficient for minimizing the makespan, subjective to the load-balancing problem constraints (equation (7.3) and (7.4)):*

$$\max F(\boldsymbol{l}) \Rightarrow \min \max(\boldsymbol{l}) \tag{7.25}$$

Proof. For a vector of workloads $\boldsymbol{l} = [l_1, \ldots, l_N]$ on each server $j \in [N]$, by the definition of fairness,

$$\max F(\boldsymbol{l}) = \max \frac{\prod_{j \in [N]} l_j}{\max_{k' \in [N]} l_{k'}} \tag{7.26}$$

WLOG, let $l_k = \max_{k' \in [N]} l_{k'}$, then,

$$\max F(\boldsymbol{l}) = \max \prod_{j \in [N], j \neq k} l_j \tag{7.27}$$

Similar to the proof of proposition 7.1, given the stability constraint in equation (7.3) $\sum_{i=1}^{M} w_i(t) \leq \sum_{j=1}^{N} v_j$, denote the total amount of workload in the system $C = \sum_{j=1}^{N} l_j$. The constraint in equation (7.4) gives $C \geq 0$, $l_j(t) \geq 0$. By means inequality,

$$\left(\prod_{j \in [N], j \neq k} l_j \right)^{\frac{1}{N-1}} \leq \frac{\sum_{j \in [N], j \neq k} l_j}{N-1} = \frac{C - l_k}{N-1}. \tag{7.28}$$

with the equivalence achieved when $l_i = l_j, \forall i, j \neq k, i, j \in [N]$ holds. Therefore,

$$\max F(\boldsymbol{l}) \Rightarrow \max \frac{C - l_k}{N-1} \tag{7.29}$$

$$\Leftrightarrow \min l_k \tag{7.30}$$

$$\Leftrightarrow \min \max_{j \in [N]} l_j \tag{7.31}$$

The inverse may not hold, since $\max \frac{C - l_k}{N-1}$ does not indicate $\max F(\boldsymbol{l})$. Thus, maximizing the linear product-based fairness is sufficient but not necessary for minimizing the makespan. This finishes the proof. □

Propositions 7.1 and 7.2 show that the two types of fairness can serve as an effective alternative objective for optimizing the makespan, which will be leveraged in the proposed MARL method as valid reward functions.

7.3 Game Theory Framework

A Markov decision game is defined as $\mathcal{MG}(H, M, \mathcal{S}, \mathcal{A}_{\times M}, \mathbb{P}, \Pi_{\times M}, r_{\times M})$, $\Pi = \{\Pi_i\}, i \in [M]$, where H is the horizon of the game, M is the number of player in the game, $\mathcal{S}$ is the state space, $\mathcal{A}_{\times M}$ is the joint action space of all players, $\mathcal{A}_i$ is the action space of player i, $\mathbb{P} = \{\mathbb{P}_h\}, h \in [H]$ is a collection of transition probability matrices $\mathbb{P}_h : \mathcal{S} \times \mathcal{A}_{\times M} \times \mathcal{S} \to [0, 1]$, $r_{\times M} = \{r_i | i \in [M]\}, r_i : \mathcal{S} \times \mathcal{A}_{\times M} \to \mathbb{R}$ is the reward function for i-th player given the joint actions. The stochastic policy space for the i-th player in $\mathcal{MG}$ is defined as $\Pi_i : \mathcal{S} \times \mathcal{A}_i \to [0, 1]$.

For the Markov decision game $\mathcal{MG}$, the state value function $V^{\boldsymbol{\pi}}_{i,h} : \mathcal{S} \to \mathbb{R}$ and state-action value function $Q^{\boldsymbol{\pi}}_{i,h} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ for the i-th player at step h under policy $\boldsymbol{\pi} \in \Pi_{\times M}$ is defined as:

$$V^{\boldsymbol{\pi}}_{i,h}(s) := \mathbb{E}_{\boldsymbol{\pi},\mathbb{P}}\left[\sum_{h'=h}^{H} r_{i,h'}(s_{h'}, \boldsymbol{a}_{h'})\middle| s_h = s\right], Q^{\boldsymbol{\pi}}_{i,h}(s, \boldsymbol{a}) := \mathbb{E}_{\boldsymbol{\pi},\mathbb{P}}\left[\sum_{h'=h}^{H} r_{i,h'}(s_{h'}, \boldsymbol{a}_{h'})\middle| s_h = s, a_h = \boldsymbol{a}\right]. \tag{7.32}$$

Definition 7.4. *(ε-approximate Nash equilibrium) Given a Markov decision game:*

$$\mathcal{MG}(H, M, \mathcal{S}, \mathcal{A}_{\times M}, \mathbb{P}, \Pi_{\times M}, r_{\times M}),$$

let π_{-i} be the policies of the players except for the i-th player, the policies (π^*_i, π^*_{-i}) is an ε-Nash equilibrium if $\forall i \in [M], \exists \varepsilon > 0$,

$$V_i^{\pi^*_i, \pi^*_{-i}}(s) \geq V_i^{\pi_i, \pi^*_{-i}}(s) - \varepsilon, \forall \pi_i \in \Pi_i. \tag{7.33}$$

If $\varepsilon = 0$, it is an exact Nash equilibrium.

Definition 7.5. *(Markov Potential Game) A Markov decision game:*

$$\mathcal{MG}(H, M, \mathcal{S}, \mathcal{A}_{\times M}, \mathbb{P}, \Pi_{\times M}, r_{\times M})$$

is a Markov potential game (MPG) if $\forall i \in [M], \pi_i, \tilde{\pi}_i \in \Pi_i, \pi_{-i} \in \Pi_{-i}, s \in \mathcal{S}$,

$$V_i^{\pi_i, \pi_{-i}}(s) - V_i^{\tilde{\pi}_i, \pi_{-i}}(s) = \phi^{\pi_i, \pi_{-i}}(s) - \phi^{\tilde{\pi}_i, \pi_{-i}}(s), \tag{7.34}$$

where $\phi(\cdot)$ is the potential function, independent of the player index.

Lemma 7.2. *Pure NE (PNE) always exists for PG, local maximizers of potential function are PNE. PNE also exists for MPG. [282]*

Theorem 7.6. *Multi-agent load balancing is an MPG with the VBF $F_i(\boldsymbol{l}_i)$ as the reward r_i for each LB agent $i \in [M]$, then suppose for $\forall s \in \mathcal{S}$ at step $h \in [H]$, the potential function is time-cumulative total fairness: $\phi^{\pi_i, -\pi_i}(s) = \sum_{t=h}^{H} F^{\pi_i, -\pi_i}(\boldsymbol{l}(t))$.*

The proof of the theorem is based on lemma 7.1:

Proof.

$$V_i^{\pi_i, \pi_{-i}}(s) - V_i^{\tilde{\pi}_i, \pi_{-i}}(s) = \mathbb{E}_{\pi_i, \pi_{-i}}\left[\sum_{t=h}^{H} r_{i,t}(s_t, \boldsymbol{a}_t)\middle| s_h = s\right] - \mathbb{E}_{\tilde{\pi}_i, \pi_{-i}}\left[\sum_{t=h}^{H} r_{i,t}(s_t, \tilde{a}_{i,t}, a_{-i,t})\middle| s_h = s\right] \tag{7.35}$$

$$= \mathbb{E}_{\pi_i, \pi_{-i}}\left[\sum_{t=h}^{H} F_i(\boldsymbol{l}_i(t))\right] - \mathbb{E}_{\tilde{\pi}_i, \pi_{-i}}\left[\sum_{t=h}^{H} F_i(\tilde{\boldsymbol{l}}_i(t))\right] \tag{7.36}$$

$$= \sum_{t=h}^{H}\left(F^{\pi_i, -\pi_i}(\boldsymbol{l}) - F^{\tilde{\pi}_i, -\pi_i}(\tilde{\boldsymbol{l}})\right) \quad \text{(lemma 7.1)} \tag{7.37}$$

$$= \phi^{\pi_i, -\pi_i}(s) - \phi^{\tilde{\pi}_i, -\pi_i}(s) \tag{7.38}$$

Notice that s is the ground truth state of the environment, therefore involving the expected time $\boldsymbol{l}$ to finish remaining tasks. □

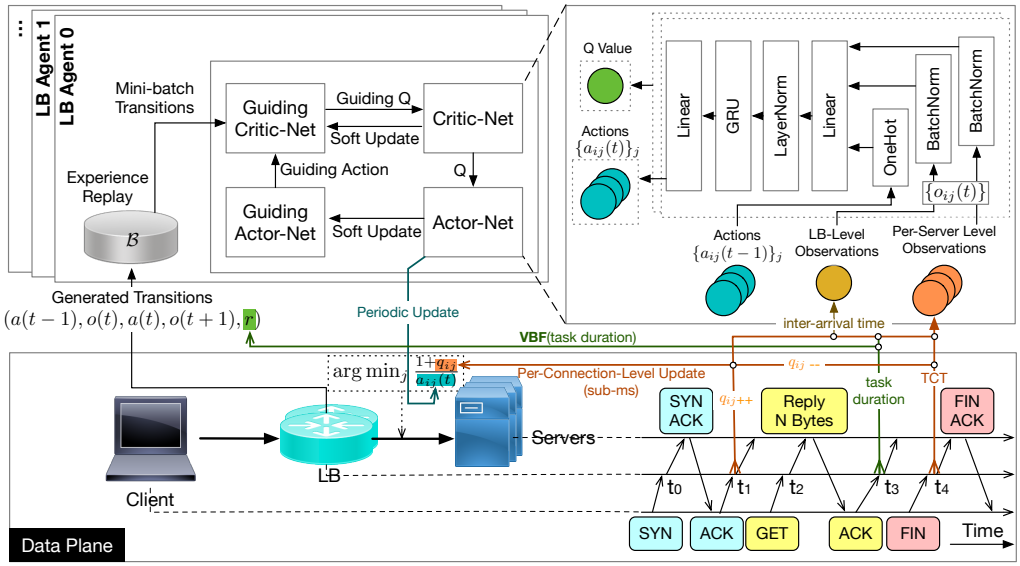


Figure 7.4: Overview of the proposed distributed MARL framework for network LB.

This theorem is essential for establishing the proposed Distr-LB, since it proves that the multi-agent load-balancing problem can be formulated as an MPG with the time-cumulative VBF as its potential function. Also, the choice of per-LB VBF as the reward function for each individual agent is critical for making it an MPG. Whereas, PBF does not provide such property. Lemma 7.2 shows that the maximizer of potential function is the NE of MPG, and from proposition 7.1 it is known that maximizing the VBF gives the sufficient condition for minimizing the makespan. Therefore, an effective independent optimization with respect to the individual reward function specified in the above theorem will lead the minimizer of makespan for load-balancing tasks. The effective independent optimization, in this context, means that the NE of MPG is achieved.

Lemma 7.3. *NE for MPG is ε-approximate NE for ε-approximate MPG. [297]*

Proof. *Given the NE (π_i^*, π_{-i}^*) for an MPG,*

$$V_i^{\pi_i^*,\pi_{-i}^*}(s) - V_i^{\tilde{\pi}_i,\pi_{-i}^*}(s) = \phi^{\pi_i^*,\pi_{-i}^*}(s) - \phi^{\tilde{\pi}_i,\pi_{-i}^*}(s) \geq 0 \tag{7.39}$$

the policies can be ε-approximate NE for another game with a different value function $\widehat{V}$ but the same potential function,

$$\widehat{V}_i^{\pi_i^*,\pi_{-i}^*}(s) - \widehat{V}_i^{\tilde{\pi}_i,\pi_{-i}^*}(s) \geq \varepsilon, \forall i \in [N], \tilde{\pi}_i \in \Pi_i, s \in \mathcal{S} \tag{7.40}$$

thus,

$$\left| \left(\widehat{V}_i^{\pi_i^*,\pi_{-i}^*}(s) - \widehat{V}_i^{\tilde{\pi}_i,\pi_{-i}^*}(s) \right) - \left(\phi^{\pi^*,\pi_{-i}^*}(s) - \phi^{\tilde{\pi},\pi_{-i}^*}(s) \right) \right| \leq \varepsilon \tag{7.41}$$

which satisfies the definition of ε-approximate MPG. □

7.4 Distributed LB Method

With the above analysis, the load-balancing problem can be formulated as an episodic version of a multi-player, partially observable, Markov game, denotes $\mathcal{POMG}(H, M, \mathcal{S}, \mathcal{O}_{\times M}, \mathbb{O}_{\times M}, r_{\times M})$. M is the number of LB agent in the game. $\mathcal{O}_{\times M}$ contains the observation space O_i for each player. $\mathbb{O} = \{\mathbb{O}_h\}, h \in [H]$ is a collection of observation emission matrices, $\mathbb{O}_{i,h} : \mathcal{S} \times \mathcal{O}_i \to [0, 1]$. $r_{\times M} = \{r_i | i \in [M]\}, r_i : \mathcal{O}_i \times \mathcal{A}_{\times M} \to \mathbb{R}$ is the reward function for i-th LB agent given the joint actions. The stochastic policy space for the i-th agent in $\mathcal{POMG}$ is defined as $\Pi_i : \mathcal{O}_i \times \mathcal{A}_i \to [0, 1]$. As discussed in section 7, the partial observability comes from the fundamental configuration of

Algorithm 5 Distributed LB for MPG

1: **Initialise:**
2: LB policy π_{θ_i} and critic Q_{ϕ_i} networks, replay buffer $\mathcal{B}_i, \forall i \in [M]$;
3: server processing speed function $v_j, \forall j \in [N]$;
4: initial observed instant queue length on server j by the i-th LB: $q_{ij} = 0, \forall i \in [M], j \in [N]$.
5: **while** not converge **do**
6: Reset server load state $X_j(1) \leftarrow 0, \forall j \in [N]$
7: Each LB agent i ($i \in [M]$) receives individual observation $o_i(1)$
8: **for** $t = 1, \ldots, H$ **do**
9: Initialise distributed workload $m_{ij}, w_i(t) \leftarrow 0, i \in [M], j \in [N]$
10: Get actions $a_i(t) \leftarrow \{a_{ij}(t)\}_{j=1}^{N} = \pi_{\theta_i}(o_i(t)), i \in [M]$
11: **for** task $\tilde{w}$ arrived at LB i between timestep $[t, t+1)$ **do**
12: LB i assigns $\tilde{w}$ to server $j = \arg\min_{k \in [N]} \frac{q_{ik}(t)+1}{a_{ik}(t)}$
13: $m_{ij} \leftarrow m_{ij} + \tilde{w}$, $w_i(t) \leftarrow w_i(t) + \tilde{w}$
14: $\alpha_{ij}(t) \leftarrow \frac{m_{ij}}{w_i(t)}$
15: **for** each server j **do**
16: Update workload: $X_{ij}(t+1) \leftarrow \max\{X_{ij}(t) + w_i(t)\alpha_{ij}(t) - \frac{v_j}{M}, 0\}$
17: $X_j(t+1) \leftarrow \sum_{i=1}^{M} X_{ij}(t)$
18: Each agent receives individual reward $r_i(t)$
19: Each agent i collects observation $o_i(t+1), i \in [M]$
20: Update replay buffer: $\mathcal{B}_i = \mathcal{B}_i \bigcup (a_i(t-1), o_i(t), a_i(t), r_i(t), o_i(t+1)), i \in [M]$
21: Update critics with gradients: $\nabla_{\phi_i} \mathbb{E}_{(o_i, a_i, r_i, o_i') \sim \mathcal{B}_i} \left[\left(Q_{\phi_i}(o_i, a_i) - r_i - \gamma V_{\tilde{\phi}_i}(o_i') \right)^2 \right]$
22:
23: where $V_{\tilde{\phi}_i}(o_i') = \mathbb{E}_{(o_i', a_i') \sim \mathcal{B}_i}[Q_{\tilde{\phi}_i}(o_i', a_i') - \alpha \log \pi_{\theta_i}(a_i'|o_i')], i \in [M]$
24: Update policies with gradients: $-\nabla_{\theta_i} \mathbb{E}_{o_i \sim \mathcal{B}_i}[\mathbb{E}_{a \sim \pi_{\theta_i}}[\alpha \log \pi_{\theta_i}(a_i|o_i) - Q_{\phi_i}(o_i, a_i)]], i \in [M]$
return final models of learning agents

network LBs in data center networks, which allows LBs to observe only a part of network traffic and does not give LBs information about the tasks (*e.g.,* expected workload) distributed from each LB. The reward functions in the experiments are variants of distribution fairness introduced in section 7.2. The potential functions can be defined accordingly based on the VBF and the PBF. The overview of the proposed distributed MARL framework is shown in figure 7.4.

In MPG, an independent policy gradient allows finding the maximum of the potential function, which is the PNE for the game. Leveraging the policy optimization in a decomposed manner, RL is distributed on each LB agent for policy learning. However, due to the partial observability of the system and the challenge of directly estimating the makespan (equation (7.1)), each agent cannot have direct access to the global potential function. To address this problem, the aforementioned fairness (section 7.2) can be deployed as the reward function for each agent, which makes the value function as a valid alternative for the potential function as an objective. This also transforms the joint objective (makespan or potential) to individual objectives (per LB fairness) for each agent. Proposition 7.1 and 7.2 verify that optimizing towards these fairness indices is sufficient for minimizing the makespan.

Algorithm 5 shows the proposed distributed LB for the load-balancing problem, which is a partially observable MPG. The distributed policy optimization is based on Soft Actor-Critic (SAC) [298] algorithm, which is a type of maximum-entropy RL method. It optimizes the objective $\mathbb{E}[\sum_t \gamma^t r_t + \alpha \mathcal{H}(\pi_\theta)]$, whereas $\mathcal{H}(\cdot)$ is the entropy of the policy π_θ. Specifically, the critic Q network is updated with gradient $\nabla_\phi \mathbb{E}_{o,a} \left[\left(Q_\phi(o,a) - r(o,a) - \gamma \mathbb{E}_{o'}[V_{\tilde{\phi}}(o')] \right)^2 \right]$, where $V_{\tilde{\phi}}(o') = \mathbb{E}_{a'}[Q_{\tilde{\phi}}(o',a') - \alpha \log \pi_\theta(a'|o')]$ and $Q_{\tilde{\phi}}$ is the target Q network; the actor policy π_θ is updated with the gradient $\nabla_\theta \mathbb{E}_o[\mathbb{E}_{a \sim \pi_\theta}[\alpha \log \pi_\theta(a|o) - Q_\phi(o,a)]]$. Other key elements of RL methods involve the observation, action and reward function, which are detailed as following.

Observation. Each LB agent partially observes the network flows that traverse through itself, including per-server and LB-level measurements. For each LB, per-server observations consist of the number of ongoing tasks, sampled task duration, and task completion times (TCT). Specifically, in algorithm 5 line 12-14, w_i is the coming workload on servers assigned by i-th LB, and it is not observable for any LB. $q_{ik} + 1$ is the locally observed number of tasks on k-th server, that are placed by i-th LB. The "+1" is for taking into account the newly arrived task. Observations of task duration and TCT samples, along with LB-level measurements which sample the task inter-arrival time as an indication of overall system load state, are reduced to 5 scalars – *i.e.,* average,

Related Work	Testbed Scale	Note
6LB [135]	2 LB + 28 servers (2-CPU each)	This chapter uses the same network trace as input traffic.
Ananta [64]	14 LBs for 12 VIPs	The exact number of servers per VIP and the in-production traffic is not documented in the paper.
Beamer [192]	2 LB + 8 servers (small) 4 LB + 10 servers (large)	Large scale experiments are conducted with 700 active HTTP requests max.
Duet [58]	3 software LB + 3 hardware LB + 34 servers	Synthetic traffic is applied so that the server cluster behind VIP processes 60k (identical) packets per second.
SilkRoad [190]	1 hardware LB or 3 software LB per VIP	Real-world PoP traffic is applied, where one server cluster behind VIP processes on average 309.84 active flows per second.
Cheetah [144]	2 LB + 24 servers	A Python generator creates 1500-2500 synthetic requests/s.

Table 7.2: Survey on real-world testbed configurations.

90th-percentile, standard deviation, discounted average, and weighted discounted average[3] – as inputs for LB agents.

Action. To bridge the different timing constraints between the control plane and data plane, each LB agent assigns the j-th server to newly arrived tasks using the ratio of two factors, $j = \arg\min_{k\in[N]} \frac{q_{ik}+1}{a_{ik}}$, where the number of ongoing tasks, q_{ik} helps track dynamic system server occupation at a per-flow level. This allows making load-balancing decisions at μs-level speed – and a_{ik} is the periodically updated RL-inferred server processing speed. As in line 14 of algorithm 5, $\alpha_{ij}(t)$ is a statistical estimation of workload assignment distribution at time interval $[t, t+1)$.

Reward. The individual reward for distributed MPG LB is chosen as the VBF (as Def. 7.2) of the discounted average of sampled task duration measured on each LB agent, such that the LB group jointly optimize towards the potential function defined in equation (7.6). Task duration information is gathered as the time interval between the end of flow initialization (*e.g.,* 3-way handshake for TCP traffic) and the acknowledgment to the first data packet (*e.g.,* the first ACK packet for TCP traffic). Given the limited and partial observability of LB agents, task duration information approximates the remaining workload $\boldsymbol{l}$ by measuring the queuing and processing delay for new-coming tasks on each server. These PBF- and MS-based rewards are also implemented for the CTDE MARL algorithm as a comparison.

Model. The architecture of the proposed RL framework is depicted in figure 7.4. Each LB agent consists of a replay buffer, and a pair of actor-critic networks, whose architecture is depicted on the top right. There is also a pair of guiding actor-critic networks, with the same network architectures but updated in a delayed and soft manner. Each LB agent takes observations $o_i(t)$ extracted from the data plane (*e.g.,* numbers of ongoing tasks $\{q_{ij}\}$, task duration, TCT) and actions from the previous timestep $a_i(t-1)$ as inputs, and periodically generates new actions $a_i(t)$, which is used to update the server assignment function $\arg\min_{j\in[N]} \frac{q_{ij}+1}{a_{ij}}$ in the data plane. The gated recurrent units (GRU) [299] are applied for all agents to leverage the sequential history information for handling partial observability.

7.5 Implementation

This chapter uses an event-based simulator (as in section 6.4.2) to study the distance between the NE achieved by the proposed algorithm and the NE achieved by the theoretical optimal load-balancing policy (with perfect observation). This chapter uses a realistic testbed (as in section 6.4.1) deployed on physical servers in a data center network, and providing Apache web services, with real-world network traffic [226], to evaluate the real-world performance of the proposed algorithm, in comparison with in-production state-of-the-art LB [65].

To justify the setups of the experiment reflect the "real-world" scenarios, a brief survey of real-world data center setup is presented based on a set of state-of-the-art load-balancing research papers, which are summarized below (table 7.2). The configuration using 2 physical servers (48 CPUs each) allows conducting experiments similar to real-world setups. Based on the survey, the experiments conducted in this chapter have reasonable scale – not only in terms of the number

[3]Discounted average weights are computed as $0.9^{t'-t}$, where t is the sampled timestamp and t' is the moment of calculating the reduced scalar.

	Hyperparameter	Simulation	Experiments	
		Small-Scale	Small-Scale	Large-Scale
Distributed LB	Learning rate	3×10^{-4}	1×10^{-3}	1×10^{-3}
	Batch size	25	25	12
	Hidden dimension	64	64	128
	Hypernet dimension	32	32	64
	Replay buffer size	3000	3000	3000
	Episodes	500	120	200
	Updates per episode	10	10	10
	Step interval	0.5s	0.25s	0.25s
	Target entropy	$-\|\mathcal{A}\|$	$-\|\mathcal{A}\|$	$-\|\mathcal{A}\|$
LB System	TCT Distribution	Exponential	Real-world trace	Real-world trace
	Average TCT	1s	200ms	200ms
	Average bytes per task	-	12KiB	12KiB
	Traffic rate	20.28tasks/s	$[650, 800]$tasks/s	2000tasks/s
	Number of LB agents	2	2	6
	Total number of servers	8	7	20
	Server group 2	4 (1-CPU)	3 (2-CPU)	10 (2-CPU)
	Server group 1	4 (2-CPU)	4 (4-CPU)	10 (4-CPU)
	Episode duration	60s	60s	60s

Table 7.3: Hyperparameters in MARL-based LB.

of agents (2/6 LBs) and servers (7/20 servers), but also in terms of traffic rates – more than 2k queries per second per VIP and more than 1150.76 concurrent flows in large scale experiments —- and are representative of real-world circumstances.

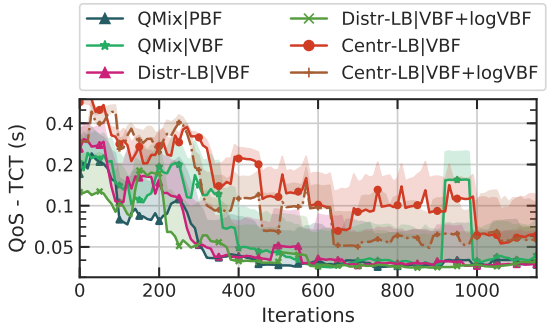
7.5.1 Hyperparameters

MARL-based load-balancing methods are trained in both simulator, and small- and large-scale testbed setups for various amounts of episodes. At the end of each episode, the RL models are trained and updated for 10 iterations. Given the total provisioned computational resource, the traffic rates of network trace for training are carefully selected so that the RL models can learn from sensitive cases where workloads should be carefully placed to avoid overloaded less powerful servers. The set of hyper-parameters is listed in table 7.3.

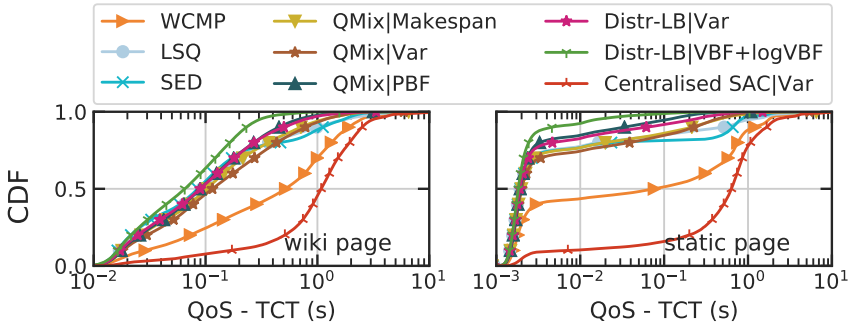
7.5.2 Benchmark Load Balancing Methods

To compare load-balancing performance, 4 state-of-the-art workload distribution algorithms are implemented. Equal-cost multi-path (ECMP) randomly assigns servers to tasks with a server assignment function $\mathbb{P}(j) = \frac{1}{n}$, where $\mathbb{P}(j)$ denotes the probability of assigning the j-th server [193]. Weighted-cost multi-path (WCMP) assigns servers based on their weights derived, and has an assignment function as $\mathbb{P}(j) = \frac{v_j}{\sum v_j}$ [65]. The local shortest queue (LSQ) algorithm assigns the server with the shortest queue, *i.e.,* $\arg\min_{j\in[n]} |w^j(t)|$ [261]. The shortest expected delay (SED) algorithm[4] assigns a task to server the shortest queue normalized by the number of processors, *i.e.,* $\arg\min_{j\in[n]} \frac{|w^j(t)|+1}{v_j}$ [236], and is expected to have the best performance among conventional heuristics. In the simulator, an *Oracle* LB algorithm is implemented, which distributes flows to the server which is expected to finish all its tasks with the lowest delay (including the new flow). The Oracle LB is aware of the remaining time of each flow, which is otherwise not observable for network LBs in real-world setups. When receiving a new flow, the Oracle LB algorithm calculates the remaining time to process on each server (assuming the newly received flow is assigned on the server as well) and assigns the server with the lowest remaining time to process the new-coming flow, to make sure that the makespan is always minimized with the global observation, which is not possible to be achieved in a real-world system. The load-balancing decisions for the Oracle algorithm are also made immediately for the Oracle LB algorithm. This Oracle LB is representative

[4] As studied in Chapter 6, HLB is able to achieve similar performance as SED without pre-configure server weights. However, HLB fails to map to the performance of SED under heavy traffic (*e.g.,* $> 90\%$ expected resource utilization). Therefore, for clarity, SED is selected as the benchmarked algorithm, which is superior to HLB.



(a) Learning curves.



(b) CDF of TCT.

Figure 7.5: Experimental results show that the proposed distributed RL framework (Distr-LB) using proposed VBF as rewards converge and effectively achieves better load-balancing performance (lower TCT and better QoS) than existing LB algorithms and CTDE RL algorithms.

of the application-layer load balancers [263, 300, 301], which makes more informed load-balancing decisions based on their observations with finer granularity.

Method		Period I (796.315 queries/s)		Period II (687.447 queries/s)		Period III (784.522 queries/s)		Period IV (784.522 queries/s)	
		Wiki	Static	Wiki	Static	Wiki	Static	Wiki	Static
WCMP		0.435 ± 0.083	0.171 ± 0.055	0.254 ± 0.087	0.073 ± 0.056	0.412 ± 0.101	0.134 ± 0.059	0.834 ± 0.323	0.492 ± 0.276
LSQ		0.141 ± 0.073	0.023 ± 0.030	0.143 ± 0.040	0.023 ± 0.011	0.620 ± 0.442	0.339 ± 0.316	0.357 ± 0.373	0.173 ± 0.299
SED		0.137 ± 0.076	0.020 ± 0.023	0.131 ± 0.067	0.027 ± 0.035	0.215 ± 0.210	0.051 ± 0.081	0.346 ± 0.496	0.169 ± 0.330
RLB-SAC [28]	Jain	0.137 ± 0.020	0.009 ± 0.006	0.125 ± 0.035	0.012 ± 0.008	0.193 ± 0.073	0.026 ± 0.022	0.204 ± 0.084	0.039 ± 0.047
	G	0.140 ± 0.053	0.015 ± 0.019	0.103 ± 0.022	0.010 ± 0.007	0.149 ± 0.049	0.015 ± 0.011	0.155 ± 0.052	0.011 ± 0.011
QMix-LB	MS	0.258 ± 0.174	0.071 ± 0.087	0.142 ± 0.073	0.030 ± 0.034	0.217 ± 0.157	0.048 ± 0.069	0.263 ± 0.202	0.073 ± 0.092
	logMS	0.167 ± 0.031	0.009 ± 0.004	0.132 ± 0.034	0.011 ± 0.008	0.844 ± 1.376	0.635 ± 1.249	0.278 ± 0.130	0.041 ± 0.038
	VBF	0.128 ± 0.052	0.014 ± 0.017	0.132 ± 0.075	0.016 ± 0.025	0.141 ± 0.025	0.008 ± 0.004	0.286 ± 0.162	0.068 ± 0.066
	logVBF	**0.106 ± 0.011**	0.007 ± 0.001	0.109 ± 0.032	0.011 ± 0.009	0.171 ± 0.043	0.022 ± 0.013	0.223 ± 0.045	0.026 ± 0.017
	VBF+logVBF	0.112 ± 0.005	**0.005 ± 0.002**	0.101 ± 0.010	0.005 ± 0.001	0.187 ± 0.090	0.024 ± 0.029	0.201 ± 0.080	0.021 ± 0.020
	PBF	0.142 ± 0.035	0.012 ± 0.006	0.099 ± 0.011	**0.004 ± 0.001**	0.211 ± 0.153	0.047 ± 0.078	0.181 ± 0.042	0.018 ± 0.009
	CV	0.407 ± 0.505	0.201 ± 0.340	0.113 ± 0.036	0.009 ± 0.008	0.203 ± 0.089	0.039 ± 0.037	0.219 ± 0.072	0.031 ± 0.017
Centr-LB	VBF	0.690 ± 0.211	0.284 ± 0.181	0.152 ± 0.041	0.016 ± 0.011	1.068 ± 0.386	0.570 ± 0.378	1.378 ± 0.377	0.867 ± 0.350
	logVBF	0.676 ± 0.231	0.265 ± 0.151	0.160 ± 0.023	0.013 ± 0.005	0.938 ± 0.200	0.446 ± 0.179	0.972 ± 0.288	0.495 ± 0.268
	VBF+logVBF	0.520 ± 0.034	0.167 ± 0.017	0.192 ± 0.040	0.019 ± 0.014	0.759 ± 0.254	0.306 ± 0.222	1.013 ± 0.168	0.520 ± 0.167
Distr-LB	VBF	**0.106 ± 0.013**	0.007 ± 0.002	**0.090 ± 0.016**	0.007 ± 0.005	0.159 ± 0.054	0.017 ± 0.009	0.196 ± 0.091	0.032 ± 0.033
	logVBF	0.139 ± 0.021	0.011 ± 0.004	0.129 ± 0.032	0.012 ± 0.011	0.250 ± 0.156	0.057 ± 0.077	0.226 ± 0.059	0.038 ± 0.019
	VBF+logVBF	0.126 ± 0.038	0.009 ± 0.006	0.094 ± 0.023	0.006 ± 0.006	**0.108 ± 0.022**	**0.004 ± 0.001**	**0.104 ± 0.013**	**0.006 ± 0.003**
	CV	0.150 ± 0.040	0.011 ± 0.009	0.149 ± 0.060	0.026 ± 0.025	0.301 ± 0.146	0.066 ± 0.072	0.267 ± 0.156	0.051 ± 0.052

Table 7.4: Complete results of *average* QoS (s) for comparison in small-scale real-world network setup (data center network and traffic).

In addition to these rule-based heuristic load-balancing algorithms, multiple RL-based load-balancing algorithms are implemented and compared. As an instance of CTDE-RL-based load balancer, QMix [281] is employed as the learning agent for the load balancer – QMix-LB, using the same framework as depicted in figure 7.4. The only difference between QMix-LB and Distr-LB is that, QMix-LB requires communications among LB agents to collect all transitions for training. As an instance of single-agent load balancer, a centralized RL agent for load balancing (Centr-LB) is implemented also using the framework depicted in figure 7.4. Finally, a state-of-the-art distributed RL-based load balancer without using the MARL framework [28] is implemented and compared with Distr-LB .

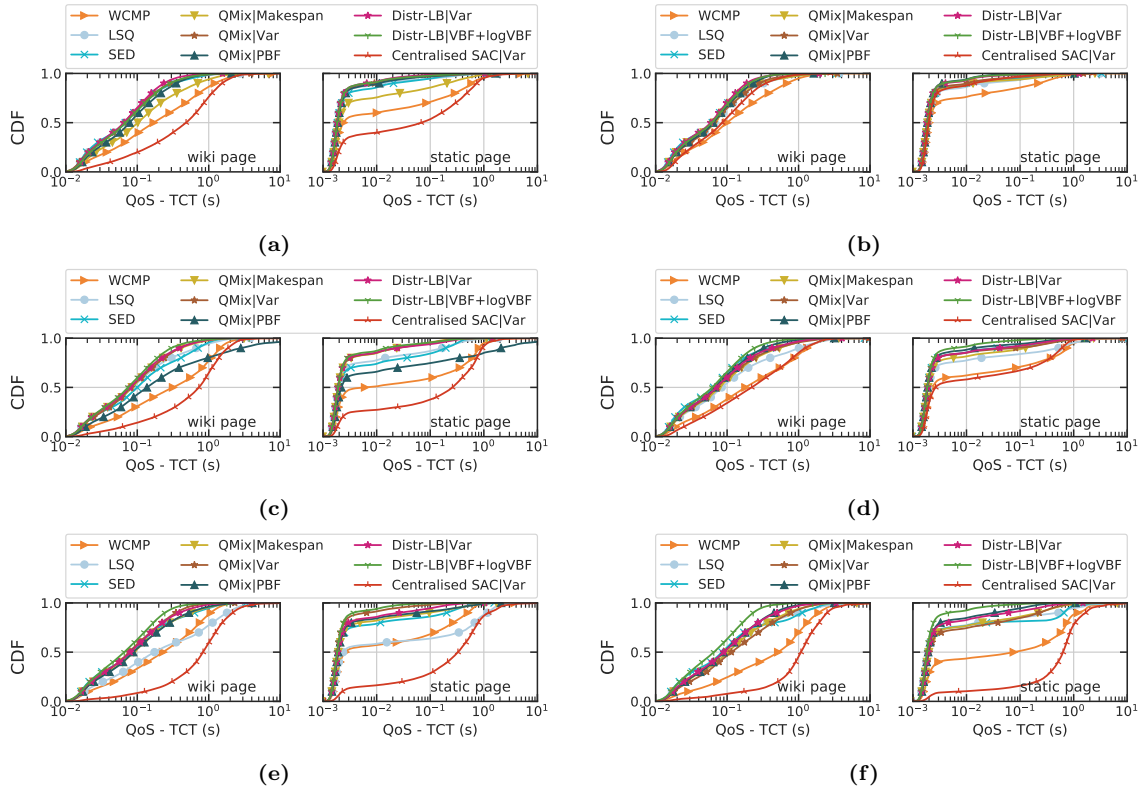


Figure 7.6: Experimental results with real-world network traces from different periods of time during a day demonstrate the effectiveness of the proposed distributed RL framework with VBF as rewards.

Method		Period I (796.315 queries/s)		Period II (687.447 queries/s)		Period III (784.522 queries/s)		Period IV (784.522 queries/s)	
		Wiki	Static	Wiki	Static	Wiki	Static	Wiki	Static
WCMP		5.801 ± 4.519	4.462 ± 3.867	4.019 ± 3.601	3.192 ± 3.543	3.239 ± 2.721	2.305 ± 2.700	8.066 ± 7.025	6.733 ± 5.329
LSQ		0.722 ± 0.487	0.195 ± 0.314	0.814 ± 0.478	0.288 ± 0.259	1.846 ± 1.915	1.168 ± 1.575	1.257 ± 1.921	0.831 ± 2.002
SED		0.706 ± 0.399	0.208 ± 0.246	0.697 ± 0.460	0.217 ± 0.291	0.726 ± 0.554	0.203 ± 0.261	0.909 ± 1.180	0.450 ± 1.112
RLB-SAC [28]	Jain	0.858 ± 0.240	0.159 ± 0.125	0.830 ± 0.358	0.227 ± 0.186	1.227 ± 0.489	0.354 ± 0.246	1.283 ± 0.594	0.408 ± 0.374
	G	0.945 ± 0.495	0.185 ± 0.214	0.682 ± 0.255	0.177 ± 0.162	1.003 ± 0.459	0.225 ± 0.176	0.973 ± 0.389	0.166 ± 0.156
QMix-LB	MS	1.469 ± 0.789	0.584 ± 0.547	1.095 ± 0.694	0.444 ± 0.423	1.182 ± 0.801	0.420 ± 0.483	1.447 ± 0.885	0.751 ± 0.772
	logMS	0.985 ± 0.264	0.117 ± 0.043	0.909 ± 0.388	0.172 ± 0.142	7.043 ± 12.237	6.427 ± 12.479	1.326 ± 0.584	0.371 ± 0.305
	VBF	0.732 ± 0.395	0.159 ± 0.239	0.665 ± 0.550	0.157 ± 0.278	0.744 ± 0.278	0.123 ± 0.093	1.028 ± 0.694	0.279 ± 0.365
	logVBF	0.682 ± 0.100	0.124 ± 0.019	0.772 ± 0.313	0.205 ± 0.159	1.174 ± 0.323	0.382 ± 0.183	1.426 ± 0.323	0.327 ± 0.153
	VBF+logVBF	0.664 ± 0.057	**0.087 ± 0.056**	0.611 ± 0.097	0.055 ± 0.027	1.171 ± 0.568	0.302 ± 0.293	1.206 ± 0.501	0.278 ± 0.239
	PBF	0.661 ± 0.193	**0.087 ± 0.099**	0.505 ± 0.119	0.048 ± 0.029	0.768 ± 0.728	0.205 ± 0.465	0.726 ± 0.433	0.128 ± 0.136
	CV	1.928 ± 2.228	1.281 ± 2.095	0.708 ± 0.405	0.131 ± 0.130	1.331 ± 0.593	0.481 ± 0.297	1.344 ± 0.329	0.451 ± 0.218
Centr-LB	VBF	3.101 ± 1.582	1.985 ± 1.790	0.903 ± 0.350	0.328 ± 0.353	4.409 ± 2.693	3.629 ± 3.219	6.649 ± 4.562	6.120 ± 4.721
	logVBF	2.715 ± 0.444	1.718 ± 0.547	1.016 ± 0.229	0.264 ± 0.092	3.247 ± 0.725	2.136 ± 0.832	4.286 ± 2.091	3.459 ± 2.323
	VBF+logVBF	2.459 ± 0.101	1.309 ± 0.063	1.243 ± 0.358	0.285 ± 0.189	2.796 ± 0.900	1.702 ± 1.287	3.466 ± 0.820	2.628 ± 1.142
Distr-LB	VBF	**0.651 ± 0.151**	0.119 ± 0.072	0.571 ± 0.237	0.133 ± 0.136	1.039 ± 0.302	0.298 ± 0.125	1.187 ± 0.594	0.355 ± 0.318
	logVBF	0.923 ± 0.162	0.193 ± 0.086	0.933 ± 0.415	0.243 ± 0.302	1.491 ± 0.764	0.579 ± 0.531	1.481 ± 0.473	0.558 ± 0.286
	VBF+logVBF	0.745 ± 0.316	0.185 ± 0.152	**0.385 ± 0.094**	**0.023 ± 0.003**	**0.595 ± 0.199**	**0.051 ± 0.030**	**0.563 ± 0.180**	**0.100 ± 0.073**
	CV	0.865 ± 0.261	0.147 ± 0.121	1.109 ± 0.668	0.433 ± 0.431	1.730 ± 0.468	0.612 ± 0.420	1.383 ± 0.666	0.446 ± 0.345

Table 7.5: Complete results of *99th percentile* QoS (s) for comparison in small-scale real-world network setup (data center network and traffic).

7.6 Evaluation

Small-Scale Real-World Testbed: As depicted in figure 7.5a, in a small-scale real-world data center network setup with 2 LB agents and 7 servers, after 120 episodes of training, the proposed distributed LB (Distr-LB) algorithm can learn from the environment based on VBF as rewards, and it converges to offer better QoS than QMix. Centralized RL agent (Centr-LB) has difficulties learning within 120 episodes because of the increased state and action space. An empirical finding is that, by adding a log term to the VBF-based reward for Distr-LB, LB agents become more sensitive to close-to-0 VBF during training ($\nabla_x \log f(x) > \nabla_x f(x)$ when $f(x) < 1$), therefore achieving better load-balancing performance.

As depicted in figure 7.5b, when comparing with in-production LB algorithms (WCMP, LSQ, SED), Distr-LB shows clear performance gains and reduced TCT for both types of web pages – Wikipedia pages require making queries to the SQL databases thus they are more CPU-intensive,

		50%-CPU+50%-IO	75%-CPU+25%-IO	100%-CPU
Oracle		6.437 ± 1.006	1.469 ± 0.102	1.291 ± 0.075
QMix-LB	PBF	10.230 ± 0.108	1.828 ± 0.054	2.200 ± 0.288
	VBF	10.936 ± 0.470	2.023 ± 0.255	2.125 ± 0.074
Distr-LB	VBF	10.335 ± 0.362	$\mathbf{1.695 \pm 0.104}$	$\mathbf{1.643 \pm 0.016}$
	VBF+logVBF	$\mathbf{8.797 \pm 0.459}$	1.873 ± 0.328	2.004 ± 0.042

Table 7.6: Comparison of average QoS (s) in simulator for different types of applications.

while static pages are IO-intensive. The comparison of average TCT using different LB algorithms is shown in table 7.4 (99th percentile TCT in table 7.5). The proposed Distr-LB also shows superior performance than the RL-based solution (RLB-SAC) [28] because of (i) a critically-designed MARL framework, and (ii) the use of a recurrent neural network to handle load-balancing problem as a sequential problem.

Reward Engineering: To verify the effectiveness of the proposed potential function VBF, it is compared with a set of different reward functions, including makespan (MS), PBF, and coefficient of variation (CV). During the study based on a real-world testbed, it is found that, when using VBF as the reward, the convergence is fast at the beginning of the training process and the sample variance of average flow duration (as an estimation of the queuing and processing delay) on each server becomes close to zero. However, it does not necessarily mean that the load-balancing policy is optimal and the NE is achieved. To capture the small variance which is close-to-zero, the logarithm of VBF (logVBF) is calculated as reward. And the combination of VBF + logVBF is an empirical design aiming at faster convergence towards the NE policy. The complete comparison results are shown in table 7.4 (average QoS) and in table 7.5 (99th percentile QoS), where the proposed distributed MARL framework achieves the best performance for most cases. To provide a complete view of all comparison results besides the one shown in figure 7.5b, the CDF of task completion time under all test cases is depicted in figure 7.6 Accompanying the evaluaion results of average QoS in large-scale testbed in table 7.7, table 7.8 also shows the 99th percentile QoS in a large-scale testbed.

NE Gap Evaluation with Simulation: To evaluate the gap between the performance of Distr-LB and the theoretical optimal policy, an Oracle LB is implemented in the simulator, which has perfect observation (inaccessible in the real world) over the system and minimizes makespan for each load-balancing decision. The configurations of the simulation are the following. There are 2 LB agents and 8 servers. 4 servers have 1 CPU worker-thread each while the other 4 servers have 2 CPU worker-threads each, to simulate the different server processing capacities. Three types of applications are compared. 100%-CPU application is a single-stage application, whose expected time to process is 1s in the CPU queue and 0s in the IO queue. 75%-CPU+25%-IO application is a two-stage application, whose expected time to process is 0.75s in the CPU queue and 0.25s in the IO queue, simulating the CPU-intensive applications. 50%-CPU+50%-IO application is a two-stage application, whose expected time to process is 0.5s in both the CPU and IO queue. The actual time to process of each task follows an exponential distribution. The traffic rate is normalized to consume on average 84.5% resources.

Table 7.6 shows that, for different types of applications, Distr-LB can achieve closer-to-optimal performance than is QMix. As the simulator is implemented based on the load-balancing model formulated in this chapter, the theoretical analysis can be directly applied, and VBF – as a potential function – helps independent cooperative LB agents to achieve good performance. The additional *log* term shows empirical performance gains in a real-world system, yet it is not necessarily the case in these simulation results. First, the generated traffic of tasks in the simulation has a higher expected workload ($>$ 1s mean and stddev), while the *log* terms are more sensitive to close-to-0 variances, which is the case in real-world experimental setups. In addition, though the simulator models the formulated LB problem, it fails to capture the complexity in the real-world system – *e.g.,* Apache backlog, multi-processing optimization, context switching, multi-level cache, network queues etc. For instance, batch processing [121] helps reduce cache and instruction misses, yet yields similar processing time for different tasks, thus the variance of task processing delay decreases and becomes closer to 0 in the real-world system. The additional *log* term exaggerates the low variance differences to better evaluate load-balancing decisions.

Method		Period I (2022.855 queries/s)		Period II (2071.129 queries/s)	
		Wiki	Static	Wiki	Static
WCMP		0.473 ± 0.102	0.194 ± 0.090	0.460 ± 0.241	0.239 ± 0.212
LSQ		0.266 ± 0.127	0.063 ± 0.065	0.218 ± 0.246	0.082 ± 0.152
SED		0.169 ± 0.062	0.020 ± 0.025	0.166 ± 0.141	0.050 ± 0.070
RLB-SAC-G [28]		0.182 ± 0.049	0.013 ± 0.009	0.111 ± 0.029	0.010 ± 0.009
QMix-LB	VBF	0.181 ± 0.062	0.019 ± 0.020	0.188 ± 0.147	0.052 ± 0.075
	PBF	0.210 ± 0.041	0.013 ± 0.006	0.104 ± 0.009	0.005 ± 0.003
Distr-LB	VBF	0.228 ± 0.055	0.019 ± 0.011	0.174 ± 0.102	0.035 ± 0.039
	VBF+logVBF	$\mathbf{0.161 \pm 0.033}$	$\mathbf{0.008 \pm 0.003}$	$\mathbf{0.094 \pm 0.015}$	$\mathbf{0.004 \pm 0.001}$

Table 7.7: Comparison of average QoS (s) in large-scale real-world network setup.

Method		Period I (2022.855 queries/s)		Period II (2071.129 queries/s)	
		Wiki	Static	Wiki	Static
WCMP		3.014 ± 0.612	2.152 ± 0.907	4.290 ± 3.593	3.300 ± 3.308
LSQ		1.863 ± 0.888	0.843 ± 0.773	1.243 ± 1.389	0.675 ± 1.223
SED		0.891 ± 0.475	0.208 ± 0.251	1.074 ± 0.751	0.592 ± 0.650
RLB-SAC-G [28]		1.064 ± 0.283	0.210 ± 0.132	0.739 ± 0.317	0.186 ± 0.214
QMix-LB	VBF	1.104 ± 0.481	0.241 ± 0.264	1.223 ± 1.169	0.634 ± 0.983
	PBF	1.201 ± 0.321	0.196 ± 0.112	0.583 ± 0.103	0.071 ± 0.050
Distr-LB	VBF	1.350 ± 0.311	0.263 ± 0.139	1.180 ± 0.702	0.448 ± 0.371
	VBF+logVBF	$\mathbf{0.890 \pm 0.250}$	$\mathbf{0.103 \pm 0.064}$	$\mathbf{0.531 \pm 0.149}$	$\mathbf{0.057 \pm 0.039}$

Table 7.8: Comparison of 99th percentile QoS (s) in large-scale real-world network setup (data center network and traffic).

		Wiki			Static		
		Mean	95th-percentile	99th-percentile	Mean	95th-percentile	99th-percentile
WCMP		1.792 ± 0.393	7.534 ± 1.817	2.366 ± 1.685	1.512 ± 0.385	6.571 ± 1.996	1.084 ± 1.842
LSQ		0.453 ± 0.178	1.958 ± 0.827	3.482 ± 1.257	0.202 ± 0.130	0.975 ± 0.617	1.801 ± 1.064
SED		0.340 ± 0.268	1.225 ± 0.812	30.600 ± 6.718	0.130 ± 0.206	0.519 ± 0.571	29.893 ± 7.042
QMix-LB	MS	0.373 ± 0.177	1.621 ± 0.830	4.046 ± 6.632	0.144 ± 0.112	0.663 ± 0.523	2.655 ± 6.899
	PBF	0.368 ± 0.375	1.529 ± 1.581	2.436 ± 1.468	0.159 ± 0.338	0.733 ± 1.437	0.974 ± 1.204
	VBF	0.282 ± 0.166	1.186 ± 0.799	3.187 ± 1.479	0.081 ± 0.104	0.395 ± 0.518	1.654 ± 1.181
	VBF+logVBF	0.533 ± 0.179	2.525 ± 0.913	4.864 ± 1.635	0.266 ± 0.129	1.409 ± 0.680	3.374 ± 1.626
Distr-LB	VBF	0.262 ± 0.100	1.086 ± 0.454	2.190 ± 0.792	0.057 ± 0.044	0.305 ± 0.234	0.683 ± 0.510
	VBF+logVBF	$\mathbf{0.221 \pm 0.112}$	$\mathbf{0.895 \pm 0.530}$	$\mathbf{1.903 \pm 0.976}$	$\mathbf{0.039 \pm 0.057}$	$\mathbf{0.197 \pm 0.284}$	$\mathbf{0.480 \pm 0.650}$

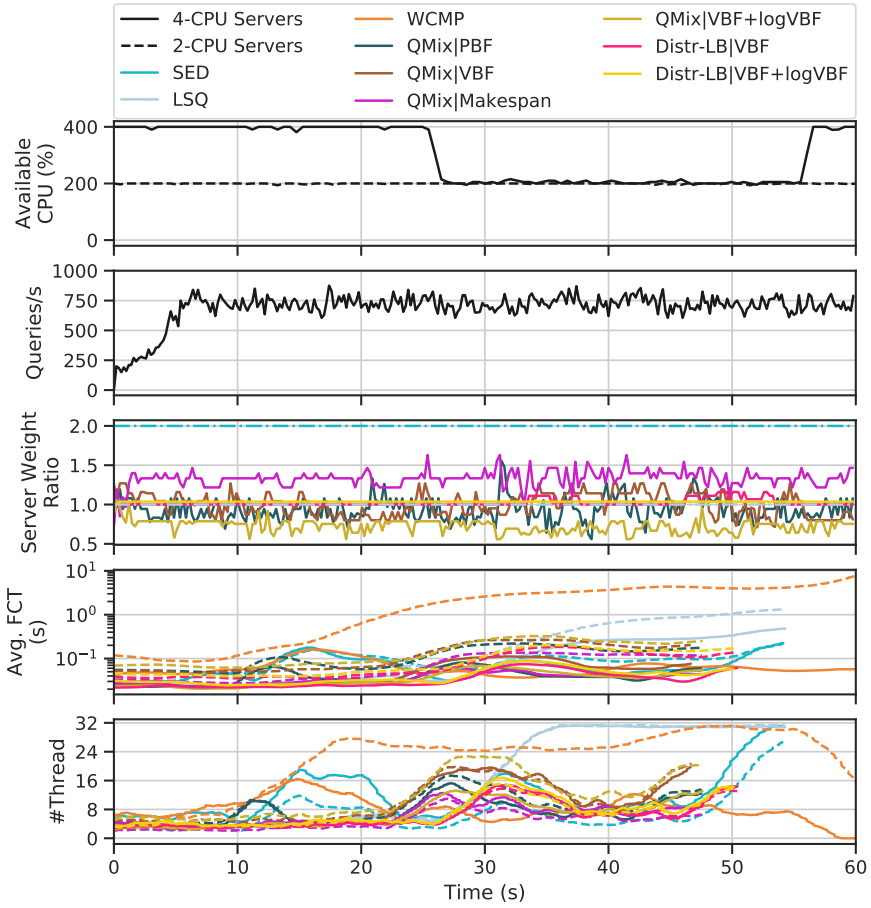
Table 7.9: Comparison of QoS (mean, 95th-percentile, and 99th-percentile task completion time in s) when server processing capacity changes over time.

Large-Scale Real-World Testbed: To evaluate the performance of Distr-LB in large-scale data center networks in the real world, the testbed is scaled up to have 6 LB agents and 20 servers and apply heavier network traffic ($>$ 2000 queries/s) to evaluate the performance of the LB algorithms that achieved the best performance in small scale setups, in comparison with in-production LB algorithms. The test results after 200 episodes of training are shown in table 7.7, where Distr-LB achieves the best performance in all cases. The QMix-LB also outperforms in-production LB algorithms. But as a CTDE algorithm, similar to the Centr-LB, it requires agents to communicate their trajectories, which – after 200 episodes of training – become 221MiB communication overhead at the end of each episode (episodic training), whereas 95%-percentile per-destination-rack flow rate is less than 1MiB/s [1].

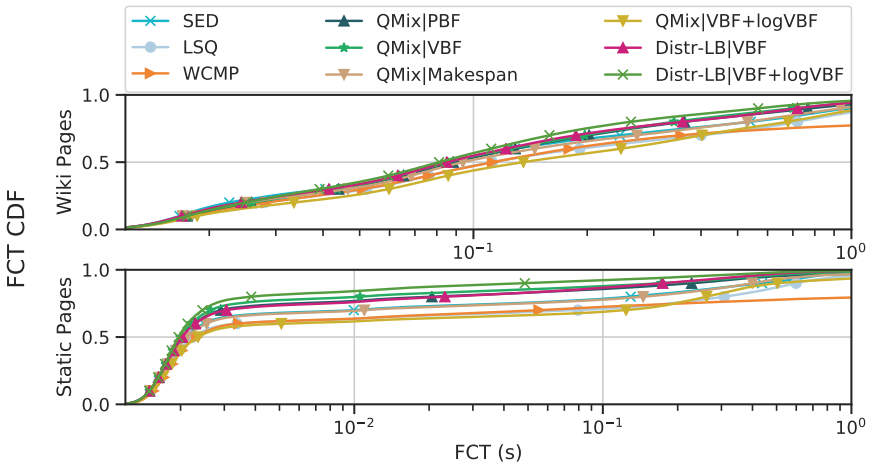
7.6.1 MARL Robustness

Given elastic and server-less computing environments, where tenants in data center networks can share physical resources (*e.g.,* CPU, disk, memory), servers can have different processing capacities, which may also change over time dynamically —- because of *e.g.,* updated server configuration (upgrading an Amazon EC2 `a1.xlarge` instance to `a1.4xlarge`) or resource contention (co-located workloads) [69]. According to [190], there are 32% of server clusters in data center networks that update more than 10 times per minute based on the measurements collected over 432 minutes up time in a month. 3% of clusters have more than 50 updates perf minute. Therefore, dynamic changes prevail in real-world data center networks.

Therefore, this section illustrates the robustness of the proposed distributed RL-based LB framework to react to dynamic changes in server processing speeds, *e.g.,* when server VMs are migrated to a new physical architecture. Using the same small-scale real-world testbed with 2 LB agents, additional CPU-bound workloads are applied on the 4-CPU server group starting from 25s.



(a) Additional workloads are applied on servers with 4 CPUs at around 25s.



(b) FCT CDF comparisons for two types of tasks.

Figure 7.7: Load balancing performance comparison in dynamic environments.

As depicted in figure 7.7a, when subjected to heavy Wikipedia traffic, MARL-based LB agents adapt server weights over time and achieve better performance than heuristic LB algorithms – finishing the same amount of workloads faster, maintaining a lower amount of active number of threads, even when server processing capacity is reduced. As depicted in figure 7.7b, over multiple runs (10 runs for each LB algorithm), RL-based LB algorithms effectively achieve lower task completion time in dynamic environments. They help avoid human intervention and make the LB agents autonomously adapt to the changes in the system. Table 7.9 lists the performance of all LB algorithms in terms of the QoS (measured as the average and 95th-percentile task completion time).

7.7 Summary

Load balancing problems have been formalized as cooperative games, yet this chapter provided the first study of their applications in real-world networking systems. Based on the Markov potential game formulation, this chapter provided a formal description of the network load-balancing problem in the context of RL. While RL models using the centralized-training-decentralized-execution (CTDE) scheme have achieved superior performance on different applications, this chapter has shown that these CTDE RL models incur significant communication overhead in networking systems. To avoid additional performance overhead, chapter proposed a distributed multi-agent RL (MARL) approach for the network load-balancing problem. Using a critically designed variance-based fairness as the reward function, as well as the potential function of the Markov potential game, each LB agent can achieve the pure Nash equilibrium by optimizing its local reward. Through this setting, the redundant communication overhead among LB agents is removed, thus improving the overall training and deployment efficiency in real-world systems, with local observations only.

Under such formulation, both the effectiveness of the proposed distributed LB algorithm, and the proposed fairness evaluation function, are theoretically justified and experimentally validated. The proposed load-balancing algorithm – Distr-LB – outperforms state-of-the-art load balancers. When compared with other learning-based algorithms, including centralized training methods (QMix-LB), centralized RL agent, or fully distributed RL-based load balancer (RLB-SAC), Distr-LB has demonstrated superior performance in both simulation and realistic experiments in physical testbeds of different scales. While chapter 6 has shown that the load-balancing algorithm based on "open-loop" control can achieve performance similar to heuristics with given information (SED), this chapter has shown that "closed-loop" control helps achieve visible performance gains than SED.

This work paves the way for interdisciplinary studies that involve both advanced learning algorithms and realistic application scenarios in the cloud. The results from this chapter have been published in [28,183,185]. The source code and data of both simulation and real-world experiment are available under an open-source license at: https://github.com/ZhiyuanYaoJ/MARLLB.

Part IV

Conclusion

Chapter 8

Conclusion

Applications and services have become more complex, while the Internet has become increasingly difficult to evolve both regarding its physical infrastructure, and its protocols and performance. Being responsible for policy configurations as well as network management and performance tuning, network operators are shifting towards the use of more and more automated tools to accomplish these tasks. The concept of "programmable networks" has emerged to alleviate these challenges, and to facilitate network evolution. This includes paradigms such as (i) software-defined networking (SDN) and (ii) network function virtualization (NFV), which decouple the forwarding hardware into control plane and data plane, and which seek to abstract network forwarding, and other networking functions, from the hardware. In support of cloud computing, these paradigms have enabled rich network traffic processing services, while having also reduced the granularity of task allocation in data centers. It has been recognized that shifting controllers from logically centralized to distributed will increase not only scalability, but also robustness to inconsistency. Machine-Learning (ML)-based approaches have been proposed to deploy more intelligence in networks, when using decoupled control and data planes. However, the increasing scale, complexity, and heterogeneity of networking infrastructure, and protocols, as well as the demand for virtualization and cloud support services in terms of efficient resource management, rapid provisioning, and scalability presents a set of new challenges in effective network organization, management, and optimization, which are addressed in this manuscript.

This thesis has, in part II investigated, how to design and implement a feature collection and exploitation framework in real-world networking systems that enables **data-driven network functions**. Networking features and system state information help VNFs make informed decisions, and intelligently manage and update networking policies in cloud data centers. Actively collecting features and system state information entails substantial control signaling and management overhead, in particular in large-scale data center networks. Specifically, chapter 3 has proposed Aquarius, a framework that collects, infers, and supplies accurate networking state information with little additional processing latency, in a scalable buffer layout. It has illustrated significant performance gains of using of Aquarius for various ML-based VNFs and evaluated experimentally the impact of Aquarius on the system performance.

This thesis has also, in part III explored, how **network load balancers** can be optimized with the advance of hardware capacities and learning algorithms. Specifically, chapter 4 set the basis for the part, by exploring the feasibility of a hardware implementation of a load-aware load balancer. Using covert channels in packet headers to embed task queue lengths and server processing speed, Charon statelessly tracks server load states, and fairely distributes workloads, while guaranteeing PCC. Simulation results show that Charon helps network load balancers achieve an improved quality of service. Implemented as a prototype on the NetFPGA-SUME platform, it shows line-rate performance with high throughput and negligible per-packet processing latency. And then, this is followed by, the description of HLB, in chapter 6, which used Aquarius for systematically studying the factors that impact workload distribution fairness of different load-balancing algorithms. A stateful open-loop load-balancing algorithm called HLB has been proposed, taking 2 key factors – server occupancies and processing speeds – into account. With no *a-priori* knowledge or manual configurations and no additional control traffic, HLB has shown optimized performance in both simulations and testbed experiments. Finally, the possibility of applying reinforcement learning to network load-balancing problems has been explored in chapter 7. A distributed MARL

approach for multi-agent load-balancing problems has been studied, based on Markov potential game formulation. Under such formulation, the effectiveness of the proposed distributed RL-based LB algorithm together with the proposed fairness index as the reward function is both theoretically justified and experimentally verified. With no communication overhead among LB agents, it demonstrates a performance gain over state-of-the-art load-balancing algorithms in both simulation and real-world tests with different scales.

In sum, the work and the results, presented in this thesis consist of:

1. **Aquarius**, a framework that enables the deployment of different data-driven network functions on commodity hardware, out of which 2 papers have been published in a conference (best paper recipient) and a journal, with open-sourced code[1];

2. **Charon**, a stateless, line-rate, load-aware load balancer based on a heuristic algorithm, implemented on programmable hardware (NetFPGA), out of which 1 paper has been published in a workshop and 1 US patent has been filed, with open-sourced code[2];

3. **MLB**, a stateful, Machine-Learning-based load balancer based on Aquarius, that investigated in the use of networking features and different models, and showcased the potential performance gains when data-driven algorithms, out of which 1 paper has been published in a workshop;

4. **HLB**, a stateful load-aware load balancer based on a statistical learning algorithm, by way of "open-loop" control, that has been extensively evaluated, out of which 1 paper has been published in a journal and 1 US patent has been filed, with open-sourced code[3];

5. **Distr-LB**, a stateful load-aware load balancer based on a multi-agent reinforcement learning (MARL) framework, by way of "closed-loop" control, out of which 1 paper has been published in a workshop and 2 papers have been published in 2 conferences, with open-sourced code[4].

The results achieved in this thesis have demonstrates that it is possible to provide generic datacenter primitives where data-driven decisions are taken *directly in the data plane* in a distributed way, rather than by a centralized controller. This, therefore, complements the traditional software defined networking architectures, wherein the data plane would be constantly updated to reflect the last decisions taken by the controller, by going one level further in granularity and realtimeness. The results presented in this thesis have shown this, which provides better resource usage and lower network usage, while requiring less monitoring and centralization.

To conclude, such an approach provides greater scalability in data center premises, both (i) by unifying the network layer and decreasing the need for out-of-band monitoring, and (ii) by providing better resource utilization as a result of this ability to take local decisions. Such architectures are therefore desirable, for reasons concerning both operational complexity and reduction of energy consumption.

As a consequence, the next research questions arise at the intersection of (i) scalable and high performance networking systems, and (ii) adaptive, robust, and resilient data-driven algorithms. Both the Aquarius framework and the data-driven load-balancing algorithms serve as a cornerstone that supports more scalable and intelligent network functions for autonomous service management in the cloud.

[1] https://github.com/ZhiyuanYaoJ/Aquarius
[2] https://github.com/ZhiyuanYaoJ/SimLB/tree/charon
[3] https://github.com/ZhiyuanYaoJ/SimLB/tree/hlb
[4] https://github.com/ZhiyuanYaoJ/MARLLB

Appendix A

Résumé en français

Cette thèse étudie l'utilisation de méthodes basées sur les jeux de données afin d'optimiser certaines fonctions réseau — par exemple l'équilibrage de charge — et d'améliorer les performances dans les réseaux de centres de données. Elle comprend 4 parties et 8 chapitres, structurés comme suit.

La partie I fournit une discussion d'introduction. Le chapitre 1 introduit des informations générales sur les architectures de centre de données, ainsi que sur les protocoles réseau associés et les architectures correspondantes. Ensuite, le chapitre étudie l'exigences croissantes d'intégrer davantage de programmabilité directement dans le réseau, ce qui permet de prendre des décisions plus éclairées et d'appliquer des politiques de configuration réseau basées sur les jeux de données. Quatre fonctions réseau d'intérêt sont présentées (passerelle VPN, adaptation automatique du nombre de serveurs, équilibrage de charge et classification du trafic) afin de démontrer le rôle des différentes fonctions réseau, ainsi que leur comportement attendu dans les réseaux de centres de données modernes. Enfin, le chapitre discute la Enfin, le chapitre discute la façon dont l'utilisation de différents types de méthodes basées sur les jeux de données peut aider 'a améliorer les centres de données, en fournissant des primitives directement au niveau de la couche réseau. Le chapitre 2 résume comment ce concept a été appliqué tout au long de cette th'ese et décrit les grandes lignes et les contributions de cette th'ese.

Pour relever les défis susmentionnés, la partie II présente un outil pour collecter et exploiter des caractéristiques (*features*) réseau dans les centres de données. Le chapitre 3 (publié dans [179,194]) présente une architecture qui permet de mettre en place des fonctions réseau utilisant efficacement des jeux données. Afin de gérer et de mettre à jour dynamiquement les politiques de configuration réseau dans les centres de données cloud, les fonctions virtuelles de réseau (VNF) utilisent, en les collectant activement, les informations sur l'état du réseau. Ce faisant, cela entraîne des coûts supplémentaires en termes de trafic de signalisation, en particulier dans les grands centres de données. Pour palier à cela, les VNF en production préfèrent des heuristiques simples et distribuées plutôt que des algorithmes d'apprentissage avancés, afin d'éviter une latence de traitement supplémentaire, indésirable dans les réseaux à haute performance et faible latence. Ce chapitre identifie les défis du déploiement d'algorithmes d'apprentissage dans le contexte des centres de données cloud, et propose Aquarius pour faire le pont entre l'application des techniques d'apprentissage automatique (ML) et la gestion des services réseau. Aquarius rassemble passivement mais efficacement des observations fiables et permet l'utilisation de techniques ML pour collecter, déduire et fournir des informations précises sur l'état du réseau, sans entraîner de trafic de signalisation supplémentaire. Il offre une visibilité fine et programmable aux VNF distribuées et permet un contrôle en boucle ouverte et fermée sur les systèmes réseau. Ce chapitre illustre l'utilisation d'Aquarius avec un classificateur de trafic, un système d'adaptation automatique du nombre de serveurs, et unéquilibreur de charge. Il démontre ainsi l'utilisation de trois paradigmes ML différents - apprentissage non supervisé, supervisé et par renforcement - au sein d'Aquarius, pour l'inférence de l'état du réseau. Les évaluations sur une implémentation réelle montrent qu'Aquarius améliore de manière appropriée la visibilité de l'état du réseau et apporte des gains de performances notables pour divers scénarios, et ce à un faible surcoût.

La partie III étudie le problème de l'équilibrage de charge dans les réseaux de centres de données. Dans le chapitre 4 (publié dans [195]), un mécanisme d'équilibrage de charge basé sur NetFPGA est proposé. Le suivi des états de connexion permet aux équilibreurs de charge de déduire les états de

charge du serveur et de prendre des décisions éclairées, mais au prix d'une consommation d'espace mémoire supplémentaire. Cela le rend difficile à mettre en œuvre sur du matériel programmable, qui a une mémoire limitée mais offre un débit élevé. Ce chapitre présente Charon, un équilibreur de charge implémenté dans P4-NetFPGA, prenant en compte l'état de charge des applications sans pour autant maintenir d'état, et qui peut atteindre le débit maximal. Charon collecte passivement les états de charge des serveurs d'applications et utilise un schéma de "power-of-2-choices" pour prendre des décisions d'équilibrage de charge en fonction de la charge, et ainsi améliorer l'utilisation des ressources. Le maintien de la cohérence des connexions est garanti, et ce sans garder d'état, en encodant l'ID du serveur dans un canal caché. Le chapitre décrit la conception d'un prototype, avant de fournir les résultats de simulations. Ceux-ci montrent des gains de performances en termes d'équité de répartition de la charge, de qualité de service, de débit et de latence de traitement.

Le chapitre 5 (publié dans [152]) décrit la première tentative connue d'appliquer des algorithmes ML pour optimiser leséquilibreurs de charge réseau. Les algorithmes de répartition de la charge utilisent normalement des heuristiques, par exemple Equal-Cost Multi-Path (ECMP), Weighted-Cost Multi-Path (WCMP) ou des algorithmes naïfs de ML, par exemple la régression ridge. Les approches avancées basées sur ML permettent d'obtenir des gains de performances dans différents problèmes de réseau et de système. Cependant, il est difficile d'appliquer des algorithmes ML à des problèmes de réseau dans des systèmes réels. Cela nécessite une connaissance du domaine, afin d'être en mesure de collecter les bonnes caractéristiques dans des réseaux à faible latence et à haut débit, évolutifs, dynamiques et hétérogènes. Ce chapitre effectue tout d'abord une analyse de données hors ligne et la conception d'un modèle. Il décrit ensuite un déploiement en-ligne dans un système réel basé sur Aquarius. Les résultats montrent que les modèles ML améliorent les performances d'équilibrage de charge, mais ils révèlent également des défis à résoudre pour appliquer ML aux réseaux, y compris le manque de généralisation dans les environnements dynamiques.

Dans le chapitre 6 (publié dans [196]), un algorithme d'équilibrage de charge en boucle ouverte basé sur un filtre de Kalman est proposé pour obtenir une répartition de la charge de travail prenant en compte l'état de charge des applications, avec une bonne généralisation. Ce chapitre propose unéquilibreur de charge hybride (HLB) distribué et indépendant des applications sous-jacentes qui, sans surveillance ni signalisation explicites, déduit les taux d'occupation et les vitesses de traitement des serveurs, ce qui permet de prendre des décisions de placement de charge optimisées. Cette approche est évaluée à la fois par des simulations et des expériences approfondies, y compris des charges synthétiques, mais aussi une trace issue de Wikipedia, également basé sur Aquarius. Les résultats montrent des gains de performances significatifs, en termes de temps de réponse et d'utilisation du système, par rapport aux algorithmes d'équilibrage de charge existants. En collectant des caractéristiques réseau passivement, HLB est capable d'atteindre des performances similaires à celles de l'algorithme Shortest Expected Delay (SED), qui lui nécessite une configuration manuelles pour informer les équilibreurs de charge des capacités de traitement des serveurs.

Dans le chapitre 7 (publié dans [28,183,185]), un algorithme d'équilibrage de charge en boucle fermée basé sur l'apprentissage par renforcement (RL) est proposé. Les équilibreurs de charge classiques fonctionnent dans des environnements dynamiques avec une surveillance limitée des charges des serveurs d'applications. Bien que HLB soit capable d'atteindre des performances similaires sans aucune connaissance préalable de la configuration du système, les équilibreurs de charge récents reposent toujours sur des algorithmes heuristiques qui nécessitent des configurations manuelles pour obtenir une bonne équité de répartition de charge. Pour palier à cela, ce chapitre propose un mécanisme d'apprentissage par renforcement distribué et asynchrone pour améliorer l'équité de la répartition de la charge obtenue par unéquilibreur de charge, et ce sans observation active de l'état du réseau et des serveurs. Étant donné que les centres de données utilisent souvent plusieurséquilibreurs de charge pour la redondance, l'apprentissage par renforcement multi-agents (MARL) est utilisé. Les défis de ce problème sont une architecture de traitement hétérogène et un environnement dynamique. De plus, chaque agent LB a une observabilité limitée et partielle du réseau, ce qui peut largement dégrader les performances des algorithmes d'équilibrage de charge dans des configurations réelles en production. L'algorithme Centralised Training and Distributed Execution (CTDE) a déjà été proposé pour améliorer les performances MARL, mais il entraîne - en particulier dans les systèmes de réseau distribués - des côuts supplémentaires en termes de communication et de gestion. Nous formulons le problème d'équilibrage de charge multi-agents comme un jeu potentiel de Markov, avec pour fonction potentielle, un fonction d'équité de distribution de charge bien conçue. Un algorithme MARL entièrement distribué est proposé pour approximer l'équilibre de Nash du jeu. Les évaluations expérimentales impliquent à la fois des simulations et un

déploiement en conditions réelles. L'algorithme d'équilibrage de charge MARL proposé montre des performances proches de l'optimum dans les simulations, et des résultats supérieurs par rapport aux LB en production dans les déploiements en conditions réelles.

Enfin, la partie IV conclut ce manuscrit.

List of Figures

This thesis comprises 113 figures, listed below.

List of Tables

This thesis comprises 24 tables, listed below.

List of Algorithms

Bibliography

[1] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. ACM, 2015, p. 123–137, event-place: London, United Kingdom.

[2] S. Yan, X. Wang, X. Zheng, Y. Xia, D. Liu, and W. Deng, "Acc: Automatic ecn tuning for high-speed datacenter networks," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 384–397.

[3] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, no. 8. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[4] D. Gedia and L. Perigo, "Performance evaluation of sdn-vnf in virtual machine and container," in *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2018, pp. 1–7.

[5] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *ACM Sigcomm Demo*, vol. 4, no. 5, p. 6, 2009.

[6] D. Gedia and L. Perigo, "Latency-aware, static, and dynamic decision-tree placement algorithm for containerized sdn-vnf in openflow architectures," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2019, pp. 1–7.

[7] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014.

[8] T. Swamy, A. Rucker, M. Shahbaz, and K. Olukotun, "Taurus: An intelligent data plane," *arXiv preprint arXiv:2002.08987*, 2020.

[9] M. Ghaznavi, E. Jalalpour, B. Wong, R. Boutaba, and A. J. Mashtizadeh, "Fault tolerant service function chaining," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 198–210.

[10] Y. Han, J. Li, D. Hoang, J.-H. Yoo, and J. W.-K. Hong, "An intent-based network virtualization platform for sdn," in *2016 12th International Conference on Network and Service Management (CNSM)*. IEEE, 2016, pp. 353–358.

[11] A. Mahimkar, C. E. de Andrade, R. Sinha, and G. Rana, "A composition framework for change management," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 788–806.

[12] Y. Geng, S. Liu, F. Wang, Z. Yin, B. Prabhakar, and M. Rosenblum, "Self-programming networks: Architecture and algorithms," in *2017 55th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2017, pp. 745–752.

[13] F. Zhang, G. Liu, X. Fu, and R. Yahyapour, "A survey on virtual machine migration: Challenges, techniques, and open issues," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1206–1243, 2018.

[14] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[15] N. Feamster and J. Rexford, "Why (and how) networks should run themselves," *arXiv preprint arXiv:1710.11583*, 2017.

[16] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, pp. 1–99, 2018.

[17] J. Xie, F. R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, and Y. Liu, "A survey of machine learning techniques applied to software defined networking (sdn): Research issues and challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 393–430, 2018.

[18] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin, and J. Aguilar, "Towards the deployment of machine learning solutions in network traffic classification: A systematic survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1988–2014, 2018.

[19] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *Proceedings of the 18th ACM workshop on hot topics in networks*, 2019, pp. 25–33.

[20] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 256–269.

[21] M. K. Putchala, "Deep learning approach for intrusion detection system (ids) in the internet of things (iot) network using gated recurrent neural networks (gru)," Ph.D. dissertation, Wright State University, 2017.

[22] A. Tuor, S. Kaplan, B. Hutchinson, N. Nichols, and S. Robinson, "Deep learning for unsupervised insider threat detection in structured cybersecurity data streams," *arXiv preprint arXiv:1710.00811*, 2017.

[23] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 123–134, 2013.

[24] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein, "Pantheon: the training ground for internet congestion-control research," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 731–743.

[25] V. Sivakumar, T. Rocktäschel, A. H. Miller, H. Küttler, N. Nardelli, M. Rabbat, J. Pineau, and S. Riedel, "Mvfst-rl: An asynchronous rl framework for congestion control with delayed actions," *arXiv preprint arXiv:1910.04054*, 2019.

[26] L. Chen, J. Lingys, K. Chen, and F. Liu, "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 191–205.

[27] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," *arXiv preprint arXiv:1810.01963*, 2018.

[28] Z. Yao, Z. Ding, and T. H. Clausen, "Reinforced workload distribution fairness," in *5th Workshop on Machine Learning for Systems at 35th Conference on Neural Information Processing Systems (NeurIPS 2021)*, 2021.

[29] D. Minarolli and B. Freisleben, "Distributed resource allocation to virtual machines via artificial neural networks," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2014, pp. 490–499.

[30] S. Fu, S. Gupta, R. Mittal, and S. Ratnasamy, "On the use of ml for blackbox system performance prediction." in *NSDI*, 2021, pp. 763–784.

[31] *Cisco Data Center Infrastructure 2.5 Design Guide*. San Jose, CA: Cisco, Nov. 2011.

[32] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an {RDMA-enabled} distributed persistent memory file system," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 773–785.

[33] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: a system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[34] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, p. 56–65, oct 2016. [Online]. Available: https://doi.org/10.1145/2934664

[35] X. Wei, Z. Dong, R. Chen, and H. Chen, "Deconstructing {RDMA-enabled} distributed transactions: Hybrid is better!" in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 233–251.

[36] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, "Gram: Scaling graph computation to the trillions," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 408–421.

[37] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-performance design of hadoop rpc with rdma over infiniband," in *2013 42nd International Conference on Parallel Processing*. IEEE, 2013, pp. 641–650.

[38] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling spark on hpc systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 97–110.

[39] H. Al Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 843–857.

[40] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati *et al.*, "Remote regions: a simple abstraction for remote memory," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 775–787.

[41] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda, "High-performance design of apache spark with rdma and its benefits on various workloads," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 253–262.

[42] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng, "Hydradb: a resilient rdma-driven key-value middleware for in-memory cluster computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–11.

[43] P. Namyar, S. Supittayapornpong, M. Zhang, M. Yu, and R. Govindan, "A throughput-centric view of the performance of datacenter topologies," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 349–369.

[44] A. Andreyev and A. Andreyev, "Introducing data center fabric, the next-generation facebook data center network," Apr 2022. [Online]. Available: https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/

[45] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan *et al.*, "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 514–528.

[46] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan *et al.*, "When cloud storage meets {RDMA}," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 519–533.

[47] K. Bilal, S. U. R. Malik, O. Khalid, A. Hameed, E. Alvarez, V. Wijaysekara, R. Irfan, S. Shrestha, D. Dwivedy, M. Ali *et al.*, "A taxonomy and survey on green data center networks," *Future Generation Computer Systems*, vol. 36, pp. 189–208, 2014.

[48] B. Wang, Z. Qi, R. Ma, H. Guan, and A. V. Vasilakos, "A survey on data center networking for cloud computing," *Computer Networks*, vol. 91, pp. 528–547, 2015.

[49] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Burlington, MA: Morgan Kaufmann, 2004.

[50] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

[51] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[52] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," *ACM SIGCOMM computer communication review*, vol. 45, no. 4, pp. 183–197, 2015.

[53] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, "F10: A {Fault-Tolerant} engineered network," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 399–412.

[54] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 225–238.

[55] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: a scalable and fault-tolerant network structure for data centers," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 75–86.

[56] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: a high performance, server-centric network architecture for modular data centers," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 63–74, 2009.

[57] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, and S. Lu, "Ficonn: Using backup port for server interconnection in data centers," in *IEEE INFOCOM 2009*. IEEE, 2009, pp. 2276–2285.

[58] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.

[59] R. Gandhi, Y. C. Hu, C.-K. Koh, H. H. Liu, and M. Zhang, "Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing," in *Proc. USENIX Annual Technical Conference (ATC)*, 2015, pp. 473–485.

[60] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34–45, 1974.

[61] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[62] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with {OpenLambda}," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[63] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[64] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu *et al.*, "Ananta: cloud scale load balancing," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 207–218.

[65] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *Proc. 13th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2016, pp. 523–535.

[66] "Amazon elastic compute cloud," https://aws.amazon.com/ec2/.

[67] N. Bitar, S. Gringeri, and T. J. Xia, "Technologies and protocols for data center and cloud networking," *IEEE Communications Magazine*, vol. 51, no. 9, pp. 24–31, 2013.

[68] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 270–282.

[69] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *Proceedings of the International Symposium on Quality of Service*, 2019, pp. 1–10.

[70] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[71] A. Whitaker, R. S. Cox, S. D. Gribble *et al.*, "Configuration debugging as search: Finding the needle in the haystack." in *OSDI*, vol. 4, 2004, pp. 6–6.

[72] A. Kumar, I. Narayanan, T. Zhu, and A. Sivasubramaniam, "The fast and the frugal: Tail latency aware provisioning for coping with load variations," in *Proceedings of The Web Conference 2020*, 2020, pp. 314–326.

[73] "Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model," ISO/IEC 7498-1, Geneva, Nov. 1994.

[74] W. Goralski, *The illustrated network: how TCP/IP works in a modern network*. Burlington, MA: Morgan Kaufmann, 2017.

[75] H. F. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, Jun. 1999. [Online]. Available: https://rfc-editor.org/rfc/rfc2616.txt

[76] E. Rescorla, "HTTP Over TLS," RFC 2818, May 2000. [Online]. Available: https://rfc-editor.org/rfc/rfc2818.txt

[77] J. Klensin, "Simple Mail Transfer Protocol," RFC 5321, Oct. 2008. [Online]. Available: https://rfc-editor.org/rfc/rfc5321.txt

[78] J. Postel and J. Reynolds, "File Transfer Protocol," RFC 5321, Oct. 1985. [Online]. Available: https://rfc-editor.org/rfc/rfc959.txt

[79] P. V. Mockapetris, "Domain names-concepts and facilities," RFC 1034, Nov. 1987. [Online]. Available: https://rfc-editor.org/rfc/rfc1987.txt

[80] J. Postel, "Transmission Control Protocol," RFC 793, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc793.txt

[81] ——, "User Datagram Protocol," RFC 768, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc768.txt

[82] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda, "MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking," in *ACM CoNEXT '10*, Philadelphia, PA, December 2010.

[83] J. Postel, "Internet Protocol," RFC 791, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc791.txt

[84] S. Deering and B. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 8200, Jul. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8200.txt

[85] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.

[86] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang *et al.*, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[87] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks," IEEE Std 802.1Q-2014, Piscataway, NJ, USA, Dec 2014.

[88] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," RFC 7348, Aug. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7348.txt

[89] P. Garg and Y.-S. Wang, "NVGRE: Network Virtualization Using Generic Routing Encapsulation," RFC 7637, Sep. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7637.txt

[90] Y. Rekhter and K. Kompella, "Virtual Private LAN Service (VPLS) Using BGP for Auto-Discovery and Signaling," RFC 4761, Jan. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4761.txt

[91] J. Drake, W. Henderickx, A. Sajassi, R. Aggarwal, D. N. N. Bitar, A. Isaac, and J. Uttaro, "BGP MPLS-Based Ethernet VPN," RFC 7432, Feb. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7432.txt

[92] A. Viswanathan, E. C. Rosen, and R. Callon, "Multiprotocol Label Switching Architecture," RFC 3031, Jan. 2001. [Online]. Available: https://rfc-editor.org/rfc/rfc3031.txt

[93] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM conference on Internet measurement (IMC)*. ACM, Nov. 2010, pp. 267–280.

[94] D. Thaler and C. Hopps, "Multipath issues in unicast and multicast next-hop selection," in *Requests For Comments*. Internet Engineering Task Force, 2000, no. 2991.

[95] D. Allan, P. Ashwood-Smith, N. Bragg, J. Farkas, D. Fedyk, M. Ouellete, M. Seaman, and P. Unbehagen, "Shortest path bridging: Efficient control of larger ethernet networks," *IEEE Communications Magazine*, vol. 48, no. 10, pp. 128–135, 2010.

[96] D. Eastlake 3rd, M. Zhang, R. Perlman, A. Banerjee, A. Ghanwani, and S. Gupta, "Transparent Interconnection of Lots of Links (TRILL): Clarifications, Corrections, and Updates," RFC 8249, Feb. 2016. [Online]. Available: https://rfc-editor.org/rfc/rfc8249.txt

[97] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *Proc. 7th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, vol. 10, 2010, pp. 19–19.

[98] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 503–514.

[99] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 2, pp. 51–62, 2007.

[100] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 266–277, 2011.

[101] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "Ictcp: Incast congestion control for tcp in datacenter networks," *IEEE/ACM transactions on networking*, vol. 21, no. 2, pp. 345–358, 2012.

[102] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010, pp. 63–74.

[103] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[104] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 15–26.

[105] M. J. S. Smith, *Application-specific integrated circuits*. Addison-Wesley Boston, 1997, vol. 7.

[106] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *Proc. ACM SIGCOMM Demos*, vol. 4, no. 5, pp. 1–2, 2009.

[107] R. Wang, D. Butnariu, J. Rexford *et al.*, "Openflow-based server load balancing gone wild," *Hot-ICE*, vol. 11, p. 12, 2011.

[108] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[109] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "Policycop: An autonomic qos policy enforcement framework for software defined networks," in *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*. IEEE, 2013, pp. 1–7.

[110] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[111] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[112] J. Schulist, D. Borkmann, and A. Starovoitov, "Linux socket filtering aka berkeley packet filter (bpf)," *Linux kernel documentation. https://www.kernel.org/doc/Documentation/networking/filter. txt(4.06. 2020)*, 2018.

[113] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[114] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[115] The DPDK Project, "Data Plane Development Kit (DPDK)," https://www.dpdk.org.

[116] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[117] J. Corbet, "BPF: the universal in-kernel virtual machine," https://lwn.net/Articles/599755/, 2014.

[118] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3:{Energy-Efficient} microservices on {SmartNIC-Accelerated} servers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 363–378.

[119] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[120] Alexey N. Kuznetsov, "Traffic Controller," https://linux.die.net/man/8/tc.

[121] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi, "High-speed software data plane via vectorized packet processing," *IEEE Communications Magazine*, vol. 56, no. 12, pp. 97–103, December 2018.

[122] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4 → NetFPGA workflow for line-rate packet processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 1–9.

[123] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[124] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, "The rise of "big data" on cloud computing: Review and open research issues," *Information systems*, vol. 47, pp. 98–115, 2015.

[125] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[126] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[127] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[128] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[129] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4d approach to network control and management," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 5, pp. 41–54, 2005.

[130] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, 2008, pp. 51–62.

[131] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. ACM, 2014, p. 163–174, event-place: Chicago, Illinois, USA. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626313

[132] S. Tata, M. Mohamed, T. Sakairi, N. Mandagere, O. Anya, and H. Ludwig, "rsla: A service level agreement language for cloud services," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 2016, pp. 415–422.

[133] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[134] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *11th International Conference on Autonomic Computing (ICAC 14)*, 2014, pp. 57–64.

[135] Y. Desmouceaux, P. Pfister, J. Tollet, M. Townsley, and T. Clausen, "6lb: Scalable and application-aware load balancing with segment routing," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 819–834, 2018.

[136] E. D. Zwicky, S. Cooper, and D. B. Chapman, *Building Internet Firewalls: Internet and Web Security.* " O'Reilly Media, Inc.", 2000.

[137] T. Chen, R. Bahsoon, and X. Yao, "A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–40, 2018.

[138] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.

[139] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacgümüş, "Smartsla: Cost-sensitive management of virtualized resources for cpu-bound database services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1441–1451, 2015.

[140] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling virtualized applications using machine learning techniques," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012, pp. 3–14.

[141] S. Rezaei and X. Liu, "Deep learning for encrypted traffic classification: An overview," *IEEE communications magazine*, vol. 57, no. 5, pp. 76–81, 2019.

[142] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[143] A. Aghdai, C.-Y. Chu, Y. Xu, D. Dai, J. Xu, and J. Chao, "Spotlight: Scalable transport layer load balancing for data center networks," *IEEE Transactions on Cloud Computing*, 2020.

[144] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. Maguire Jr, P. Papadimitratos, and M. Chiesa, "A high-speed load-balancer design with guaranteed per-connection-consistency," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 667–683.

[145] A. Aghdai, M. I.-C. Wang, Y. Xu, C. H.-P. Wenz, and H. J. Chao, "In-network congestion-aware load balancing at transport layer," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2019, pp. 1–6.

[146] S. Naseer, Y. Saleem, S. Khalid, M. K. Bashir, J. Han, M. M. Iqbal, and K. Han, "Enhanced network anomaly detection based on deep neural networks," *IEEE access*, vol. 6, pp. 48 231–48 246, 2018.

[147] M. Usama, J. Qadir, A. Raza, H. Arif, K.-L. A. Yau, Y. Elkhatib, A. Hussain, and A. Al-Fuqaha, "Unsupervised machine learning for networking: Techniques, applications and research challenges," *IEEE access*, vol. 7, pp. 65 579–65 615, 2019.

[148] A. Finamore, M. Mellia, M. Meo, and D. Rossi, "Kiss: Stochastic packet inspection classifier for udp traffic," *IEEE/ACM Transactions on Networking*, vol. 18, no. 5, pp. 1505–1515, 2010.

[149] T. Auld, A. W. Moore, and S. F. Gull, "Bayesian neural networks for internet traffic classification," *IEEE Transactions on neural networks*, vol. 18, no. 1, pp. 223–239, 2007.

[150] N. Williams, S. Zander, and G. Armitage, "A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 5, pp. 5–16, 2006.

[151] M. C. Belavagi and B. Muniyal, "Performance evaluation of supervised machine learning algorithms for intrusion detection," *Procedia Computer Science*, vol. 89, pp. 117–123, 2016.

[152] Z. Yao, Y. Desmouceaux, M. Townsley, and T. H. Clausen, "Towards intelligent load balancing in data centers," in *5th Workshop on Machine Learning for Systems at 35th Conference on Neural Information Processing Systems (NeurIPS 2021)*, 2021.

[153] M. M. Raikar, S. Meena, M. M. Mulla, N. S. Shetti, and M. Karanandi, "Data traffic classification in software defined networks (sdn) using supervised-learning," *Procedia Computer Science*, vol. 171, pp. 2750–2759, 2020.

[154] G. Nenvani and H. Gupta, "A survey on attack detection on cloud using supervised learning techniques," in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. IEEE, 2016, pp. 1–5.

[155] T. Chen, X. Zhang, M. You, G. Zheng, and S. Lambotharan, "A gnn-based supervised learning framework for resource allocation in wireless iot networks," *IEEE Internet of Things Journal*, vol. 9, no. 3, pp. 1712–1724, 2021.

[156] W. Zhang, Z. Zhang, H.-C. Chao, and M. Guizani, "Toward intelligent network optimization in wireless networking: An auto-learning framework," *IEEE Wireless Communications*, vol. 26, no. 3, pp. 76–82, 2019.

[157] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 10 012–10 022.

[158] B. Zhou, H. Zhao, X. Puig, T. Xiao, S. Fidler, A. Barriuso, and A. Torralba, "Semantic understanding of scenes through the ade20k dataset," *International Journal of Computer Vision*, vol. 127, no. 3, pp. 302–321, 2019.

[159] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[160] A. D. Desai, A. M. Schmidt, E. B. Rubin, C. M. Sandino, M. S. Black, V. Mazzoli, K. J. Stevens, R. Boutin, C. Ré, G. E. Gold *et al.*, "Skm-tea: A dataset for accelerated mri reconstruction with dense image labels for quantitative clinical evaluation," *arXiv preprint arXiv:2203.06823*, 2022.

[161] Y. M. Asano, C. Rupprecht, A. Zisserman, and A. Vedaldi, "Pass: An imagenet replacement for self-supervised pretraining without humans," *arXiv preprint arXiv:2109.13228*, 2021.

[162] J. T. Wu, N. N. Agu, I. Lourentzou, A. Sharma, J. A. Paguio, J. S. Yao, E. C. Dee, W. Mitchell, S. Kashyap, A. Giovannini *et al.*, "Chest imagenome dataset for clinical reasoning," *arXiv preprint arXiv:2108.00316*, 2021.

[163] M. Hemani, A. Patel, T. Shimpi, A. Ramesh, and B. Krishnamurthy, "What ails one-shot image segmentation: A data perspective," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[164] C. Garcin, A. Joly, P. Bonnet, J.-C. Lombardo, A. Affouard, M. Chouet, M. Servajean, J. Salmon, and T. Lorieul, "Pl@ ntnet-300k: a plant image dataset with high label ambiguity and a long-tailed distribution," in *NeurIPS 2021-35th Conference on Neural Information Processing Systems*, 2021.

[165] A. Aakerberg, K. Nasrollahi, and T. B. Moeslund, "Rellisur: A real low-light image super-resolution dataset," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[166] Y. Zhang, D. S. Park, W. Han, J. Qin, A. Gulati, J. Shor, A. Jansen, Y. Xu, Y. Huang, S. Wang *et al.*, "Bigssl: Exploring the frontier of large-scale semi-supervised learning for automatic speech recognition," *IEEE Journal of Selected Topics in Signal Processing*, 2022.

[167] J. Kahn, M. Rivière, W. Zheng, E. Kharitonov, Q. Xu, P. Mazaré, J. Karadayi, V. Liptchinsky, R. Collobert, C. Fuegen, T. Likhomanenko, G. Synnaeve, A. Joulin, A. Mohamed, and E. Dupoux, "Libri-light: A benchmark for asr with limited or no supervision," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 7669–7673.

[168] D. Galvez, G. Diamos, J. Ciro, J. F. Cerón, K. Achorn, A. Gopi, D. Kanter, M. Lam, M. Mazumder, and V. J. Reddi, "The people's speech: A large-scale diverse english speech recognition dataset for commercial usage," *arXiv preprint arXiv:2111.09344*, 2021.

[169] G. Y. Kebe, P. Higgins, P. Jenkins, K. Darvish, R. Sachdeva, R. Barron, J. Winder, D. Engel, E. Raff, F. Ferraro *et al.*, "A spoken language dataset of descriptions for speech-based grounded language learning," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[170] N. Pavlichenko, I. Stelmakh, and D. Ustalov, "Crowdspeech and voxdiy: Benchmark datasets for crowdsourced audio transcription," *arXiv preprint arXiv:2107.01091*, 2021.

[171] S. Evain, M. H. Nguyen, H. Le, M. Z. Boito, S. Mdhaffar, S. Alisamir, Z. Tong, N. Tomashenko, M. Dinarelli, T. Parcollet *et al.*, "Task agnostic and task specific self-supervised learning from speech with lebenchmark," in *Thirty-fifth Conference on Neural Information Processing Systems (NeurIPS 2021)*, 2021.

[172] "Predict," https://www.predict.org/.

[173] "NSL-KDD," http://nsl.cs.unb.ca/NSL-KDD/.

[174] "CAIDA," https://www.caida.org/.

[175] "Internet Traffic Archive," http://ita.ee.lbl.gov/.

[176] K. Shafi and H. A. Abbass, "Evaluation of an adaptive genetic-based signature extraction system for network intrusion detection," *Pattern Analysis and Applications*, vol. 16, no. 4, pp. 549–566, 2013.

[177] G. Creech and J. Hu, "Generation of a new ids test dataset: Time to retire the kdd collection," in *2013 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2013, pp. 4487–4492.

[178] A. McGregor, M. Hall, P. Lorier, and J. Brunskill, "Flow clustering using machine learning techniques," in *International workshop on passive and active network measurement*. Springer, 2004, pp. 205–214.

[179] Z. Yao, Y. Desmouceaux, J.-A. Cordero-Fuertes, M. Townsley, and T. Clausen, "Aquarius-enable fast, scalable, data-driven service management in the cloud," *IEEE Transactions on Network and Service Management*, 2022.

[180] J. Taghia, F. Moradi, H. Larsson, X. Lan, M. Ebrahimi, and A. Johnsson, "Policy-induced unsupervised feature selection: A networking case study," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 750–759.

[181] L. H. A. Reis, L. C. S. Magalhães, D. S. V. de Medeiros, and D. M. Mattos, "An unsupervised approach to infer quality of service for large-scale wireless networking," *Journal of Network and Systems Management*, vol. 28, no. 4, pp. 1228–1247, 2020.

[182] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett *et al.*, "Knowledge-defined networking," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 3, pp. 2–10, 2017.

[183] Z. Yao, Z. Ding, and T. Clausen, "Multi-agent reinforcement learning for network load balancing in data center," in *31st ACM International Conference on Information and Knowledge Management (CIKM'22)*, 2022.

[184] Y. Xu, W. Xu, Z. Wang, J. Lin, and S. Cui, "Load balancing for ultra-dense networks: A deep reinforcement learning based approach," *IEEE Internet of Things Journal*, vol. 6, no. 6, p. 9399–9412, Dec 2019, arXiv: 1906.00767.

[185] Z. Yao and Z. Ding, "Learning distributed and fair policies for network load balancing as markov potentia game," *arXiv e-prints*, pp. arXiv–2206, 2022.

[186] A. Aghdai, M. I.-C. Wang, Y. Xu, C. H.-P. Wenz, and H. J. Chao, "In-network congestion-aware load balancing at transport layer," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2019, pp. 1–6.

[187] J. Zhang, S. Wen, J. Zhang, H. Chai, T. Pan, T. Huang, L. Zhang, Y. Liu, and F. R. Yu, "Fast switch-based load balancer considering application server states," *IEEE/ACM Transactions on Networking*, p. 1–14, 2020.

[188] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 561–575.

[189] M. Ahmed, A. N. Mahmood, and J. Hu, "A survey of network anomaly detection techniques," *Journal of Network and Computer Applications*, vol. 60, pp. 19–31, 2016.

[190] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017, pp. 15–28.

[191] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "Hpcc: high precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 44–58.

[192] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with Beamer," in *Proc. 15th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*. USENIX Association, 2018, pp. 125–139.

[193] J. T. Araújo, L. Saino, L. Buytenhek, and R. Landa, "Balancing on the edge: Transport affinity without network state," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 111–124.

[194] Z. Yao, Y. Desmouceaux, J. A. C. Fuertes, M. Townsley, and T. H. Clausen, "Efficient data-driven network functions," in *30th International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2022)*, 2022.

[195] C. Rizzi, Z. Yao, Y. Desmouceaux, M. Townsley, and T. Clausen, "Charon: Load-aware load-balancing in p4," in *2021 17th International Conference on Network and Service Management (CNSM)*. IEEE, 2021, pp. 91–97.

[196] Z. Yao, Y. Desmouceaux, J.-A. Cordero-Fuertes, M. Townsley, and T. Clausen, "Hlb: Toward load-aware load balancing," *IEEE/ACM Transactions on Networking*, 2022.

[197] M. Ahmed, A. N. Mahmood, and J. Hu, "A survey of network anomaly detection techniques," *Journal of Network and Computer Applications*, vol. 60, pp. 19–31, 2016.

[198] OPNFV, "Open Platform for NFV (OPNFV) Project Portal," https://www.opnfv.org/, 2019.

[199] OpenStack, "OpenStack Project Portal," https://www.openstack.org/, 2019.

[200] Y. Jie, Y. Lun, H. Yang, and L.-y. Chen, "Timely traffic identification on p2p streaming media," *The Journal of China Universities of Posts and Telecommunications*, vol. 19, no. 2, pp. 67–73, 2012.

[201] K. Lalitha and V. Josna, "Traffic verification for network anomaly detection in sensor networks," *Procedia Technology*, vol. 24, pp. 1400–1405, 2016.

[202] R. Cziva and D. P. Pezaros, "Container network functions: Bringing nfv to the network edge," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 24–31, 2017.

[203] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Io-lite: a unified i/o buffering and caching system," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 1, pp. 37–66, 2000.

[204] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in *Proceedings of the 32nd international conference on Very large data bases*, 2006, pp. 607–618.

[205] D. Zhou, Z. Yan, Y. Fu, and Z. Yao, "A survey on network data collection," *Journal of Network and Computer Applications*, vol. 116, pp. 9–23, 2018.

[206] N. Schottelius, "High speed nat64 with p4," Master's thesis, ETH Zurich, 2019.

[207] F. Ruffy, M. Przystupa, and I. Beschastnikh, "Iroko: A framework to prototype reinforcement learning for data center traffic control," *arXiv preprint arXiv:1812.09975*, 2018.

[208] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *International Conference on Machine Learning*. PMLR, 2019, pp. 3050–3059.

[209] M. Hutter, A. Szekely, and J. Wolkerstorfer, "Embedded system management using wbem," in *2009 IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, 2009, pp. 390–397.

[210] J. Yu, H. Lee, M.-S. Kim, and D. Park, "Traffic flooding attack detection with snmp mib using svm," *Computer Communications*, vol. 31, no. 17, pp. 4212–4219, 2008.

[211] P. Gonçalves, J. L. Oliveira, and R. L. Aguiar, "An evaluation of network management protocols," in *2009 IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, 2009, pp. 537–544.

[212] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford, "Clove: Congestion-aware load balancing at the virtual edge," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, 2017, pp. 323–335.

[213] A. Rizzi, A. Iacovazzi, A. Baiocchi, and S. Colabrese, "A low complexity real-time internet traffic flows neuro-fuzzy classifier," *Computer Networks*, vol. 91, pp. 752–771, 2015.

[214] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE communications surveys & tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[215] M. A. Qadeer, A. Iqbal, M. Zahid, and M. R. Siddiqui, "Network traffic analysis and intrusion detection using packet sniffer," in *2010 Second International Conference on Communication Software and Networks*. IEEE, 2010, pp. 313–317.

[216] A. Papadogiannakis, D. Antoniades, M. Polychronakis, and E. P. Markatos, "Improving the performance of passive network monitoring applications using locality buffering," in *2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2007, pp. 151–157.

[217] B. Pit-Claudel, Y. Desmouceaux, P. Pfister, M. Townsley, and T. Clausen, "Stateless load-aware load balancing in p4," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, Sep 2018, p. 418–423.

[218] J. Papamichael, T. M. M. Liu, D. Haselman, L. A. M. Ghandi, S. Sapek, and G. W. L. Woods, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture, ser. ISCA*, vol. 18, 2018.

[219] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," *arXiv preprint arXiv:1903.06701*, 2019.

[220] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

[221] R. Cohen, M. Kadosh, A. Lo, and Q. Sayah, "Lb scalability: Achieving the right balance between being stateful and stateless," *arXiv:2010.13385 [cs]*, Oct 2020.

[222] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[223] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li, "In-band network telemetry: A survey," *Computer Networks*, vol. 186, p. 107763, 2021.

[224] Facebook Engineering, "Reinventing Facebook's data center network," https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/, Mar 2019.

[225] E.-J. van Baaren, "Wikibench: A distributed, Wikipedia based web application benchmark," Master's thesis, VU University Amsterdam, 2009.

[226] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, vol. 53, no. 11, pp. 1830–1845, Jul. 2009.

[227] "The Apache HTTP server project." [Online]. Available: http://www.apache.org

[228] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[229] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 139–152.

[230] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," Stanford, Tech. Rep., 2006.

[231] D. A. Reynolds, "Gaussian mixture models." *Encyclopedia of biometrics*, vol. 741, no. 659-663, 2009.

[232] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: why and how you should (still) use dbscan," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–21, 2017.

[233] E. Schubert and M. Gertz, "Improving the cluster structure extracted from optics plots," in *LWDA*, 2018.

[234] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," https://www.tensorflow.org/, 2015, software available from tensorflow.org.

[235] "Katran," https://github.com/facebookincubator/katran, Apr 2020.

[236] W. Zhang *et al.*, "Linux virtual server for scalable network services," in *Ottawa Linux Symposium*, vol. 2000, 2000.

[237] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[238] P. Patel, A. H. Ranabahu, and A. P. Sheth, "Service level agreement in cloud computing," Cloud Workshops at OOPSLA09, [Online] Available: https://corescholar.libraries.wright.edu/knoesis/78, 2009.

[239] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *ACM Sigcomm computer communication review*, vol. 39, no. 1, pp. 50–55, 2008.

[240] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 51–66.

[241] S. Shi, Y. Yu, M. Xie, X. Li, X. Li, Y. Zhang, and C. Qian, "Concury: A fast and light-weight software cloud load balancer," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 179–192.

[242] V. Olteanu and C. Raiciu, "Datacenter scale load balancing for multipath transport," in *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, ser. HotMIddlebox '16. ACM, 2016, p. 20–25, event-place: Florianopolis, Brazil. [Online]. Available: http://doi.acm.org/2940147.2940154

[243] Theo Julienne, "GLB: GitHub's open source load balancer," 2018. [Online]. Available: https://github.blog/2018-08-08-glb-director-open-source-load-balancer/

[244] D. Borman, R. T. Braden, V. Jacobson, and R. Scheffenegger, "TCP Extensions for High Performance," RFC 7323, Sep. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7323.txt

[245] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing Architecture," RFC 8402, Jul. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8402.txt

[246] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[247] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The p4-netfpga workflow for line-rate packet processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–9. [Online]. Available: https://doi.org/10.1145/3289602.3293924

[248] A. J. Walker, "An efficient method for generating discrete random variables with general distributions," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 253–256, 1977.

[249] T. Li, D. Farinacci, S. P. Hanks, D. Meyer, and P. S. Traina, "Generic Routing Encapsulation (GRE)," RFC 2784, Mar. 2000. [Online]. Available: https://rfc-editor.org/rfc/rfc2784.txt

[250] "Vivado." [Online]. Available: https://www.xilinx.com/support/university/vivado.html

[251] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend tcp?" in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference - IMC '11*. ACM Press, 2011, p. 181. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2068816.2068834

[252] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Hudson, MA: Eastern Research Laboratory, Digital Equipment Corporation, 1984, vol. 38.

[253] M. Hawari, "System and networking aspects of the transition of high-performance applications from dedicated to commodity hardware: the example of media production for professional broadcast," Ph.D. dissertation, Institut polytechnique de Paris, 2021.

[254] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben, "On the diversity of cluster workloads and its impact on research results," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 533–546.

[255] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, 2019.

[256] R. S. Sutton, A. G. Barto *et al.*, *Introduction to reinforcement learning*. MIT press Cambridge, 1998, vol. 135.

[257] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[258] X. Hong, H. Wang, Y. Zhang, K. Narayanaswamy, R. Nair, and H. Han, "Server load balancing using a fair weighted hashing technique," Jun. 24 2014, uS Patent 8,762,534.

[259] Y. Bi, G. Han, C. Lin, Y. Peng, H. Pu, and Y. Jia, "Intelligent qos-aware traffic forwarding for sdn/ospf hybrid industrial internet," *IEEE Transactions on Industrial Informatics*, p. 1–1, 2019.

[260] H. Jamal and K. Sultan, "Performance analysis of tcp congestion control algorithms," *International journal of computers and communications*, vol. 2, no. 1, pp. 18–24, 2008.

[261] G. Goren, S. Vargaftik, and Y. Moses, "Distributed dispatching in the parallel server model," *arXiv:2008.00793 [cs]*, Aug 2020, arXiv: 2008.00793.

[262] W. Wang and G. Casale, "Evaluating weighted round robin load balancing for cloud web services," in *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 2014, pp. 393–400.

[263] R. Gandhi, Y. C. Hu, and M. Zhang, "Yoda: A highly available layer-7 load balancer," in *Proc. 11th European Conference on Computer Systems*. ACM, 2016, article no. 21.

[264] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, "Wcmp: Weighted cost multipathing for improved fairness in data centers," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 5.

[265] M. Shafiee and J. Ghaderi, "A simple congestion-aware algorithm for load balancing in datacenter networks," *IEEE/ACM Transactions on Networking (TON)*, vol. 25, no. 6, pp. 3670–3682, 2017.

[266] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," *SIAM journal on computing*, vol. 29, no. 1, pp. 180–200, 1999.

[267] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, "The Segment Routing architecture," in *Proc. IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–6.

[268] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.

[269] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 183–196.

[270] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang, "Quality-of-service in cloud computing: modeling techniques and their applications," *Journal of Internet Services and Applications*, vol. 5, no. 1, pp. 1–17, 2014.

[271] A. Hadoop, "Apache hadoop," *URL http://hadoop. apache. org*, 2011.

[272] A. Spark, "Apache spark," *Retrieved January*, vol. 17, p. 2018, 2018.

[273] X. Chen, H. Kim, J. M. Aman, W. Chang, M. Lee, and J. Rexford, "Measuring tcp round-trip time in the data plane," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020, pp. 35–41.

[274] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.

[275] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing sla violations," in *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*. IEEE, 2007, pp. 119–128.

[276] M. Andreolini, S. Casolari, M. Colajanni, and M. Messori, "Dynamic load management of virtual machines in cloud architectures," in *International Conference on Cloud Computing*. Springer, 2009, pp. 201–214.

[277] Y. Yang and J. Wang, "An overview of multi-agent reinforcement learning from game theoretical perspective," *arXiv preprint arXiv:2011.00583*, 2020.

[278] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls *et al.*, "Value-decomposition networks for cooperative multi-agent learning," *arXiv preprint arXiv:1706.05296*, 2017.

[279] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson, "Counterfactual multi-agent policy gradients," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[280] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. Pieter Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *Advances in neural information processing systems*, vol. 30, 2017.

[281] T. Rashid, M. Samvelyan, C. Schroeder, G. Farquhar, J. Foerster, and S. Whiteson, "Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4295–4304.

[282] D. Monderer and L. S. Shapley, "Potential games," *Games and economic behavior*, vol. 14, no. 1, pp. 124–143, 1996.

[283] W. H. Sandholm, "Potential games with continuous player sets," *Journal of Economic theory*, vol. 97, no. 1, pp. 81–108, 2001.

[284] J. R. Marden, G. Arslan, and J. S. Shamma, "Cooperative control and potential games," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 39, no. 6, pp. 1393–1407, 2009.

[285] O. Candogan, I. Menache, A. Ozdaglar, and P. A. Parrilo, "Flows and decompositions of games: Harmonic and potential games," *Mathematics of Operations Research*, vol. 36, no. 3, pp. 474–503, 2011.

[286] D. Fudenberg, F. Drew, D. K. Levine, and D. K. Levine, *The theory of learning in games*. MIT press, 1998, vol. 2.

[287] R. Fox, S. M. Mcaleer, W. Overman, and I. Panageas, "Independent natural policy gradient always converges in markov potential games," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2022, pp. 4414–4425.

[288] S. Leonardos, W. Overman, I. Panageas, and G. Piliouras, "Global convergence of multi-agent policy gradient in markov potential games," *arXiv preprint arXiv:2106.01969*, 2021.

[289] S. V. Macua, J. Zazo, and S. Zazo, "Learning parametric closed-loop policies for markov potential games," *arXiv preprint arXiv:1802.00899*, 2018.

[290] D. H. Mguni, Y. Wu, Y. Du, Y. Yang, Z. Wang, M. Li, Y. Wen, J. Jennings, and J. Wang, "Learning in nonzero-sum stochastic games with potentials," in *International Conference on Machine Learning*. PMLR, 2021, pp. 7688–7699.

[291] H. Li and H. He, "Multi-agent trust region policy optimization," *arXiv preprint arXiv:2010.07916*, 2020.

[292] C. S. de Witt, T. Gupta, D. Makoviichuk, V. Makoviychuk, P. H. Torr, M. Sun, and S. Whiteson, "Is independent learning all you need in the starcraft multi-agent challenge?" *arXiv preprint arXiv:2011.09533*, 2020.

[293] Z. Ding, T. Huang, and Z. Lu, "Learning individually inferred communication for multi-agent cooperation," *Advances in Neural Information Processing Systems*, vol. 33, pp. 22 069–22 079, 2020.

[294] S. Q. Zhang, Q. Zhang, and J. Lin, "Efficient communication in multi-agent reinforcement learning via variance based control," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[295] J. Zhang, F. R. Yu, S. Wang, T. Huang, Z. Liu, and Y. Liu, "Load balancing in data center networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2324–2352, 2018.

[296] S. Sen, D. Shue, S. Ihm, and M. 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*, 2013, pp. 151–162.

[297] M. S. Ali, P. Coucheney, and M. Coupechoux, "Reinforcement learning algorithm for load balancing in self-organizing networks," 2019.

[298] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.

[299] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[300] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[301] "HAProxy: the reliable, high-performance TCP/HTTP load balancer." [Online]. Available: http://www.haproxy.org

ECOLE
DOCTORALE

Titre : La Gestion Autonome des Services dans le Cloud

Mots clés : Réseau de Centre de Données, Fonctions Réseau, Équilibrage de Charge, Algorithmes d'Apprentissage

Résumé : Les applications et services sont devenus plus complexes, rendant la configuration et la gestion des politiques de mise en réseau plus difficiles pour des performances optimisées. Les paradigmes de réseau programmable, tels que le réseau définie par logiciel (SDN) et la virtualisation des fonctions réseau (NFV), ont émergé pour faciliter l'évolution du réseau et permettre des services riches de traitement du trafic réseau. Cette thèse explore comment offrir des fonctions de mise en réseau génériques basées sur les données dans les réseaux de centres de données pour construire des systèmes autonomes qui optimisent les performances de mise en réseau avec un minimum d'intervention humaine et complexité opérationnelle. Elle étudie également comment les fonctions et primitives du réseau peuvent être améliorées par des algorithmes basés sur les données, tout en gardant à l'esprit les exigences de production des réseaux de centres de données. Pour relever les défis de la collecte de mesures et du déploiement de politiques de mise en réseau basées sur les données, un outil générique est construit pour extraire les fonctionnalités de mise en réseau du plan de données et déployer des algorithmes basées sur l'apprentissage automatique (ML) pour diverses fonctions de mise en réseau dans des systèmes de mise en réseau du monde réel. Cette thèse se concentre sur les problèmes d'équilibrage de charge réseau dans les réseaux de centres de données, sur lesquels un état de l'art des équilibreurs de charge est fourni. Des algorithmes d'équilibrage de charge d'apprentissage sont proposés sur la base d'algorithmes d'apprentissage, montrant de meilleures performances que les méthodes d'équilibrage de charge de pointe.

Title : Autonomous Service Management in the Cloud

Keywords : Data-Center Networking, Data-Driven, Network Functions, Load Balancing, Learning Algorithms

Abstract : Applications and services have become more complex, while the Internet has become increasingly difficult to evolve both regarding its physical infrastructure, and its protocols and performance. Being responsible for policy configurations as well as network management and performance tuning, network operators are shifting towards the use of more and more automated tools to accomplish these tasks. The concept of "programmable networks" has emerged to alleviate the challenges, and to facilitate network evolution. This includes paradigms such as (i) software-defined networking (SDN) and (ii) network function virtualization (NFV), which decouple the forwarding hardware into the control plane and data plane, and seek to abstract network forwarding, and other networking functions, from the hardware. In the era of "big data" on cloud computing, these paradigms have enabled rich network traffic processing services, while having also reduced the granularity of task allocation in data centers. It has been recognized that shifting controllers from logically centralized to distributed will increase not only scalability but also robustness to inconsistency. Machine-Learning (ML)-based approaches have been proposed to deploy more intelligence in networks, when using decoupled control and data planes. In this context, the question explored in this thesis is whether, and how, it is possible to offer generic, data-driven networking functions in data center networks as services, for constructing autonomous networking systems which optimize networking performances with minimal human intervention and operational complexity. This thesis investigates the increasing scale, complexity, and heterogeneity of networking infrastructure, and protocols, as well as the demand for virtualization and cloud support services in terms of efficient resource management, rapid provisioning, and scalability present a set of new challenges in effective network organization, management, and optimization. This is accomplished by studying how certain network functions and primitives (traffic classification, auto-scaling, load balancing) can be reliably enhanced by various data-driven algorithms, while bearing in mind the in-production requirements in data center networks – high scalability, high throughput, low latency, and low overheads.

Institut Polytechnique de Paris
91120 Palaiseau, France