Doctoral Dissertation
Doctoral Program in Electrical, Electronics and Communications Engineering
(XXXIII cycle)

Traffic Optimization in Data Center and Software-Defined Programmable Networks

German Sviridov

* * * * * *

Supervisors
Prof. Paolo Giaccone, Supervisor
Prof. Andrea Bianco Co-supervisor

Politecnico di Torino
2020

. . . .

German Sviridov

Turin, 2020

Abstract

In the past decade, the field of telecommunications went through a radical change in the way users interact with the network and the way networks evolved in the function of user needs. Data Centers have become the backbone of modern digital society and industry. This led to their rapid spread with the goal of providing fast and readily-available services to customers around the globe. Consequently, wide-area networks have grown exponentially more complex to accommodate the surge in bandwidth demand and new paradigms such as Software Defined Networking emerged to combat the increase in complexity. Nevertheless, even nowadays, most of the wide-area networks still rely on legacy network management algorithms and protocols which significantly limit the pace at which current network infrastructures are able to adapt to modern traffic scenarios. Similarly to what happens in the wide-area networks, data center networks still employ classic flow management mechanisms which, although being able to guarantee a good level of performance, do not fully exploit the potential of modern network architectures.

In this Thesis a major focus is devoted towards analyzing the impact of the radical change introduced by modern network infrastructures on the traffic flow performance. As a first contribution, we show that programmable data planes, although being able to overcome some of the shortcomings of traditional Software Define Networking, still fall short for many applications relevant to the operations of wide-area networks. This in turns contributes to the deterioration of traffic flow performance, for a wide range of applications. We address this issue by proposing a novel paradigm of designing programmable data planes-ready network applications that exploit replicated states inside the network, ultimately permitting to improve traffic flow performance.

A second contribution relates to the performance optimization of data center networks. Traditional flow scheduling mechanisms employed in data center networks are unable to keep up with the ever-increasing demand for highly responsive applications deployed in modern data centers. At the same time, solutions proposed in the literature rely on complex control mechanisms. Those solutions are capable of significantly improving flow scheduling policies, thus traffic flow performance. Yet, even by relaxing some of the requirements of those solutions, they still remain too complex and require complex modifications to the hardware of underlying devices composing the network. This ultimately results in prohibitively expensive solutions, thus inapplicable in realistic scenarios. We propose a flow scheduling mechanism based on aggregate flow statistics that is capable of achieving traffic flow performance close to state-of-the-art solutions while keeping the complexity low, thus making it accessible for the already available network infrastructures.

While our contributions in the field of programmable data planes and data center flow scheduling are capable of considerably improving traffic flow performance we show that blindly optimizing aggregate traffic flow performance does not lead to optimal results in terms of user experience. Indeed, observing aggregate flow information does not give any insight on the nature of the service carried in single flows, thus precluding any possibility of understanding the impact of the network parameters on those flows. Our final contribution addresses this issue by proposing an efficient way of assessing the impact of the latency on interactive applications which dominate the modern Internet. We consider cloud gaming as a reference application and highlight the heterogeneity in network requirements for apparently similar flows. This is done by developing an automatic Quality of Experience assessment procedure which exploits modern advances in the field of artificial intelligence and deep reinforcement learning. Finally, we show that the proposed methodology can be employed to define and enforce fine-grained flow optimization policies capable of taking into account service-level network requirements.

Acknowledgements

First of all, I would like to thank my incredible advisors Prof. Paolo Giaccone and Prof. Andrea Bianco who had the dedication and patience of guiding me through this long journey. I am very grateful to Andrea Bianco for transmitting to me his invaluable research expertise which taught me how to approach new challenges and to always question everything. I am also very thankful to Paolo Giaccone for his incredible dedication to what he does and his incredible knowledge. Never in my life I have found a person who cares about his students and their future as much as Paolo does. Not only he contributed incredibly to my professional growth, but he also contributed to shaping the way I am today which will resonate within me for the rest of my life.

A special thank goes to Dario Rossi and the amazing people of the Networks AI team of Huawei Research Center of Paris who hosted me for my internship. They made those six cold and rainy months fly like an arrow. While they helped me in developing my professional skills at the same time, I've never felt so welcomed and for this, I will forever cherish the moments spent together.

I would like to express the greatest appreciation to Irene Barbieri and to my family who were always there when I needed them most, for supporting me through my PhD and for contributing to who I became during this long journey. A special token of appreciation goes to Riccardo Peloso for the cheerful moments spent together and to Giovanni Conte for being an amazing friend you could always count on.

I would like to thank Abubakar Siddique Muqaddas and Giuseppe Bianchi's wonderful research group of Roma Tor Vergata for their collaboration. Finally, special gratitude goes to the amazing people of the TNG Group of Politecnico di Torino for creating a warm and welcoming research environment and their invaluable feedback throughout my studies.

Contents

Chapter 1

Introduction

In the past decade, the field of telecommunications went through a radical change in the way users interact with the network and the way networks evolved based on users' needs. Data Centers (DCs) became the backbone of the modern digital economy which led to their rapid spread throughout the globe. Consequently, wide-area networks grew exponentially more complex to accommodate the surge in bandwidth demand. Nevertheless, even nowadays, most of the wide-area networks still rely on legacy network management algorithms and protocols which significantly limit the pace at which current network infrastructures are able to adapt to modern traffic scenarios. Similarly to what happens in wide-area networks, data center networks still employ classic flow management mechanisms which, although being able to guarantee a good level of performance, do not fully exploit the potential of modern Data Center Network (DCN) architectures.

In this chapter, we will address the aforementioned issues related to the management of wide-area and data center networks. First we will analyze how Software Defined Networking (SDN) was introduced to solve the issue of the ever-growing management complexity of modern network infrastructures. Later, instead, we will discuss the architecture of modern data centers and the challenges which arise in optimizing the performance of intra-data center flow transmission. Finally, we will present the structure of the Thesis, outlining the addressed challenges and the contribution to the state-of-the-art.

1.1 Software Defined Networking

Users have an ever-increasing demand in *fast and reliable* service which forces network operators to implement more complex and intricate services inside their infrastructures. At the same, legacy network management mechanisms are quickly becoming unsustainable due to their lack of flexibility and difficulty of adapting to new, more dynamic scenarios. Such was the case of Facebook's video autoplay feature, which in 2014 in the US led to a surge in the total mobile data traffic by 60% overnight [1] which had a dramatic impact on the network infrastructure in different parts of the world. On the other hand, instances of network misconfiguration had led numerous times to major portions of local Internet traffic being rerouted through an ISP located on the opposite part of the world, possibly exposing sensitive data and causing service disruption for millions of customers.

Episodes like these spread light on the necessity of implementing novel network management mechanisms with simplicity and fast adaptability to new scenarios at their core. Such was the motivation behind the introduction of SDN which caused a radical change in the way modern networks are managed.

SDN sparked considerable interest both in the industry and in the research community which was enough to promptly highlight not only the potential benefits, but also the inevitable drawbacks of employing such an approach. Among the main drawbacks with SDN was the issue related to the reactivity and increased control plane latency.

SDN introduced a separation of, traditionally tightly coupled, control and data planes, as depicted in Figure 1.1. While in classic networks the control plane lived alongside to the data plane on each switch, in SDN the control plane is moved to an external central entity, namely the *controller*. This translates in SDN switches being devoid of any decision-making capability, leaving bare metal without any network-related logic within. The role of the controller is to provide decision-making capabilities to each switch by interacting with them over a dedicated control interface, i.e., the *southbound interface*. This interaction is performed each time a, previously unseen, network event occurs. To deal with different events and to provide custom network configuration the SDN controller interacts with a set of *network applications*. These applications represent an entry point for the network administrator to the network infrastructure. Indeed,

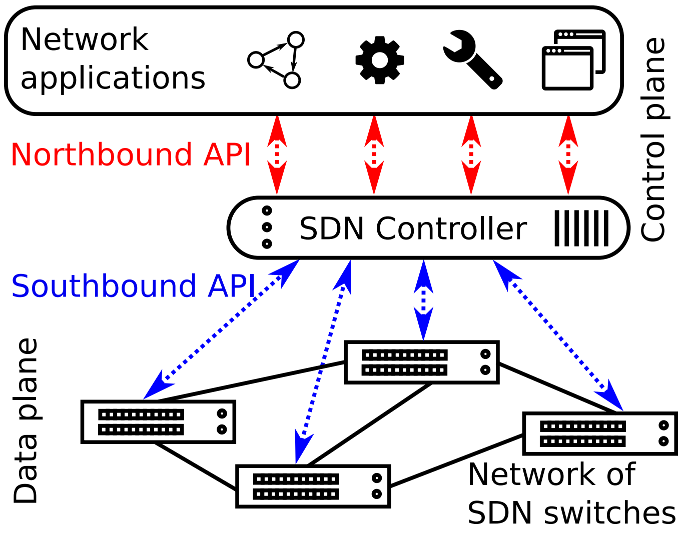


Figure 1.1: High-level overview of an SDN architecture.

they allow the programmer to easily describe high-level network behavioral policies, such as routing schemes or energy management policies, which are then translated into device-specific instructions by the controller through the southbound APIs.

SDN has greatly reduced the complexity of modern network infrastructures by bringing flexibility and ease of management. Nevertheless, such improvements came at a cost of increased control plane latency and communication overhead [2] between switches and the controller, which consequently heavily impacted the reactiveness of the whole system to critical networking events.

As SDN led to a more simplified and unified network management scheme, the reactivity of SDN to critical network events became constrained to the communication latency between switches and the controller, and the latency incurred by the processing and reaction routines at the controller. The concept of programmable data plane [3] was created to fill the newly opened gap in network performance. Programmable data planes performed a step backward from fully centralized control planes by offering the possibility of executing custom code directly inside the packet processing pipeline of the switch. This lightened the switches' dependency on the controller by providing a limited amount of decision-making capabilities directly inside switches.

While its main objective was that of removing the communication delay with the controller for latency-sensitive in-switch decisions, the introduction of programmable data planes opened a wide range of possibilities for the development of novel network applications. Stateful switches offered the opportunity of *embedding some of the services which previously required dedicated devices, directly inside the switches*, thus flattening the overall network architecture. At the same time, they enabled the possibility of defining completely new network applications that previously were unattainable without expensive custom hardware devices. While those applications permitted to dramatically increase the network performance, it also led to a new set of challenges related to the scalability of such an approach.

1.2 Data center networks

DCs took the world by storm with the explosion of data-based services, cloud solutions for businesses and entertainment systems for the general public.

From the point of view of the network interconnection, DCNs present very ordered tree-like structures, as shown in Figure 1.2. Most of the DCNs exploit multi-level hierarchical design with servers being at the bottom of the hierarchy. Such design is what permits DCs to easily scale to the enormous amount of servers (millions of them in the case of large data centers). At the same time, such design permits to achieve large bisection bandwidth which enables the possibility of obtaining high data rates among any pair of servers.

The main peculiarity of DCNs is that, differently from conventional Internet networks, the DC operators are in full control of every element of the network. This means that fine-grained optimization can be easily done at any layer of the architecture, i.e., both at the software and hardware of servers and switches composing the network. Such kind of unprecedented fine-grained control opens a wide range of opportunities for designing and implementing network algorithms, aiming at improving the performance of specific DC architectures. The average amount of time required to transmit a flow from one part of a DC to another one is among the most critical types of performance indicators of a DCN. The latter, usually referred to as *flow completion time*, is usually used as the main performance metric since it is very representative of what an average user desires, i.e., a fast response after a request for given data has been made. Furthermore, flow completion time summarizes information about many aspects of the

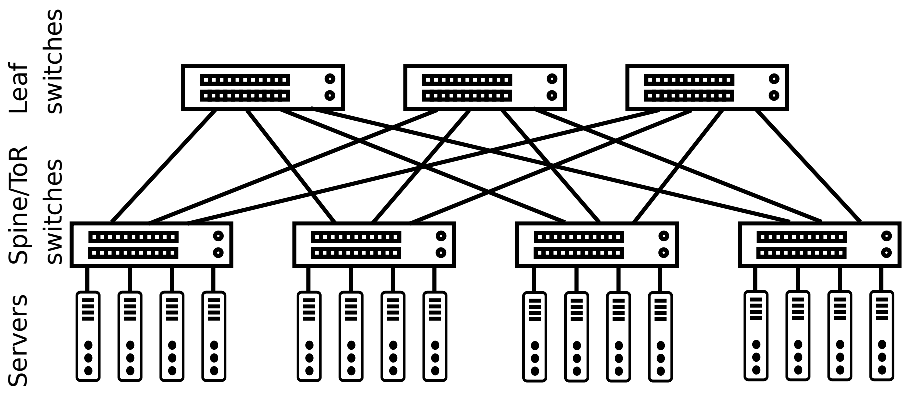


Figure 1.2: High-level overview of an SDN architecture.

DC such as the efficiency of the load balancing, congestion control, and scheduling algorithms employed in the system. Among these algorithms, flow scheduling has a major impact on the overall DCN performance.

Modern applications distinguish themselves by an ever-increasing requirement for higher interactiveness among participants. This trend is manifested by a broad spread of realtime services such as video conferences and online gaming which became the dominant type of applications in DCs. Nevertheless, less interactive applications such as video streaming and bulk data transfer, although not being the dominant application type, remain the main contributors to the overall network usage. From the network point of view, the coexistence of those two types of applications pose numerous challenges.

Whenever a short, highly delay-sensitive flow contends the same link with a long, delay-insensitive flow, the performance of the short flow suffers considerably in terms of flow completion time. Indeed, the short flow is forced to share its bandwidth for its entire lifetime with another flow leading to a potentially double flow completion time. At the same time, the long flow is not penalized significantly since the time of coexistence with the short flow represents a small percentage of its overall lifetime. The role of the flow scheduler is to decide the order in which flows must be served inside the network. In such a trivial example it is evident that an ideal flow scheduler would prioritize the short flow over the long one. Nevertheless, the optimal way of performing flow scheduling in realistic scenarios and constrained to the real capabilities

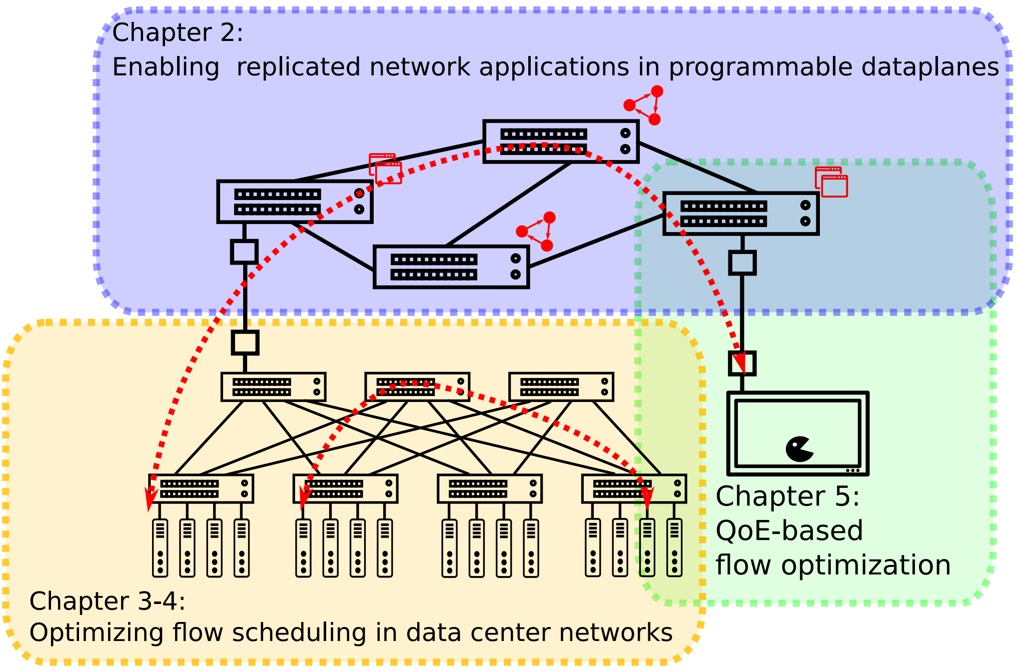

Chapter 2:
Enabling replicated network applications in programmable dataplanes
Chapter 5:
QoE-based
flow optimization
Chapter 3-4:
Optimizing flow scheduling in data center networks

Figure 1.3: Roadmap of the Thesis

of DC switches is still an open research problem.

1.3 Structure of the Thesis

In this Thesis we focus on new research questions opened by the introduction of programmable data planes (Chapter 2), the wide spread of data center networks (Chapter 3-4) and the raise of data-driven algorithms (Chapter 5). Figure 1.3 shows an overview of the main topics covered in this Thesis which combined provide an end-to-end traffic flow performance optimization scheme on both the network and the application layers. This is achieved by analyzing how programmable data planes can improve the performance of flows in the wide-area networks, moving to flow scheduling in data center networks and finally descending into fine-grained optimization of traffic flow performance based on the user-perceived Quality of Experience (QoE).

While programmable data planes have been widely used to embed, previously controller-specific applications, directly inside the data plane, traditional approaches of embedding network applications still rely on having a single instance of a given network application for the entirety of the network. This inevitably leads to issues related to scalability and resilience to faults. As an example, following such an approach, a network application designated to monitor all of the incoming traffic in the network will require all of the traffic to traverse a single switch. This will inevitably lead to traffic flow performance degradation, major congestion, and in the worst case, total service disruption. Although being a simple example, it spreads lights on how traditional embedding approaches are not suitable for the majority of network applications operating on *global network statistics*.

To solve this issue, as a first contribution, we devise a state replication algorithm which is capable of replicating network application-related persistent states among different switches without relying on the controller. This enables replicated instances of the same network application in different parts of the network, ultimately leading to increased scalability and better traffic flow performance.

While designing a suitable scheme for replicating states, we provide the programmer an abstraction model enabling him to easily develop network applications operating on replicated states. We provide an end-to-end solution, namely LOADER, starting from *the programming abstraction*, going through the *compilation phase* and finally choosing the *optimal amount of replicas per each state* and their *optimal embedding in the network*.

Through extensive analysis, we show that LOADER is capable of improving the traffic flow performance thanks to the presence of distributed network applications orchestrated by a centralized manager. While LOADER can greatly boost the traffic flow performance in wide-area networks, centralized approaches such as SDN have been widely used also in the context of DCNs for what concerns network management, configuration and monitoring. At the same time, most of the data flows nowadays originate from a Data Center (DC) as pictured in Figure 1.3. Thus, optimizing the traffic flow performance inside DCNs becomes of paramount importance. Yet, while in wide-area networks to improve the traffic flow performance the main objective is that of minimizing the network congestion, in DCNs minimizing congestion alone is not sufficient. Load balancing, optimized transport protocols, and, most importantly, the use of a refined flow scheduling algorithm play a fundamental role in the DC environment. This leads to the inapplicability of the general congestion minimization approach proposed by LOADER in such a peculiar context. While there exist a plethora of distributed versions of the former schemes, the usage of centralized approaches for fine-grained operations such as flow scheduling in DCNs has not been extensively explored. As flow scheduling has the most influence on traffic flow performance in DCNs, taking inspiration from LOADER we analyze whether centralized approaches for flow scheduling are a viable solution for improving the network performance in DCNs. Specifically, we compare two different approaches which, although both exploiting a central controller to schedule flows, significantly differ in complexity. Specifically, we compare synchronous and asynchronous centralized flow scheduling algorithms. We show that both of them can be implemented in a realistic scenario and finally we highlight the performance and practicality of each of them through an extensive set of simulations.

The main outcome of our investigation of centralized flow scheduling algorithms is that they are mostly unpractical and sometimes even unfeasible in realistic scenarios. Due to the scale of modern DCs and very tight latency requirements distributed approaches are currently the only feasible solution for flow scheduling. While there exist multiple variations of different distributed flow scheduling algorithms, most of them either lead to unsatisfactory performance or require expensive hardware which is not readily available on currently employed commercial switches. For this reason, we devise a simple, yet efficient flow scheduling mechanism, namely NOS2, for DCNs which offers a high degree of practicality. We design NOS2 by keeping in mind the restraints of realistic DCN architectures and by considering the presence of off-the-shelf components, thus not requiring any expensive modification to the already present infrastructure. The proposed mechanism exploits in-network flow prioritization algorithm with a central controller responsible of configuring the parameters of the prioritization algorithm. NOS2 is capable of achieving close to state-of-art traffic flow performance while keeping the complexity as low as possible as shown by our extensive set of simulations.

As previously mentioned, most of the flows nowadays originate from DCs and terminate at end-users. In the previous Chapters we aimed at the optimization of flow-level performance by either trying to minimize the congestion in the network or by trying to reduce the average flow completion time. Although being a suitable approach for most of the applications on the Internet, such an approach is not able to take into account fine-grained requirements in terms of bandwidth and latency of user-centric applications. Such is the case of different interactive applications ranging from voice/video calls or remote control to online gaming. The main issue in guaranteeing a particular level of network performance for such applications is that typically the required level of performance is unknown. For this reason, such applications are usually grouped into a macro-category of latency-sensitive applications. This inevitably leads to all of them being offered the same level of performance, without considering the actual level of satisfaction of the user. Knowing the response of the user to different network conditions for a particular service is notoriously difficult to achieve as it requires a complex and expensive QoE assessment phase.

The final contribution of this Thesis addresses the issue with the difficulty of performing QoE assessment by proposing a framework for a fast and efficient QoE assessment in the context of *cloud gaming* using Deep Reinforcement Learning (DRL). We choose the scenario of cloud gaming as it offers a vast set of different games with different sensitiveness to latency, thus providing a way of emulating different interactive services. Following our analysis, we exploit the proposed framework to devise a data-driven QoE-aware flow scheduler which, contrarily to what has been done before, is capable of optimizing the performance of each interactive service based on the user-perceived QoE.

Chapter 2

LOADER: Local Decisions on Replicated States in Programmable data planes

Part of the work presented in this chapter has been published in:

- Sviridov G, Bonola M, Tulumello A, Giaccone P, Bianco A, Bianchi G. ”LODGE: LOcal decisions on global statEs in programmable data planes.” In *4th IEEE Conference on Network Softwarization and Workshops (NetSoft).* 2018.
- Muqaddas AS, Sviridov G, Giaccone P, Bianco A. ”Optimal state replication in stateful data planes.” In *IEEE Journal on Selected Areas in Communications.* 2020.
- Sviridov G, Bonola M, Tulumello A, Giaccone P, Bianco A, Bianchi G. ”LOcAl DEcisions on Replicated States (LOADER) in programmable data planes: programming abstraction and experimental evaluation.” In *Computer Networks.* 2020.

Future wide-area networks are called to efficiently and flexibly support an ever-growing variety of heterogeneous network applications such as network address translation, tunneling, load balancing, traffic engineering, monitoring, intrusion detection, and so on. Software-based programmability of such type of applications has been first pioneered by early Software Defined Networking (SDN) proposals, and then by the more recent trend of Network Function Virtualization (NFV). However, both these approaches have shown shortcomings. Indeed, original SDN approaches (and, more specifically, the OpenFlow-based ones), were relying on stateless switching architectures, and thus suffered from the need to centralize *any* state update and maintenance to a centralized controller, thus paying a significant toll in terms of latency and communication overhead. On the other side, NFV has addressed the design of middlebox functionalities in software, typically using commodity CPUs. However, early NFV implementations appeared to be performance-limited: it is a fact that there exists a substantial gap (a 50× factor [4]) between the speed attainable in software opposed to dedicated HW devices, and such gap is not going to decrease in the future, with HW switches capable to attain many Terabit per seconds, opposed to the tens of Gigabit per second attainable by their SW counterparts.

In order to overcome such limitations, starting from 2014 with OpenState [5] and P4 [6], a new innovation trend emerged with the introduction of *programmable / stateful* data planes. Stateful data planes offer an additional level of programmability with respect to the traditional stateless SDN paradigm, by introducing the possibility of keeping and manipulating persistent states locally at the network device. Opposed to stateless switches, persistent states can now be directly deployed and managed inside network devices in the form of simple user-defined memory elements. Furthermore, arbitrary algorithms for packet/flow processing, e.g., described in terms of simple Mealy Finite State Machines [5] or more sophisticated Extended Finite State Machines [7], [8], can be directly loaded and run inside the processing pipeline of individual network devices, thus providing opportunities of implementing network applications directly within the network device at line rate.

The crucial advantage of stateful data plane technologies consists in the possibility to significantly reduce the interaction between switches and the controller. Opposed to a stateless data plane, in which any change of the forwarding decision requires the intervention of the controller, a stateful data plane permits to take *localized* decisions, i.e., adapt the forwarding behavior to network events and handle changing states locally inside the switch. This approach significantly reduces the reliance upon a centralized controller, and mitigates the relevant severe penalties in terms of latency and signaling overhead [2], hence greatly improving the reactivity of the network control applications.

Unfortunately, the benefits of distributing network applications on stateful switches cannot be achieved in cases where non-local states need to be considered. For example, an application that identifies the occurrence of a particular event based on multiple statistics gathered from different switches, operates on a global state that is the combination of different local statistics of different switches. Even in the case of stateful data planes, the control and update of the global state is still delegated to a centralized entity, either to a controller or a single switch [9]. The traditional approach of employing a centralized controller for global state management greatly simplifies the implementation, but non-local states can be accessed and updated only at the price of extra delay, thus affecting the overall reactivity. On the other hand, solutions employing global states centralized in a stateful switch lead to performance impairments. Indeed, all flows affected by/ affecting a global state should traverse the switches storing it. This ultimately leads to overall higher network utilization and traffic concentration, thus affecting network congestion and available capacity. Furthermore, any failure to the switch can jeopardize the state integrity due to the presence of a single replica of the global state.

In [10] we propose a novel framework, namely LOADER (LOcAl DEcisions on Replicated states), which enables a new possibility for stateful data planes: the states and the corresponding control logic are distributed across the switches and the controller while permitting multiple replicas of the same state/control logic to be present in the network. This permits to run network applications operating on global states without a unique central entity. Switches can take instantaneous decisions based on local replicas of non-local states, without any controller intervention, thus re-establishing the beneficial effects of stateful data planes also for non-local states. A preliminary analysis of using multiple replicas was presented in [11], mainly focusing on some implementation issues related to state synchronization and replication from the data plane point of view and providing some experimental results. In the meanwhile In [10] instead, we build upon [11] and implement:

- the programming abstractions to define generic (either local or non-local) states and the control logic of any network application;
- the engine to optimally embed the states and the control logic into the network devices and the controller, to optimize performance while taking into account the

available resources in terms of processing and state storage capabilities;

- the mechanism to transparently replicate non-local states across multiple network devices.

To this end, in this Chapter we discuss the issues and possible solutions for offloading network applications to the data plane. By analyzing the outcome of our discussion and thoroughly revising existing solutions we first provide a high-level abstraction of the LOADER framework by defining its core modules and later delve into the details of each module and the way LOADER abstraction is exposed to the network programmer. After discussing consistency-related issues when dealing with replicated states and how to overcome them we deep-dive into details regarding our lightweight implementation of the LOADER framework in ONOS [12] with major emphasis on the data plane implementation in P4 [6] and Open Packet Processor (OPP) [7]. By showcasing LOADER on these two different architectures we prove the generality of the proposed programming model, which, by design, is agnostic of the adopted data plane implementation. Finally, we show how to program a distributed Deny-of-Service (DoS) detection application in LOADER and experimentally assess the performance for both P4 and OPP based implementations. Furthermore, to highlight the versatility of the proposed framework, we provide details about the implementation of other relevant network applications.

2.1 Related work

Data plane embedding of network applications is steadily gaining attention from the industry and the research community. Numerous frameworks and abstraction models have been proposed which try to expose to the programmer data plane resources allowing them to embed custom logic and persistent states directly inside the data plane. Yet no significant effort have been put into dealing with scalability issues which inevitably arise when embedded network applications must operate on network-wide states.

Numerous studies considered employing replicated states in the data plane [13]–[15], demonstrating the scalability benefits of such an approach and treating it as an enabler for new network applications. Yet all of the above studies considered specific applications with tailored implementation.

On the other hand, there have been studies [16]–[20] concerning the development of general-purpose programming abstractions for network applications, none of which considered having replicated states in the data plane. In particular, in [16]–[18] the authors tried to address the issues of defining a general enough programming language for network applications. Yet they considered that states are kept at the controller in a centralized fashion, thus not only neglecting the available data plane resources but also leading to scalability issues due to the centralization of all policies at the same controller. In [19] the authors addressed the former issue by providing an abstraction model including replicated states and distributed network applications among different controllers. Although solving the issue with scalability, applications still reside in the control plane, mitigating the benefits of having stateful data planes.

On the contrary, works such as [9], [20] proposed novel network programming abstractions, which permits to define complex network applications for stateful data planes. In particular, SNAP [9] addressed the problem of performing optimal embedding of states across the network switches, taking into account the dependency between states and the traffic flows. Nevertheless, by design, SNAP is limited to just one replica of each state within the network, thus still precluding a wide variety of possible network applications.

LOADER, instead, enables multiple replicas of the state, extending the single replica approach of SNAP. Furthermore, LOADER closes the gaps of previously proposed programming models by providing a programmer-level abstraction for the definition of network applications while transparently dealing with replication and embedding problems.

The optimal replication problem for multiple replicas has been defined and investigated in [21] and is presented later in Section 2.4. Given a network application and the corresponding states, the problem considers all the traffic flows that are affected by/affect such states and, based on a generic cost function, computes (i) the optimal number of replicas, (ii) their placement within the network and (iii) the corresponding optimal traffic routing. Although the optimal solution for the state replication problem is challenging to obtain in practice, in [21] we propose a simple heuristic able to significantly reduce the problem complexity. While LOADER provides the programming framework and the implementation for replicated states, the heuristic proposed in [21] is used as a building block for the LOADER embedding module.

In general, the problem of maintaining consistency across replicated states has been deeply investigated in the past in the field of distributed systems [22] and many solutions have been proposed, depending on the nature of the states, the desired properties, and the available resources. There have been however, few works concerning replication in stateful data planes. Although we do not treat the issue of developing a sophisticated replication algorithm in this work, we design LOADER in an agnostic way to the actual consistency scheme given future research in the field.

2.2 Offloading network applications

Classic SDN management schemes present a series of limitations such as poor reactiveness, big communication overhead, and compromised fault-tolerance caused by the excessive centralization of the control plane. Stateful data planes introduce the possibility of embedding custom logic inside network devices, thus offering a new way of mitigating the aforementioned issues by providing means of *offloading control plane functionalities to the data plane.*

2.2.1 Network application in stateless SDN

In traditional SDN networks a logically centralized entity, namely the controller, is responsible for managing the whole network operations by means of user-defined network applications.

Being centralized, to function correctly, network applications are required to operate on an accurate snapshot of the network. The task of constructing such a snapshot is delegated to the controller which continuously gathers network statistics in the form of network *states*, which provide a synthetic description of the network in the form of a generic data structure holding a variable or a compound of variables. Given this information at the controller, applications are able to detect the presence of certain events (e.g., load unbalance, security risks, misconfiguration, etc.) by performing a set of operations over the states and, whenever possible, take actions to correct the network operations.

2.2.2 Network applications offloading with stateful data planes

Although being suitable for coarse-grained network operations, due to the poor reactiveness, classic SDN approaches come to their limits when it comes to supporting network applications performing fine-grained operations. Such is the case of, e.g., per-packet processing or fine-grained traffic engineering [23]. Stateful data planes offer the possibility of mitigating this limitation by offloading the related logic to the data plane.

Offloading network applications implies the embedding of some or all of the application elements into the network devices. This involves application states being embedded under the form of stateful primitives natively supported by the network devices and action logic under the form of data plane packet processing modules. Although being feasible from the theoretical point of view, application offloading creates considerable challenges in practice.

2.2.3 Satisfying resource constraints

When it comes to offloading, the type and the corresponding amount of available resources at each network device pose hard constraints for the embedding of application elements.

Dedicated hardware devices, such as switches and routers, lead to almost zero latency during the execution of local processing, but typically have limited resources in terms of processing capabilities and memory. On the other hand, general-purpose network devices such as SDN controllers provide resource flexibility at the cost of large processing latency. To minimize the application execution latency, during the embedding phase, network applications exceeding the resources constraint at a single network device may be *split across multiple devices*. If application splitting still does not satisfy the resources constraint, the application may be fully delegated to the controller, thus, reverting to a traditional stateless SDN scheme.

2.2.4 Inter and intra-application dependencies

In addition to the resource constraints, most of the applications involve a dependency among different elements of a network application, i.e., states are accessed/modified and actions are executed according to a well-defined order which is tightly bound to the definition of the application. The complexity is further increased when considering that a given state of an application may be accessed by different network applications such as in the case of two network applications reading a common counter. This inter and intra-application dependency imposes a constraint on how the traffic must be routed across individual elements of the split application to ensure the correctness on the execution of the application [9], [21].

2.2.5 Offloading shared states

Considering a general case of network applications embedded in different network devices we can define two macro-categories of states: i) given a generic state s stored in a given network device n, s is said to be *local* if it can be accessed (read/write) only by n itself. In such a scenario, s can be internally embedded in n (provided that n is capable of supporting it). ii) On the contrary, when s is accessed (read/write) by multiple network devices that share the state, s is said to be non-local. If all states related to a network application are local, the offloading does not present any considerable challenge as states can be embedded into a single network device, assuming no violation of the capacity constraint. However, when a state is non-local, multiple network applications or multiple parts of the same application must be able to access the state.

In classic stateless SDN, non-local states are managed by states polling and aggregation at the controller. Instead, in stateful SDN, non-local states can be supported with one of the two approaches:

- *Single replica.* As proposed in SNAP [9], a non-local state can be embedded in a single network device, thus a unique replica is made available in the entire network. Consequently, to support inter/intra application dependency all traffic affected by/affecting the state must be routed through that device. In SNAP, the choice of the network device to embed the state in is optimized according to some optimization criteria, e.g., minimization of the distance among dependent states, equal load balancing across the network devices, etc.

  The approach proposed by SNAP may lead to major scalability and performance impairments, specifically when a state is affected by/affects a large amount of traffic.

- *Multiple replicas*: A single state is made available in different parts of the network by

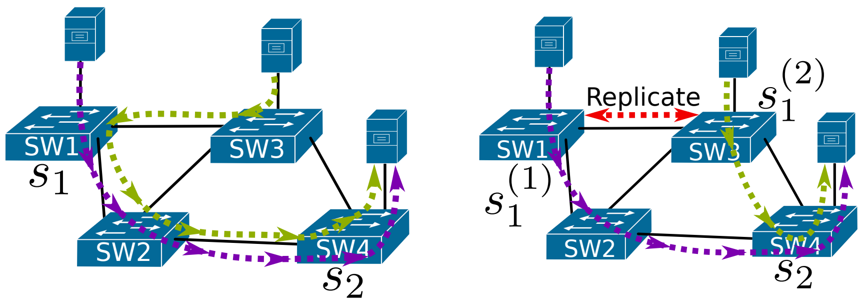


Figure 2.1: Example routing without replicated states (left) and with replicated states (right), as enabled by LOADER.

providing copies (i.e., replicas) of it inside different network devices. This approach permits to distribute the traffic across multiple network devices while also providing robustness to failures. However, although this approach provides more embedding flexibility, it requires the presence of a replication protocol between the replicas, to keep all replicas consistent. In the absence of such a replication mechanism, the values of each replicated state will start to diverge, thus leading to different distinct states which will not be representative anymore of the global network dynamics.

An example of the two approaches is depicted in Figure 2.1. Assume a network application composed of two states, namely s_1 and s_2, and two flows originating from H1 and H2 and directed towards H3. For a single replica in SW1, the green flow is forced to make a detour from its shortest path to traverse SW1 storing s_1. On the contrary, in the presence of multiple replicas the green flow can reach its destination following the shortest path thanks to the presence of two replicas of s_1, namely $s_1^{(1)}$ and $s_1^{(2)}$, embedded respectively inside SW1 and SW2. Although being a simple example, it highlights the importance of using replicated states. Detouring flows from their shortest path adds considerable data overhead in the network which in turn leads to ineffective use of network resources. Furthermore, in extreme cases, such as sudden traffic spikes, this resource mismanagement may lead to scenarios of excessive overload of network devices storing the state, thus degrading the traffic flow performance. State replication mitigates these issues by providing multiple copies of the same state which, in the best-case scenario, are all located on the shortest path for each flow. Furthermore, flow processing and updating network states are delegated to multiple switches, thus the overall workload is distributed across the switches.

2.2.6 Managing inconsistency of replicated states

The management of state inconsistencies is among the most challenging aspect of the approach employing state replication. Whenever a given replica of a state propagates its update to other replicas a period of inconsistency is created. During this time interval read operations on different replicas of the same state may lead to different outcomes. When developing the application, the programmer must be able to take into account the presence of these errors and specify the maximum amount of error that can be tolerated. Consequently, operating on replicated states requires an additional abstraction layer capable of translating user-defined consistency constraints into embedding constrains.

In addition of defining a formal model for managing inconsistency errors, LOADER provides a general abstraction model and a framework for developing network applications based on replicated states. In the following, we identify a common abstraction for network applications permitting LOADER to be target independent and completely agnostic to the underlying network hardware. The abstraction is made generic by: i) supporting network applications operating only on local states, as they fall into the special-case category of single-replica states, ii) supporting the absence of stateful switches, iii) being target-independent from the technologies employed in the data plane.

2.3 LOADER abstraction model and framework

LOADER naturally extends functionalities of previously proposed frameworks based on single-replica states. As shown in Figure 2.2, the proposed framework is based on three main blocks which define the lifecycle of deploying a LOADER application: i) application are defined by means of a predefined set of APIs which expose to the programmer LOADER-specific functionalities; ii) once defined, the applications undergo a compilation phase by means of a compiler capable of translating them into basic primitives supported by network devices; iii) finally, the compiled network applications undergo an embedding phase during which the embedder will try to place the basic primitives composing the network application inside the available network devices.

In the following section we define an abstraction model for LOADER that permits the decomposition of a network application in basic elements that can be directly embedded into network devices.

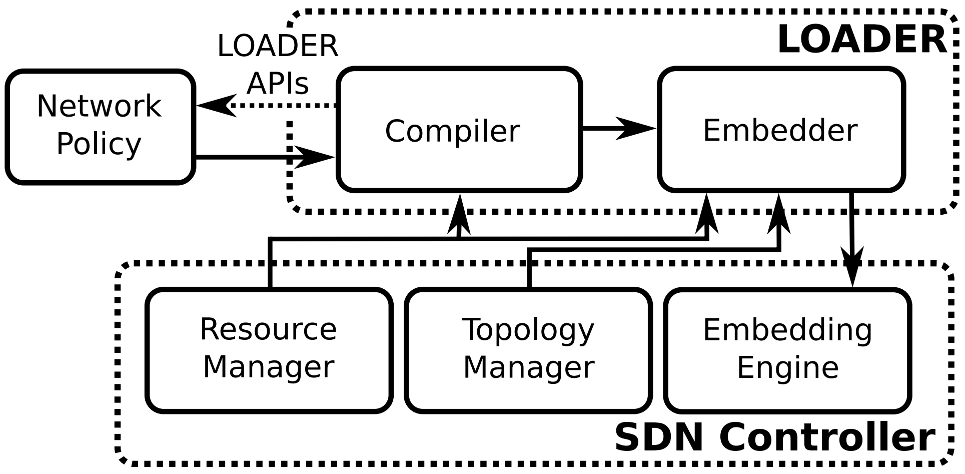


Figure 2.2: Main building blocks of LOADER framework.

At the top layer, users define network applications by employing a set of predefined building blocks, namely *application elements*, in a completely agnostic way with respect to the remaining components of the framework. The application elements supported by LOADER are the only part of the framework exposed to the programmer by means of APIs and generic language libraries. While maintaining generality, the use of these elements permit an efficient decomposition of user-defined applications during the compilation phase and provide a comprehensive abstraction for the compiler during their translation to device-specific primitives.

2.3.1 Building blocks of a network application

Figure 2.3 depicts an example of a generic network application employing LOADER abstraction. Each application is composed of four main types of application elements which have to be implemented by the programmer: *states*, *reduction functions*, *trigger functions* and *activity functions*.

For ease of explanation, in the following, we will present the role of each application element by considering a reference data center load balancing application. More details about the topology and the functionality of data center load balancers will be presented in Chapter 3. This application works as follows: (1) whenever the load on the data center servers is medium-low, the application distributes the user's request among the available servers in a load balancing fashion, i.e., an arriving request is forwarded to

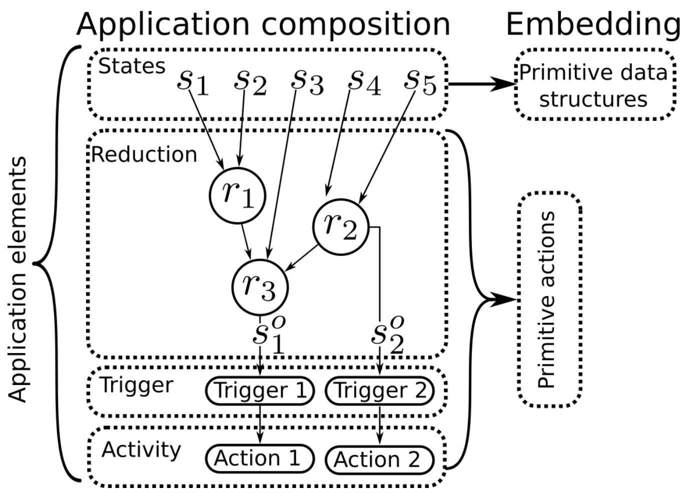


Figure 2.3: DAG representation of a LOADER network application and its mapping to primitive elements.

the least loaded server, in terms of CPU utilization; (2) otherwise, when the data center is highly loaded, users' requests are sent to the controller for further processing.

Each application element is defined as follows:

- **States:** Let $\Omega_P = \{s_i\}_i$ be the set of states associated with a network application P, with $s_i^{(k)}$ be the k-th replica of state s_i, with $k \in \mathbb{N}$. For the reference load balancing application, state s_i represents the current CPU load of a generic server i, where $i = 1, \dots, n$, and n is the number of available servers.

- **Reduction function:** The reduction function is a generic multivariate function that maps states in Ω_P to a reduced version s_1^o of the input states. It is obtained by combining a set $\mathscr{R} = \{r_j\}$ of primitive reduction actions natively available in the network device. In the reference application, $\mathscr{R} = \{r_1, r_2\}$ with $r_1 = \arg\min()$ and $r_2 = \text{mean}()$, which compute the index corresponding to the minimum and the average of an array of values, respectively. Consequently, the reduced versions are just two scalars: $s_1^o = \arg\min(s_1, \dots, s_n)$ and $s_2^o = \text{mean}(s_1, \dots, s_n)$.

- **Trigger function:** Based on s_i^o, the trigger function evaluates the presence of a particular event and decides whether a reaction is required or not. The reference application operates concurrently on two trigger functions leading to different activity functions. The first trigger function checks if the average data center

```
from Controller import TopologyManager
from LOADER.PrimitiveActions import SetEgress, Rate
from LOADER.Scope import Pkt, ExtScopeHelper

THR = 0.8  # threshold CPU load percentage

# Get the average CPU load of servers in the form of a list of states. We omit
  the details.
loads = ExtScopeHelper(scope="ServerLoad")

r1 = ReductionFunction(
      states = [loads]
      operation=argmin([i.Value() for i in loads]))

r2 = ReductionFunction(
      states = [loads]
      operation=mean([i.Value() for i in loads]))

a1 = ActivityFunction(
      scope = Pkt(filter = (TCP.Flag.SYN == 1)),
      action = SetEgress,
      args = r1.Result())

a2 = ActivityFunction(
      scope = Pkt(filter = (TCP.Flag.SYN == 1)),
      action = SetEgress,
      args = CONTROLLER_PORT)

tr1 = TriggerFunction(
      s0=r2.Result(),
      trigger=(r2.Result() <= THR),
      inconsistencyLevel=UpdateError(15),
      activity=a1)

tr2 = TriggerFunction(
      s0=r2.Result(),
      trigger=(r2.Result() > THR),
      inconsistencyLevel=UpdateError(15),
      activity=a2)
```

Listing 2.1: Resource-aware load balancing with LOADER.

load s_2^o is below a given threshold (corresponding to a low load scenario). The second trigger function instead is activated whenever s_2^o is above the predefined threshold.

- **Activity function:** The activity function is a sequence of actions that are executed when the events associated with a trigger function occur. In the reference application, two action functions are defined. If the first trigger function is satisfied (i.e., s_2^o is smaller than the threshold), then Action 1 is executed and the user's request is sent to the least loaded server, otherwise, Action 2 is triggered and the request is forwarded to the controller. Both action functions are executed at the same switch where the request has been received.

Table 2.1: Example applications enabled by LOADER and their mapping the the LOADER programming model

Application	States	Reduction function	Trigger function	Action function
DDoS detection [11]	Average rate of inbound SYN packets traversing each edge router	Sum of all states	Comparison against a fixed threshold	Controller notification
Distributed rate-limiting [14]	Average rate of inbound traffic traversing each edge router	Sum of all states	Comparison against a random threshold	Packet drop
Link-aware load balancing [13]	Average load on uplink and downlink ports connecting a pair of two ToR switches	Argmin among the maximum of all uplink and downlink pairs sharing a common path	Change in the reduction function output	Insertion of a per-flow forwarding rule
Resource-aware load balancing [24]	Instantaneous CPU utilization of servers	i) Argmin among all states, ii) Mean among all states	Comparison of the average global CPU utilization against a fixed threshold	Insertion of a per-flow forwarding rule if the average CPU utilization is below a threshold, controller notification otherwise.

The actual implementation of the reference load balancing application is shown in Listing 2.1. The listing highlights the simplicity of defining the application, by depicting how each of the previously discussed application elements can be defined and manipulated by the programmer thanks to the APIs provided by the LOADER programming model.

Table 2.1 depicts other example applications which operate on replicated states. Those applications have been proposed in the literature with either custom hardware implementation inside switches or by employing ad-hoc P4 code. We show how those applications can be easily mapped to the LOADER abstraction by employing the application element provided by our abstraction model. A detailed description of those applications, alongside with the evaluation of selected applications, will be presented in Section 2.7.

We limit the discussion of the syntax and the implementation details of the proposed programming model while focusing on its core functionalities. One core functionality is the explicit management of inconsistencies, which is specified as a parameter in both trigger functions (as explained in Section 2.5), and the semantic of the language which is discussed in the following.

2.3.2 Semantics and order of execution

By default, in LOADER all operations on states are executed in parallel. Such kind of disaggregation for the order of execution significantly reduces the embedding complexity as each element of the network application can be treated independently and will not require order synchronization.

Nevertheless, some applications may require a specific order for the execution of the activity functions (e.g., appending new packet headers in a given order). Such kind of constraints may significantly increase the complexity of the overall approach by requiring additional ordering mechanisms whenever the activity functions are distributed across different switches.

LOADER provides means of specifying particular order for the execution of operations by exposing to the programmer the `SequentialActivityFunction` class. This class imposes hard constrains on the compiler forcing it to treat the enclosed activity functions as a single sequential activity function. This in turn forces the embedder to perform co-located embedding of those activity functions, forcing them to be embedded in the same network devices, ultimately permitting sequential execution.

2.3.3 Compilation phase

Although in our experimental evaluation we implemented a minimal proof-of-concept compiler, implementing a network application compiler compatible with a broad variety of different network devices requires immense effort and in-depth knowledge about each device architecture. For this reason, for the purpose of this work we discuss what the compiler must perform and how the proposed framework facilitates its operations.

Network applications are compiled through the *LOADER Compiler*, as shown in Figure 2.2. The compiler takes as input the network capabilities in the form of available *basic primitives*, and the user-defined application in the form of LOADER application elements. The catalog of available primitives depends on the specific network devices operating in the network and is stored in the resource management module of the network controller and it is updated through the network management plane, e.g., at device installation time. The application is then represented by the compiler in the form of a DAG (Directed Acyclic Graph) composed of its basic elements, as shown in Figure 2.3. The compiler then reconstructs the dependency among each application element and maps them to basic primitives supported by the network devices composing the network so that, as depicted in Figure 2.3:

- states are mapped into *primitive data structures*, such as counters, registers, hash tables, etc., to store application states;
- reduction, trigger, and activity functions are mapped into *primitive actions*, i.e. basic processing/decision capabilities offered by network devices.

2.3.4 Optimal embedding and application reaction latency

The embedding consists in mapping the primitive elements provided by the compiler into a set of physical network devices. This is performed by exploiting the target-specific drivers and southbound APIs (e.g., P4Runtime, gRPC, OpenFlow, etc.) offered by the embedding engine of the controller.

To perform the actual embedding, as depicted in Figure 2.2, the embedder takes as input: i) the set of primitive elements provided by the compiler, ii) the resource availability inside the network provided by the controller resource manager, and iii) the actual location of the resources inside the network provided by the controller topology manager. Given this information, it is possible to find a set of feasible embeddings of the decomposed application inside the network devices supporting the required primitives. Notably, each element of the network application is not required to be embedded in a single network device. Instead, individual primitives composing the network application can be embedded in different network devices, based on the types of supported primitives, their amount and their location inside the network. The adopted

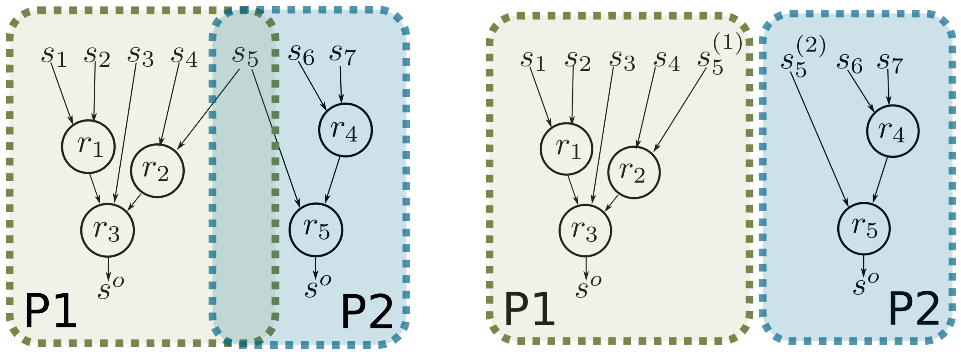

s_1 s_2 s_3 s_4 s_5 s_6 s_7
r_1 r_2 r_3 r_4 r_5
s^o
P1
P2
$s_5^{(1)}$
$s_5^{(2)}$

Figure 2.4: Reduction function decomposition in case of two network applications sharing a state s_5 without replicated states (left) and with replicated states (right).

algorithm to optimize the embedding (i.e., computing the optimal number of replicas and their placement within the network) has been already investigated in our previous work [21], which serves as a natural integration to LOADER. In the following, we give insights regarding the functionalities and restrains of the embedding mechanism, while in Section 2.4 we provide the Integer Linear Programming (ILP) model for solving the problem of optimal state replication.

Constraints on primitives location

In the absence of co-location at the same network device of primitive actions and primitive data structures directly operated by those primitive actions, state replication is mandatory. Indeed, to perform the reduction of a given set of states, the states must be locally available at the network device operating the reduction function. This requires either to provide co-location of the states and reduction functions or to perform state replication at the network device storing the corresponding reduction primitive.

Inter-application state sharing

States may be shared among different network applications. Figure 2.4 shows an example of two network applications P_1 and P_2 sharing a common state s_5. Using a single replica approach, s_5 is required to be embedded into a single network device. As a consequence, the device storing s_5 must serve both P_1 and P_2, which, as previously discussed, may lead to scalability issues whenever the number of applications employing s_5 grows large. Instead, with state replication, the two applications can be made independent by replicating s_5 in $s_5^{(1)}$ and $s_5^{(2)}$. Note that the concurrent access of two different applications to two replicas of the same state is equivalent to the concurrent access of two instances of the same application on such replicated states, as in the DDoS detection scheme discussed in Section 2.7.

Application reaction latency

Given an application embedding, it is possible to evaluate the corresponding *reaction latency*, by considering the position of the primitives in the network, the propagation delays between the involved network devices, and the replication delay. For a single-replica state, the replication delay is by construction null as no replication occurs whatsoever. On the other hand, in the case of multiple replicas, the reaction latency models the latency required to propagate a new value of the state to all the replicas and will be explained in detail in Section 2.5.1. Interestingly enough, as investigated in [21], an optimal embedding might lead to multiple replicas. Although multiple replicas imply non-null replication delays, this delay can be compensated by a much smaller application execution latency. The distributed DDoS detection application, considered later in Section 2.7, is an example of such a scenario, clearly showing the advantage of keeping multiple replicas for some network-wide applications.

Objective-based embeddings

The optimal embedding is chosen by minimizing a particular cost function. The definition of the cost function highly influences the way the embedding is performed, as shown in [21]. As an example, a cost function aiming at reducing the network energy consumption or reducing the synchronization traffic between replicas may lead to scenarios in which the application is embedded into few network devices or eventually to a single network device (e.g., the SDN controller). On the other hand, a cost function aiming at minimizing the network congestion may lead to multiple replicated states across different network devices to balance the traffic across the network. Thus, the definition of the cost function highly affects the level of distribution of the application, ranging from completely distributed implementations to completely centralized and stateless ones.

LOADER in stateless SDN

In the case of stateless SDN networks, with network devices able to perform only basic forwarding/routing operations, the LOADER approach is still viable. Indeed, LOADER provides only an abstraction layer between the actual application and its mapping to the network devices. As previously discussed, the controller is seen as part of the available embedding targets during the embedding phase. Being typically a general-purpose machine the controller is seen as a network device with unlimited computation resources, thus giving the embedder the possibility of eventually placing the network application at the controller. Nevertheless, as previously mentioned, the latency between the controller and the network devices is typically high as it includes both the network latency and the in-software processing delays at the controller. As shown in the following, this latency plays a fundamental role during the embedding as it directly affects the *state inconsistency level* which is among the main user-defined constraints in LOADER.

2.4 Optimal state replication problem

As described in the previous, the optimal state replication problem must incorporate a broad variety of constraints and must take into account both the traffic statistics in the form of the traffic matrix and the available resources of the network devices. In the following we will present a simplified version of the optimal state replication problem which does not take into consideration the resource requirements. The presented formulation will focus exclusively on finding the optimal embedding for replicated states while satisfying intra-application dependencies. For the sake of presentation clarity we will refer to all of the network application primitives as states, thus assuming reduction/trigger and action functions to be stateful elements.

Given a network graph, the objective of the state replication problem is to identify the best set of nodes (i.e., network devices) where to place the replicas of each state and to compute the optimal routing. Coherently with [9], the nodes are selected to minimize the overall traffic in the network and to guarantee that all flows affecting (or affected by) a given state will traverse at least one state replica. Differently from [9], the traffic in the network is composed not only of data traffic, but also of the traffic introduced by the synchronization protocol required to keep consistent the replicas of a given state.

We propose an ILP formulation, as in the original SNAP model [9]. The relevant notation is reported in Table 2.2 and further discussion and the evaluation of this model is deeply discussed in [21]. Our formulation takes the following input parameters:

- **Network:** Let $G = (V, E)$ be the network graph with N nodes. Let c_e be the capacity of edge $e \in E$.
- **Traffic flows:** Let $\mathscr{F}$ be the set of all flows. The traffic demands are assumed to be known in advance. In particular: let λ_f be the demand of traffic flow $f \in \mathscr{F}$, being $f_s \in V$ and $f_d \neq f_s \in V$ respectively the source and the destination nodes of the flow.
- **State variables:** Let S be the set of all state variables. Let $S_f \subseteq S$ be the *ordered* sequence of state variables for flow $f \in \mathscr{F}$, obtained from the primitives dependency graph of the corresponding application.
- **Maximum number of replicas:** Let C_s be a given upper bound on the number of replicas for a state variable s, chosen by the network designer. Note that the optimal number of replicas for state s, denoted by $\hat{C}_s$, will be computed while satisfying the constraint $\hat{C}_s \leq C_s$. Furthermore setting $C_s = 1$ for primitive functions (i.e., reduction, trigger and action functions) will lead to them not being replicated the final embedding.

Let H_f be the set of all possible sequences of state replicas for a flow f. Consider a toy example in which a flow f requires 3 state variables $\mathscr{A}$, $\mathscr{B}$, $\mathscr{C}$, i.e., $S_f = [\mathscr{A}, \mathscr{B}, \mathscr{C}]$. Each state has 2 replicas (denoted as “1” and “2”). Now H_f ={[1 1 1], [1 1 2], [1 2 1], [1 2 2], [2 1 1], [2 1 2], [2 2 1], [2 2 2]}, and, as example, the sequence $h = [121]$ implies that f traverses replica 1 of state $\mathscr{A}$, then replica 2 of state $\mathscr{B}$, and finally replica 1 of state $\mathscr{C}$. Let h_s be the replica of state variable s in sequence $h \in H_f$. For the above example with $h = [121]$, $h_{\mathscr{A}} = 1$, $h_{\mathscr{B}} = 2$ and $h_{\mathscr{C}} = 1$.

The output of the solver is described as follows, and the relevant notation is reported in Table 2.3:

- *Placement of the replicas of each state.* Let P_{scn} be a binary variable equal to 1 iff replica c of state s is stored at node n.

  Note that the optimization problem might place multiple replicas on the same node, but this would correspond to a single instance of the state. Thus, the optimal number of distinct replicas $\hat{C}_s$ of state s across the whole network can be computed as follows[1]:

$$\hat{C}_s = \sum_{n \in V} \mathbb{1}\left\{ \sum_{c \leq C_s} P_{scn} > 0 \right\}$$

- *Data traffic routing.* Let R_{fhe} be a binary variable equal to 1 iff flow f traverses the sequence of state replicas h on edge e. The set of such variables describes the complete routing of all flows in the network, taking also into account the constraint for the required sequence of traversed replicas. To avoid out-of-sequence problems, we do not permit flow splitting between different sequences of replicas.

- *Synchronization traffic routing.* Let $\hat{R}_{snme}$ be a binary variable equal to 1 iff there are replicas of the state variable s on nodes n and m and the flow from node n to node m traverses edge e. This set of variables describes the routing of the synchronization traffic between different replicas of the same state. Let $\hat{\lambda}_s$ be the traffic generated by each state replica to update each other single replica of the same state.

Finally, Table 2.4 reports the list of auxiliary variables adopted in the ILP formulation.

In the optimal state replication problem, the total traffic in the whole network is minimized:

$$\min \sum_{e \in E} \sum_{f \in \mathcal{F}} \sum_{h \in H_f} R_{fhe} \lambda_f + \sum_{e \in E} \sum_{s \in S} \sum_{n \in V} \sum_{\substack{m \in V \\ n \neq m}} \hat{R}_{snme} \hat{\lambda}_s \tag{2.1}$$

The first term represents the total data traffic in the network. It is obtained by summing all the traffic due to f on all the possible sequences of state replicas and on all of the

[1]Let $\mathbb{1}_{\{A\}}$ be the indicator function of A, equal to 1 iff condition A is true.

Table 2.2: Input variables

Context	Variable	Description	Range
Network definition	V	set of all nodes	$\{1, \ldots, N\}$
	N	number of nodes (i.e., $\|V\|$)	$\mathbb{N}$
	E	set of all edges	
	c_e	capacity of edge $e \in E$	> 0
Flow definition	$\mathcal{F}$	set of all the flows	
	λ_f	traffic demand for flow $f \in \mathcal{F}$	> 0
	f_s	source node for flow $f \in \mathcal{F}$	$1, \ldots, N$
	f_d	destination node for flow $f \in \mathcal{F}$	$1, \ldots, N$
State definition	S	set of all state variables	
	C_s	max number of replicas for state s	≥ 1
	S_f	sequence of state variables for flow $f \in \mathcal{F}$	$\subseteq S$
	$\hat{\lambda}_s$	synchronization traffic between any pair of replicas for state $s \in S$	> 0

Table 2.3: Output variables

Context	Variable	Description	Range
Data traffic routing	R_{fhe}	1 iff flow f along sequence of replicas h traverses edge e	Binary
Synchronization traffic routing	$\hat{R}_{snme}$	1 iff synchronization traffic from node n to node m containing replicas of state variable s traverses edge e	Binary
Replica placement	P_{scn}	1 iff replica c of state s is stored in node n	Binary

Table 2.4: Auxiliary Variables

Variable	Description	Range
$E_I(n)$	set of edges entering node $n \in V$	$\subseteq E$
$E_O(n)$	set of edges leaving node $n \in V$	$\subseteq E$
$E(n)$	set of all edges incident to node $n \in V$	$\subseteq E$
H_f	set of all sequences of replicas for flow $f \in \mathcal{F}$	-
h_s	replica id of state s for flow $f \in \mathcal{F}$ in sequence $h \in H_f$	$1, \ldots, C_s$
P_{fsce}	1 iff flow f on edge e has passed replica c of state s	Binary
X_{fh}	1 iff flow f is assigned $h \in H_f$	Binary
U_{sn}	1 iff at least one replica of state variable s is on node n	Binary
Y_{snme}	1 iff $\hat{R}_{snme} > 0$	Binary

edges. Instead, the second term is the synchronization traffic between replicas of the same state, summed across all states and edges in the graph. Notably, (2.1) is similar to the objective function used by the SNAP framework in [9], but with the introduction of the second term that takes into account the synchronization traffic, not included in SNAP.

As an alternative, the objective function could be modified to minimize the maximum congestion on a link, obtained by summing data and synchronization traffic, as follows:

$$\min \max_{e \in E} \Big(\sum_{f \in \mathcal{F}} \sum_{h \in H_f} R_{fhe} \lambda_f + \sum_{s \in S} \sum_{n \in V} \sum_{\substack{m \in V \\ n \neq m}} \hat{R}_{snme} \hat{\lambda}_s \Big) \tag{2.2}$$

and could be easily integrated in the following formulation, using well-known ILP modeling techniques.

2.4.1 Constraints in the optimization problem

We now discuss all the constraints considered in the ILP model. In some cases, we will get products of binary variables, but the corresponding constraint can be easily linearized according to well-known techniques.

Data routing constraints

Constraints (2.4)-(2.7) are similar to the constraints for the classic multi-commodity flow problem. However, our modification consists of assigning a commodity for each sequence $h \in H_f$ of state variable replicas directly at the source of the flow f, to model the sequence of states required by each flow.

We introduce an auxiliary variable, which is an indicator function X_{fh} equal to 1 if sequence $h \in H_f$ is assigned to flow $f \in \mathcal{F}$.

$$X_{fh} = \sum_{e \in E_O(f_s)} R_{fhe} - \sum_{e \in E_I(f_s)} R_{fhe} \tag{2.3}$$

Indeed, whenever a particular sequence h is adopted, similar to (2.4), the net outgoing data traffic from source f_s is 1. Notably, the second term considers the special case in which the flow is re-entering (and leaving) f_s in the path to reach the state and then the destination. We now force only one sequence h to be assigned to flow f. $\forall f \in \mathcal{F}$:

$$\sum_{h \in H_f} X_{fh} = 1 \tag{2.4}$$

A similar constraint is defined for flow f's destination f_d, but now the net incoming flow should be 1. $\forall f \in \mathcal{F}$:

$$\sum_{h \in H_f} \Big(\sum_{e \in E_I(f_d)} R_{fhe} - \sum_{e \in E_O(f_d)} R_{fhe} \Big) = 1 \tag{2.5}$$

The sum of all the data and synchronization traffic passing an edge must not exceed its capacity. $\forall e \in E$:

$$\sum_{f \in \mathcal{F}} \sum_{h \in H_f} R_{fhe} \lambda_f + \sum_{s \in S} \sum_{n \in V} \sum_{\substack{n^* \in V \\ n \neq n^*}} \hat{R}_{snn^*e} \hat{\lambda}_s \leq c_e \tag{2.6}$$

Finally, the standard flow conservation condition must be satisfied at any node. $\forall h \in H_f, \forall f \in \mathcal{F}$:

$$\sum_{e \in E_I(n)} R_{fhe} = \sum_{e \in E_O(n)} R_{fhe} \quad \forall n \in V \setminus \{f_s, f_d\} \tag{2.7}$$

Placement constraints

Each replica can only be placed at one switch. $\forall s \in S$, $\forall c \leq C_s$:

$$\sum_{n \in V} P_{scn} = 1 \tag{2.8}$$

We now constrain the flows to be routed through the corresponding states, i.e., all flows dependent on a state must traverse the node where the replica of such state is located (except at source f_s and destination f_d). $\forall n \in V \setminus \{f_s, f_d\}, \forall f \in \mathcal{F}, \forall h \in H_f, \forall s \in S_f$:

$$\sum_{e \in E_I(n)} R_{fhe} \geq P_{sh_sn} + X_{fh} - 1 \tag{2.9}$$

Indeed, if a particular sequence h is adopted for f, then (2.9) becomes $\sum_{e \in E_I(n)} R_{fhe} \geq P_{sh_sn}$ and in the case the node contains a replica h_s of the state s, then $\sum_{e \in E_I(n)} R_{fhe} \geq 1$, which forces at least one R_{fhe} variable to be one on the incoming edges to e. Otherwise, if the sequence h is not adopted for f, then (2.9) becomes a useless bound.

We now define a variable that tracks the fact that a flow has already traversed a particular state along its path. For a flow f traversing a replica h_s of state s, we define

$P_{fsh_se} = 0$ for all edges along the path before entering the node with replica h_s of s, and $P_{fsh_se} = 1$ for all edges on the path after h_s. It is initialized to zero for all unused replica sequences h. $\forall f \in \mathcal{F}, \forall s \in S_f, \forall h \in H_f, \forall e \in E$:

$$P_{fsh_se} \leq R_{fhe} \tag{2.10}$$

To model the fact that P_{fsh_se} changes from 0 to 1 whenever the flow leaves a node where the state is stored, we set: $\forall f \in \mathcal{F}, \forall s \in S_f, \forall h \in H_f, \forall e \in E, \forall n \in V \setminus \{f_s, f_d\}$:

$$P_{sh_sn}X_{fh} + \sum_{e \in E_I(n)} P_{fsh_se} = \sum_{e \in E_O(n)} P_{fsh_se} \tag{2.11}$$

Indeed, only when $P_{sh_sn}X_{fh} = 1$ (i.e., node n has replica h_s and f exploits h including it), the net flow of P_{fsh_se} entering n is 0 and the corresponding one leaving n is 1.

We now impose that the data flow reaches the destination f_d after having traversed all the states required in h, i.e. $P_{fsh_se} = 1$ for one edge entering f_d. $\forall f \in \mathcal{F}, \forall s \in S_f, \forall h \in H_f$:

$$P_{sh_sf_d}X_{fh} + \sum_{e \in E_I(f_d)} P_{fsh_se} = X_{fh} \tag{2.12}$$

So far, the constraints (2.10)-(2.12) force the flows to pass through all the required state variables, but not necessarily in sequence. We model here the correct sequence of traversed states, if the flow f has to cross $h_s \in H_f$ of s, followed by replica $h_{s'} \in H_f$ of s'. $\forall f \in \mathcal{F}, \forall s, s' \in S_f, \forall h \in H_f, \forall n \in V$

$$P_{sh_sn} + \sum_{e \in E_I(n)} P_{fsh_se} \geq P_{s'h_{s'}n} + X_{fh} - 1 \tag{2.13}$$

Indeed, if either flow f has been assigned sequence h, i.e., $X_{fh} = 1$, or replica $h_{s'} \in H_f$ exists at node n, or replica $h_s \in H_f$ does not exist at node n, then (2.13) becomes $\sum_{e \in E_I(n)} P_{fsh_se} \geq 1$. This forces P_{fsh_se} to be 1 before entering node n, which means that the flow must have traversed h_s before entering the node containing $h_{s'}$. This ensures that the flow traverses the correct sequence of states as dictated by h.

Constraint (2.14) ensures that if flow has traversed state variable replica h_s on edge e, i.e., $P_{fs'h_{s'}e} = 1$, then it must have already crossed state variable replica h_s, which ensures $P_{fsh_se} = 1$. $\forall f \in \mathcal{F}, \forall s, s' \in S_f, \forall h \in H_f, e \in E$:

$$P_{fsh_se} \geq P_{fs'h_{s'}e} \tag{2.14}$$

State synchronization

State synchronization implies the generation of synchronization traffic between any pair of replicas of the same state. Thanks to the routing variable $\hat{R}_{snme}$, we can model the traffic between any pair of nodes n and m containing replicas of the state variable s and consider its contribution in the total traffic, as in (2.1) and (2.2), and in the constraint (2.6) regarding the edge capacity.

In the optimization model, multiple replicas of the state variable can be hosted on the same node n. Hence, to track that there is at least one replica at node n, we define the variable U_{sn} in (2.15). $\forall c \in C_s$, $\forall s \in S$, $\forall n \in V$:

$$U_{sn} \geq P_{scn} \tag{2.15}$$

For the synchronization traffic from node n to node m, the routing variable $\hat{R}_{snme}$ is treated as a commodity from node n such that $U_{sn} = 1$ to node m such that $U_{sm} = 1$. We constrain the routing to ensure the standard flow conservation equation at the intermediate node.

We define a new intermediate variable Y_{snme}, set to 1 iff $\hat{R}_{snme} > 0$. This is ensured using the big-M method [25] as in (2.16) where M is sufficiently larger than $\hat{R}_{snme}$. $\forall s \in S$, $\forall n \in V$, $\forall m \neq n \in V$, $\forall e \in E$

$$0 \leq -\hat{R}_{snme} + MY_{snme} \leq M - 1 \tag{2.16}$$

To fix a large enough value for M, assume $\hat{R}_{snme} = 1$, $\forall e \in E_O(n)$, then $Y_{smne} = 1$ from (2.16). In this case, for the condition $M \geq \hat{R}_{snme}$ to be true, M must be equal to or greater than the maximum degree of G:

$$M \geq \Delta_G \tag{2.17}$$

with $\Delta_G = \max_{n \in V} |E_O(n)|$.

We require the egress synchronization flow from a state replica containing node to use only one outgoing edge. This can be done by exploiting Y_{snme} as in (2.18). $\forall s \in S$, $\forall n \in V$, $\forall m \neq n \in V$:

$$\sum_{e \in E_{O(n)}} Y_{snme} \leq 1 \tag{2.18}$$

The following constraints (2.19)-(2.22) model the multi-commodity flow problem for the synchronization traffic. Specifically, constraints (2.19) and (2.20) are for the originating synchronization flow from the source node n and the sink flow in the destination node m containing the state replicas respectively. $\forall s \in S$, $\forall n \in V$, $\forall m \neq n \in V$:

$$\sum_{e \in E_{O(n)}} Y_{snme} \geq U_{sn} \tag{2.19}$$

$$\sum_{e \in E_{I(m)}} Y_{snme} \geq U_{sm} \tag{2.20}$$

Instead, constraints (2.21)-(2.22) are for the flow conservation at intermediate nodes. $\forall s \in S$, $\forall n \in V$, $\forall m \neq n \in V$:

$$\sum_{e \in E_{O(n)}} Y_{snme} \leq \sum_{e \in E_{I(n)}} Y_{snme} + U_{sn} \leq 1 \tag{2.21}$$

$$\sum_{e \in E_{I(n)}} Y_{snme} \leq \sum_{e \in E_{O(n)}} Y_{snme} + U_{sm} \leq 1 \tag{2.22}$$

2.4.2 Approximate solution

The complexity to solve an ILP model is $O(2^{2^{k_v+2}} k_c)$ [26], where k_v is the number of variables and k_c is the number of constraints. As a worst case, assume that all flows $f \in \mathcal{F}$ require to traverse all state variables $s \in S$, where each $s \in S$ has C replicas. In this case, it can be shown that $k_v = O(\max(N^2 C^{|S|}, |S|N^4))$ and $k_c = O(\max(N|S|C^{|S|}, |S|N^4))$. In a simple scenario when only one state variable required by all the flows, $k_v = O(N^4)$ and $k_c = O(N^4)$. Thus, the final complexity is lower bounded by $O(2^{2^{N^4+2}} N^4)$. Clearly, the presented ILP formulation does not scale for large instances of the problem. This advocates the design of approximation algorithms to solve the optimal replication problem in real scenarios. To address this issue, we propose PlaceMultiReplicas (PMR) algorithm which is computationally scalable and will be later shown to approximate well the optimal solution obtained by the ILP solver for small problem instances.

The pseudocode of PMR is given in Algorithm 1. It takes as input the network graph G, the state variable s and the maximum number of replicas C_s of s and the set of flows $\mathcal{F}$ requiring s. As output, the algorithm returns: the routing variables of the data flows R_{fhe} and of the state synchronization flows $\hat{R}_{smne}$ and the replicas placement variables P_{scn}. The algorithm works through 3 phases:

- *Phase 1.* The network graph G is partitioned into C_s clusters, in order to minimize the maximum distance among the elements within a cluster. This allows to distribute the replicas across the whole network in a balanced way, exploiting the spatial diversity offered by each cluster.
- *Phase 2.* In each cluster, a replica is placed in the "most central" node, i.e., the one with the highest betweenness centrality, in order to minimize the data traffic for each flow.
- *Phase 3.* The position of each replica is perturbed at random using a local search to improve the solution with respect to one obtained in the previous two phases.

Algorithm 1 comprises all the mentioned phases. After having initialized the routing and the replica placement variables (lines 2-4), Phase 1 is executed in line 5 by calling ComputePartitions. This method solves the k-means clustering problem [27] with $k = C_s$ using Lloyd's algorithm [28] in which the node with the highest betweenness centrality is chosen as the center of the partition.

As part of Phase 2 (lines 6-9), within each subgraph G_c the node n' with the highest betweenness centrality is assigned a state variable replica through NodeWithHighestBC.

Lines 11 to 18 refer to a local search procedure with I iterations. Within each iteration, RouteFlows is used to route flows through the location of the replicas identified in Phase 2, following two sub-paths: one from the flow source node to the closest replica and one from this replica to the destination node. The procedure works on the set of flows $\mathcal{F}$ and the location of state variables P_{scn} and returns the routing variables for

Algorithm 1 PlaceMultiReplicas (PMR)

```
1:  procedure $[\{R_{fhe}\}, \{\hat{R}_{smne}\}, \{P_{scn}\}]$ = PlaceMultiReplicas($G, s, C_s, \mathscr{F}$)
2:      $R_{fhe} = 0, \forall f \in \mathscr{F}, h \in H_f, \forall e \in E$        ▷ Init routing
3:      $\hat{R}_{smne} = 0, \forall c, g \neq c \leq C_s, \forall e \in E$        ▷ Init state sync
4:      $P_{scn} = 0, \forall c \leq C_s, \forall n \in V$        ▷ Init state $s$ location
5:      $\{G_c\} \leftarrow$ ComputePartitions($G, C_s,$)        ▷ **Phase 1:** Graph partitions $\{G_c\}$
6:      for $c \leq C_s$ do        ▷ **Phase 2:** Replica placement
7:          $n' \leftarrow$ NodeWithHighestBC($G_c$)        ▷ Find best candidate in partition $G_c$
8:          $P_{scn'} = 1$        ▷ Store the state replica location
9:      end for
10:     $T_{\min} = \infty$        ▷ Init minimum traffic
11:     for $I$ iteration do        ▷ **Phase 3:** Local search
12:         $[T', \{R'_{fhe}\}, \{\hat{R}'_{smne}\}] \leftarrow$ RouteFlows($\mathscr{F}, \{P_{scn}\}$)        ▷ Route flows through the replicas
13:         if $T' < T_{\min}$ then        ▷ Check if the traffic is smaller
14:             $T_{\min} = T'$        ▷ Store current best solution
15:             $R_{fhe} = R'_{fhe}$ $\hat{R}_{smne} = \hat{R}'_{smne}, P'_{scn} = P_{scn}, \forall f \in \mathscr{F}, \forall h \in H_f, \forall c, g \neq c \leq C_s, \forall e \in E, \forall n \in V$
16:         end if
17:         $\{P'_{scn}\} \leftarrow$ PerturbReplicaLocation($\{P_{scn}\}$)        ▷ Change existing location of state replicas
18:     end for
19: return $[\{R_{fhe}\}, \{\hat{R}_{smne}\}, \{P_{scn}\}]$
20: end procedure

21: procedure $[T_{\text{CURRENT}}, R'_{fce}, \hat{R}'_{smne}]$ = RouteFlows($\mathscr{F}, P_{scn}$)
22:     $T_{\text{current}} = 0$        ▷ Init total traffic
23:     for $f \in \mathscr{F}$ do        ▷ For each flow
24:         minDist = $\infty$        ▷ Init minimum distance
25:         $c_b \leftarrow$ null        ▷ Init best replica for current flow
26:         $\mathscr{P}_{best} \leftarrow$ null        ▷ Path with minimum length for $f_s \rightarrow n_c \rightarrow f_d$
27:         for $c \in C_s$ do        ▷ For all state replicas
28:             $\mathscr{P}$ = ShortestPath($f_s, n_c$) $\cup$ ShortestPath($n_c, f_d$)
29:             if $\mathscr{P}$.length < minDist then
30:                 minDist = $\mathscr{P}$.length        ▷ Update minimum distance
31:                 $\mathscr{P}_{best} \leftarrow \mathscr{P}$        ▷ Store path with minimum length
32:                 $c_b \leftarrow c$        ▷ Store best replica for this flow
33:             end if
34:         end for
35:         for $e \in \mathscr{P}_{best}$ do        ▷ For each edge in the minimum length path
36:             $R'_{fc_be} = R'_{fc_be} + \lambda_f$        ▷ Store the routing
37:             $T_{\text{current}} = T_{\text{current}} + \lambda_f$        ▷ Store the traffic value
38:         end for
39:     end for
40:     for $c \in C_s$ do        ▷ For each c^th replica of state variable $s$
41:         for $g \neq c \in C_s$ do        ▷ For each g^th replica of state variable $s$
42:             $\mathscr{P}_{cg} \leftarrow$ ShortestPath($n_c, n_g$)        ▷ Shortest path from $n_c \rightarrow n_g$
43:             for $e \in \mathscr{P}_{cg}$ do        ▷ For each edge in the path $n_c \rightarrow n_g$
44:                 $\hat{R}_{smne} = \hat{R}_{smne} + \alpha$        ▷ Store the state sync flow
45:                 $T_{\text{current}} = T_{\text{current}} + \alpha$        ▷ Update total traffic
46:             end for
47:         end for
48:     end for
49: return $[T_{\text{current}}, R'_{fce}, \hat{R}'_{smne}]$
50: end procedure
```

data flows R'_{fce} and for state synchronization $\hat{R}'_{smne}$, and the corresponding total traffic T' in the network. Lines 23 to 39 route the data flows from their source f_s to the destination f_d while traversing the replica c_b which has the minimum path length among all other replicas. For each flow, in lines 25 and 26, the replica c_b and the path $\mathscr{P}_{best}$ traversing it are initialized. Then for each replica (in lines 27-34), first, the shortest path $f_s \rightarrow n_c \rightarrow f_d$ is computed. n_c is the vertex for which $P_{scn} = 1$. If the path length $\mathscr{P}$.length is less than the previous minimum minDist in line 29, then the current path $\mathscr{P}$ is stored as the best path $\mathscr{P}_{best}$ and the current replica c as the best replica c_b. In lines 35-38, for each edge in $\mathscr{P}_{best}$, the routing as well as the traffic value is updated. Lines 40 to 48 generate flows from each state replica c to all the other state replicas g for state synchronization using the shortest path. This includes the synchronization flows $\hat{R}_{scge}$ being updated in line 44 for each edge in the path $\mathscr{P}_{cg}$ before updating the total traffic in line 45. If T' is less than the previous minimum, then the minimum traffic value and all the decision variables are updated (lines 14-15). In Phase 3 (line 17), a local search procedure perturbs the existing state replica locations. This proceeds by randomly selecting one node where a replica is located and moving it to one of its neighbor nodes. This new solution is then compared with the current one (line 13) after having evaluated the corresponding routing and total traffic.

2.4.3 Performance comparison

To highlight the effectiveness of the approximation algorithm we perform a detailed comparison of the PMR algorithm with respect to the ILP solution. The local search in PMR runs with $I = 1000$ iterations. In the case of small instances of the problem, we run an ILP solver, coded using IBM CPLEX optimizer [29], implementing the optimization model. We compute the *approximation ratio*, i.e., the ratio between the total traffic obtained by PMR and the optimal traffic obtained by the ILP solver. We consider two standard topologies for the network graph:

- *Unwrapped Manhattan* is a $\sqrt{N} \times \sqrt{N}$ grid.
- *Watts-Strogatz* [30] adds a few long-range links to regular graph topologies to reduce the distances between pairs of nodes and emulate a small-world model. It is generated by taking a ring of N nodes, where each node is connected to k nearest neighbors. In each node, the edge connected to its nearest clockwise neighbor is disconnected with probability p and connected to another node chosen uniformly at random over the entire ring. Thus, the final topology maintains the original

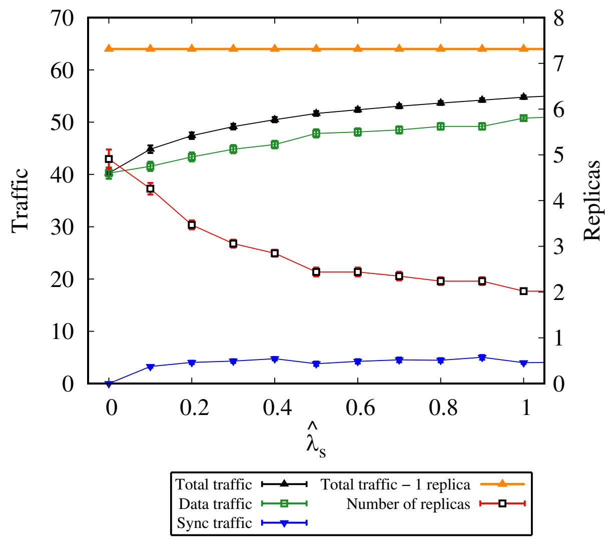


Figure 2.5: Optimal traffic and number of replicas in a 4×4 Manhattan graph for uniform traffic, using the ILP solver.

average degree k while being connected. In the following, we will use $p = 0.1$ and $k = 8$.

We utilize random traffic matrices with the number of flows equal to the number of nodes in the graph ($|\mathcal{F}| = N$) and with unity demands ($\lambda_f = 1$). The source-destination pairs for the flows were generated according to two models. In the case of *uniform traffic*, all the source nodes were associated with a random permutation of nodes as a destination; thus each node is a source and a destination of exactly one flow. In the case of *clustered uniform traffic*, we partitioned the nodes of the graph in half and generated a random permutation between the nodes of the same partition; thus all the flow are local within the same partition. All the results were obtained with 1000 different runs to get very small 95% confidence intervals (in all cases within 4.2% accuracy).

In Figure 2.5 we evaluate the effect of varying the number of replicas for state s and of the synchronization rate $\hat{\lambda}_s$, through the optimal ILP solver. We consider a 4×4 Manhattan graph and set $C_s = 7$. As expected, when increasing the traffic required to synchronize the replicas ($\hat{\lambda}_s$), the optimal number of replicas reduces, since the higher costs of synchronization compensates the beneficial effect of multiple replicas on the data traffic. Instead the synchronization traffic is almost constant, since, for a smaller

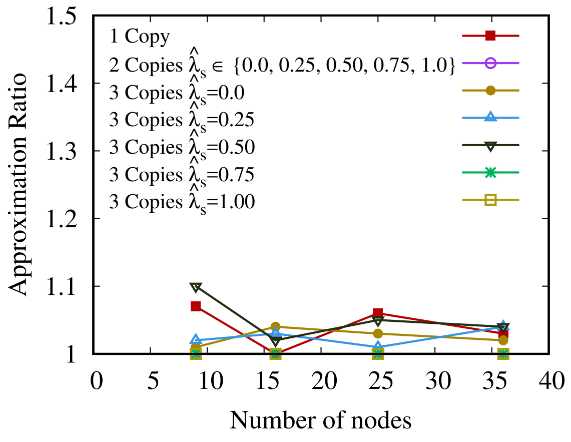


Figure 2.6: Approximation ratio of PMR in a Manhattan graph under uniform traffic.

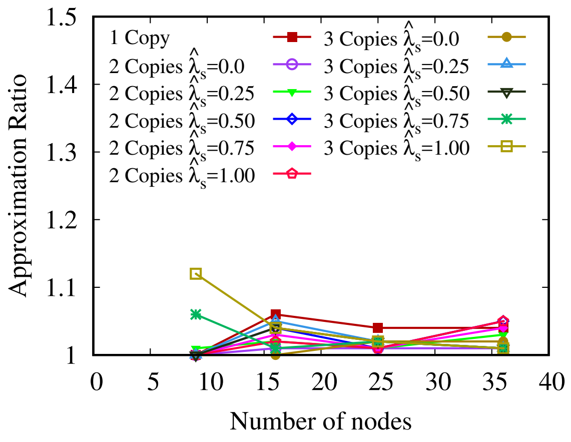


Figure 2.7: Approximation ratio of PMR in Watts-Strogatz graph under uniform traffic.

number of replicas, their relative distances grows, to "cover" a larger area of the network.

Figures. 2.6-2.7 show the approximation ratio for different number of nodes N, of replicas C_s and different values of $\hat{\lambda}_s$, under uniform traffic. The two graphs refer to Manhattan and Watts-Strogatz graphs, respectively. The approximation ratio in all cases is always ≤ 1.15, thus PMR approximates well the ILP solution. For larger graphs, we could not provide the results as the ILP solver is not computationally feasible.

2.5 Bounding inconsistency among states

To provide the correct functionality of the application, all replicas of a state must be consistent. Consequently, a read operation of any replica at any given time should eventually return the same result. The CAP theorem [31] states that, for a replication scheme, out of Consistency, Availability and Partition tolerance, only two properties can be picked at the same time. Considering that network failures may occur, partition tolerance cannot be left out of the design of the replication algorithm, leaving us with two main reference models:

- *Strong consistency.* This model privileges consistency over availability, meaning that a read operation on any non-faulty replica will return the most recent committed value (same for all replicas) or an error. This property is achieved at the cost of reduced availability due to the requirement of multiple interactions between replicas and is based on complex consensus protocols [32].

- *Eventual consistency.* This model privileges availability and results in instantaneous operations on all replicas with a considerably reduced protocol complexity. Although it introduces transient inconsistency, the latter can be seen as an error in the value of a local replica.

The choice between the two models depends on the level of tolerance of the considered network application in the presence of temporary inconsistencies between replicas of the same state. The majority of network applications require small packet processing latencies. Indeed, excessive latencies may lead to noticeable performance degradation in the case of real-time traffic and applications performing per-packet processing. This leads to the necessity of privileging high availability when state changes occur.

For highly mutable states, replication schemes based on strong consistency may lead to excessive latency due to the complex protocol needed to reach the consensus, ultimately leading to excessive commit delays which will preclude the correct functionality of applications. However, the majority of network applications operate on statistical network measurements and remain robust even in the presence of small errors for the value of the global state, making strong consistency less essential.

2.5.1 Replication delays and state inconsistency

LOADER does not impose any constraint on the adopted replication scheme, leaving to the programmer the freedom of implementing any replication protocol alongside with the suitable reconciliation scheme supported by network devices. It is generally true that replication schemes based on strong consistency are more complex and introduce larger latency to commit a value than the schemes based on eventual consistency. Thus, without loss of generality, in the following discussion, we focus on supporting eventual consistency. More in detail we focus on the case of optimistic replication realized with basic gossiping for which the precise sequence of concurrent writes on the different replicas is not affecting the application correctness.

In an eventual consistency scheme, each state is associated with a certain *replication delay* d_i, i.e., the maximum amount of time required to convey a state update to all of its replicas. Note that d_i corresponds also to the worst-case inconsistency time. Assume now that a state is replicated with period d_i^R (i.e., the inverse of the replication frequency). Let d_{nm}^P be the communication latency between network devices n and m, taking into account the propagation delay (we assume isolation of replication traffic from data traffic, thus negligible queueing delays). If $\mathcal{N}_i$ is defined as the set of nodes storing replicas of s_i, we can claim:

$$d_i = d_i^R + \max_{n,m \in \mathcal{N}_i} d_{nm}^P \tag{2.23}$$

The programmer is required to develop network applications by keeping in mind that different state replicas may suffer from inconsistency intervals during which their values may differ. To cope up with this, LOADER exposes to the programmer the possibility of defining an explicit inconsistency level for the replicated states. This is made possible by defining a level of *state inconsistency* inside the trigger function. The output of a network application is driven by the outcome of the trigger function, and for this reason, specifying the inconsistency level at the trigger function is sufficient to determine also the overall state inconsistency of the application.

We foresee two main inconsistency metrics which can be defined by the programmer: (1) *time obsolescence* ϵ_t and (2) *update error* ϵ_r. The former metric provides means of defining an upper bound on the time freshness of the state replicas and guarantees that at any given time any replica will contain a value not older than ϵ_t in time. The latter instead specifies the maximum admissible inconsistency in terms of uncommitted writes for any state variable, thus ensuring that the difference between all the replicated states does not exceed a number ϵ_r of state writes. The actual choice of the adopted inconsistency metric and the corresponding value is left to the programmer and it largely depends on the particular network application.

LOADER guarantees that the constraints specified by the programmer in terms of inconsistency metrics are satisfied. During the embedding phase LOADER first assigns replicas positions in the network so to minimize the maximum communication latency between any pair of replicas, i.e., minimize the second term of (2.23). Following this operation, two scenarios are possible. If time obsolescence ϵ_t is specified, then the replication periodicity d_i^R must be set such that:

$$d_i^R \leq \epsilon_t - \max_{n,m \in \mathcal{N}_i} d_{nm}^P \tag{2.24}$$

If instead an update error ϵ_r is specified, now d_i^R must be related to the rate of write operations on the state over time. To satisfy this constraint for a generic state x, it is sufficient to evaluate δ_τ^* as the maximum number of write operations performed on x over a time interval τ. Note that δ_τ^* depends on the specific meaning of the considered state and should be evaluated a priori. E.g., for a packet counter at an interface, it is obtained by the data rate divided by the transmission time of a minimum size packet. Let $|x|_t$ denote the number of writes for state x up to time t. By construction, it holds:

$$|x|_{t+\epsilon_t} - |x|_t \leq \delta_{\epsilon_t}^* \epsilon_t \tag{2.25}$$

By definition, we can bound (2.25) with ϵ_r and obtain:

$$\delta_{\epsilon_t}^* \epsilon_t \leq \epsilon_r \tag{2.26}$$

Based on (2.24), d_i^R is chosen such that:

$$d_i^R \leq \frac{\epsilon_r}{\delta_{\epsilon_t}^*} - \max_{n,m \in \mathcal{N}_i} d_{nm}^P \tag{2.27}$$

Note that in the case of states which permit a definition of absolute state error based on some norm (e.g., scalars, arrays, graphs), knowing the nature of write operations permits to translate the update error into *absolute value error*. Assuming that a write operation can alter the state by a maximum amount, it is possible to rewrite δ_τ^* in terms of absolute state variation and derive the temporal constraints following the same above formulation.

Listings 2.2, 2.3, 2.4 provide an example of the definition of a trigger function in LOADER for a simple scenario (i.e., sum of two states). The listings show respectively a trigger function with a given value of time obsolescence (ϵ_t), a trigger function with update error (ϵ_r) and a trigger functions which does not tolerate any state inconsistency.

```
r = ReductionFunction(states=[s1, s2], operation=stateSum)

tr = TriggerFunction(s0=r.Result(), trigger=(r.Result() > 0),
         inconsistencyLevel=TimeObsolescence(2, ”ms”))
```

Listing 2.2: Example of trigger function with time obsolescence ϵ_t equal to 2ms.

```
r = ReductionFunction(states=[s1, s2], operation=stateSum)

tr = TriggerFunction(s0=r.Result(), trigger=(r.Result() > 0),
         inconsistencyLevel= UpdateError(10))
```

Listing 2.3: Example of trigger function with update error ϵ_r equal to 10 writes.

```
r = ReductionFunction(states=[s1, s2], operation=stateSum)

tr = TriggerFunction(s0=r.Result(),trigger=(r.Result() > 0))
```

Listing 2.4: Example of trigger function without inconsistency (i.e. replication is not permitted).

2.5.2 Replication traffic generation

To replicate a state, network devices generate by themselves update packets, based on the required replication periodicity d_i^R. This generation is not currently supported in off-the-shelf hardware for stateful switches as a fundamental primitive, since, for performance reasons, packet generation events are triggered only by packet arrivals. Depending on the actual hardware, we foresee different solutions which provide a way of generating new packets without any hardware modification of current off-the-shelf chipsets, which are briefly discussed in the following.

Controller-triggered updates

The generation is triggered by the controller. In the case of periodic updates, the controller sends periodic trigger messages to the network devices, where they are processed and used to generate the update packets, by acting upon the reception of the trigger messages. Despite its simplicity, this approach has many limitations. First, the required control bandwidth from the controller to each switch can become relevant for small update periods. Second, the controller is loaded with an additional task, impairing its scalability.

Traffic-triggered updates

The generation is triggered directly by the reception of data packets received at any interface of the network device. This permits to self-adapt the amount of replication traffic on the dynamicity of the states, whenever these depends on the arrived traffic. In terms of implementation, the update message is generated by cloning a data packet and then modifying it to carry the update value. For what concerns stateful SDN switches, we consider two possible approaches to regulate the replication traffic rate based on native internal primitives:

- packet period p. By keeping a packet counter, a new update packet is generated every p received packets, i.e., $d_i^R \leq p/r_{\text{min}}$ where r_{min} is the minimum packet arrival rate over the whole switch. This can be used in (2.23) to choose p and satisfy the given inconsistency metrics. Intuitively, the update rate is proportional to the arrival rate of data packets which may suit well particular traffic-monitoring applications. On the other hand, for other applications this approach may lead to shortcomings since in the absence of transit traffic no updates will be generated.

- time period τ^R. An update packet is generated at the first packet arrival after τ^R time and thus $d_i^R \leq \tau^R + 1/r_{\text{min}}$. This can be used in (2.23) to choose τ^R and satisfy the given inconsistency metrics. Intuitively, this case results in periodic updates, i.e., a fixed replication rate approximately independent from the traffic.

In terms of the message format, the replication packet must carry the state identifier, the state value and the identifier of the switch originating the update. All identifiers can be predetermined by the controller at the time of application instantiation. This mechanism guarantees the state's uniqueness while providing flexibility in terms of the state format encoding. Finally, to route properly the replication traffic, the position of each application primitive in the network is considered. LOADER exploits the network knowledge at the controller to install updates forwarding rules through a Steiner tree, either shared across all the states or one specific for each state.

2.6 LOADER implementation

To prove the feasibility if the LOADER approach, we developed a lightweight implementation of the framework. We integrated LOADER into ONOS v1.14 while using P4 [6] and Open Packet Processor (OPP) [7] switches for the data plane. The choice of these two distinct data plane architectures aims at showing the generality of the proposed approach, which results to be independent of the specific type of devices adopted in the network.

2.6.1 Control plane implementation

LOADER has been integrated inside the ONOS controller in the form of an ONOS application with custom control logic overriding the default controller behavior.

Application definition

We consider a set of predefined application elements supported by the switches. This assumption permits to drastically simplify the implementation of the application definition phase inside ONOS. In particular, we specify each application element by means of predefined ad-hoc classes for each type of application element, based on the primitives supported by the switches. Thus no interaction with the resource manager of ONOS is performed.

Application elements embedding

For the purpose of this work, we consider a homogeneous network with devices composed of programmable switches having the same type and amount of resources. Since the algorithm to solve the optimal embedding problem is out of the scope of this work, we consider the following simple embedding scheme, inspired to the one proposed in [21]. The position of each replicated primitive inside the network is determined by considering the betweenness centrality [33] of each network device, weighted by the amount of traffic flowing through it. The main idea is to privilege the devices that are traversed by most of the traffic. Furthermore, the number of replicas of each primitive is fixed a priori and not optimally chosen. The replication traffic between the different replicas is routed on a single Steiner tree shared across all the replicas. This permits to reduce both the amount of replication traffic and the amount of flow table entries.

State identification

LOADER requires a unique identifier for each state, to guarantee the correct processing of update packets. Similarly to other network programming frameworks [19] LOADER assigns a unique identifier to each state during the application compilation phase. For replicated states an additional identifier is assigned to distinguish between different replicas of the same state.

2.6.2 P4 implementation

P4 [3] is a novel stateful data plane programming language designed for next-generation SDN switches. The main motivation behind the development of P4 is to provide greater flexibility for the data plane by mitigating issues present in existing technologies. In order to achieve such a level of elasticity P4 was built around four main concepts:

- **Protocol-independence**: Among the main novelties of P4 in respect to traditional SDN approaches is the introduction of a programmable parser and deparsed directly inside the language specifications. This provides means of defining matching actions for all fields present in currently available protocol headers. Furthermore, it permits to define custom protocol headers ex-novo and even extend the matching rules to the payload of processed packets.

- **Target-independence**: P4 leverages its functionalities on a flexible compiler which, given the specifications of the target hardware, is able to efficiently translate P4 code in hardware instructions and map it to the underlying architecture.

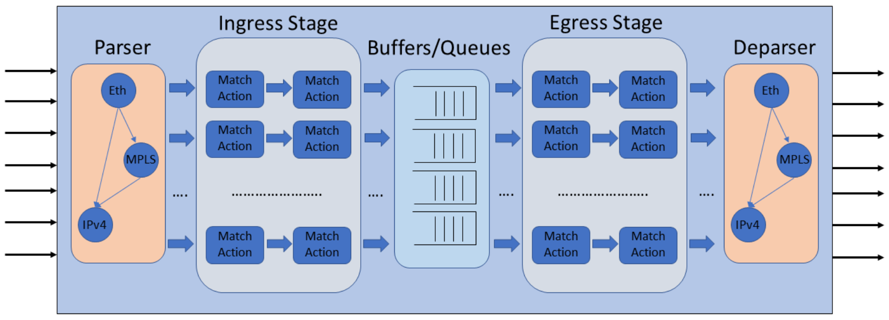


Figure 2.8: High level abstraction of a P4 Simple Switch Architecture [34] target.

This effectively enables the possibility of having a unified common high-level abstraction for all of the P4-enabled devices.

- **Reconfigurability**: While traditional SDN switches are capable of changing only the forwarding behavior, P4-enabled switches are capable of changing the entirety of the packet processing pipeline without any service disruption. This effectively enables plug-and-play logic at runtime, ultimately leading to more flexible and dynamic network management schemes.

- **Stateful operations and externs**: Although not being among the main motivation behind P4, the inclusion of stateful elements inside the language represents a big factor behind its success. P4-enabled devices must be capable of supporting stateful primitives such as registers, counters or meters. Furthermore these primitives must be accessible during packet processing, thus operating at line rate. While such a feature alone enable a rich set of novel functionalities, the presence of extern modules provides a truly flexible abstraction for custom extensions. Indeed, extern modules permit to expose any hardware-specific functionalities inside the P4 language, thus providing the possibility of plugging-in vendor-specific functions such as encryption or packet inspection directly into the packet processing pipeline.

Figure 2.8 depicts a high-level overview of a processing pipeline of a P4-enabled switch employed in our evaluation. Before entering the main processing pipeline all packets go through the programmable parser. We program the parser to support a custom header type used to identify LOADER packets and to discriminate among different content types carried within. Following the parsing stage packets are moved into the ingress stage which, together with the egress stage, implements the network policies in the form of match-action rules combined with stateful primitives and actions. After processing and before departing from the switch packets go through a deparsing stage which suitably modifies the protocol headers stack by appending LOADER-specific headers to the packet if needed.

To provide connectivity between ONOS and P4 switches (version 1.1), we exploited P4Runtime [35]. At the time of this work, P4Runtime implementation in ONOS v1.14 performed only basic flow tables manipulations without providing support for features such as runtime pipeline modification and manipulation of extern objects such as registers and counters. Due to these limitations, we implemented the required primitive data structures and the replication control logic directly in P4 instead of letting the controller push them to each switch at application creation time. However, the controller is left with the possibility of activating or deactivating application elements inside a switch, which is equivalent to pushing new logic.

Replication traffic format

Replication traffic is transported through packets that are formatted with a custom header carried by Ethernet packets, identified by an unused protocol type (`LOADER_ETHTYPE`) in the Ethernet header. We leverage P4 to define custom packet formats and we implemented LOADER header format directly inside the programmable parser.

Listing 2.5 shows the full header format of LOADER packets. As previously mentioned, all identifiers are assigned by the controller during application initialization. Being `srcSwID`, `stateID` and `replicaID`, respectively, source switch, state, and replica identifiers, which are required to correctly interpret and process the update packets at the destination switches. On the other hand, the inclusion of `dstSwID` permits to implement more sophisticated replication schemes instead of employing ours based on shared spanning trees. In our experiments we implemented a broadcast transmission among all switches holding the replicas and for this reason `dstSwID` field remained not utilized. The `stateValue` field carries the actual value of the replicated state and its length is upper bounded by a constant number of bit, i.e., `STATE_MAX_WIDTH`. Finally,

```
header LOADER_t {
  bit<32> srcSwID;
  bit<32> dstSwID;
  bit<32> stateID;
  bit<32> replicaID;
  bit<STATE_MAX_WIDTH> stateValue;
  bit<16> L3ProtocolType;
}
```

Listing 2.5: LOADER header definition in P4

```
state parse_LOADER {
  packet.extract(hdr.LOADER);
  transition select(hdr.LOADER.L3ProtocolType){
    LOADER_ETHTYPE : parse_LOADER;
    IP_ETHTYPE : parse_IP;
    default : accept;
  }
}
```

Listing 2.6: LOADER parser implementation in P4

the `L3ProtocolType` field permits to attach LOADER packets to transit packets, i.e., to piggyback replication information on data traffic.

We generate nested LOADER headers to carry multiple state updates in a single packet. This functionality is depicted in Listing 2.6 which shows the implementation of the LOADER protocol parser. Although in this work we opted to define a custom LOADER header, replication traffic transport can be also implemented by employing Inband Network Telemetry (INT) format [36] defined by the P4 Language Consortium.

Generation of periodic update packets

Commercial implementations of stateful switches generally do not support the generation of self-triggered events, precluding the possibility of employing periodic updates. However, in conformity with their purpose, switches are able to execute routines during packets reception and departure. Such routines may be related to simple packet processing up to more complicated user-defined routines in programmable switches. This behavior can be exploited to provide a simple mechanism to approximate a periodic traffic generation without hardware modifications.

We exploit traffic-triggered updates, as described in Section 2.5.2, in which the temporal periodicity d_i^R is obtained as follows. During the execution of a replication routine, the current timestamp t_{clk} is saved as t'. For each subsequent incoming packet we check the value of the internal clock t_{clk} and compare it against the expected execution time of the routine, i.e., against $t' + d_i^R$. If $t_{\text{clk}} \geq t' + d_i^R$ a new replication routine is executed generating an update packet and t' is updated. Consequently, the first packet arriving after d_i^R time will trigger the generation of the update packet.

The replication routine generates and transmits a state-update packet filled with the state-related information. To generate these packets we employ the packet-cloning extern provided in P4 v1 model [34]. Once the update has been triggered by an arriving packet, such packet is cloned to the egress port that has been assigned to it by its prior processing. Subsequently the original packet undergoes a transformation which substitutes its original header with the LOADER header filled with all the information related to the state which needs to be updated. At the same time the payload of the packet that triggered the update is dropped. Following this operation, the newly created LOADER packet is transferred to the corresponding output queue without undergoing further processing. Since the triggering packet needs to be fully processed at the time of cloning, this functionality, which is illustrated in Listing 2.7, resides at the very end of the ingress processing pipeline. In this way the replication traffic generation routine does not impact in any way the transit packets.

Replication traffic routing

The generated replication packets are transmitted on one or more egress ports following a Steiner tree shared among all replicas. The distribution tree consists of a mapping *(Switch, PortList)* which assigns to each switch of the Steiner tree the set of ports connected to the corresponding links. All newly generated or transit LOADER packets match against a specific match-action table which sends a copy of the packets for each port specified in *PortList*. To avoid loops for transit LOADER packets, at the egress stage the original ingress port of each packet is compared against the current egress port. If the two ports are the same, the packet is dropped. This mechanism permits to keep the amount of flow entries related to LOADER routing as low as one entry per state per switch.

Both the P4 switch and the LOADER framework implementations are publicly available at the LOADER repository [37].

```
if( meta.LOADER_meta.state == UPDATE_NEEDED ){
  clone_pkt_to_egress(sm.egress_spec);
  fillLOADERHeaderTable.apply(meta.LOADER_meta.state_id);
  set_state_update_time(meta.LOADER_meta.state_id);
}
```

Listing 2.7: Generation of replication packet in P4

2.6.3 OPP implementation

Open Packet Processor (OPP) [7] is a programmable data plane abstraction in which Extended Finite State Machines (EFSM) are used to model stateful forwarding algorithms. The OPP machine model extends the match-action tables pipeline model assumed by OpenFlow. Such tables are substituted with *stages*, which can be either stateless or stateful. A stateless stage is in fact an OpenFlow-like match-action table. The pipeline processes packet headers to define corresponding forwarding behaviors. The packets are processed by the ingress pipeline, which is composed of a parser stage and several stateless and stateful blocks after the processed packets go into the internal switch memory that holds the packet queues.

An OPP application requires the definition of the following components:

- *Lookup/update extractors*: these two blocks are configured by defining a combination of packet fields that are used to retrieve/update flow state information.
- *Conditions*: conditions are arithmetic comparison operations of global/local variables and packet header fields; conditions are matched in the EFSM table along with the flow state and packet fields.
- *EFSM table*: programming the EFSM table requires the definitions of a set of EFSM entries formed by a *match* section (as defined in the list item above) and an *action* section, which defines the state transition and a set of packet actions (drop, push header, forward, etc.) and update functions over the local registers. The EFSM table is configured as a standard OpenFlow table and is usually realized in ASIC switches using TCAMs.
- *Global data variables*: OPP global variables are independent of a particular flow and can be used in the condition block.

The OPP protocol used between the OPP switches and controllers is a modified version of OpenFlow 1.3 standard, extended to support the configuration of an OPP pipeline. In particular, the configuration of the lookup/update extractors and the conditions are realized with two new experimental OpenFlow messages that carry the list of packets fields to be extracted from the packet headers and the arithmetic operations whose results are matched from the EFSM tables. Furthermore, the configuration of the EFSM table requires the extension of the OpenFlow FLOW_MOD message to support new match fields (conditions, and flow states) and new actions (state transition and data variables updates).

The OPP switch implementations is publicly available at the OPP source repository [38].

Replication traffic format

In the OPP prototype, we decided to format the replication packets by employing the 20-bit labels provided by the MPLS protocol. This design choice was taken for mainly two reasons: the MPLS header is a widely used protocol supported by most of the Internet nodes, and in our OPP implementation it was simple to handle such an encapsulation header since adding a custom protocol would have resulted in a static implementation of the parser code to support a custom header. The Switch ID is encoded in the source and destination fields of the overlay IP protocol, assigned to each node by the control plane at configuration time. The State ID is inserted in the Experimenter field of MPLS (3 bits) and as such, confined to a maximum of 8 different states supported. Finally, the MPLS label (20 bits) carries the State Value.

Generation of periodic update packets

OPP does not support a time-based generation of periodic events so, as in the P4 implementation discussed in Section 2.6.2, the generation of time-related events is triggered only by the reception of packets. To emulate a timer expiration we use per-flow registers to store the time difference between packet arrivals. This difference is then compared with the replication period d^R. The result of this comparison is then matched by the EFSM table, resulting in the execution of the corresponding action present in the table entry.

To generate the replication traffic, we implemented two approaches. In the first approach, we clone the arrived packet to generate an update packet. When the packet generation event is triggered, the cloned packet is attached with an MPLS header containing the correct state information, while the original packet continues its normal processing. In the second approach, the update packet is instead generated ex-novo by using a predefined template that already contains the MPLS header. The header fields are then modified according to the state information to be written in the packet. Differently from the first approach, this one has the advantage of reducing the size of update packets since they do not carry any data above the MPLS layer.

Replication traffic routing

As discussed in Section 2.6.2 for the P4 implementation, the mapping switch-to-output port to route the replication traffic is statically assigned by the controller at configuration time. In such a way, the forwarding decision is taken through the normal OpenFlow stateless match-action strategy.

2.7 Implementation and evaluation of network applications with LOADER

As a proof of concept, we used the LOADER programming model to developed a simple yet significant application for the *distributed detection* of Distributed Denial of Service (DDoS) attacks, denoted as DDoSD. The main idea of the distributed detection is to exploit the typical temporal correlation between the increase of traffic across all the network devices at the border of the network, due to the distributed nature of the attack. Clearly, the correlated traffic increase across the edge routers is a much more reliable way to detect an attack with respect to monitoring the traffic on a single network device only. Consequently a network application performing DDoSD must be able to capture this sudden increase in the network traffic.

With traditional SDN approaches, the controller is involved in the detection process by being notified about the transit packets by switches. This leads to a large overhead in terms of traffic and of detection latency. Instead, LOADER enables a *distributed* detection process operating directly at the switches, without any controller involvement. Furthermore, the actions to counter the attack are executed in a distributed way, by each network device involved in the detection.

In a single replica approach (i.e., in the absence of LOADER) the DDoSD application would require all the traffic entering the network to traverse a single switch holding the state monitoring the incoming traffic. Thus the network load would grow, increasing the congestion, and could not be compatible with some traffic management schemes (e.g., load balancing) that require to control the routing arbitrary within the network.

LOADER instead permits to replicate the entire DDoSD application over multiple switches, thus minimizing the data overhead over the whole network. At the same time, LOADER introduces an overhead in terms of replication traffic, whose amount depends on the allowed inconsistency level. The replication traffic will be evaluated experimentally for the DDoSD application in Section 2.7.3.

Notably, DDoSD is robust to possible transient inconsistencies between the values of total traffic estimated at each switch, thus employing an eventual consistency replication scheme will not create noticeable degradation due to estimation errors of replicated states.

As shown in Figure 2.9, we consider a large network (e.g., an Autonomous System - AS) connected to other networks (e.g., other ASs) through different edge routers and the attack targets a set of internal servers. Since the definition of a realistic DDoSD algorithm is a well-known problem in the literature [39] and it is completely out of the scope of this work, we employ a simple proof-of-concept threshold-based detection scheme, which demonstrates the correct operation of the replication mechanism and can be used as a foundation for more sophisticated DDoSD algorithms.

2.7.1 Network application definition

The total traffic entering the whole network and directed toward the targeted servers is defined as the sum of the inbound traffic over each edge router (`SW1`-`SW4` in our reference topology). Based on the value of the inbound traffic the network application must perform some retaliation to counteract the DDoS attack. Consequently it is straightforward to map this kind of application to a LOADER application as described in the following.

- **States**: Given N edge routers, we define s_i as the average rate of inbound traffic traversing the border router i, with $i = 1, \dots, N$. As a monitoring target, we employ the rate of incoming SYN packets directed towards the internal servers.
- **Reduction function**: The reduction function employed by the application is composed of a single primitive action, namely $r_1 = \text{sum}()$. Consequently, the output of the reduction function is defined as $s^o = \text{sum}(s_1, \dots, s_N)$.
- **Trigger function**: Following the previous discussion, we define the threshold function simply as a simple comparison of s^o against a predefined threshold. Thus, a DDoS attack is detected locally at each switch if s^o is larger than a given threshold, above which the attack is considered as detected. The threshold is determined with standard test-based statistical methods.
- **Activity function**: We employ a simple activity function which notifies the controller once the application has been triggered.

The implementation of the DDoS application with LOADER programming model is available in Appendix A.

2.7.2 Implementation

The considered DDoSD scheme has been implemented on top of two different programmable data plane platforms: (1) $P4_{14}/P4_{16}$; (2) OPP. Furthermore, the definition of the DDoSD application was performed inside ONOS with LOADER abstraction which permits to automatically offload and configure the developed network application.

Control plane implementation

We implemented basic LOADER functionalities related to this particular use case inside ONOS. We employ the routing algorithms and the embedding mechanism based on the betweenness centrality discussed in Section 2.6.1 with a maximum amount of admissible replicated states equal to C. We assume a sufficiently large amount of resources inside switches, thus permitting function co-location with consequent replication of all application elements.

Data plane implementation with P4

Our prototype is developed and tested in a virtual environment using Mininet [40] and P4-enabled virtual switches targeting using the V1 Model and using the Simple Switch Architecture [34]. We estimate the rate of incoming TCP SYN packets by employing a sampling window equal to δ. Let $r_k(t_n)$ be the estimated rate in the time interval $(t_n - (k+1)\delta, t_n - k\delta]$ with $t_n = n\delta, n \in \mathbb{N}$. The average rate is estimated at each switch i as

$$s_i(t_n) = \frac{1}{w}\sum_{k=0}^{w-1} r_k(t_n) \tag{2.28}$$

and represents the local state to be shared across all the other border routers, coherently with the description of Section 2.7.1. In particular, w is chosen as a power of 2 due to the hardware limits in P4 switches imposed to the types of operations that can be implemented, i.e., shift operations are supported, divisions are not [41]. Notably, The w most recent samples of the estimated rate are stored in a circular buffer. Replicated states are instead saved in dedicated registers.

Data plane implementation with OPP

The OPP implementation requires a sequence of three stages: stage 0 extracts the state from update messages; stage 1 stores the state from the metadata notified by the previous table, performs monitoring and detection and generates update messages; stage 2 performs simple L3-forwarding. Stage 0 represents the stateful processing core of replicated states. The processed flows are identified by the IPv4 destination addresses of the target servers. Stage 0 also considers one flow data variable containing the switch-local state and the $C-1$ variables storing the replicated states. Switch-local state s_i is computed by employing a hardware-implemented Exponential Weighted Moving Average (EWMA) counting the number of TCP SYN packets in a given preconfigured time window.

2.7.3 Experimental evaluation and validation

We configure a Mininet-based emulation environment deploying the topology shown in Figure 2.9, where, for the sake of simplicity, each cluster and each AS is represented

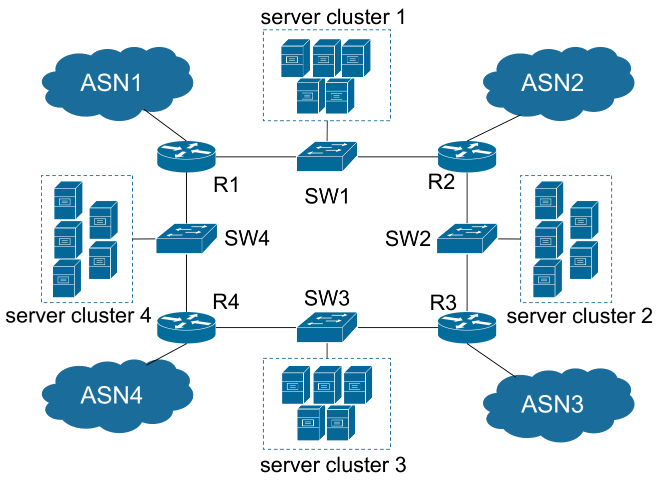


Figure 2.9: Reference topology for DDoS Detection use case.

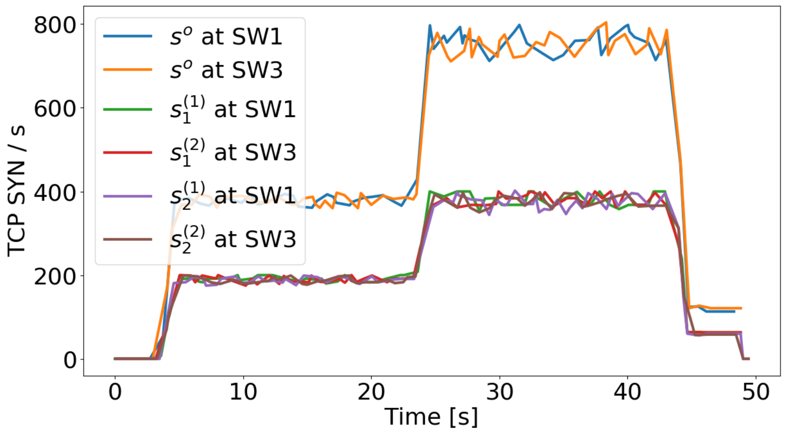


Figure 2.10: Temporal evolution of the local, remote and global states for the stateful switches in case of 2 replicas for the global state in P4 implementation.

by a Mininet host. To simulate the DDoS attack, we use `hping3` tool to send TCP SYN requests from all ASs to all internal servers. In each experiment, during the first 20 seconds, we send the request at a slow rate, and then we increase the rate of all senders in such a way to trigger the execution of the activity function. We consider experiments with varying C: (i) single replica embedded in SW1 ($C = 1$), (ii) 2 replicas ($C = 2$) embedded in SW1 and SW3, and (iii) 4 replicas ($C = 4$) embedded in SW1, SW2, SW3, SW4. We repeated the experiments to achieve negligible 95% confidence intervals if shown in the plots.

Figure 2.10 shows the evolution of application states s_i alongside with the evolution of s^o for the case of 2 replicas, implemented in P4. Identical results are obtained with OPP. As expected, the values of s^o evaluated at SW1 and SW3 are coherent, and permit a contemporary detection of the DDoS attack in the two switches, without any interaction with the controller. This experimental result validates our proposed implementation for both P4 and OPP.

In Figures 2.11, 2.12 we show the average utilization of the links present in the ring topology connecting all switches, for different values of C, with both P4 and OPP implementations. Clearly, for one replica (i.e, single replica approach) the load on the link is greatly unbalanced and in general higher for all the links. By increasing the number of replicas to 2, the load of the data traffic decreases by a factor of 1.6 both in P4 and OPP and is much better balanced across the links. The slightly different values depend on the different mechanisms adopted for triggering the update event by the incoming traffic: in P4 the update rate depends on the traffic, whereas in OPP it is independent. Adding two other replicas reduces the data traffic by around 20% in both implementations, but now the replication traffic becomes more relevant due to the higher number of replicas. Indeed, the fraction of update packets increases from 14% (for 2 replicas) to 24% (for 4 replicas) in P4 and from 11% (for 2 replicas) to 23% (for 4 replicas) in OPP. Thus, the two implementations behave very similarly and show a beneficial effect on the overall traffic in the network due to the presence of multiple replicas.

2.7.4 LOADER-induced overhead

As previously discussed and shown in Figure 2.11 and Figure 2.12, LOADER adds some network overhead in the form of added synchronization traffic. The actual characterization of the amount of synchronization traffic highly depends on the network topology and the definition of the state. The impact of those factors has been exhaustively analyzed in our previous work [21].

From the point of view of device resource utilization, the amount of memory required to manage replicated states scales linearly with the degree of replication. Specifically, every switch must store their own local state values and the remote state values. Alongside those states, switches must also store an aggregate value combining local and

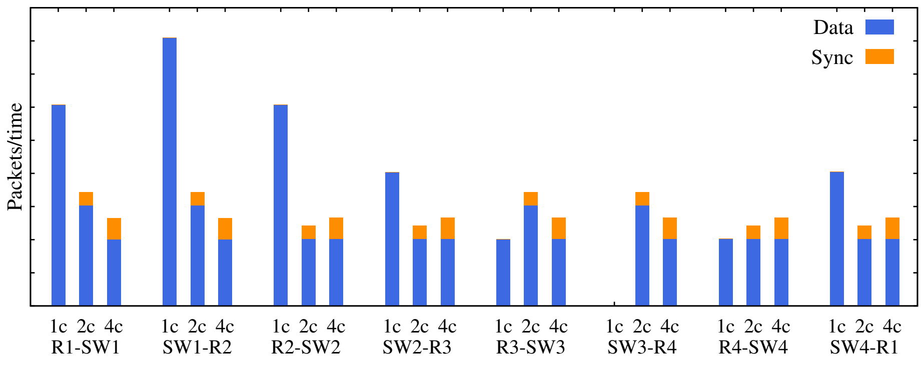


Figure 2.11: Average link utilization for data and for replication traffic in case of $1, 2, 4$ replicas for global state in P4 implementation.

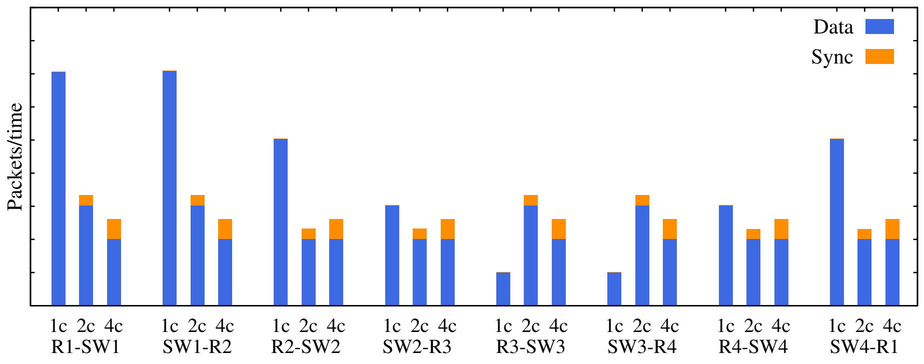


Figure 2.12: Average link utilization for data and replication traffic in case of $1, 2, 4$ replicas for global state in OPP implementation.

remote states into a global state, which is then used as input to the reduction function. This translates into a total requirement of $A(C + 1)$ bits of register memory per switch, being A the size in terms of bits of a generic state to replicate (e.g., $A = 32$ bits in the case of simple counters considered in the DDoS use case).

2.7.5 Other applications enabled by LOADER

Although being significant, the DDoS use-case does not highlight the whole versatility of the proposed programming models. For this reason, in the following we describe some examples of network applications (previously described in Table 2.1) which are shown to benefit from state replication. We show how those applications can be implemented with LOADER by providing their elements mapping and a code example for each of them. The actual implementation of those applications with the LOADER programming model is presented in Appendix A.

Distributed rate-limiting

In [14] the authors propose a network-wide global token bucket. Similarly to a local token bucket, a global one permits to rate limit all the incoming traffic in a given network thanks to a network application performing probabilistic dropping at the edge routers of the network. However, differently from a local one, a global token bucket involves an instance of the same token bucket run independently at each border router and using a single shared state accounting for the *total* inbound traffic.

This kind of application can be easily mapped to LOADER by considering the DDoSD scheme and by changing only the trigger and the activity functions as follows:

- **States**: Given N edge routers, we define state s_i as the average rate of inbound traffic traversing edge router i, with $i = 1, \dots, N$.
- **Reduction function**: The reduction function performs a sum operation among all local state s_i with $s^o = \text{sum}(s_1, \dots, s_N)$.
- **Trigger function**: In order to perform probabilistic dropping the trigger function must invoke the activity function proportionally to the rate of the incoming traffic and the desired rate.
- **Activity function**: Identically to the DDoSD case, the activity function must perform dropping of incoming packets whenever invoked as to guarantee that the total incoming traffic is less than a given threshold.

In Figure 2.13 we show an example of the distributed rate-limiting application in action. We create two flows: Flow 1 from AS 1 directed towards server cluster 1 and another flow from AS 3 directed towards server cluster 3. We consider shortest path routing and place state replicas in SW1 and SW3. Flow 1 starts at time 0 with a rate of 5 Mbps while flow 2 starts with an offset of 20 s and with the same rate. Although the flows do not cross each other at any point in the network when flow 2 starts both of them are rate limited to a predefined aggregate 8 Mbps threshold. Note that oscillations in throughput are due to the adopted probabilistic dropping scheme.

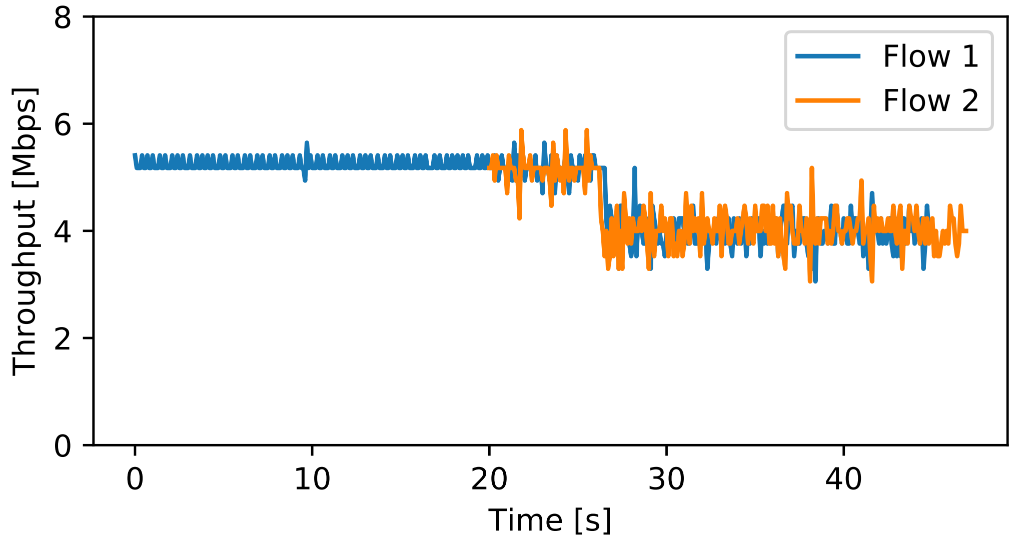


Figure 2.13: Distributed rate limiter with two flows at different edges of the network.

Link-aware load balancing

LOADER can find its applicability also in the context of DCNs, such as for the case of link-aware load balancing. More details about DC load balancing will be given in Chapter 3. In [13] the authors propose a load balancing scheme for data center networks, based on the congestion level of individual links from the source ToR switch to the destination ToR switch (refer to Figure 3.1 for a reference DC topology). Source ToR switches keep track of local uplink congestion and of the downlink congestion from each spine switch to the destination ToR. When a new flow starts, the source ToR switch selects a path to the destination by considering the one that minimizes the maximum congestion on the whole path, i.e., local uplink congestion and the downlink congestion on the spine.

For the sake of simplicity, we present a reduced version of the application with some omitted details and by assuming that the application targets a single ToR switch with P spine switches. The application can be easily extended to many ToR switches by simply instantiating multiple instances of the same application and the states related to downlink congestion must be shared across multiple ToR switches.

This network application can be mapped to LOADER as follows:

- **States**: Given a ToR switch, we define state s_i as the average load on the P uplink ports, with $i = 1, \dots, P$. Additionally, we define state s_j as the average downlink load on the port leading to the destination ToR switch of spine switch $j - P$, with $j = P + 1, \dots, 2P$.

- **Reduction function**: The reduction function is composed of two primitive actions, namely $r_1 = \max()$ and $r_2 = \arg\min()$. Consequently, the reduced version of the states is obtained as: $s^o = \arg\min(\max(s_1, s_{P+1}), \dots, \max(s_P, s_{2P}))$

- **Trigger function**: Differently from previous use cases, the trigger function in this network application triggers the activity function each time a new s^o is obtained and does not require any additional checks.

- **Activity function**: The activity function involves a simple insertion of a new per-flow forwarding rule for each new flow based on the outcome of the reduction function.

2.8 Discussion

In this Chapter we propose a novel framework, namely LOADER which enables an effortless development of distributed applications and we show that distributed network applications can be beneficial for the network performance and can be efficiently implemented in high-performance programmable stateful switches.

While enabling novel application types, LOADER is capable of substantially reducing the traffic overhead introduced by data flows. However, in this work it was not possible to increase the testbed size due to its demand of computation resources. Yet, results from Section 2.4.2 show that, for medium-sized topologies, distributing network applications can lead to a substantial decrease in the data traffic overhead introduced in the network. Indeed, scaling from a single replica (i.e., traditional SDN approach) to just two replicas can lead to a potential decrease in the total traffic introduced by the related flows by a factor 1.6.

Chapter 3

To Sync or not to Sync: Why Asynchronous Traffic Control is Good Enough for Your Data Center

Part of the work presented in this chapter has been published in:

- Sviridov G, Bianco A, Giaccone P. ”To Sync or Not to Sync: Why Asynchronous Traffic Control Is Good Enough for Your Data Center.” In *IEEE Global Communications Conference (GLOBECOM).* 2018.

Data centers (DC) are constituted by a large concentration of servers, providing computing and storage resources, typically to run cloud computing services and real-time applications. Servers are connected through a Data Center Network (DCN) that interconnects them to the rest of the data center and provides access to the Internet, as shown in Figure 3.1. Interestingly, most of the data traffic within a DCN is local, mainly due to the high exploitation of parallel processing, of the redundancy in the data storage and of the internal control mechanisms. Thus, performance perceived by the users heavily depends on the performance of the data transfers within the data center.

In recent years, the demand for low latency and high bandwidth in the DCN has grown dramatically, partially compensated by a scalable design of data centers, exploiting multi-layer Clos-based topologies [42]. The memories internal to the switches have

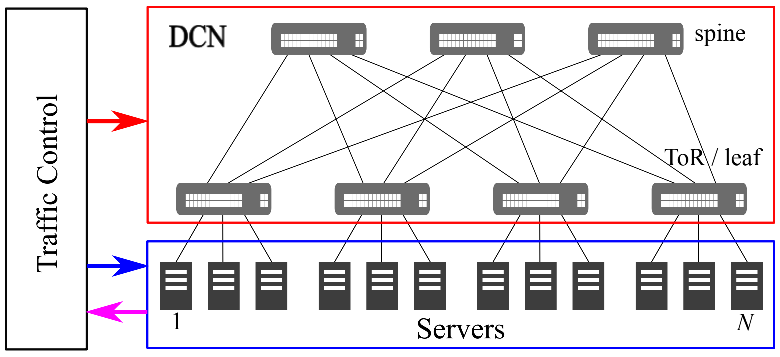


Figure 3.1: Architecture of a multilayer data center with a central traffic control for packet scheduling and routing

instead kept growing slowly. Consequently the ratio of RTT over bandwidth for the intra-data center communications have kept shrinking. As a consequence, under such conditions, congestion control schemes in standard TCP protocols, which were originally tailored to WAN/LAN with high/medium RTT and medium/low bandwidth, are not able to converge fast enough, ultimately leading to poor performance. This is also exacerbated by the small size of most of the traffic flows. Indeed, the majority of flows do not last long enough to trigger congestion control mechanisms, which were originally designed for long-lasting flows.

This fact has motivated the networking community to devise ad-hoc *proactive transport schemes*, exclusively designed for DCNs, able to minimize the latency within the data center. Being proactive, those schemes require additional knowledge about what is happening in the network in order to operate correctly. Such a concept is similar to SDN which was exhaustively discussed in 2. Indeed, in 2 we showed that LOADER finds excellent applicability in scenarios such as wide-area networks (Section 2.7.5) and DCNs (Section 2.7.5) by conveying global state information directly to network devices required to take performance-critical decisions. Yet, as discussed in Section 2.5 in contexts characterized by high network dynamics the effectiveness of LOADER may suffer. Such is the case of DCNs which have a limited physical extension and small propagation delays, ultimately leading to high state mutability and, consequently, high state inconsistency for potential distributed network applications. In such a scenario, keeping all states at a central entity would lead to a more accurate representation of the network, although penalizing the scalability of the approach.

Centralized control provides the possibility to globally monitor the network state in real-time and to optimize the packet transmissions in a proactive way[1]. An example of such an approach has been proposed by the authors of Fastpass [43] who define a centralized scheduler for the DC capable of performing fine-grained proactive coordination of the data transfer between pairs of servers. The main idea is to abstract the DCN as a logical "big switch" in which each server is connected to a single port. Fastpass aggregates information about the offered load of each server and performs per-packet scheduling in such a way to ensure that at any time at most one packet is transferred to and from each server. This guarantees almost no congestion within the DCN switches and very low, close to ideal, DCN crossing latencies, accounting only for the store-and-forward delays and for the link propagation delays. However, this property comes at a cost: mainly the requirement of DC-wide synchronization among servers. Indeed, all servers must have a common time reference to trigger the transmission of each individual packet at a predefined time instant, chosen by the packet scheduler. Hence, Fastpass mimics a synchronous TDM-based network. The downside of minimizing the DCN latency is that most of the delay is now experienced at the servers, as later shown in the numerical results in Section 3.3.7, which investigate the queuing spreading across the various levels of the data center hierarchy, from servers to the higher layer switches. Nevertheless, the predictability of DCN crossing delays permits to better control the overall performance of data transfers, providing a nice solution in the design of high-performance data centers.

In [44], we address the following question: Is it possible to achieve performance similar to an almost ideal synchronous architecture by relaxing the constraint on *synchronization*, thus reducing its cost and complexity? We show that the answer is positive. Indeed, fully *asynchronous* data transfers can achieve performance similar to Fastpass without any strict synchronization among the servers. The asynchronous solution is still based on a centralized controller, that now only orchestrates the actual rates at which each server injects packets in the DCN, instead of controlling the exact time when every single packet is transmitted by the servers. Consequently, we claim that architectures based on rate allocation are able to achieve a better trade-off in terms of

[1]Note that these ad-hoc solutions coexist with the legacy transport protocols adopted in the servers.

performance and complexity than synchronous architectures.

To this end, in this Chapter we describe the considered synchronous and asynchronous traffic control models for DCNs. For a fair comparison, we provide a general framework to compare the two traffic control schemes while highlighting the benefits and the drawback of each approach. After such an analysis, to motivate our main claim regarding the limited advantages of the synchronous architecture, we conduct an extensive simulation campaign, with a detailed and realistic simulation model of a DC. Finally, we analyze the obtained results which allow us to assess the performance of the two traffic control approaches and confirm our initial claims.

3.1 Related work

The most relevant work to our is the already mentioned Fastpass [43]. Fastpass provides proactive network-wide packet scheduling for the DCN by i) actively gathering server transmission requests ii) optimally scheduling packet transmissions among all servers present in the network and iii) routing packets through the network in such a way to avoid packet collisions.

Flowtune [45] follows all the three phases described for Fastpass architecture. Instead of operating on a per-packet basis Flowtune operates on a per-flowlet granularity. Instead of a per-packet scheduler, it employs a centralized rate normalization algorithm based on the solution of a network utility maximization (NUM) problem. Yet, operating on such a coarse-grained scale makes it impossible to perform collision-free load balancing, forcing it to employ the suboptimal ECMP load balancing scheme. While proposing a scalable alternative to Fastpass, in their evaluation the authors of [45] focus completely on the scalability aspect of centralized rate assignment while obviating the comparison of the proposed approach with the synchronous one.

Numfabric [46] leverages the same architecture as in [45] but with the NUM problem being solved in a distributed way by relaxing the constraint on the maximum utilization of a link, which leads to suboptimal rate allocations. Finally, due to its distributed nature, the solution for the NUM problem requires multiple RTT in order to converge.

Hedera [47] provides a load balancing scheme based on fine-grained load estimation, thus neglecting the rate normalization. Differently from the model proposed in this

Chapter, the load balancing scheme reroutes flows to balance the offered traffic across the topology, leaving to TCP the overall maximization of the network utilization. Thus, the rate estimation is based on the link load, without the interaction with the servers.

Finally, all these previous works [45]–[47] do not provide any performance comparison with a fully synchronous architecture, which is instead our main contribution.

3.2 Scheduling and routing in DCN

As shown in Figure 3.1, we assume a data center with N servers connected by a multilayer DCN, in which each server is connected to a ToR switch. ToR switches are interconnected through a multilayer Clos-based topology, e.g., leaf-spine in the case of a two-layers topology. For simplicity, we assume that the bisection bandwidth of the DCN is maximum, thus no over-subscription is present[2]. We also assume a homogeneous scenario with all the links have the same capacity. For inhomogeneous scenarios with link rates, at some layer, f times faster than at another layer, it is possible to construct an equivalent topology in which the faster links are split into f parallel disjoint slower links, replicating f times the switches with the higher rate ports. The process can be iterated across all the layers until all the links in the DCN equivalent topology have the same capacity.

Traffic flows are transferred between pairs of servers. The packet transmission from the servers to the ToR switches and the corresponding routing path are coordinated by central traffic control. Transmission queues at the servers are organized on a per-server destination basis, to avoid throughput degradation due to the well-known head-of-line problem. Thus, a maximum of N (logical) queues are managed by each server.

3.2.1 Synchronous (SYN) architecture

This architecture is based on Fastpass and the implementation issues are discussed in [43]. We assume that all server linecards are synchronized; time is slotted according to a fixed packet transmission time. At a generic timeslot t, a synchronous controller runs a sequence of three phases, as depicted in Figure 3.2:

[2]The model can be easily adapted to DCNs with over-subscription.

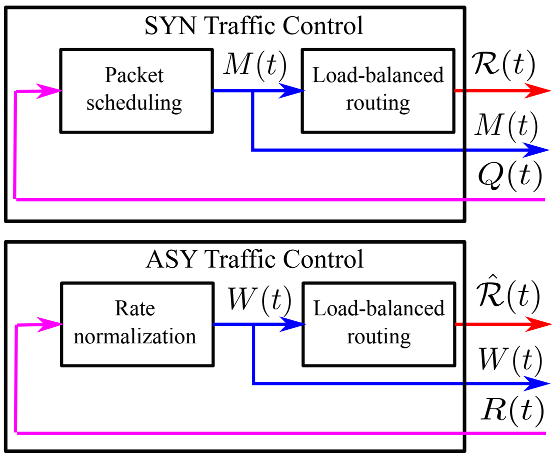


Figure 3.2: Synchronous (SYN) vs Asynchronous (ASY) architecture

1. *Queue state collection.* The state of all the transmission queues at the servers is retrieved (e.g., by interacting with proper socket monitoring tools installed in the server kernel [43]). Let $Q(t) = [q_{ij}(t)]$ be a $N \times N$ matrix such that $q_{ij}(t)$ denotes the priority to transmit a packet from server i to server j, at timeslot t. As an example, $q_{ij}(t)$ can be the queueing delay of the packet at the head-of-line of the corresponding transmission queue. Thus, packets with higher queueing delay have a higher transmission priority.

2. *Packet scheduling.* Based on $Q(t)$, a packet scheduler chooses a set of source-destination server pairs for transmission during timeslot t, such that at there is at most one concurrent transmission to and from each server. Thus, the scheduler computes a *matching* between the source servers and the destination servers. The matching is described by a binary $N \times N$ matrix $M(t) = [m_{ij}(t)]$, such that $m_{ij}(t) = 1$ if server i transmits the head-of-line packet directed to server j during timeslot t.

3. *Load-balanced routing.* The controller computes the routing path for each transmitted packet in $M(t)$ to balance the packets across the links of the DCN, to guarantee that all packets traverse different links. The outcome is a data structure $\mathscr{R}(t)$ that describes the routing path for all the packets in $M(t)$.

The adopted constraints in the packet scheduling and routing phases completely avoid queuing at each interface of the DCN, if assuming the same number of hops in the multilayer DCN and the same propagation delays of all links. In practice, very limited congestion can be experienced to compensate different path lengths and different link propagation delays. Section 3.3.7 will be devoted to evaluate the actual queuing.

One major practical issue of the considered SYN approach is the requirement to guarantee synchronous behavior of all the servers. Indeed, the required time accuracy becomes more strict with high port bitrates. As a reference example, consider that the transmission time of the smallest Ethernet packet (e.g., a TCP ACK) is about 50 ns at 10 Gbps and the one of an Ethernet MTU is about 1.2 μs. Given the quartz clock generators present in the servers and their unavoidable thermal drift (affected also by the server computation load), the precise synchronization across all servers can be achieved if relying on expensive dedicated hardware [48].

The choice of the timeslot duration is also very critical for performance, due to the possible partial filling of the transmission timeslots. In terms of throughput, small timeslots permit to reduce the waste due to partial filling, but at the expense of introducing some control overhead to manage the fragmentation of large packets into multiple timeslots. On the contrary, large timeslots remove or mitigate the fragmentation problem, but suffer from the partial filling of the timeslots due to small packets. In terms of control information, small timeslots require higher bandwidth for the control channel between the controller and the data center components. We will evaluate in Section 3.3.6 the effects of the partial filling on the performance.

3.2.2 Asynchronous (ASY) architecture

We now remove the constraint of synchronization and propose an architecture based on *rate control*; the centralized controller assigns transmission rates at each server, one for each possible destination server, instead of assigning packets to timeslots as in the SYN scenario. The considered scheme runs each time the offered load changes at the servers, tracking flow-level dynamics instead of packet-level dynamics as in the SYN case. As shown in Figure 3.2, a sequence of three phases occurs at time t:

1. *Offered rate estimation.* The control gathers the statistics about the offered load between any pair of servers. Similarly to the previous scenario, the statistics can be obtained by interacting with proper socket monitoring tools installed in the

server kernel. Let $R(t) = [r_{ij}(t)]$ be a $N \times N$ matrix, denoted as *offered rate matrix*, with $r_{ij}(t)$ be the offered load from server i to server j at time t, normalized by the link rate, i.e. $r_{ij} \in [0,1]$.

2. *Rate normalization.* Now $R(t)$ is renormalized to become admissible and avoid link overloading, i.e. the overall transmission rate from any server and towards any server must be lower than the link rate. The outcome of this phase is a $N \times N$ matrix $W(t) = [w_{ij}(t)]$, denoted as *transmission rate matrix* and $w_{ij}(t)$ be the actual transmission rate to adopt from server i to server j at time t. The algorithm maximizes the overall throughput $\sum_i \sum_j w_{ij}(t)$. By construction $W(t)$ is a double sub-stochastic matrix, i.e., $\sum_i w_{ij}(t) \leq 1$ and $\sum_j w_{ij}(t) \leq 1$.

3. *Load-balanced routing.* Based on $W(t)$, the controller chooses the paths to balance the traffic across the DCN. The outcome is a proper data structure $\hat{\mathscr{R}}(t)$ that describes the routing paths for the packets transferred starting from t.

Similarly to the SYN architecture, the ASY one can be easily extended to support traffic priorities, by properly scaling each value in $R(t)$.

Whenever the offered rate changes, i.e. $R(t)$ varies, the controller must re-run the three above steps. In the worst case, this occurs with a frequency which is related to the packet transmission time. Thus, the ASY system incurs (in the worst case) in an overhead similar to the SYN system in terms of exchanged control information. Instead, during the time intervals in which $R(t)$ does not change, ASY incurs in a much lower overhead than SYN.

3.3 Performance evaluation

To analyze the performance of SYN and ASY architectures, we first describe the specific algorithms adopted for the controllers. Then, we detail the simulation methodology and, finally, we present the numerical results.

3.3.1 Algorithms for SYN architecture

We assume a greedy maximal matching adopted for packet scheduling, based on the state of the transmission queues at the server. We consider different priority functions to define the $\boldsymbol{Q}(t)$ matrix:

- Oldest cell first (OCF): the matching is performed by looking at the queuing delay of the head-of-line packets.
- Shortest remaining job first (SRJF): the matching is performed by looking at the amount of residual bytes of each flow needed to be transferred. Shortest flows are prioritized over the longest ones.
- Max-min fair (MMF): the matching is performed by looking at the waiting time of the packets since they become the head of the queues.

Routing computation, coherently with Fastpass, is performed to minimize the contention on the switch output ports. To achieve this, we implemented the solution of the edge coloring problem for bipartite graphs starting from $\boldsymbol{M}(t)$ by adapting the classical Paul algorithm for Clos networks in [49]. Notably, in the case of a two layers DCN, each color is associated with one distinct spine switch.

3.3.2 Algorithms for ASY architecture

To implement rate normalization, we consider a simple algorithm based on iterative matrix renormalization presented in [50]. The algorithm iteratively renormalizes rows and columns of the rate matrix $\boldsymbol{R}(t)$, as shown in Algorithm 2. The convergence of such an algorithm is guaranteed if all elements of the matrix are strictly positive while in the presence of zeros the process oscillates around the solution. In spite of the oscillations, for sufficiently large N, the error in respect to the theoretical convergence point is negligible. To achieve the desired rate of convergence we define the target error rate ϵ_t which once reached terminates the renormalization procedure. At each iteration of the algorithm the error is computed after the column normalization is performed which guarantees that all inbound traffic at each server is admissible while allowing bandwidth overallocation at each servers' egress. Nevertheless, even in the presence of outbound rate overallocation backpressure mechanisms employed At the same time, we define a maximum number of allowed iterations $i_{\max}$ which once reached terminates the execution of the procedure which precludes infinite oscillations around the optimal solution.

Algorithm 2 Iterative rate matrix renormalization algorithm employed in ASY architecture

```
1:  procedure $\{w\}$ = RateNormalization($R_{ij}, i_{\max}, \epsilon_t$)
2:      $w_{ij} \leftarrow R_{ij} \quad \forall i, j = 1, .., N$                ▷ $N \times N$ rate matrix
3:      $\epsilon \leftarrow \epsilon_t$
4:      $i \leftarrow 0$
5:      while $\epsilon \geq \epsilon_t$ or $i > i_{\max}$ do     ▷ While the current error is greater or equal
                                                  to the target error or until the maximum
                                                  number of allowed iterations is reached
6:          $w_{ij} \leftarrow w_{ij} / \sum_{j=0}^{N} w_{ij} \quad \forall i = 1, .., N$          ▷ Row normalization
7:          $w_{ij} \leftarrow w_{ij} / \sum_{i=0}^{N} w_{ij} \quad \forall j = 1, .., N$       ▷ Column normalization
8:          $\epsilon \leftarrow |1 - \frac{1}{N} \sum_{i=0}^{N} \sum_{j=0}^{N} w_{ij}|$     ▷ Average outgoing rate over-allocation
9:          $i \leftarrow i + 1$
10:     end while return $w$
11: end procedure
```

Each server implements one leaky bucket scheduler for each active per-destination queue, to guarantee an instantaneous rate $w_{ij}(t)$ between server i and server j. For the purpose of this analysis we use a fluid leaky bucket whose rate is updated instantaneously whenever there is a variation of the assigned rate.

For load-balanced routing, we deployed a standard flow-by-flow ECMP, which will be shown in Section 3.3.7 to be good enough to provide low buffer occupancy, and, as a consequence, small queuing delays inside the DCN.

3.3.3 Simulation methodology

We performed the analysis using the discrete-event simulator OMNeT++ [51] in combination with the libraries of the INET framework, which provides detailed simulation models for the Internet protocols stacks from the MAC layer up to the application layer.

We considered a standard Ethernet-based leaf-and-spine topology for the DCN to compare SYN with ASY architectures, as shown in Figure 3.1. The chosen topology provides full bisection bandwidth, connecting $N = 120$ servers, built with 3 spine switches, 4 leaf switches and 30 servers per leaf switch. All servers are connected to the leaf switches via a 1 Gbps link while leaf switches are connected to spine switches via a 10 Gbps link. The buffers at the servers and at the switches are assumed infinite.

Traffic flows are generated according to a Poisson process with bursty arrivals of packets belonging to the same flow and the flow size is exponentially distributed with an average of 46 kB. This value has been derived by the Facebook data center [52].

IP packet lengths are chosen according to one of the following methods:

1. fixed and equal to 1500B to simulate bulk data transfers.

2. randomly chosen according to a bimodal distribution with 0.4 probability of generating 40B packets and 0.6 probability of generating 1500B packets; this scenario approximates the scenario typical of many applications such as Hadoop, as observed in [52];

3. randomly chosen according to the Facebook Web Server distribution (FBW) taken from [52], which refers to a web service scenario in a data center. In this scenario only 15% of the packets have are 1500B size with a median centered around 150B.

In SYN architecture, we set the timeslot corresponding to 1500B at the IP layer, coherently with Fastpass [43].

We considered two different traffic patterns:

1. *incast*, where all the servers sends traffic to the same hot-spot server;

2. *uniform*, where the traffic is uniformly distributed across all servers;

The data center load is defined as the average normalized amount of traffic destined to the servers. Finally, the centralized controller is assumed to operate out of bandwidth and with zero latency.

We evaluate the normalized per-server throughput and the Flow Completion Time (FCT), in terms of average and coefficient of variation. FCT is measured from the generation of the first packet belonging to a flow until the reception of the last packet by the destination application. Each FCT is then normalized by the theoretical minimum FCT that would be achieved by that flow in an empty DCN.

FCT has been chosen among the primary confrontation metrics due to its importance in assessing the performance of typical DC delay-sensitive applications, which directly affects the quality of experience of end-users. We further highlight the composition of FCT by analyzing the server latency, i.e. the average amount of time packets spend in the transmission queues inside each server before entering the DCN and the

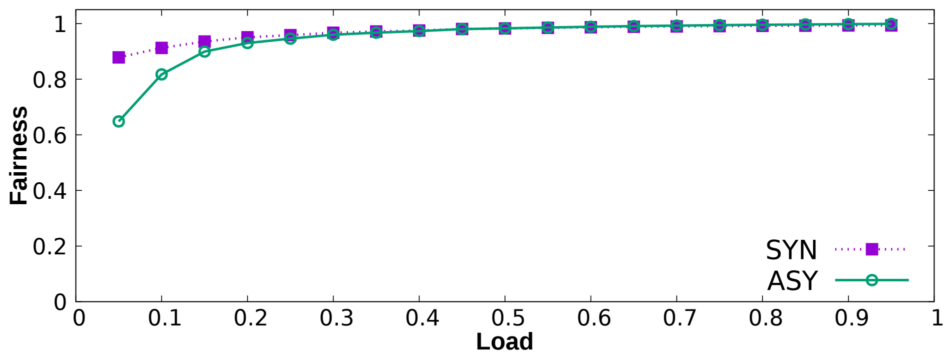


Figure 3.3: Fairness comparison between SYN MMF packet scheduling and ASY rate normalization algorithms under uniform traffic

network latency, which is the average time it takes for packets to arrive to the destination server once they enter the DCN.

3.3.4 Fairness comparison

As a preliminary result, we show that the two architectures behave in a similar way in terms of fairness. We compare the ASY architecture computing the MMF matching with the SYN architecture with the previously described rate normalization algorithm. Figure 3.3 depicts the Jain's fairness index of the two systems under uniform traffic for different loads. For a small load, the ASY system achieves lower fairness due to the sparseness of the offered rate matrix. However, for load higher than 0.2, which is a typical scenario in DCNs, the matrix becomes denser and the two architectures rapidly converge to the same, close to the optimal one, fairness index.

3.3.5 Influence of the matching policy on FCT

We investigate the effect of the metrics adopted in the computation of maximal size matching, as described in Section 3.3.1, under fixed-length packets. Similar results have been obtained for variable-size packets.

Figure 3.4 depicts the influence of each matching metric on the FCT, under the scenario of the incast traffic. OCF yields the highest average FCT but at the same time, the lowest variance (results not reported for the sake of brevity). On the contrary SRJF by its nature minimizes the FCT but may lead to unfairness which in return increases

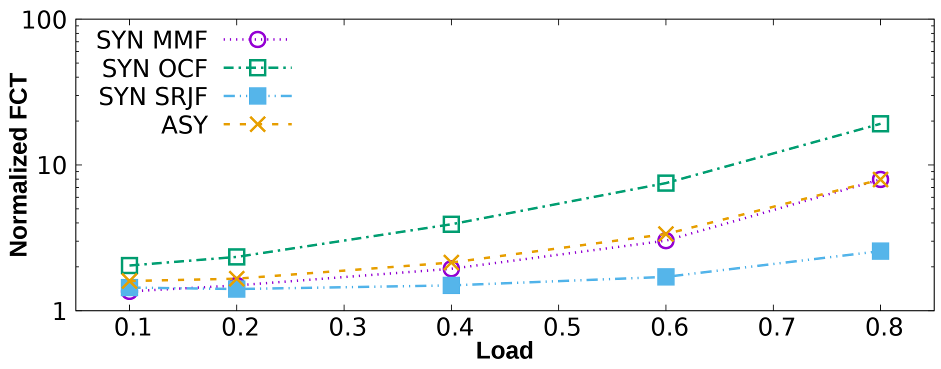


Figure 3.4: Comparison among different maximal size matchings under incast traffic pattern.

variance and the amount of potentially missed application deadlines. MMF is a reasonable trade-off between average FCT and the corresponding variance as it balances the two metrics. All the metrics permit to achieve the maximum throughput and the same (optimal) fairness. The main factor responsible for the slight reduction in FCT in the case of the ASY system is packet contention at the switches, which will be shown in Section 3.3.7 to be negligible.

3.3.6 Influence of packet-length distribution

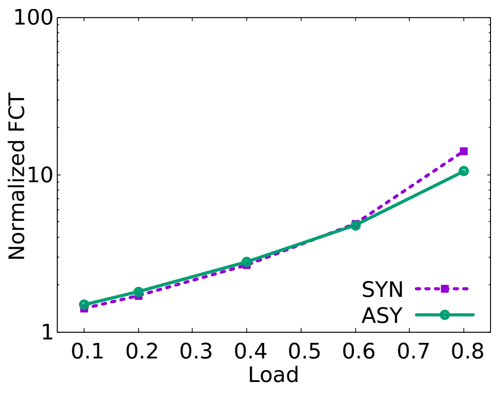
In our subsequent analysis we considered the influence of packet-length distribution on the two systems.

Fixed packet lengths

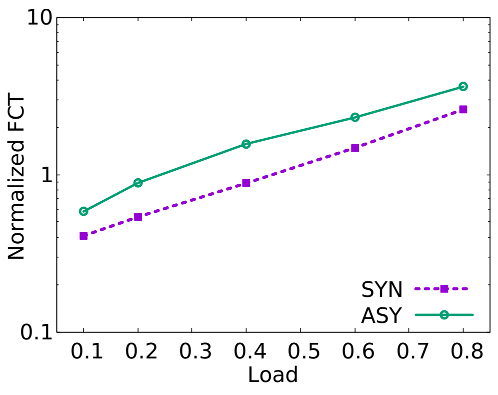
We obtained results similar to the case of the incast traffic pattern. From Figure 3.5a it can be seen that there is no significant variation in terms of FCT between the two systems. Noticeably at higher loads the ASY system yields smaller FCT while still maintaining comparable variance. The reason behind this behavior will be later explained in Section 3.3.7.

Bimodal packet lengths

Bimodal packet length distribution is common to applications using TCP as the transport protocol. In the case of fixed-size packets, each timeslot of the SYN system was



(a) Average



(b) Coefficient of variation

Figure 3.5: FCT for fixed-length packets and uniform traffic pattern.

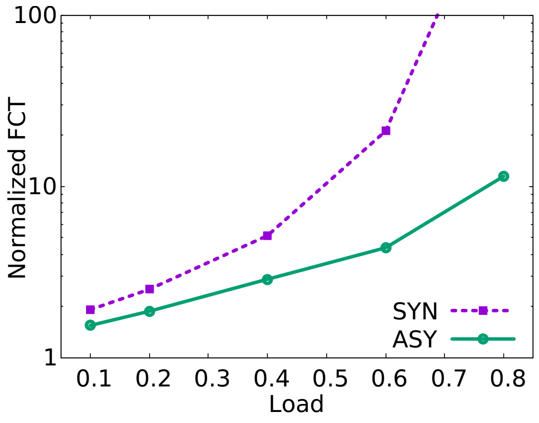
fully exploited because the packet lengths were tailored in such a way to perfectly fit inside the timeslot, leading to 100% throughput inside each timeslot. We observed that the performance of the SYN system changes significantly in the presence of variable-size packets. For a bimodal packet distribution, the FCT of the SYN system quickly diverges from the ASY one, which as it can be seen from Figure 3.6a increases by more than one order of magnitude. It is easy to build an adversarial traffic pattern repeatedly composed of one MSS-sized packet followed by one ACK packet which may reduce the throughput of the SYN system down to $\approx 50\%$.

FBW packet lengths

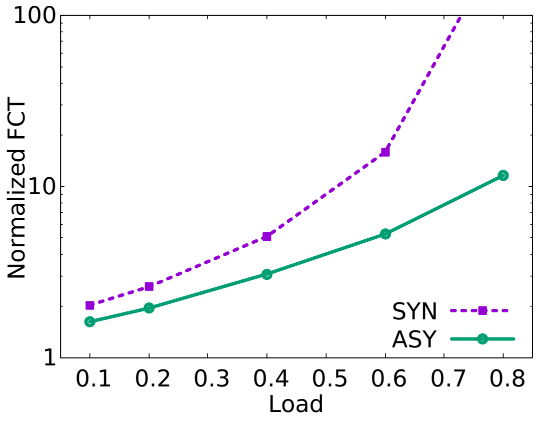
Similarly to the bimodal packet length distribution, in the case of FBW the SYN system does not achieve 100% throughput due to the partial timeslot filling. Results depicted in Figure 3.6b show how at low load the two distributions lead to similar FCT. However, at higher load, even if FBW packet lengths lead to smaller FCT with respect to the pure bimodal one, it is still one order of magnitude larger with respect to the ASY case.

3.3.7 Queueing within the data center

Figures 3.7, 3.9 show a 100ms trace of buffer occupancy averaged across all servers, for 0.8 load under uniform traffic pattern and fixed packet lengths. It is immediate to notice that the ASY system provides a more balanced distribution of buffer occupancies across the entire data center. The ASY system is able to keep the server memory 40% lower



(a) Bimodal



(b) FBW

Figure 3.6: FCT for bimodal and FBW packet lengths and uniform traffic pattern.

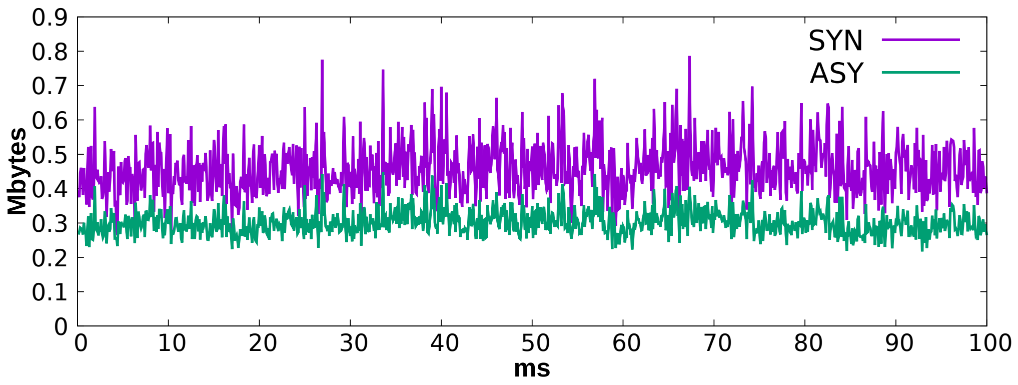


Figure 3.7: Transmission buffer occupancy at the servers

with respect to the SYN architecture at the expense of slightly bigger buffering at the switches. Notably, this reduction is relevant because it mitigates the resource overhead of managing per-destination queuing at the servers. In particular, we observed that the 40% reduction in the buffer occupancy at the servers corresponded to a 45% increase in the average buffer occupancy at leaf switches (but no significant variation in the 99th percentile with respect to the SYN case) and 100% increase in the average buffer occupancy at spine switches (with a 99-percentile around 7.5 kB, which is still very small).

In Figure 3.10a we show the impact of queuing on the experienced delays under uniform traffic pattern with fixed-length packets. Although the 50th percentile of the packet network latency for ASY architecture is double that of the SYN, the overall FCT

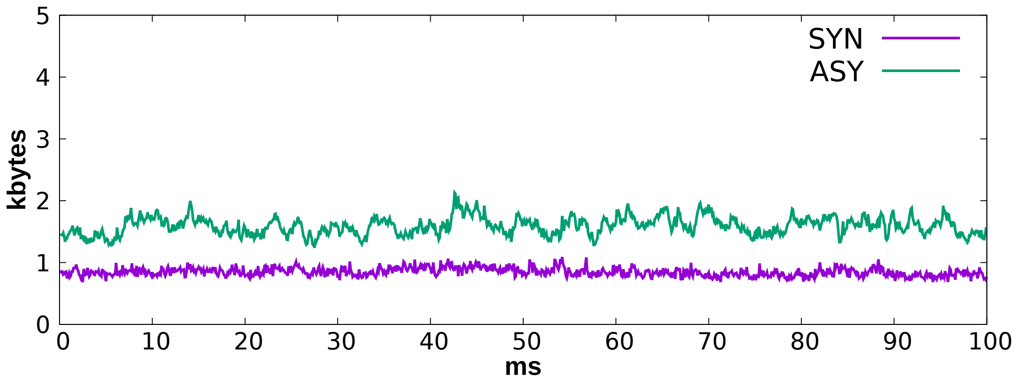


Figure 3.8: Per-port buffer occupancy at the leaf switches

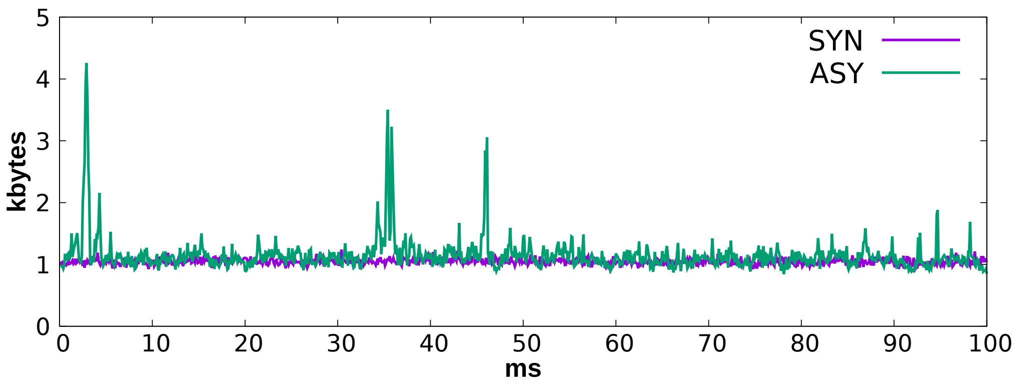
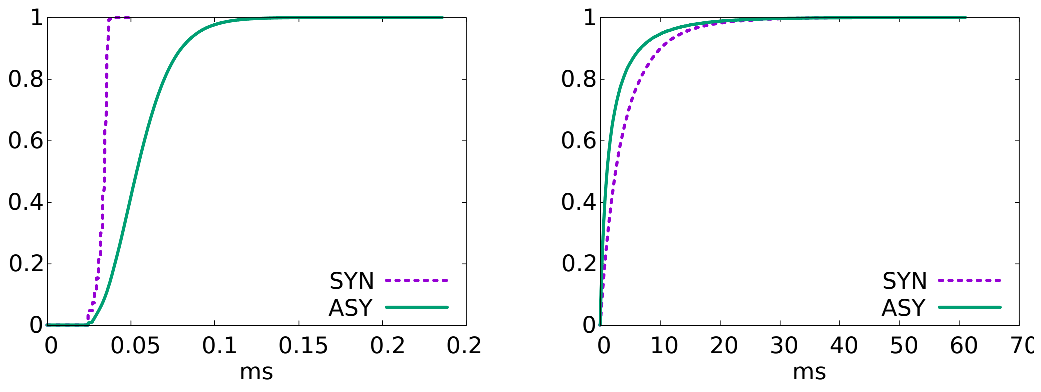


Figure 3.9: Per-port buffer occupancy at the spine switches



(a) CDF of network latency.

(b) CDF of FCT.

Figure 3.10: Latency distribution inside the network for SYN and ASY systems under 0.8 load and uniform MTU-sized traffic.

is still dominated by the latencies at server queues. In fact, when compared with the

CDF of FCT in Figure 3.10b, the contribution due to network latency can be seen to be negligible. The cost of providing protection against contention inside the DCN in SYN architecture is that of experiencing larger delays at the servers, which, as shown, for high load becomes the dominant cause of an increased FCT.

3.4 Discussion

Asynchronous architectures appear to be very promising for their trade-off between performance and complexity. Nevertheless, employing centralized schemes for such a critical task as traffic control has its own drawbacks. Scalability is among the main factors pushing vendors away from centralized traffic management solutions, leaving distributed approaches as the preferred ones. However distributed approaches typically are not capable of reaching the same level of performance as the centralized ones unless employing expensive dedicated hardware. In Chapter 4 we will show that distributed flow scheduling can be performed on commodity switches, thus not requiring any hardware modification, while at the same time achieving performance close to the state of the art expensive solution.

Chapter 4

NOS2: Simplifying Flow Scheduling in Data Center Networks

Part of the work presented in this chapter has been published in:

- Sviridov G, Bianco A, Giaccone P. ”Low-complexity Flow Scheduling for Commodity Switches in Data Center Networks.” In *IEEE Global Communications Conference (GLOBECOM)*. 2019.

In Chapter 3 we analyzed how an asynchronous traffic control is capable of minimizing the congestion in DCNs by accurately adjusting the transmission rate of each server, ultimately leading to low flow completion times. Nevertheless, in our discussion we conclude that centralized traffic control schemes are difficult to realize in practice. Furthermore, building upon those solutions to support more sophisticated flow scheduling mechanisms can result in even higher complexity and poorer scalability. Yet modern DCs call for flow scheduling mechanisms capable of discriminating among different flows.

Indeed, most of the modern DCs are employed as multi-tenant environments, thus being composed of multiple servers running different tasks and services such as interactive applications, data processing and machine learning. As a consequence, DCNs observe different types of flows with different requirements in terms of latency. Multiple studies [52], [53] have shown that the majority of flows present in a DCN are short (i.e., “mice” flows). While being the dominant type of flows in the DCN, mice flows are far from being responsible for the majority of the bandwidth utilization. Very long flows (i.e., “elephant” flows), although being scarce in numbers, contribute to more than 80% of the total network utilization.

Mice flows require a small flow completion time (FCT) because they typically belong to delay-sensitive applications, such as remote procedure calls (RPCs) or real-time interactive applications. On the contrary, elephant flows are bandwidth sensible and typically require a large amount of bandwidth for a prolonged time while being able to tolerate high latency. The coexistence of the two types of flows inside DC poses significant challenges for the performance optimization of the respective type of flow. Traditionally, DC flow management schemes are based on classic transport protocols and on switches employing buffers shared among all flows without any discrimination between different types of flows. Thus, in such scenarios all flows are treated likewise. However, such approaches lead to strong penalties for the FCT of mice flows whose performance is deteriorated by the coexistence with bandwidth-hungry and aggressive elephant flows.

Multiple proposals have been made aiming to achieve the best possible performance for both mice and elephant flows by exploiting knowledge about the lengths of individual flow [54]–[56]. However, in realistic scenarios such detailed information about each flow is typically impossible to obtain.

Recent proposals like PIAS [57] and Homa [58] try to address this issue by assuming all flows to be mice up to a certain amount of transmitted data, thus performing prioritization of new flows over the already existing ones in both servers and DC switches. This approach does not require any preliminary information about the flow length and permits to reduce the FCT for mice flows, even though they may initially compete with elephant flows. At the same time, elephant flows do not starve, thus satisfying their bandwidth hungriness. From a practical point of view, such an approach employs in-network prioritization employing different priority queues (PQs) with different types of flows being assigned a different priority. This leads to the creation of leading “spatial division” of flows of different lengths between different PQs, with the highest priority being assigned to mice flows. It was shown that 8 PQs at the servers permit to achieve a reduction in the average FCT up to a factor of 3 with respect to traditional scheduling using a single buffer [57]. Unfortunately, even if high-end switches are often equipped with 4 to 8 PQs, such queues are not fully usable as they are typically devoted to DC-critical tasks. Such tasks may include separation of different types of traffic [59] such as RDMA [60], DCTCP [61] and traditional TCP-based traffic or isolation of the signaling flows. The few available PQs poses substantial restraints on the use of fine-grained PQ-based scheduling mechanisms, making them unpractical in realistic scenarios. On the other hand, flow scheduling at servers has observed small interest from the research community and its possible coupling with DCN scheduling has not been widely investigated.

In this work, we propose *Network Optimized Split 2* (NOS2), a flow scheduling algorithm which aims at a sweet spot between practicality and performance gain by combining fine-grained server-side flow scheduling with simple DCN flow scheduling. NOS2 closely approximate the performance of PIAS and Homa while employing only 2 PQs in the switches, thus making a step towards the practicality of this kind of scheduling mechanism in realistic scenarios. NOS2 employs a simple PQ-based flow scheduling at each server using 8 PQs, thus providing high scheduling granularity, while maintaining coarse-grained scheduling based on 2 PQs inside switches, effectively decoupling server-side and network scheduling.

NOS2 is tuned to achieve low FCT by leveraging the information about the DC-wide flow length distribution (denoted as *workload* in the following). We show that combining such information with fine-grained scheduling at servers is sufficient to achieve performance similar or even better (depending on the scenario) than schemes based on 8 PQs while employing a considerably smaller amount of PQs inside switches. A preliminary version of this work was presented in [62].

To this end in this Chapter, we describe the main issues related to flow scheduling employing flow prioritization in DCNs an how such issues have been addressed in previous work. Following this analysis we propose NOS2, a novel flow scheduling algorithm and describe its architecture. Finally we investigate the performance of NOS2 by performing an exhaustive simulation campaign under realistic scenarios and by comparing it to existing solutions.

4.1 Related work

The authors of [57] propose PIAS, a flow scheduling algorithm which aims at minimizing the FCT by splitting flows among different priorities without precise a priori knowledge of each flow length. While the benefits of this approach idea has already been investigated in [63], the authors of [57] provide a formulation to find an optimal split of traffic among different PQs by exploiting information on the average DC-wide flow length distribution. However, the authors do not address issues related to the heterogeneity of workload across the DC and the performance penalty due to the usage of a DC-wide global flow length distribution. Furthermore, the proposed architecture requires a large number of PQs inside switches, which may not be easily available.

In [64] the authors compute the optimal traffic split by employing reinforcement learning algorithms. While being a considerable step forward in solving the scalability issues, the proposed solution still operates on average DC-wide flow length distribution, thus not taking into account fine-grained scheduling at each host.

In [59] the authors propose a flow prioritization scheme that exploits only two PQs at network switches. However, differently from NOS2, the architecture introduces a modification of the scheduling algorithm running at the servers and requires expensive modifications of the switch to execute the proposed prioritization mechanism.

In [54] the authors propose a scheduling algorithm, namely pFabric, and show that it is able to achieve a near-optimal FCT under realistic workloads. pFabric employs a fine-grained flow prioritization scheme that, in the worst case, needs an unbounded number of PQs, limiting its practical applicability. Furthermore, it requires a priori knowledge of the length of each individual flow, which is typically unknown and/or difficult to obtain in realistic scenarios.

The work in [65] proposes to emulate the presence of PQs inside the network by acting on the congestion window of the transport protocol and giving more aggressiveness when sending the segments of short flows. However, this system provides marginal improvements with respect to a traditional transport protocol, and the performance is still worse than a system based on flow separation in different PQs.

In [58] the authors propose Homa, a credit-based transport protocol in which receivers drive the senders by assigning transmission credits. The credits serve a double

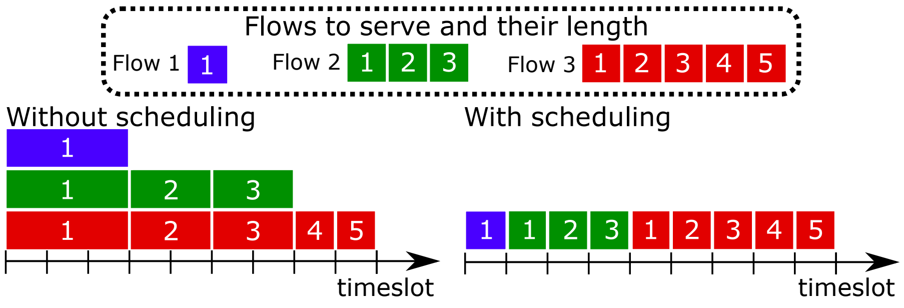


Figure 4.1: Example of flow scheduling in function of the time.

purpose: i) to specify which senders are permitted to transmit over the DCN to receivers and ii) to define the network priority to be used during the data transmission. The priority is selected by exploiting information on the flow length distribution so as to prioritize short flows over long ones. However, similarly to [57], this approach requires a large number of PQs inside switches. Furthermore, Homa requires to completely rework the transport protocol inside each server.

4.2 Reducing FCT with flow prioritization

The problem of scheduling flows consists of choosing a particular set and an order of flows to serve from an available pool of active flows so to minimize a given objective function (e.g., FCT, number of deadline violations, average throughput). Depending on the metric employed for such a choice, different scheduling algorithms can be implemented. Figure 4.1 depicts an example of a pool of 3 flows of different length, in terms of packets, to be transmitted over a single link. Time is slotted with a timeslot equal to the packet transmission time. Without any flow scheduling, all three flows will be served concurrently and they will fairly share the available link bandwidth leading to an FCT of 3, 7 and 9 for flows 1, 2 and 3 respectively, achieving an average FCT of 6.33 timeslots. On the other hand, a flow scheduler which privileges short flows over long one, for the same pool of flows, will lead to an FCT of 1, 4 and 10 with an average FCT of 5 timeslots. In the latter example, the FCT of short flows (flows 1 and 2) is reduced considerably without affecting significantly the performance of the long flows (flow 3).

In realistic workloads the majority of flows inside a data center are mice flows. Indeed, measurements from [66] show that 80% of flows are mice flows less than 10MB. Flow schedulers that privilege short flows over long ones may lead to considerable FCT reductions for mice flows at the expense of the elephants. Nevertheless, [66] showed that mice flows account for less than 1% of the global DCN traffic, thus the performance penalty of long flows is expected to be negligible. In general, the performance of elephant flows depends on the overall flow length distribution, in particular it is highly influenced by the shape of the tail of the distribution. It is expected to observe smaller performance degradation for shorter tails and vice-versa.

One way of realizing the aforementioned idea is to employ scheduling based on the flow length of each flow, as discussed in the following.

4.2.1 Flow length-aware scheduling

In flow length-aware scheduling algorithms, the length of individual flows is assumed to be known in advance and packets belonging to shorter flows are served at higher priority with respect to packets belonging to longer flows. A flow maintains the priority level for its entire lifetime. This scheduling policy, known as Shortest Job First (SJF), has been shown to have a dramatic benefit for the average FCT [67]. A preemptive version of SJF, known as Shortest Remaining Job First (SRJF), assigns priorities to each flow based on the amount of bytes left to transmit until their completion. Flows with the least amount of missing bytes are served with a higher priority with respect to flows requiring more bytes to transfer.

To implement prioritization of individual packets in real scenarios, strict priority (SP) scheduling is adopted and packets are stored in N separated queues. Let $\mathcal{Q} = \{q_p\}_{p=0}^{N-1}$ be the set of all queues, with p being the priority level (with 0 being the level corresponding to the highest priority). At each time instant the scheduler selects the highest priority queue $q_s \in \mathcal{Q}$ to serve, i.e., $s = \min_{0 \le p \le N-1}\{p : q_p \text{ is not empty}\}$.

SRJF has been widely studied and its practical implementation was proposed in pFabric [54] proving to be optimal [68] in terms of average FCT.

The main limitation of flow length-aware schedulers is that they require knowledge about the length of individual flows. Furthermore, such approaches are made further more impractical when considering that SJF and SRJF require a per-flow queueing that can be approximated with a large N for the SP scheduler.

4.2.2 Flow length-agnostic scheduling

In realistic scenarios the length of individual flows is typically unknown or difficult to be obtained without undergoing into deep modifications of the application layer [69]. Furthermore, the amount of PQs available in commercial switches typically ranges from $N = 4$ to $N = 8$ PQs, far less than the amount of PQs required for fine-grained scheduling policies as SJF or SRJF.

In the absence of any knowledge about the length of individual flows, scheduling policies leverage the amount of transferred data for each flow (which can be locally computed based on per-flow counters) to approximate flow length-aware schedulers.

Least Attained Service (LAS) scheduler [70] always gives priority to flows with the least amount of attained service (i.e., transferred bytes) leading to small FCT for mice flows as, similarly to SRJF, they are prioritized over large flows. LAS is known to be optimal when the cumulative distribution function of the flow length $F(x)$ is known in advance and its hazard rate $h(x) = \frac{F'(x)}{1-F(x)}$ is decreasing [71]. Although mitigating the issue related to the absence of information about the length of individual flows, LAS still requires a number of PQs as large as pFabric, thus making it impractical in real scenarios.

This limitation can be overcome by employing a discretized version of LAS, based on few queues (i.e., small N). This solution is known as Multilevel Feedback Queue (MLFQ). Similarly to an SP scheduler, MLFQ is composed of N queues (namely levels) served in a strict priority manner. However, differently from a normal SP scheduler, MLFQ permits flows to be demoted to lower priorities using a set of *demotion thresholds* $\Omega = \{\omega_i\}_{i=1}^{N-1}$, which are based on the amount of service (in terms of transmitted bytes) $b_f(t)$ a given flow f has obtained up to time t. Notably, $b_f(t)$ is a per-flow counter available in many commercial switches for data centers. Figure 4.2 depicts an example of an SP and 3-level MLFQ schedulers with the corresponding demotion thresholds in the latter case. Figure 4.2 depicts two implementations of an MLFQ scheduler, being black the common datapath among the two implementations; the blue datapath depicts the logical behavior of an MLFQ scheduler, while the red datapath shows the actual implementation of an MLFQ scheduler operating on a per-packet level, as used in practical

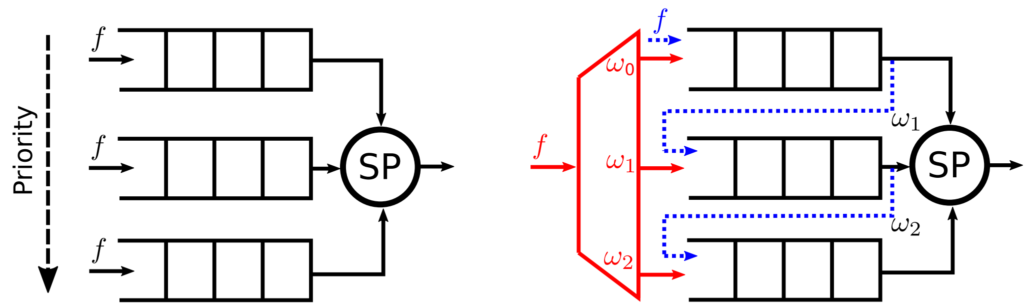


Figure 4.2: Comparison of a SP (left) and an MLFQ scheduler (right) with per-flow datapath (blue) and per-packet datapath (green)

implementations.

In a classic MLFQ scheduler each new job is initially placed in the highest priority queue and once it obtains enough service from a queue it is either demoted to a lower priority queue or, if its required service has been obtained, is removed from the system. This behavior is represented by the blue datapath in Figure 4.2. However, in an MLFQ scheduler operating on a per-packet basis, jobs (i.e., flows) arrive in chunks of packets. Here flows observe "soft demotions" with their packets being enqueued in the corresponding queues at the time of new packet arrival according to the obtained service of the flows they belong to. This behavior is represented by the red datapath in Figure 4.2. In the following, we will always consider the per-packet version of MLFQ.

Adding $\omega_0 = 0$ and $\omega_N = +\infty$ to Ω, at any given time t the priority p of a flow f is set such that $\omega_p \le b_f(t) < \omega_{p+1}$. When a new flow f arrives at the transmission buffer of a network interface, the flow is assigned a priority $p = 0$ and its packets are stored in the highest priority queue q_0. When the amount of served bytes exceeds the first threshold, i.e., $b_f(t) > \omega_1$, the flow is demoted and assigned a lower priority $p = 1$. Consequently, all the new packets belonging to the demoted flow will be stored in q_1. This process repeats until the flow terminates or eventually reaches the lowest priority queue $N - 1$, where it remains up to completion.

The most crucial aspect in an MLFQ scheduler is the definition of a proper set Ω of demotion thresholds. A simple approach, derived from [54], named Equal Split with N levels (ES-N), splits $F(x)$ in N equal percentiles, as follows:

$$\omega_i = F^{-1}\left(\frac{i}{N}\right), \qquad i = 1, \dots, N-1$$

However, such an approach may lead to early flow demotion, with mice flows quickly ending up in lower priorities together with elephant flows. Similarly, it may lead to late elephant demotion by keeping elephant flows mixed with mice flows for too long in high priority queues before demoting them. This ultimately leads to unbalanced utilization of available PQs, and, as a consequence, to FCT deterioration.

The authors of PIAS [57] try to overcome this limitation by proposing an information-agnostic scheduler able to achieve performance similar to that of pFabric while employing $N = 8$ PQs and using only the knowledge about the overall *DC-wide* flow length distribution. PIAS addresses the issue related to unbalanced PQs utilization by providing a formulation for an Optimal Threshold Assignment (OPT) so as to provide the best PQs utilization and to minimize the average FCT inside the MLFQ scheduler.

The formulation represents the MLFQ scheduler as a tandem of N M/M/1 queues, each queue corresponding to a priority level of the MLFQ scheduler. The service time of each queue is made dependent on the measured average DC-wide flow length distribution. The resulting objective function of the OPT problem is constructed in such a way to minimize the average queueing delay inside the tandem of queues, thus minimizing also the average FCT. The OPT objective function is modeled by:

$$\min_{\Omega} \sum_{i=1}^{N} \theta_i \sum_{m=1}^{i} W_m$$

being W_m the average delay of the mth queue, computed with a M/M/1 queuing model, and with $\theta_i = F(\omega_i) - F(\omega_{i-1})$ being the fraction of flows whose response time is affected by being processed in the first to ith queues.

4.2.3 Practical implementation of flow length-agnostic schedulers

Figure 4.3 shows two alternative architectures, one based on PIAS and the other on ES-N to compute Ω. Both architectures employ a MultiLevel Feedback Queue (MLFQ) but the thresholds in each server to serve data flows coming from upper layers is computed either with OPT or with ES. A central threshold controller collects statistics on the flows generated at the servers, derives the corresponding workload and executes the OPT (or ES) algorithm. At the same time, servers monitor and notify the threshold

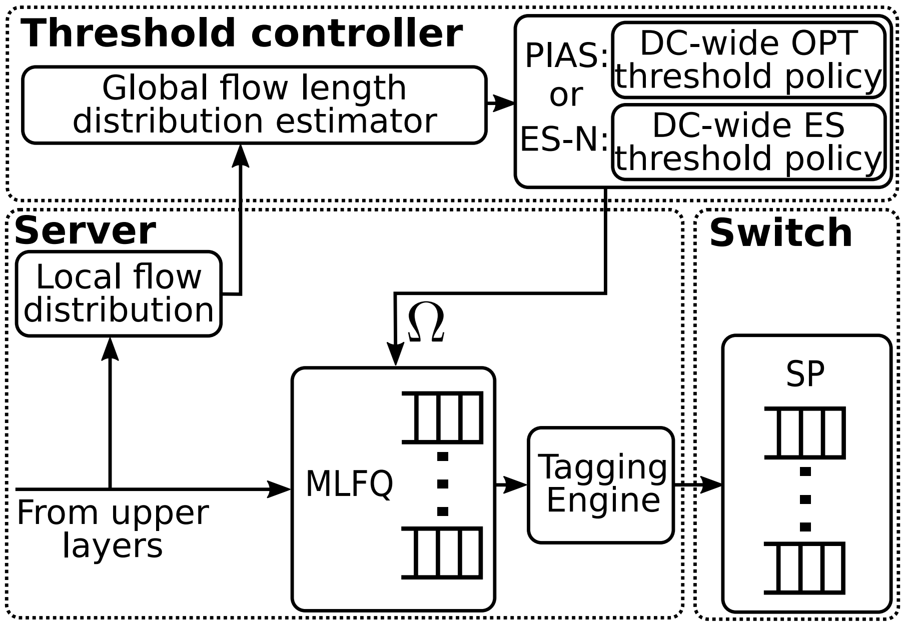


Figure 4.3: High-level architecture of PIAS and of ES-N, with many PQs at the servers and at the switches.

controller their flow statistics, and receive the updates on Ω.

Each server, prior to transmitting a packet to the network, tags the packet with a priority equal to the MLFQ level within which it has been previously enqueued. The priorities assigned by each server must be consistent throughout the DCN, meaning that switches must have a number of PQs equal to the number of levels of the MLFQ employed at servers. Upon packet reception, switches are left with the sole role of performing SP scheduling based on the priority tag. To do so, standard mechanisms based on IEEE 802.1p [72] can be exploited, being already available in most of the commercial DC switches.

4.2.4 Applicability in a realistic scenario

The OPT threshold assignment formulation employed in PIAS belongs to the family of sum-of-linear-ratios problems, which are known to be NP hard [73] and, to the best of our knowledge, no proven approximation exists for $N > 2$. Thus, the solution can only be found by means of heuristic algorithms, which do not provide any guarantees on the optimality of the obtained solution and possibly require a large amount of time to converge.

PIAS presents a major restraint in the case of a sudden change in the workload or in the case of sub-optimal solutions, which may lead to a threshold-workload mismatch, which was shown to deteriorate PIAS performance by up to 25% [57]. Furthermore, due to the fact that PIAS employs the average DC-wide flow length distribution, in the case of mixed workloads servers will use demotion thresholds which are naturally mismatched with respect to their actual fine-grained local workloads.

For a wide range of traffic types, most of the demotion thresholds in Ω are located after the 90th percentile of the flow length distribution. Since the majority of realistic workloads are heavy-tailed and their empirical estimation is hard to achieve, the estimation errors reduce the efficiency of MLFQ. Indeed, if the tail of the workload has been underestimated, flows may be demoted too early, leading to most flows ending up in the low PQs too early. On the contrary, if the tail has been overestimated, flows may never be demoted to lower priorities, thus reducing the potential performance gain of MLFQ.

Despite the previously cited drawbacks, the main disadvantage of PIAS is the large number of required PQs. Indeed, although the majority of commercial switches offer up to 8 PQs, as previously discussed, most of them are typically unavailable since they are utilized to perform isolation of different types of traffic.

4.3 NOS2 overview

Differently from switches, implementing fine-grained scheduling at hosts does not pose significant difficulties thanks to the possibility of defining custom schedulers in the protocol stacks or using hardware accelerators [74] readily available in most of the commercial DC servers. On the contrary, commodity switches used in DCNs cannot implement sophisticated scheduling algorithms and even when they are capable of doing so they are restrained by the scarce amount of stateful resources. Furthermore, employing DC-wide distribution of flow length in order to set demotion thresholds for all servers present in the data center may lead to unfairness scenarios. Since the demotion thresholds are computed by taking into account the dominant workload type, rare workloads will be scheduled using a mismatched set of thresholds before even leaving the servers.

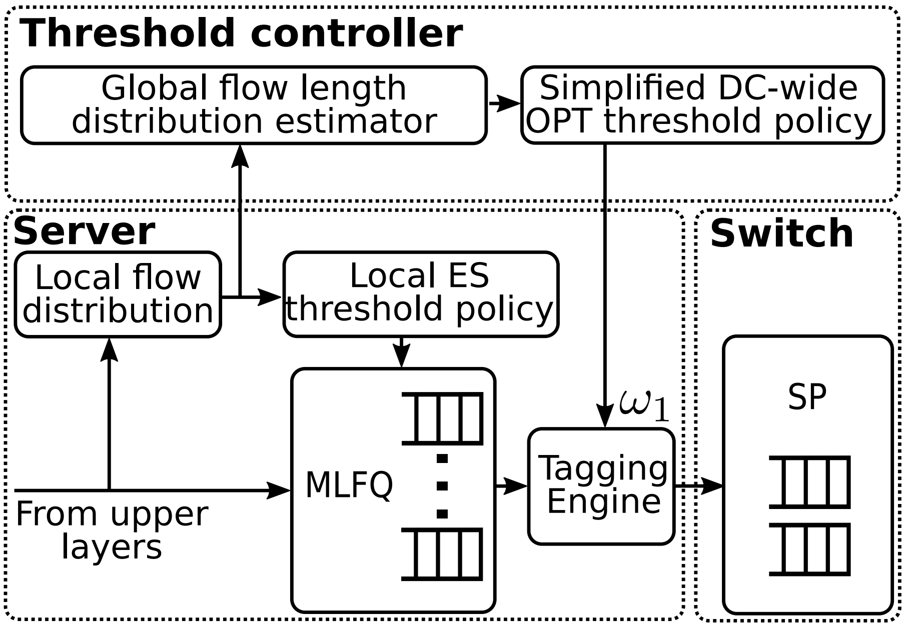

Threshold controller
Global flow length distribution estimator
Simplified DC-wide OPT threshold policy
Server
Local flow distribution
Local ES threshold policy
ω_1
MLFQ
Tagging Engine
From upper layers
Switch
SP

Figure 4.4: High-level architecture of NOS2, with multiple PQs at the servers and just 2 PQs in all the switches.

Inspired by the aforementioned observation, to overcome the shortcomings of previously proposed approaches, we propose the NOS2 architecture. Our key idea is to decouple the MLFQ system with multiple queues at the servers from the queueing occurring at the switches, which now adopt only 2 PQs, independently from the number of queues at the servers. The high priority queue at the switches is devoted to mice flows and the low priority one to elephant flows. Such choice is made on the basis that, as it will be shown in Section 4.4, fine-grained scheduling at hosts plays a fundamental role in the reduction of FCT while partially overshadowing the benefits of fine-grained network scheduling.

Figure 4.4 depicts the NOS2 architecture. Differently from PIAS and ES-N, NOS2 leaves to servers the complete freedom in managing their demotion thresholds on the basis of the locally observed workload. However, this does not introduce any expensive threshold computation since NOS2 involves simple ES thresholds for MLFQ scheduler at the servers. Furthermore, per-flow counters are not required in the switches, simplifying their operations. At the same time, to compensate for eventual performance losses due to non-optimized thresholds at servers, demotion thresholds inside switches are computed by the threshold controller based on the DC-wide flow length distribution, as in PIAS.

4.3.1 Threshold controller

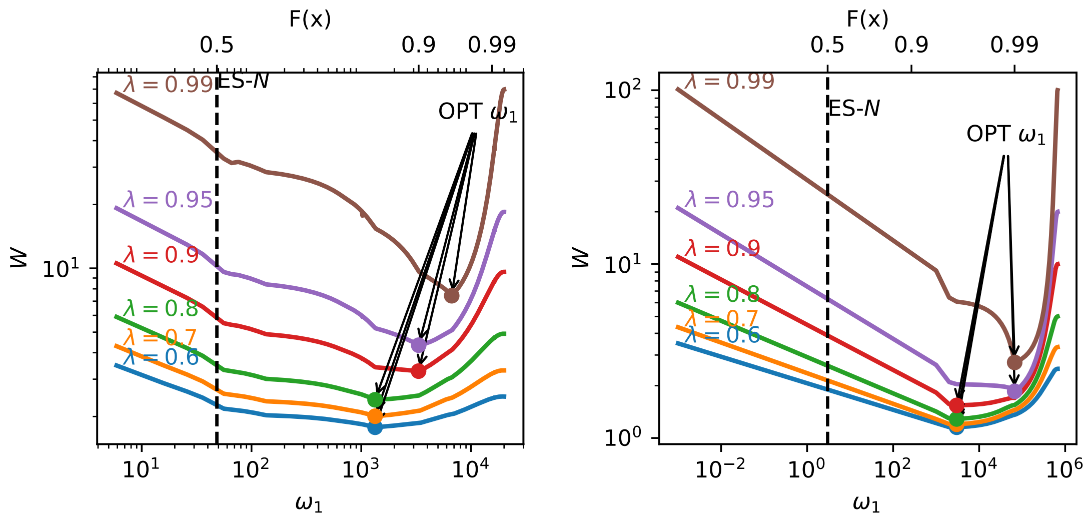
The central controller gathers flow statistics from the servers, computes the DC-global flow length distribution and computes Ω for the switches based on the OPT algorithm, which is provably optimal. Since the switches adopt only 2 queues and just one threshold is needed (i.e., $\Omega = \{\omega_1\}$) the complexity of the OPT algorithm is considerably lower with respect to the case of multiple thresholds. Intuitively, the controller must identify just the elephant and mice flows globally in the DC and instruct the switches to enqueue them in low priority and high priority queues, respectively. As shown in Figure 4.4, a single optimal threshold ω_1 is sent to all the servers.

Computing the optimal ω_1 represents a critical point in NOS2 architecture. We assume Poisson flow arrival and exponential service time. Given the average DC cumulative flow length distribution $F(x)$, the flow arrival rate λ and average flow length $1/\mu$, by recalling the formulation from Section 4.2.2 it is possible to find ω_1 by minimizing the following expression:

$$\min_{\omega_1} \frac{1}{\mu - \lambda\omega_1} + \frac{1 - F(\omega_1)}{\mu - \lambda(F^{-1}(1) - F(\omega_1)(F^{-1}(1) - \omega_1))}$$

where the first term corresponds to the normalized queueing delay W_1 for the high priority queue. This queue, modeled as an M/M/1 queue, is fed by an amount of data equal to $\lambda\omega_1$, being ω_1 an upper-bound for the average flow length observed by the queue. The second term corresponds to the normalized queueing delay W_2 for the low priority queue; here, $1 - F(\omega_1)$ is the fraction of the flows feeding the low priority queue, as $F(\omega_1)$ leave the system before reaching the second queue, and $F^{-1}(1)$ is the maximum flow length which is assumed to be finite. As in the previous case, λ is multiplied by the upper bound of the average length of flows, computed by integrating over the distribution entering the second queue.

The previous expression can be easily solved by employing numerical methods such



(a) Web-search workload (b) Data-mining workload

Figure 4.5: Average queueing delay W in function of ω_1 and λ. The ω_1 values computed according to (optimal) OPT and to ES-N are also shown.

as global minimization heuristics or by simply performing an iterative search. Furthermore, in [75] it is shown that, under non-restricting conditions, there exists a closed-form approximate expression to compute the optimal value of ω_1 with a good degree of approximation.

For the implementation of NOS2 we consider a simple heuristic based on local minimization with random sampling [76] inside the possible solution space which achieves an average convergence time in the order of seconds on an i7-6700K CPU, with a target relative accuracy of 10^{-3}. Figure 4.5 depicts how ω_1 computed with OPT and ES-N, changes in function of λ for two realistic flow length distributions, namely *web-search* and *data-mining*, described later in Section 4.4.1. It is worth noticing that the optimal ω_1, computed by OPT, is always located in the tail of the flow length distribution and leads to a considerably lower queuing delay with respect to ES-N.

4.3.2 Server scheduling and tagging

At each server NOS2 employs MLFQ with a simple ES-N threshold policy based on the local estimation of the workload, as shown in Figure 4.4. ES-N introduces greater adaptability to variable traffic patterns. Indeed, in the case of a sudden change in the workload, a new set Ω can be obtained as soon as the new flow length distribution is evaluated, without incurring additional latency of running complex OPT algorithm for multiple thresholds.

However, the aforementioned increased workload adaptability thanks to the use of ES-N comes at an inevitable cost of a decreased performance in terms of average FCT when compared to the OPT thresholds. Nevertheless, in [57] it was shown that when using ES-8, i.e., with 8 PQs, there is a mere 10% penalty in FCT with respect to using thresholds obtained by the OPT algorithm. This behavior is confirmed later in Section 4.4 by our simulation results which show an even smaller performance gap between ES-8 and OPT thresholds.

As described before, the role of discriminating flows between mice and elephant flows is left to the threshold controller. Given ω_1, servers perform a simple packet tagging with the use of a Tagging Engine depicted in Figure 4.4. Packets are tagged with either a high or low priority tag depending on the amount of already transmitted bytes for the corresponding flow.

4.3.3 Switch scheduling

As shown in Figure 4.4, the switch enqueues the incoming traffic into one of the two PQs based on the priority level specified by the tag present in the packet and set by the servers. Then, the 2 queues are served with a strict priority scheduling policy. Switches do not require to maintain any per-flow state, since the discrimination between elephant and mice flows is delegated to the servers and to the central threshold controller.

In summary, *the NOS2 approach integrates fine-grained scheduling with multiple priority levels at each server with coarse-grained scheduling at the switches that simply keep mice and elephant flows separated inside the DCN.*

4.4 Performance evaluation

To understand the effectiveness of the proposed approach, in this section we analyze the performance of NOS2 by means of extensive simulations.

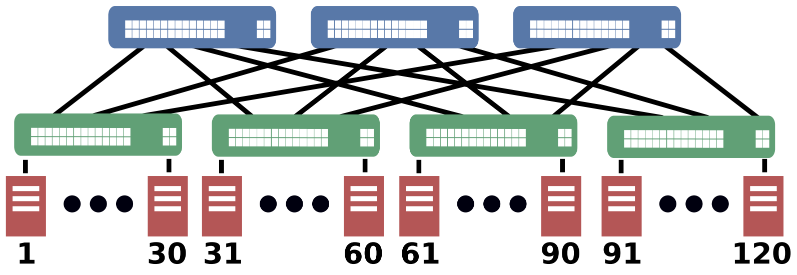


Figure 4.6: Data center topology used for the simulations.

4.4.1 Simulation methodology

We perform the analysis using the discrete-event simulator NS3 [77], which provides detailed simulation models for the Internet protocols stacks from the MAC layer up to the application layer. We consider a standard Ethernet-based leaf-and-spine topology for the DCN, as shown in Figure 4.6. The chosen topology connects 120 servers and comprises 3 spine switches and 4 leaf switches with 30 servers connected to each leaf switch. All servers are connected to the leaf switches via a 1 Gbps link, while leaf switches are connected to spine switches via a 10 Gbps link. Thanks to this link dimensioning, the DCN provides full bisection bandwidth.

The queueing mechanisms described in Section 4.3 are implemented at the output network interfaces of servers and switches. We employ DCTCP as the end-to-end transport protocol with marking thresholds and gain parameters set according to the guidelines provided in [61], while load balancing across the DCN is performed using equal-cost multi-path routing (ECMP).

Traffic flows arrive according to a Poisson process and the flow lengths are randomly generated according to the desired distribution. We classify flows into mice flows (<100kB), medium flows (>100kB and <10MB) and elephant flows (>10MB). The destination of each flow is uniformly chosen at random among the other 119 servers. The data center normalized load λ is defined as the average amount of traffic destined to the servers, normalized with respect to the bisection bandwidth. We vary λ between 0.5 and 0.8 to simulate different loads inside the DC. We do not show the results for $\lambda < 0.4$ since the difference in the performance of all the considered schemes is negligible.

We consider for our experiments two realistic, heavy-tailed flow length distributions, depicted in Figure 4.7: Data-Mining (DM) [66] and Web-Search (WS) [61]. The

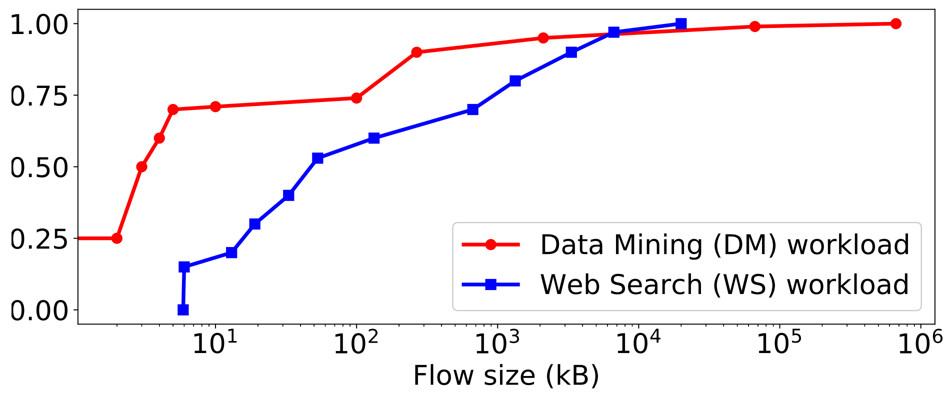


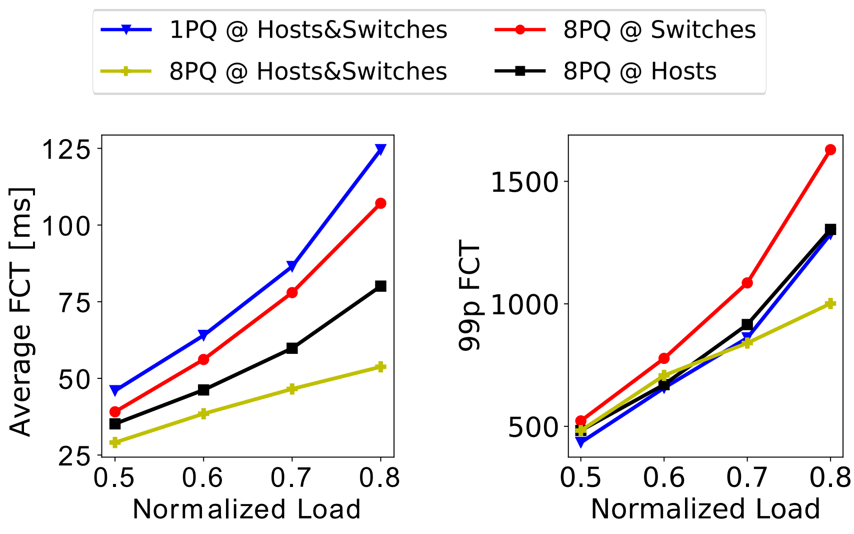
Figure 4.7: Flow length distributions employed in simulation.

former is composed predominantly of mice flows (80%) and elephant flows (20%). The latter instead shows a smoother transition from mice to elephants. We consider two scenarios: in the case of *homogeneous workload*, all the servers generate traffic according to one of the two flow length distributions (DM or WS), whereas in the case of *heterogeneous workload*, 50% of the servers, chosen at random, generate traffic according to the DM workload and the remaining servers according to the WS workload.

We report the *FCT gain*, defined as the FCT achieved by DCTCP divided by the FCT of the considered scheme. Thus, an FCT gain G >1 means that the FCT is reduced by a factor G with respect to DCTCP. We evaluate the average value and the coefficient of variation (CV) of the Flow Completion Time (FCT) gain, for all the flows, and, independently for each class of flows (mice, medium and elephant). The FCT is measured from the generation at the source server of the first packet belonging to a flow until the reception of the last packet at the destination server. Evaluating the FCT is important to infer the performance of typical DC delay-sensitive applications, which directly affects the Quality of Experience perceived by the end users accessing cloud-based applications.

We compare NOS2 with respect to the following alternative solutions:

- *ES*-8: the architecture in Figure 4.3 with MLFQ, $N = 8$ and ES-8 threshold computation.
- *PIAS*: the architecture in Figure 4.3 with MLFQ, $N = 8$ and OPT thresholds computation. The optimal threshold values are taken from the publicly available PIAS



(a) Average FCT

(b) 99-percentile FCT

Figure 4.8: Impact of the number of PQs and their location

source code [78].

- *DCTCP*: as a reference case, we run bare DCTCP at servers with a single queue at servers/switches, thus disabling completely any flow prioritization scheme.

4.4.2 MLFQ granularity comparison

To highlight the importance of isolating different types of flows in different queues, we conduct a series of preliminary experiments by using WS distribution and by varying the amount of PQs inside servers and switches. For these experiments, we consider the ES-N architecture.

Figure 4.8a shows that server scheduling plays the most significant role for the reduction of FCT with respect to switch scheduling. Indeed, using an MLFQ with 8 PQs at the server while keeping only 1 PQ at switches leads to a reduction in FCT by a factor 1.33 when compared to the opposite scheme with 1 PQ at servers and 8-level MLFQ at switches. Combining the two schemes leads to an ever further reduction in FCT with a mere increase by a factor of 2 in FCT when moving from $\lambda = 0.5$ to $\lambda = 0.8$.

Figure 4.8b instead shows the 99-percentile FCT for the same set of experiments. The FCT of the scenario employing 8-layer MLFQ at switches presents a considerable increase in FCT compared to other schemes, which remain comparable up to $\lambda = 0.7$ after which they start to diverge from the scenarios employing 8 PQs both at servers and at switches.

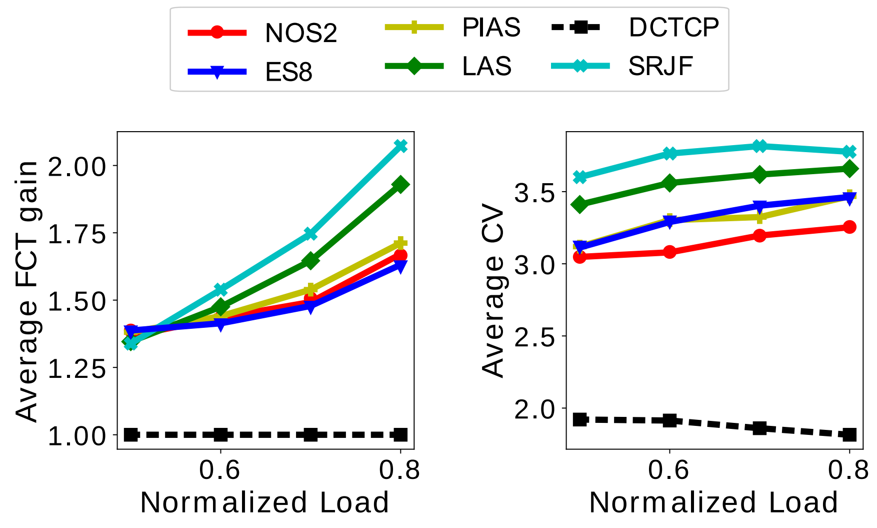
4.4.3 Comparison to other scheduling mechanisms

To show how NOS2 compares to other scheduling mechanisms, we conduct a series of additional experiments by considering the scheduling mechanisms described in Section 4.2. Notably, we consider SRJF and LAS, which, as previously discussed, require complex switch architectures and in the case of SRJF detailed information about the length of each individual flow.

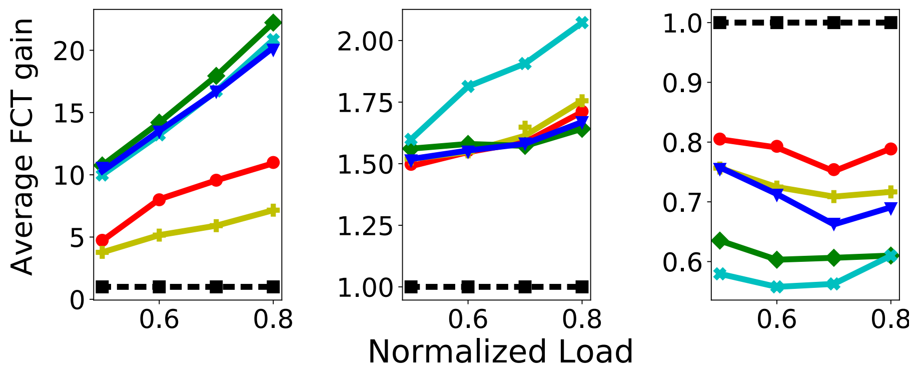
Figure 4.9 shows a comparison of the different scheduling mechanisms. Similar results have been obtained using the data-mining distribution. While all scheduling mechanisms have a comparable CV of FCT gain, Figure 4.9a shows that there is a big discrepancy in the average FCT when comparing algorithms using a fixed amount of PQs to SRJF and LAS. When observing the FCT subdivided by flow type in Figure 4.9c, both LAS and SRJF are shown to achieve considerable FCT gain both in the case of medium and mice flows, while having their performance considerably deteriorated in the case of elephant flows. Since ES8 offers fine scheduling granularity for mice flows, it achieves a comparable gain for that kind of flows. However, it penalizes medium and elephant flows, with the former case having worse performance gain than NOS2. Interestingly, PIAS targets the improvement of medium flows, which are the dominant type of flows in the web-search distribution and thus contribute to the overall better average FCT in respect to NOS2 and ES8.

4.4.4 Performance under homogeneous workload

Figures 4.10a, 4.11a show the aggregate average FCT under WS and DM workloads respectively. In both cases, NOS, ES-8 and PIAS behave very similarly in terms of average FCT. We now focus on the performance of mice, medium and elephant flows. Under the WS workload, Figure 4.10c shows that, for mice flows, the best FCT is obtained by ES-8, the worst by PIAS, with NOS2 showing an intermediate behavior. The opposite holds for the elephant flows, while for medium flows the FCT is almost unaffected by



(a) Average FCT gain (b) CV of FCT gain

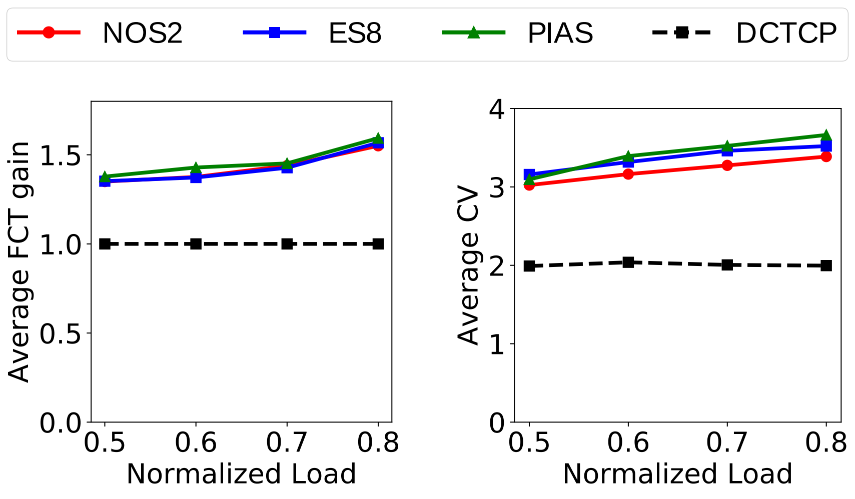


(c) Average FCT gain for different flow types: mice (left), medium (middle), elephant (right)

Figure 4.9: Comparison of different scheduling schemes

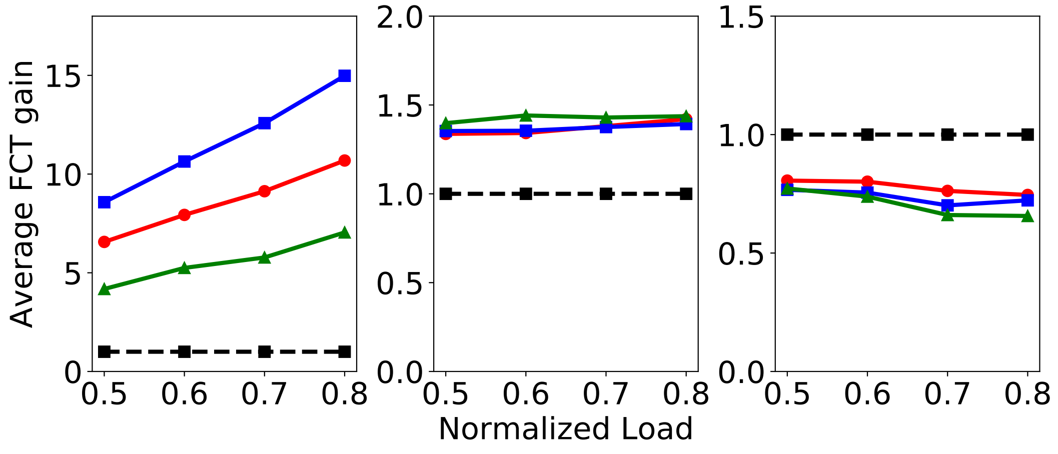
the adopted scheme. On the contrary, Figure 4.11c shows that, for the DM workload, only the performance experienced by medium flows are affected by the adopted scheme, and, in this case, the best scheme is PIAS while the (relatively) worst is NOS2, which still keeps the FCT gain larger than one. It is worth noting that under DM workload the amount of medium flows accounts only for 4% of the flows, which explains why the performance gap between the three architectures in Figure 4.11a is so mild.

The explanation behind the relative performance degradation for NOS2 for medium-sized flows is that 2 PQs cannot offer enough scheduling granularity to differentiate



(a) Average FCT gain

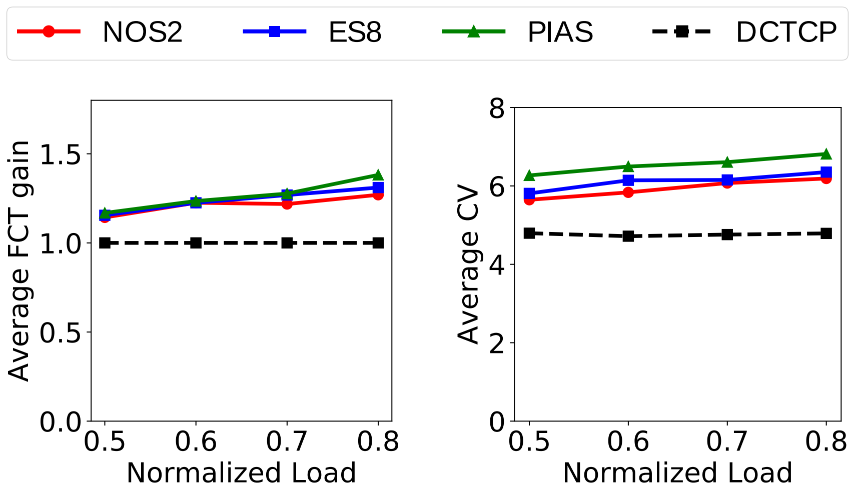
(b) Average CV of FCT gain



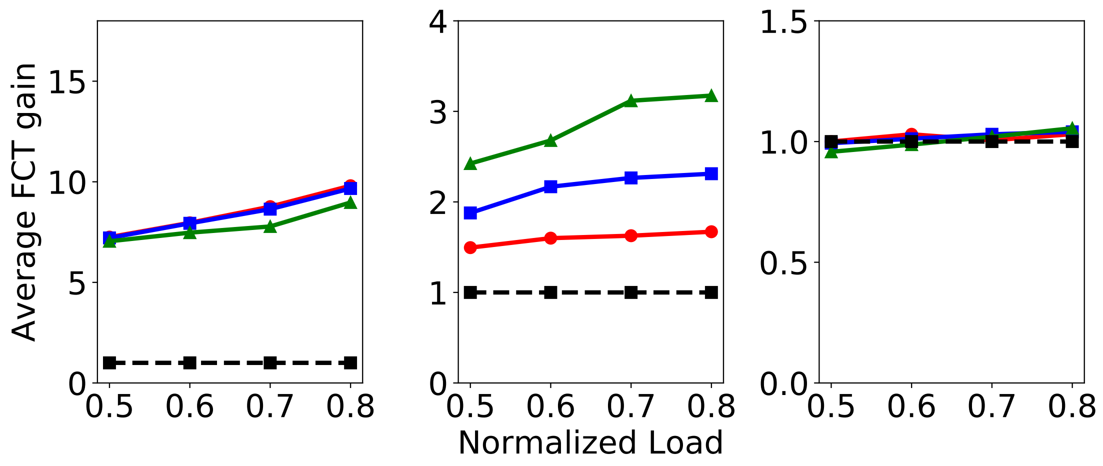
(c) Average FCT gain for different flow types

Figure 4.10: Web-search workload

among all flow types. Since the majority of flows in DM workload are either mice or elephant, the threshold setting which optimizes the average FCT tries to optimize those two types of flows at the price of sacrificing the performance of medium flows. On the contrary, the WS workload presents a higher concentration of medium flows, which leads to the threshold being optimized for those flows. Similar reasoning applies when it comes to explaining the performance of elephant flows. For the WS workload, the performance for elephant flows is penalized considerably compared to DCTCP. Although WS and DM have a similar fraction of elephant flows, WS distribution presents



(a) Average FCT gain (b) Average CV of FCT gain



(c) Average FCT gain for different flow types

Figure 4.11: Data-mining workload

a considerably shorter tail, thus a smaller average length of elephant flows. In view of the discussion made in Section 4.2, this leads to a bigger penalty when mice flows are prioritized over them. On the other hand, in the case of DM workload the tail of the distribution is considerably longer, thus this effect is less visible.

For what concerns the CV of the FCT gain, reported in Figures 4.10b, 4.11b, NOS, ES-8 and PIAS perform similarly, with NOS2 performing slightly better than ES-8 and PIAS for both WS and DM workloads. There is still a considerable gap between the three architectures and DCTCP. This is not surprising, since DCTCP aims at treating each flow fairly while NOS2, ES-8 and PIAS privilege short flows at the expense of long flows, thus increasing the variance of the FCT.

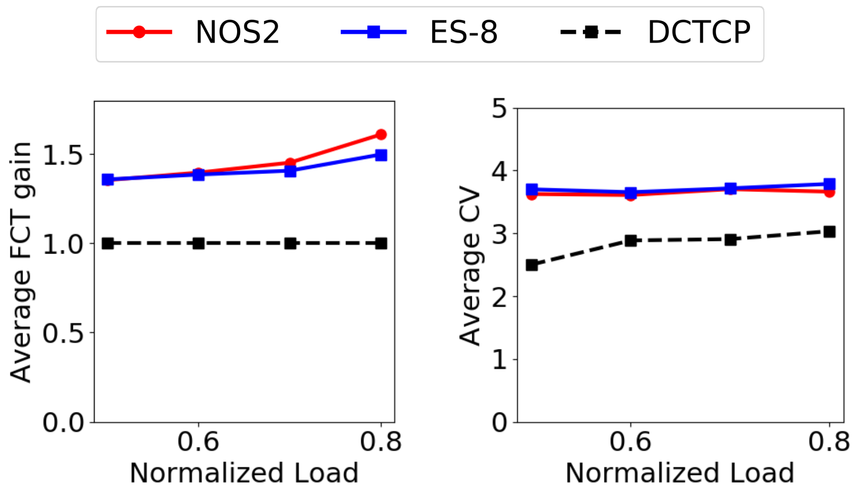
4.4.5 Performance under heterogeneous workload

Under heterogeneous workload, as defined in Section 4.4.1, we compare NOS2 only with ES-8, because the publicly available source code for PIAS provides the values of the demotion thresholds only for homogeneous DM and WS workloads. Figure 4.12a shows that, for $\lambda \geq 0.7$, NOS2 outperforms ES-8 because each server employs demotion thresholds tailored to their own specific workload instead of using those based on the average data center flow length distribution, as in ES-8. At the same time, the DC-wide optimized thresholds, which are instead based on the average flow length distribution, are able to compensate for the mixture of two different workloads. As shown in Figure 4.12b, there is no significant difference in terms of the CV of FCT gain in the heterogeneous case when compared to the homogeneous one.

To show the robustness of NOS2 to different concentrations of workloads, we vary the fraction of servers employing DM workloads from 10% to 90% for load 0.7 as to simulate different mixed workload patterns. Figure 4.13 show that the performance of NOS2 are almost independent of the fraction of DM workload with an average FCT gain that stays stable around 1.5 for all of the considered mixed workloads. Furthermore there is no substantial variation in the CV of the FCT gain which confirms the robustness of NOS2 to different workloads.

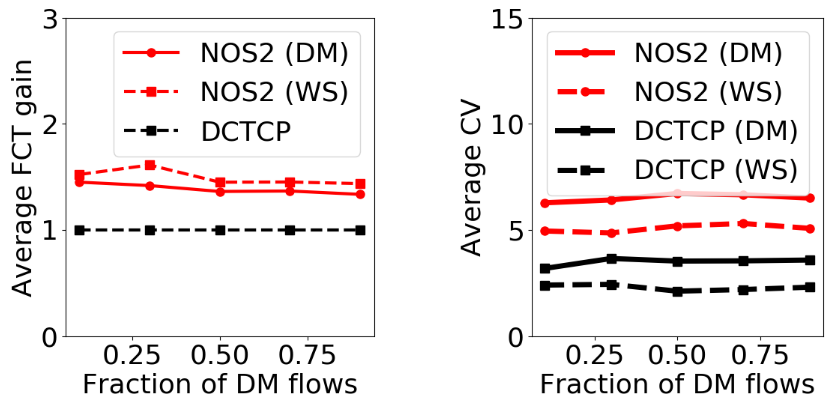
4.4.6 Robustness to wrong workload estimations

The estimation of the tail of the flow length distribution requires observing the completion of elephant flows. During workload changes, a considerable amount of time is required to estimate correctly the contribution of elephant flows and thus update the flow size distribution. During this transient phase, the assignment of thresholds is performed in a suboptimal way since it is based on incomplete information which in turn affects how different flows are distributed among different queues inside the network and ultimately affects the FCT.



(a) Average FCT gain (b) Average CV of FCT gain

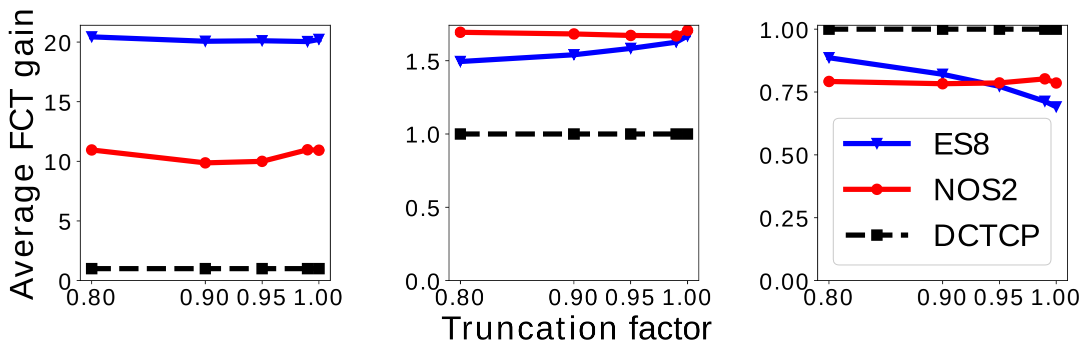
Figure 4.12: Heterogeneous workload



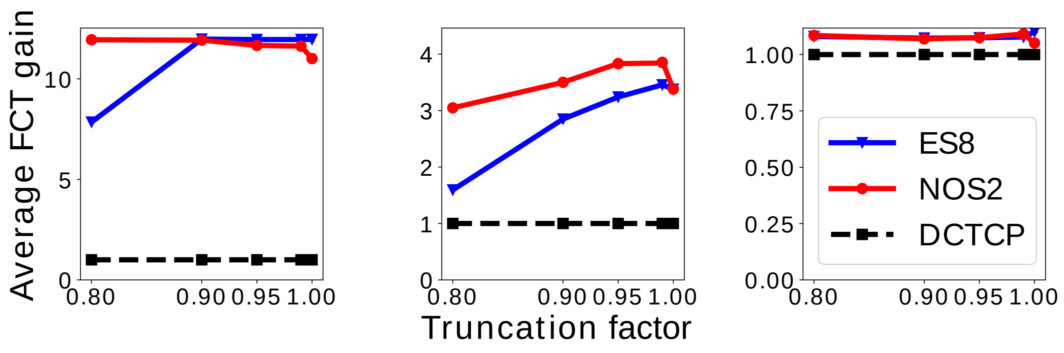
(a) Average FCT gain (b) Average CV of FCT gain

Figure 4.13: Performance under heterogeneous workload and variable fraction of DM workload, for normalized load 0.7

As shown in Figure 4.5b and Figure 4.5a the position of the demotion threshold is agnostic to the shape of the lower percentiles of the flow length distribution as it highly depends on the shape of the tail of the flow length distribution. This effect is noticeable especially for high loads which observe ω_1 being positioned after the 95-percentile of the flow length distribution. Consequently, even such a small underestimation of the flow length distribution will lead to suboptimal ω_1.



(a) Web-search workload



(b) Data-mining workload

Figure 4.14: Performance of mice (left), medium (middle) and elephant (right) flows in the case of workload underestimation.

To highlight this effect we consider a scenario in which the tail of workload is underestimated. We consider a truncated flow length distribution $F^{(\alpha)}(x)$ for the threshold assignment, while keeping invariant the actual distribution $F(x)$ for the traffic generation. In more detail, we truncate the original flow length CDF $F(x)$ at a given α factor such that the support is limited by $F^{-1}(\alpha)$, as follows (assuming $F(0) = 0$):

$$F^{(\alpha)}(x) = \frac{F(x)}{F^{-1}(\alpha)} \qquad \text{for } x \in [0, F^{-1}(\alpha)]$$

This approach provides a scenario in which the CDF are perfectly estimated up to a given percentile but elephant flows have still not been observed.

Figs. 4.14a and 4.14b show respectively the FCT of web-search and data-mining workloads with 0.8 normalized load and with α varying from 1 (i.e., the ideal case with no truncation) to 0.8 (i.e., 20% of the elephant flows have not been considered). In the case of WS workload there is no significant variation in the FCT in respect to the truncation factor for mice flows. For other types of flows ES8 trades the performance of medium flows for the performance of elephant flows while the performance of NOS2 remains consistent in respect to the truncation factor. On the other hand, in the case of DM workload the performance of elephant flows stays invariant in respect to the truncation factor, but ES8 suffers considerably when the truncation factor decreases with a loose of performance of almost a factor 2 when the truncation factor is set to 0.8. In conclusion, NOS2 appears robust to workload underestimation, occurring during workload changes.

4.5 Future work

As discussed throughout the chapter, NOS2 considerably simplifies the complexity of employing MLFQ schedulers in commodity DCNs, achieving a better trade-off between complexity and performance than alternative solutions.

Although presenting a simple and concise architecture, NOS2 still relies on a central entity to compute the demotion thresholds for the traffic split. Recent advances in the field of programmable data planes [3], [8],

and their adoption in commercial DC solutions enable to offload the estimation of the network-wide CDF directly to the DCN switches. Furthermore, programmable data planes offer enough flexibility to perform also the computation of the demotion thresholds directly within the DCN, thus removing all the limitations that derive from having a single central entity.

Although, NOS2 solves the issue of the workload-threshold mismatch when scheduling flows at servers, it still does not take into consideration the diversity in the spatial distribution of workloads. Since realistic workloads typically present a clustered behavior for different workloads [52], to achieve even better performance a higher number of demotion thresholds could be employed. Investigating the possible benefits of such an increase in scheduling granularity and the consequences from the computational point of view is an important research direction left for future work.

4.6 Discussion

In this Chapter we introduce NOS2, a low complexity flow scheduling mechanism. We design NOS2 with the aim of achieving a simple yet performant scheduling algorithm which is ready to be deployed on commodity switches already present in modern DCs. Indeed, our simulation results show that NOS2 is capable of achieving close to the state of the art performance while keeping a low complexity which remains comparable with capabilities of currently deployed commercial switches. Finally, we believe that NOS2 hits the sweet spot in terms of trade-off between performance and complexity.

Chapter 5

Removing Human Players from the Loop: AI-Assisted Assessment of Gaming QoE

Part of the work presented in this chapter has been published in:

- Sviridov G, Beliard C, Bianco A, Giaccone P, Rossi D. "Removing human players from the loop: AI-assisted assessment of Gaming QoE." In *IEEE INFOCOM Workshop on Network Intelligence.* 2020.
- Sviridov G, Beliard C, Simon G, Bianco A, Giaccone P, Rossi D. "Leveraging AI players for QoE estimation in cloud gaming." In *IEEE INFOCOM Demo and poster session.* 2020.

In previous Chapters major attention has been dedicated to analyzing and improving flow performance at different levels of the network. We went from generic minimization of congestion using programmable data planes in wide-area network in Chapter 2 to fine-grained traffic control and flow scheduling in DCNs in Chapters. 3-4. Yet, as we discussed in Chapter 1 the majority of traffic in the Internet nowadays is related to user-centric applications. Notably, users do not require the smallest possible flow completion time nor do they care. Instead they demand a given level of performance and interactivity depending on the actual application they are using. This leads to the question whether a blind reduction of the flow completion time is the correct approach for traffic optimization. Following the previous observation one must adapt the traffic optimization schemes employed in the network based on the nature of the service carried inside individual flows and to guarantee a certain level of satisfaction for the end-user. A service such as video streaming can tolerate high latency, (as long as the bandwidth stays adequate during the flow existence). On the other hand, video calls, trading applications or remote surgery have very strict requirements in terms of latency. Finally, there exists a grey area of applications for which such kinds of requirements are usually unknown or difficult to achieve. Such is the case of online video games.

The video game industry is skyrocketing, with a market value that has surpassed that of the film industry and is expected to exceed 230 B$ by 2022. Due to its booming user base and the increasing social factor in recent games, the video game industry is starting to have a significant influence on society. Single or two-player mode games are long surpassed: most games today either offer an *optional* online mode, or can be *exclusively* played online with other users. Due to the necessity of interconnecting different players and guaranteeing high levels of gaming experience, the gaming industry has begun to attract a significant interest of network operators. ISPs (such as Comcast [79]) have started to offer game-specific broadband plans that promise higher Quality of Experience (QoE) for online games. Equipment vendors (such as Ciena [80] and Barefoot [81]) put the game use-case at the heart of their network architecture evolution. Game service providers (such as Google Stadia [82]) have started to offer cloud-rendered games directly streamed to the user's home. However, the initial customer experience appears to be disappointing [83] at best. This calls for further research to ensure that networks appropriately handles gaming traffic.

The large spectrum of video game genres, and the large overall video game catalog, makes the above task very complex [84]. Some games (e.g., first-person shooters or brawlers) are extremely fast-paced and require swift reaction times from the players, while others (e.g., strategy or turn-based games) are intrinsically less sensitive to latency. Network conditions clearly play a significant role in the user experience for the former. Large network latency or packet drops can reduce the player's performance to well below the natural score. Furthermore, among similar interactive games, the effect of a given amount of latency (or packet drop rate) may have a different impact, significantly hampering playing ability in one game while and being unnoticeable to the player in another.

Due to the presence of complex in-game dynamics, it is usually wrong to assume that two apparently similar video games will lead to similar QoE under the same network conditions. Even different modes of the same first-person shooter game may lead to different responses from users depending on network conditions, as shown in [85]. This heterogeneity precludes any kind of generalization of existing results to different video games and therefore forces us to analyze QoE on a game-by-game basis. This analysis is typically done with the participation of human players and is notably a very time-consuming and expensive task that cannot keep up with the steady introduction of new games to the market.

In this work, we take a completely different approach and advocate that it is possible to *remove human players from the QoE assessment loop*. While this may seem a bold statement at first, we base our claim on the observation that player satisfaction is naturally related to the score they are able to achieve. Whereas everybody is aware of the famous Pierre de Coubertin quote "*The important thing in the Olympic Games is not to win, but to take part*", there is little doubt that the winner of the 2019 Fortnite game competitions that brought home $ 3M [86] would think the same. We can also understand the feelings of any player who fails to win the prize or is not able to compete fairly because of bad network conditions.

Our key idea is to use scores as a proxy for user satisfaction, allowing us to automate and scale up the game assessment process. More explicitly, we propose to exploiting recent advances in the field of Artificial Intelligence (AI), whereby AI agents are able to autonomously learn complex tasks such as video game playing [87] without any human intervention. Based on these agents, in [88], [89] we thus propose and implement a framework for automated assessment of game scores, and in particular of *game score degradation in presence of network impairments.* We use this framework on three games, gathering insights on their score degradation characteristics. These data are then used to explore how network device packet handling mechanisms might be designed to reinstate game score fairness, using scheduling as a proof of concept. Finally, we discuss the issues raised in this work and draw conclusions.

5.1 Related work

A first class of work related to our concerns QoE assessment in video games, which has been widely studied throughout the first decade of the 2000s. A major effort has been put into analyzing the impact of network conditions on different games (Section 5.1.1). A second, and so far separate, class of work concerns the use of AI for video game playing. This is a much more recent field with significant contributions during the last few years (Section 5.1.2).

5.1.1 Assessing video game QoE

On the track of perceived QoE, the authors of [90] analyze how the Mean Opinion Score (MOS) for the game "Call of Duty" depends on network conditions. They show that the MOS of novice players remains constant even when latency exceeds 100 ms, while it sharply decreases for experienced players after only 50 ms of latency. Similarly, in [91] the authors show that there is a substantial difference in the satisfaction of different levels of racers in an online racing game. It is shown that there is a big discrepancy on how both the game score and the perceived QoE varies for professional and novices players with former players being able to compensate better for induced network latency without score deterioration but stating that the game felt unrealistic.

More quantitative results are provided in [85], which evaluates the impact of network conditions on user score in the "Unreal Tournament 2003" game. The authors show that, depending on the game mode, a score degradation of 10 %-25 % is experienced with 250 ms of latency. This is heavily in contrast with numerous previous studies which fixed the usability threshold for fast-paced games at 100 ms [84]. On the same track in [92] the authors measure the frustration level of the players by analyzing the average game session duration of an MMORPG in function of the network conditions. They show that the session duration has an exponential decay in function of both network latency, jitter and packet loss. Although being a coarse view these results give insight on the average behavior of the users.

Given the above inconsistencies arising from human players, the research community has used in-game bots to estimate the impact of latency on the average score in “Quake” game [93]. Nevertheless, in-game bots are practically encoding expert-knowledge concerning a specific game as a set of human-programmed software heuristics.

The above approaches are thus both time consuming and cannot be generalized to other games. We seek to avoid these disadvantages by leveraging self-learning agents. *To the best of our knowledge, no literature work has so far employed AI agents for the purpose of automated score estimation across a set of games, which is thus one of our main contributions of this work.*

5.1.2 AI for video game playing

More recently, significant research effort has been devoted by the AI research community to develop artificial agents capable of generalizing to multiple environments. Whereas it is out of the scope of this work to provide full coverage, for which we refer the reader to a comprehensible survey [94], we cover here the techniques we use in this work.

In particular, [87] proposes Deep Q-Networks (DQN), an application of the classic Q-Learning algorithm in the context of Deep Neural Networks. The proposed algorithm is able to learn to play Atari games by observing raw pixels and is able to achieve super-human score levels for most of them. Many improvements over the original DQN proposal have then since been proposed in follow-up research [95]–[98] with some of them targeting more complex video games such as “Doom” game [96] or “Starcraft” game [98]. Yet in the Atari environment most of the DQN variations perform similarly to the original algorithm.

Among them, Deep Recurrent Q-Network (DRQN) [95] introduces a recurrent neural network at its core. DRQN has been extensively used to develop agents for more complex games such as Doom [96] and showed excellent results against agents trained with more complex model [97].

Our work differs from this prior work in that we leverage the proposed techniques to achieve a different goal, namely estimating QoE under network impairments.

5.2 Methodology

In a nutshell, our methodology consists in (i) emulating a controlled Cloud Gaming (CG) environment and (ii) training AI agents to let them play in our environment. We tune the network conditions and measure the agents' gaming performance.

5.2.1 System model of Cloud Gaming (CG)

Although being fairly novel, CG [99] is rapidly gaining momentum in the gaming community with numerous services having been released in recent years including the well-known Stadia platform [82]. Figure 5.1 depicts a basic CG model.

The fundamental idea of CG consists in employing *remote rendering* of the video game environment and streaming the high-definition rendering to the clients in the form of real-time video. This is in contrast with traditional video games that are rendered on a local machine owned by the player. In the case of traditional single-player games, the entire state of the game is kept locally on the local machine, so that the perceived QoE is influenced only by the performance of the user hardware. In the case of traditional multi-player games, only metadata related to player actions are transmitted to the server, that maintains the entire state of the game.

CG on the other hand, delegates the rendering responsibilities to the cloud and clients are no longer required to possess expensive dedicated hardware for local rendering. Instead, it suffices for them to be equipped with a device capable of displaying a basic video stream, such as a smartphone or a low-end laptop, for instance. CG clients interact by sending actions (e.g., keystrokes, pad or mouse movements) to the cloud, that affect the game state. The disadvantage is that, unlike in traditional video gaming rendered on local hardware, the streaming of the cloud-based rendering can be affected by varying network conditions.

From the point of view of the network requirements, CG gives rise to additional challenges as it combines elements of bandwidth-hungry applications (such as high-definition streaming) with highly delay-sensitive applications (such as real-time communication and control). It is reasonable to assume that different video games running in the cloud have roughly the same requirements in terms of bandwidth, as all require a

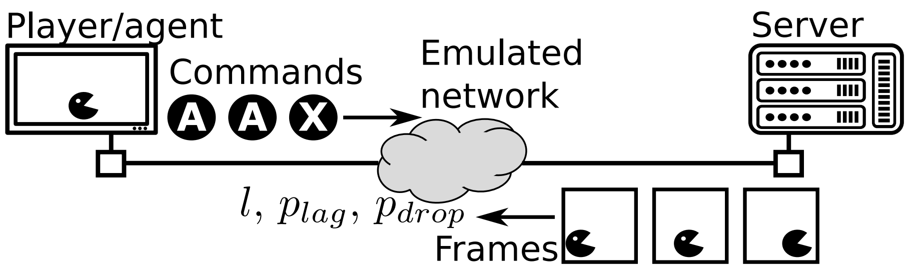


Figure 5.1: Cloud Gaming (CG) emulated system.

steady stream of 4K video data. However, when it comes to latency, under the same network conditions user-perceived QoE may vary significantly depending on the nature of the considered video game.

In our experimental setup, both the agents and the server are in the same high-performance network, so that we are able to measure the gaming experience (i.e., AI agent score) in both ideal conditions (sub-millisecond delay, high-bandwidth, no loss) as well as under controlled network degradation. In particular, as depicted in Figure 5.1, we can actively control latency and losses at the frame level, with several scenario settings and on several games, as we detailed later in the experimental section (Section 5.3).

5.2.2 Automatizing game playing

In [93] a first attempt has been made to substitute real human players by in-game bots to assess video game QoE. However, bots are not representative of real human behavior as they can exploit hidden game states, and are not limited to using only the visual information available to humans. Additionally, this approach cannot be applied when internal game states are not available, as in the case of online video games and for CG.

Recent advances in the field of AI have led to multiple models of artificial agents having great flexibility in adapting to different environments. As an example, an AI agent developed to play Atari games is equally capable of learning to play almost *any* Atari game without any human intervention or modification to the actual algorithm.

Without loss of generality, in this work we consider three different Atari games, whose screenshot is depicted in Figure 5.2. The choice of using Atari games instead of more complex ones is due to the (relative) simplicity of training artificial agents.

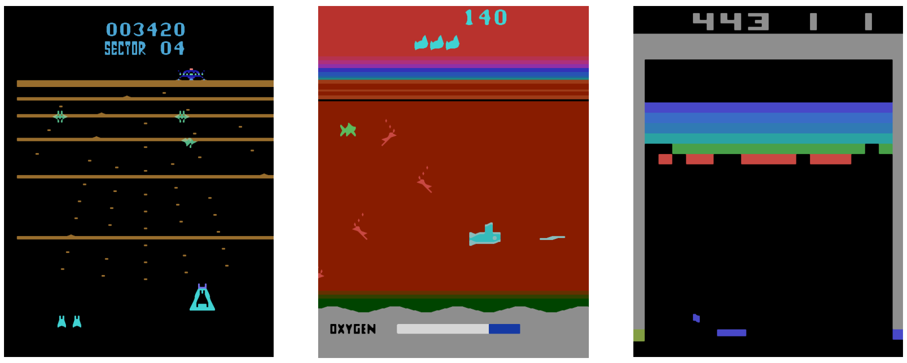


Figure 5.2: The set of Atari games considered in this work. From left to right: Beamrider, Seaquest, Breakout.

Furthermore, the Atari suite provides a large variety of games, which allow us to highlight the aforementioned heterogeneity in latency sensitiveness. We discuss how our technique applies to more complex games in Section 5.4.

Analogously to real humans, AI agents are built on top of their ability to directly employ the *raw pixels as input to their decision-making process.* Furthermore, as in the case of real humans, the agent decision process is built using a process of "trial and error", to progressively improve the gaming performance (i.e., the game score). This allows agents to generalize to different scenarios by decoupling them from the actual game engine and makes them a promising technology for replacing humans in game QoE assessment.

5.2.3 Training AI players

More formally, we use Reinforcement Learning (RL) techniques to train our artificial agents. In RL, an artificial agent interacts with an environment (i.e., our emulated CG system) by means of a closed-loop feedback system. At each time instant t the agent receives as input a representation of the environment (i.e., a state s_t that in our case is a video frame) and in turn reacts by performing an action a_t (i.e., a keypress). Following the action, the agent receives a new state s_{t+1} and the reward r_t (in our case, a possible change in the game score) corresponding to the state transition $s_t \xrightarrow{a} s_{t+1}$.

Practical limits of RL for games

An ideal agent must be able to perform actions so that at each time t, given the current state s_t, the action will lead to a maximum *future discounted reward* $G_t = \sum_{k\geq 0} \gamma^k r_{t+k}$. The discount rate $\gamma \in [0, 1]$ tunes the decision making process by making a compromise between immediate rewards (small γ) and possible future rewards (large γ).

As the definition of the future discounted reward depends on s_t only, it intrinsically incorporates a notion of agent behavior. Future rewards r_{t+k} are given by the current behavioral model of the agent, i.e., by its policy function $\pi(a|s)$. This function dictates the probability of taking an action a given the current state s. In principle, it is then trivial to build an explicit expression for the optimal policy $\pi^*(a|s)$ which maximizes $\mathbb{E}[G_t]$. However, explicitly computing $\pi^*(a|s)$ becomes burdensome when the number of states grows large.

Human-like "trial and error" and Q-Learning

A first practical limitation of RL is that learning the optimal policy proves to be challenging due to its excessive computation cost. Thus, multiple *approximate algorithms* have been proposed, including Q-Learning, which is among the most widely used and studied approximate RL techniques. The main concept behind Q-Learning is finding an optimal state-action function $Q^*(s, a)$, which rewards $\mathbb{E}[G_t]$ by performing action a while in state s at time t. Q-Learning employs a fundamental result from the field of dynamic programming, namely the *Bellman equation*, that determines Q^* in an iterative fashion. Briefly, Q-Learning works in an incremental fashion by updating the Q function at each time step t as in (5.1).

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r + \gamma \max_{a'} Q(s_{t+1}, a')) \tag{5.1}$$

Starting from sub-optimal Q and simply selecting at each time step the action a that maximizes Q, i.e., $a = \arg\max_{a'} Q(s, a')$, and updating the new estimate of Q, provides guarantees of converge with probability 1 to optimal Q^* under non-restrictive conditions. From Q^* it is immediate to define the optimal policy $\pi^*(a|s)$. *This allows agents to improve their score over time, learning to play like humans do.*

Human-like vision and CNN

A second practical limitation of RL is that approximate methods reach their limits when the state space becomes very large since building an explicit Q function is then unfeasible. Consider a simple arcade game whose state is represented by the current frame. Even a monochrome resolution as small as 64×64 pixels will lead to a state-space of $2^{64\times 64}$.

Deep Q-Network (DQN) [87] has been proposed as a solution to overcome the limitation of the exploding state-space, by leveraging recent advances in the field of computer vision achieved through Deep Learning (DL) techniques. In particular, instead of employing an exhaustive Q function, DQN employs a Convolutional Neural Network (CNN) to infer the Q function, starting from the raw pixels of the frames. Specifically, DQN extracts the most significant game features from raw pixels followed by a multi layer perceptron which performs the regression over the extracted features and outputs the estimate of the Q function. *This approach allows agents to play any game, as a human would do, through a CNN equivalent of a visual interaction with a rendered CG game.*

Training in practice

We use OpenAI Gym toolkit [100] with the ALE environment [101] to train agents to play three different Atari games shown in Figure 5.2 using the model described in [87]. The agents are trained on two NVIDIA V100 GPUs, letting them interact with our emulated CG in unperturbed settings, i.e., with ideal network performance. We report the training results in Figure 5.3 showing the average score achieved by the agent as a function of the number of update steps. On average a training time of 22 hours (equivalent to 10M training epochs) is sufficient to reach the maximum score for all of the considered games.

5.3 Experimental evaluation

In this section, we first provide details about the experimental setup (Section 5.3.1). We then use the agents trained as described in the previous section to assess the impact of

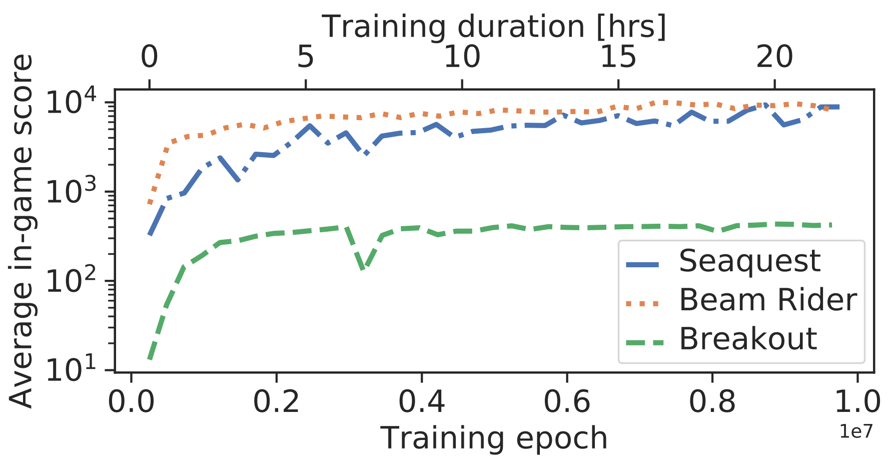


Figure 5.3: Training performance of DQN for three Atari games.

network conditions on score degradation (Section 5.3.2). Finally, we show how we can exploit our findings to improve the QoE by acting on network scheduling (Section 5.3.3).

5.3.1 Network model

With reference to the CG architecture depicted in Figure 5.1, we model the communication channel between the player and the game server as an asymmetric link characterized by the parameters depicted in Table 5.1. Notably, we tune (i) the probability that a frame is delayed p_{lag}, (ii) the amount of latency l and (iii) the keystroke loss probability p_{drop}. Note that l is defined in respect to the action taken during the delayed frame and is usually referred to as *lag*.

We consider the case where the communication channel is adequately over-provisioned in terms of bandwidth to support multiple simultaneous games, thus no network congestion is experienced. For the sake of simplicity, we assume no inter-frame video encoding. Thus for each update of the game state a full frame containing the new state of the visual playing environment is sent to the client. Frames are sent at a constant rate of 60 frames per second. This simplistic setting is a reasonable simplification given that, for the time being, we do not desire to tackle the problems of bandwidth adaptation logic in the video streaming portion of the CG (see Section 5.4).

Frame delays and losses can thus be synthetically added to control network conditions. For the sake of simplicity, we consider simple independent probabilistic models

Table 5.1: Channel parameters employed in the experiments

Variable	Description	Range
p_{drop}	Per-keystroke drop probability	[0, 0.5]
p_{lag}	Per-frame lag probability	[0, 0.8]
l	Lag duration	[0, 300] ms

to add delay or to lose a packet. With probability p_{lag} we add a given amount of lag l for each frame transmitted from the server. At the client-side, at time t_0, if the received frame relates to a time instant $t > t_0$, then the frame is rendered, otherwise the received frame is discarded and the last rendered frame is re-rendered for that time instant, essentially simulating a game freezing behavior. Note that for $p_{lag} = 1$ every frame is delayed by l corresponding to the case of a constant lag scenario. However, emulating correlated delays and losses (as would happen in the case of transient bandwidth bottlenecks) can be easily realized by using simple Markovian models. *The delay in the reception of the new frames will alter the agent's ability to play, exactly as would happen to humans in the case of sluggish network conditions.*

One last point is worth elucidating: the agent sends back an action for every received frame in the form of a keystroke. For the case of $p_{drop} > 0$, whenever a keystroke is dropped, the game server interprets it as a *no action*. The environment is then advanced by one frame without executing any action.

5.3.2 Assessing score degradation

We emulate a game played over synthetic network conditions using the trained agents. As for traditional QoE assessment, our goal is to observe how our agent playing capabilities are affected by network conditions. As typically done with human players [85], we quantify the influence of the network on QoE by considering the average score that the agent is able to achieve in a perturbed scenario. To compare scores across games, we normalize the score over the average score achieved in a non-perturbed scenario: a normalized score ≈ 1 corresponds to no noticeable game impairment (i.e., highest MOS in an Absolute Category Rating scale), whereas a score ≈ 0 corresponds to the most severe degradation (i.e., lowest MOS). We perform a set of experiments with a non-ideal communication channel and report our findings in Figure 5.4. Furthermore, we report the 95% confidence interval of the obtained score to show the significance of the results.

Fixed latency

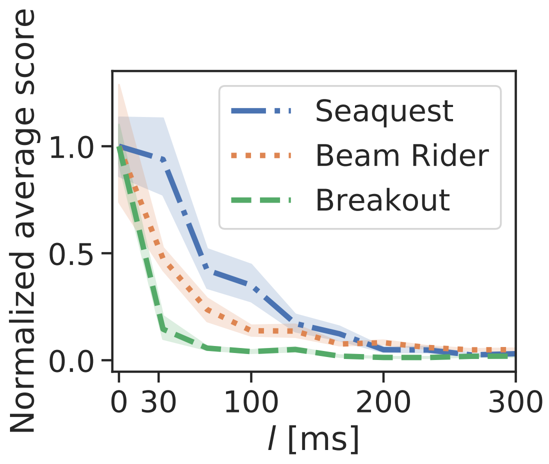
We consider the scenario of a communication channel with an added network latency. We vary this lag between 0 and 300 ms and observe the score achieved by the agent. Figure 5.4a shows that all three games have a sharp decrease in the score when additional latency is added. However, for low lags there is a large discrepancy in the score degradation across games: Breakout, Beamrider and Seaquest exhibit a significant, moderate and negligible score degradation, respectively.

This provides initial insights on the dynamics of three games. Breakout dynamics are very fast-paced, leaving small error margins. On the contrary, in Seaquest and Beamrider the game dynamics are more forgiving, with the agent being able to easily avoid obstacles and shoot at enemies even in the case of a delayed scenario. *This allows us to conclude that, in the low lag regime, not all games are equally penalized by the same network conditions.*

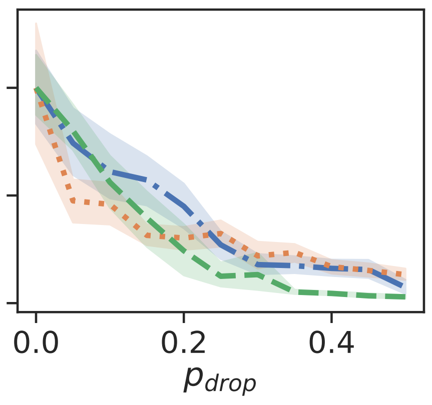
Second, it is also clear that after 100 ms of extra latency, the latter two games become unplayable. This is not different from the human perception timescale, where 100 ms is typically considered as a threshold that significantly affects the ability to retain control in interactive tasks or communication [84]. This is also found in cloud-based games, which strive to maintain latency at an even lower level [102]. *Games are therefore unplayable in these conditions and AI agents confirm this expectation.*

Random keystroke drop

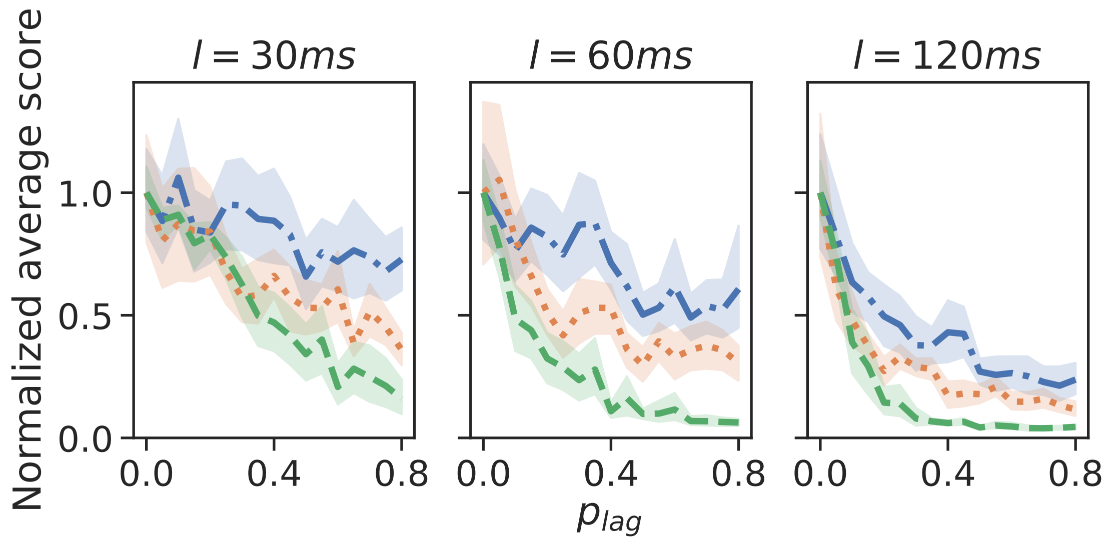
We repeated the experiment by this time-varying the keystroke drop probability p_{drop}. Figure 5.4b summarizes our findings: all three games have a similar score degradation as a function of p_{drop} with Seaquest being the most tolerant among the three. *This allows us to deduce that latency is more important than drops, as dropped actions are naturally repeated by players, realizing a sort of implicit Forward Error Correction (FEC) mechanism.*



(a) Deterministic frame lag.



(b) Keystroke loss probability.



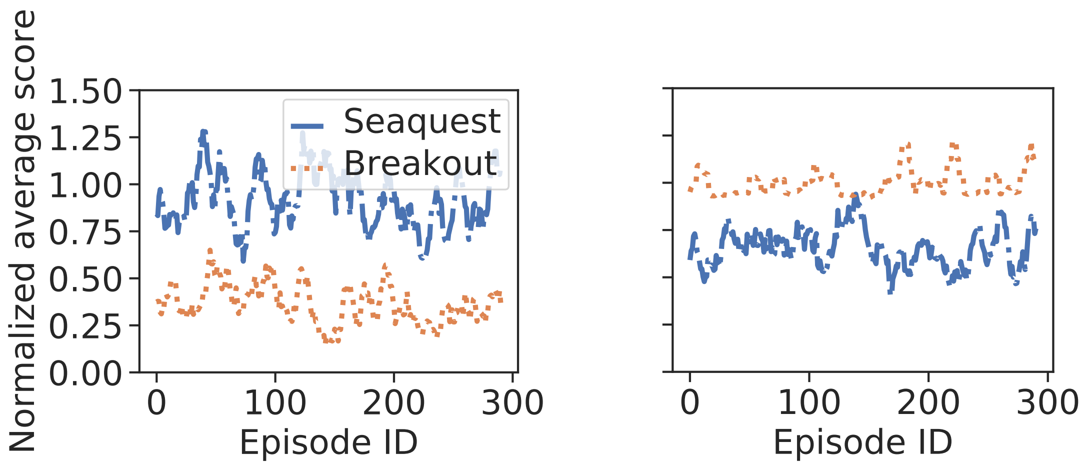
(c) Probabilistic frame lag.

Figure 5.4: Score degradation under perturbed network conditions

Random latency

We perform further evaluations by considering the case of probabilistic latency. Results depicted in Figure 5.4c show the score evolution as a function of both l and p_{lag}. These results closely mimic those obtained in Figure 5.4a with similar score degradation for all three games. As in the case of fixed latency, Seaquest is able to tolerate higher latency, while Breakout has rapid score degradation even for small l and p_{lag}.

The important takeaway from the figures is that our method is automated and is able



(a) Deterministic frame lag.

(b) Keystroke loss probability.

Figure 5.5: Normalized game scores achieved by AI agents under Blind (left) vs QoE-aware (right) schedulers

to gather very fine-grained score degradation maps for a combination of parameters. The bottleneck is mostly represented by agent training time (i.e., 22 hours per agent in our scenarios), whereas the experiments for Figure 5.4 took less than 30 minutes each.

5.3.3 Achieving per-game QoE fairness

Knowing a fine-grained QoE response to network conditions as provided by our methodology opens new opportunities for in-network QoE management such as resource placement, or QoE-aware packet scheduling.

Taking inspiration from our previous work in Chapter 3 and Chapter 4 and building upon the results of Figure 5.4a, we make a simple proof of concept, where we take scheduling decisions that are aware of the heterogeneous sensitivity of games to network impairment. ISPs and CG providers need to perform resource arbitration in the case of multiple concurrent players. The objective can be formulated loosely as enforcing inter-game fairness: in the case of two or more concurrent game sessions, a QoE-aware scheduler could prioritize the most latency-sensitive games, as the experience of players of the other games would be less affected by extra latency.

We thus perform an experiment in which we alter the scheduling policy for two concurrent sessions of the Breakout and Seaquest games. Figure 5.5 shows the time evolution of the score of both games. Both sessions share the same bottleneck link which adds a per frame delay of 30 ms with probability $p_{lag} = 0.5$, and both games are equally treated by a Round Robin (RR) scheduler. Figure 5.5a shows that although both game flows observe the same latency a big discrepancy is experienced in the obtained scores. Breakout is heavily affected by the network impairments, achieving an average normalized score of 0.35, whereas Seaquest is almost unperturbed, achieving a 0.96 average normalized score.

Using a game QoE-aware scheduler allows us to perform scheduling based on the expected impact on gaming performance.

As a proof of concept, we switch the scheduling policy to Strict Priority which has been previously discussed in Chapter 4 and depicted in Figure 4.2. Using SP scheduling we assign the highest priority to Breakout flows while leaving Seaquest flows in a lower priority. In this scenario, as reported in Figure 5.5b, the Breakout agent observes an ideal channel and is able to achieve a normalized score of 1 while Seaquest sees its normalized score decrease to 0.65. However, this priority scheduling increases the average game score which goes from 0.67 in the case of RR scheduling, to 0.82 in the latter case. The considered scenario is, of course, only considered as a proof of concept to show the feasibility of the proposed approach: in practice one would use advanced weighted scheduling mechanisms, with weights deduced from results in Figure 5.4. The evaluation of this mechanism is left for future work.

5.3.4 Beyond Atari

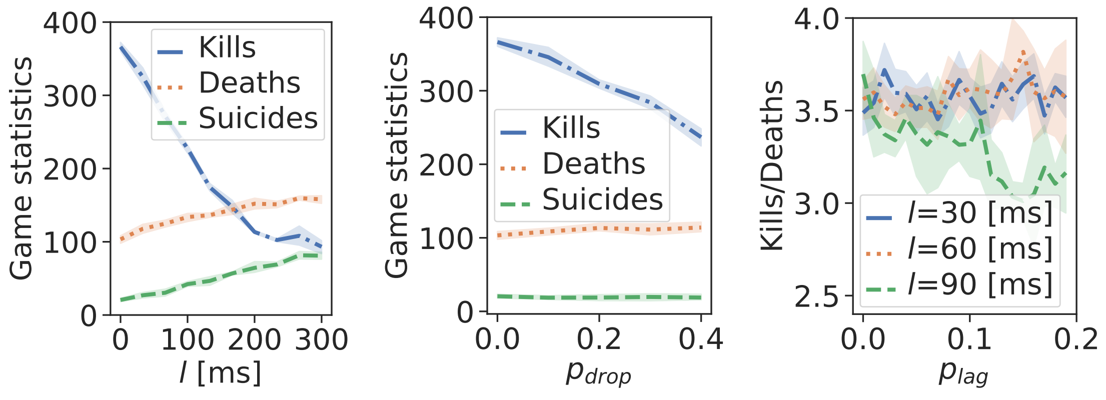
Although being quite simple, Atari represents a versatile testbed for the evaluation of the methodology as it includes a vast catalog of different games. Nevertheless, one may be persuaded that such a methodology cannot be applied to more realistic 3D games and is limited to Atari only. To highlight the applicability of the proposed methodology to more complex games we considered the scenario of cloud gaming with classic Doom instead of Atari.

Doom has rapidly become an important benchmark for the evaluation of DRL algorithms with numerous works focusing on the development of AI models aiming at reaching super-human performance in various game scenarios of the game [103] [104] [96] [105] [106]. Due to this a vast research environment has been created around the game which provides ease of integration of the game with available DRL frameworks.

We tested our methodology by employing an artificial agent trained to play Doom on the client-side. The agent is trained using the deep neural network architecture described in [96] which achieved first place in the yearly ViZDoom competition [97]. Identically to a human player, the agent receives as input raw pixels and based on it takes decisions in the form of keystrokes.

We consider the deathmatch game scenario with 8 in-game bots behaving according to a hardcoded game logic present in the original game and one AI agent. In such a scenario all players compete against each-other by trying to achieve the maximum amount of kills in a given game duration. Similarly to the Atari case, we employ the average in-game score as an indicator of the perceived QoE as previously done in [90]. We consider the kill over death ratio (K/D) as the primary QoE indicator since it captures various in-game statistics (including accuracy, avoidance, and reactiveness) and more generally reflects how easy it is to play the game in a given network configuration. Figure 5.6 depicts experimental results alongside with the 95 % confidence intervals after multiple deathmatches runs for a subset of selected network parameters.

The results provide insights similar to the atari case, highlighting the score degradation in function of the added latency. Noteworthy, Figure 5.6a shows an increasing rate in the agent suicide rate with the increase of latency. This metric can be used as a proxy for partially defining the agent's accuracy, as the employed game mode included weapons capable of self-harming the agent if hitting a wall or an enemy too close to the enemy. This is what has been empirically confirmed by observing the behavior of the agent in real-time; the added latency contributed to spatial disorientation of the agent, thus leading to the decrease of accuracy and an increase in the amount of times projectiles were hitting to a wall close to the agent instead of hitting the enemy. In the case of probabilistic keystroke drops, showed in Figure 5.4b, the suicide rate remained constant in function of the drop probability which only led to a decrease in the amount of kills performed by the agent, which can be attributed to the dropping of keystrokes related to the firing action. Finally probabilistic latency depicted in Figure 5.6c highlights how the agent is able to remain resilient to transient latencies whenever the amount of such latency is kept below $\approx$ 90 ms.



(a) Deterministic lag (b) Stochastic losses (c) Stochastic lag

Figure 5.6: Score degradation in perturbed channel scenario for the game of Doom

5.4 Discussion

Our proposal raises some interesting questions which still remain unanswered and which we now discuss briefly.

5.4.1 Game stages classification

During our analysis, we considered only games requiring the same playing style throughout the entire game. Modern games are typically composed of a series of different stages, e.g., action stages, exploration stages, dialogue stages, etc. Identifying different stages would lead to even finer-grained control over the resources to be allocated to single games. As previously shown, low lags added in a fast-paced action stage may lead to catastrophic performance degradation, whereas a 500 ms lag in a dialogue stage would be hardly noticed by the player. Automatic game stages detection and classification can be achieved using a methodology similar to the one we presented by observing how the agent reacts to channel perturbation throughout its play-through. *Including more diversity and newer games is part of our ongoing work.*

5.4.2 Delay-tolerant agents

Our results show that latency plays a fundamental role for the performance of the agent. These results are consistent with other studies performed with real human players [85],

[90]. On the other hand, studies such as [91] showed that, to a certain extent, experienced players are able to tolerate the effects of latency on their gaming performance, and incur smaller score penalties. In the field of DRL, there has been limited interest in building agents resilient to delayed environment response. Examples such as [107] or [108] perform future planning knowing the present observations, instead of using delayed observations to predict the present. Nevertheless, there exists a significant body of work in the field of classic RL that considers these issues [109] and provides strong theoretical results on model requirements [110]. *We are currently experimenting with delay-tolerant DRL agents to refine our framework.*

5.4.3 Agent calibration

Building artificial agents capable of adapting to scenarios with network impairments introduces another degree of freedom in the assessment of QoE by tuning the "experience level" [91] or the "play style" [111] of the artificial player. In [111] significant steps have been made towards creating agents with different play styles, that can make AI performance closer to human performance. *Validating the results that we gathered in this work with a study including real human subjects is a necessary step in our future work agenda.*

To summarize, in this Chapter we propose a novel methodological approach for efficient and reliable QoE assessment in Cloud Gaming scenarios. At its core, the proposed approach employs artificial players instead of real humans to assess game session QoE under different network impairments. We show performance results for three Atari games for different communication conditions that are in line with related literature employing human subjects.

We argue that this development yields a versatile and efficient tool for automating QoE assessment, by employing artificial players instead of real human subjects. We expect this to bring a substantial reduction in the cost and complexity of the entire process while introducing a rigorous, methodological and scalable strategy for the assessment of game QoE. Furthermore, in future work we will investigate whether such an approach can be applied for other interactive services, thus expanding the proposed methodology to a broader set of applications.

Chapter 6

Conclusion

In this Thesis a major focus has been devoted towards analyzing traffic flow performance in modern network infrastructures. Starting from how programmable data planes can improve the performance of flows in the wide-area networks, moving to flow scheduling in DC networks and finally descending into fine-grained optimization of traffic flow performance based on the user-perceived QoE.

In Chapter 2 we presented LOADER, a novel framework for developing network applications based on non-local states for programmable data planes. The main objective of LOADER is to overcome limitations with traditional network application embedding mechanisms. Those approaches are notorious for having poor scalability in the case of network applications operating on states with large network scopes, which may ultimately lead to poor traffic flow performance.

We designed LOADER to operate on replicated states instead of employing only local states which ultimately allowed us to obtain a scalable and versatile framework able to cope with the limitations of traditional approaches. Our contribution included an extensive analysis of the main practical design challenges of enabling switches to operate on replicated states. Furthermore, we provided a high-level programming abstraction for the development of distributed network applications based on replicated states. The developed abstraction model represents a step forward in making this kind of approach practical. This is achieved thanks to the fact that LOADER combines the expressiveness of a high-level programming model without ignoring the underlying hardware architecture of programmable switches. Thus, while making it accessible to the network programmers by its ease of developing network applications, our model can operate efficiently by providing a comprehensible abstraction for the compilation and the embedding of network applications.

We validated LOADER using both P4 and OPP stateful data planes under a realistic setting and with a realistic SDN controller. Our evaluation showed that distributed network applications enabled by LOADER can be beneficial for the network performance and can be efficiently implemented in high-performance programmable stateful switches.

With LOADER we showed how centralized network management approaches combined with network application offloading represent a big potential in improving the network performance while enabling novel network applications. Among the example applications enabled by LOADER we showed a DCN link-aware load balancer. Such application has been widely used to greatly benefit the overall traffic flow performance inside DCNs. Yet, as discussed in Chapter 3 and Chapter 4, traffic control and flow scheduling still play a major role in respect to load balancing when it comes to improving the traffic flow performance. Starting from this observation and taking inspiration from centralized management approaches, in Chapter 3 we investigated whether centralized schemes, used in SDN, can be applied to traffic control in DCNs to improve the overall network performance. We investigated the use of centralized asynchronous and asynchronous traffic control architectures. More in detail, we showed that, by relaxing the constraint on time synchronization of an ideal synchronous flow scheduler and by employing simple rate limiting at each server, it is possible to obtain performance comparable to the ideal system, both in terms of throughput, fairness and flow completion time. In our evaluation we proposed a simple asynchronous traffic control algorithm which allowed us to perform a detailed evaluation of queuing delays across all the components of the DC. Ultimately this allowed us to gather insights on possible culprits which contribute to the deterioration of flow-level performance. We showed that the asynchronous approach is capable of reducing the memory overhead of the transmission queues at the expense of increased network latency while keeping the complexity significantly smaller in respect to the synchronous system.

We showed that asynchronous architectures appear to be very promising for their trade-off between performance and complexity. Nevertheless, we concluded that, while in theory centralized approaches may be employed to implement fine-grained traffic control, they are highly complex to implement from the practical point of view and will unlikely find applicability in real Data Centers.

Following the outcome of Chapter 3 we dedicated our effort in analyzing practical flow scheduling mechanisms for DCNs that are able to achieve high performance while keeping the complexity as low as possible. In Chapter 4 we presented the result of our works, namely NOS2. NOS2 is designed as a flow scheduler aiming at achieving a good level of performance in terms of flow completion time while keeping the complexity as low as possible. To achieve this goal we exploited a fine-grained MLFQ flow scheduling at each host while adopting a simple strict priority scheduler at each switch with only 2 queues. NOS2 exploits centralized knowledge which, differently from the system presented in Chapter 3, does not impact in any way the performance of the overall system. Indeed, in NOS2 we exploit knowledge about the DC-wide estimation of the flow length distribution to configure scheduling policy at hosts and switches. Such operation can be done efficiently and rapidly, thus not introducing additional latency or overhead in the overall system.

Our simulation results showed that NOS2, exploiting only 2 priority queues in switches, is able to achieve a performance close to that of state-of-the-art flow schedulers which instead rely on 8 priority queues in switches. Thus, given the lower number of queues and the simpler and more accurate computation of the thresholds, NOS2 is shown to achieve the best trade-off between performance and complexity.

The process of optimizing both wide-area networks and DC networks traffic flow performance starts from the assumption of minimizing the average flow completion time as in Chapters 3-4 or the amount of congestion as assumed in Chapter 2. Such an assumption is almost never the case in realistic scenarios. What typically happens in reality is that different flows, even if they are similar among each other, may require different levels of performance in terms of throughput or latency. While fully understanding this kind of requirements may be a challenging task in Chapter 5 we try to do so for a small, yet important, portion of Internet services, namely for the case of Cloud

Gaming.

We propose a novel methodological approach for efficient and reliable QoE assessment in cloud gaming scenarios. At its core, the proposed approach employs artificial players instead of real humans to assess game session QoE under different network impairment scenarios. While employing simple Atari games for our extensive evaluation we prove the fact that it can be applied to more complex and realistic games such as Doom. As cloud gaming streams are typically treated equally in the network, thus observing similar network conditions, following the evaluation of the proposed QoE-assessment methodology we highlight how the gained knowledge can be exploited to provide more fine-grained QoS requirements for different games. We show that different games have a different response to latency and starting from this observation we propose an in-network strict priority scheduler. The proposed scheduler, which takes inspiration from Chapter 4, is able to maximize the average gaming performance among different flows by exploiting the latency response curves of individual games.

In conclusion in this Thesis we analyze the impact on the traffic flow performance of numerous elements of modern network infrastructure. We showed that programmable data planes are capable of improving traditional SDN performance, yet they require advanced state management techniques in case of complex network applications. We showed that the DC traffic flow performance can be improved easily and without introducing substantial complexity in the network. Finally, we showed that optimizing traffic flow performance without considering the underlying service requirements can be myopic for numerous real-time applications, thus requiring advanced analysis of the QoE of users for different services.

Appendix A

LOADER implementation of reference applications

A.1 DDoS detection with LOADER

The DDoS detection application operates on a series of network states related to the transit SYN packets on the edge routers of the network (line 20). To filter the corresponding ports and the type of packets the `scope` attribute is used during the definition of the state which is passed a helper function, namely `extPortFilter`. The reduction function is simply defined as the sum of the states (line 26) through a predefined primitive. The trigger function (line 32) simply compares the outcome of the reduction function against a predefined threshold `R` and invokes the activity function whenever the condition is satisfied. The inconsistency level is defined as time obsolescence with $\epsilon_t = 0.2\text{ms}$ (line 24) forcing state replication to occur every 0.2ms. The activity function targets all edge routers and acts by performing the notification of the controller about the presence of an attack.

A.2 Distributed rate-limiting with LOADER

The distributed rate-limiting application is a variation of the DDoS application. Notably, while the reduction function remains invariant to the DDoS case, the states are defined

```
1  from Controller import TopologyManager
2  from LOADER.PrimitiveActions import Drop, StateSum, Rate
3  from LOADER.Scope import Pkt
4
5  def extPortFilter(devices):
6      extPorts = []
7      for d in devices:
8          extPorts += [p for p in d.getPorts() if p.Type==EXTERNAL]
9      return (Pkt.ingressPort in extPorts) and (Pkt.TCP.Flag.SYN == 1)
10
11 R = 1000 # DDoS threshold in SYN pkts / s
12
13 # List of all edge routers
14 devices = TopologyManager.getEdgeRouters()
15 applicationStates = []
16
17 # Iterate over all edge routers
18 for i in range(devices):
19     # Create a state for each edge router
20     s = State(target=d,
21             scope=Rate(filter=Pkt(filter = extPortFilter([d]))))
22     applicationStates.append(s)
23
24 # Define the reduction function as the sum of application states
25 r = ReductionFunction(states=applicationStates,
26         operation=StateSum)
27
28 # Define the activity function to drop all incoming packets
29 a = ActivityFunction(target=devices, scope=Pkt(filter=extPortFilter(devices)), action
       =Controller.Notify(”DDoS detected”))
30
31 # Define trigger function to perform probabilistic dropping
32 tr = TriggerFunction(s0=r.Result(),
33             trigger=r.Result()>R,
34             inconsistencyLevel=TimeObsolescence(0.2, ”ms”)
35             activity = a)
```

Listing A.1: DDoS detection with LOADER

as the total amount of traffic going through the edge routers (line 20). To perform rate-limiting, the activity function (line 29) is defined to drop any incoming packet following the activation of the trigger function. The trigger function is randomly activated, with the probability of activating increasing whenever the total incoming traffic approaches the predefined target threshold R. As in the previous case the inconsistency level is defined as time obsolescence with $\epsilon_t = 0.2$ms.

A.3 Link-aware load balancing with LOADER

In link-aware load balancing for data center networks, the objective is to find the least congested path from a given source server to a destination one. The states are defined as the congestion level on all uplink paths (line 25) and on the corresponding downlink

```
from Controller import TopologyManager
from LOADER.PrimitiveActions import Drop, StateSum, Rate
from LOADER.Scope import Pkt

def extPortFilter(devices):
extPorts = []
for d in devices:
    extPorts += [p for p in d.getPorts() if p.Type==EXTERNAL]
return Pkt.ingressPort in extPorts

R = 100**6 # Desired rate in bps

# List of all edge routers
devices = TopologyManager.getEdgeRouters()
applicationStates = []

# Iterate over all edge routers
for d in devices:
# Create a state for each edge router
s = State(target=d,
        scope=Rate(filter=Pkt(filter=extPortFilter([d]))))
applicationStates.append(s)

# Define the reduction function as the sum of application states
r = ReductionFunction(states=applicationStates,
        operation=StateSum)

# Define the activity function to drop all incoming packets
a = ActivityFunction(target=devices,
        scope=Pkt(filter=extPortFilter(devices)),
        action=Drop)

# Define trigger function to perform probabilistic dropping
tr = TriggerFunction(s0=r.Result(),
        trigger=(rand()<(r.Result()-R)/r.Result()),
        inconsistencyLevel=UpdateError(10),
        activity = a)
```

Listing A.2: Distributed rate-limiting with LOADER

paths (line 31). The uplink paths are considered by taking into account the set of leaf switches (line 18), while the downlink paths are taken over the spine switches (line 19). We assume that the topology manager exposes the appropriate methods to access the set of those switches. The reduction function performs a minmax operation over all the possible paths, thus leading to a path that minimizes the maximum congestion on the uplink-downlink segment (lines 14-16). This kind of application presents a trigger function which always returns true (lines 46-49). Consequently, the activity function is always triggered. Since the scope of the activity function targets all SYN packets (line 40), the activity is executed whenever a new flow arrives. Whenever this condition occurs, the activity function sets a new flow entry by assigning the least congested path for the new flow (lines 41-44). Notably, in the scenario of data center networks the load dynamics may change rapidly due to the presence of a big amount of flows. For this reason the inconsistency level is specified in the form of update error with $\epsilon_r = 10$ writes (line 48), leading to a more updated information at the cost of potentially bigger synchronization traffic.

```
 1  from Controller import TopologyManager
 2  from LOADER.PrimitiveActions import SetEgress, Rate, min, max
 3  from LOADER.Scope import Pkt
 4
 5  # Filter for downlink ports (i.e. from spine to leaf)
 6  def dlPortFilter(device):
 7      return Pkt.getEgressPort() in [p for p in device.getPorts() if p.Type ==
        DOWNLINK]
 8
 9  # Filter for uplink ports (i.e. from leaf to spine)
10  def ulPortFilter(device):
11      return Pkt.getEgressPort() in [p for p in device.getPorts() if p.Type ==
        UPLINK]
12
13  # Reduction function for minimum path congestion
14  def minMaxCong(ulCong, dlCong):
15      dstLeaf = TopologyManager.getSpineID(Pkt.getDst())
16      return argmin([max(ulCong[i], dlCong[i][dstLeaf]) for i in range(len(
        TopologyManager.getSpines()))])
17
18  l = TopologyManager.getLeafSwitches()[0]
19  spines = TopologyManager.getSpineSwitches()
20
21  dlCong = []
22  ulCong = []
23
24  for p in l.getPorts(filter = ulPortFilter):
25      s = State(target=l, scope=Rate(filter = Port(p)))
26      ulCong.append(s)
27
28  for sp in spines:
29      spineLoad = []
30      for p in sp.getPorts(filter = dlPortFilter):
31          s = State(target=sp, scope=Rate(filter = p))
32          spineLoad.append(s)
33      dlCong.append(spineLoad)
34
35  r = ReductionFunction(states=[ulCong, dlCong],
36              operation=minMaxCong)
37
38  a = ActivityFunction(
39              target = l,
40              scope = Pkt(filter = (Pkt.TCP.Flag.SYN == 1)),
41              action = insertRule(
42              match = Pkt.getTuple(),
43              action = SetEgress,
44              args = r.Result()))
45
46  tr = TriggerFunction(
47              s0=r.Result(),
48              inconsistencyLevel=UpdateError(10),
49              activity = a)
```

Listing A.3: Link-aware load balancing with LOADER

Bibliography

[1] M. Trevisan, D. Giordano, I. Drago, M. M. Munafò, and M. Mellia, "Five years at the edge: Watching internet from the isp network," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 561–574, 2020.

[2] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.

[3] P. Bosshart and al., "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *ACM SIGCOMM CCR*, 2013.

[4] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and latency of virtual switching with open vswitch: A quantitative analysis," *Journal of Network and Systems Management*, vol. 26, no. 2, pp. 314–338, 2018.

[5] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch," *ACM SIGCOMM CCR*, Apr. 2014, ISSN: 0146-4833.

[6] 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.

[7] G. Bianchi, M. Bonola, S. Pontarelli, D. Sanvito, A. Capone, and C. Cascone, "Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing," *arXiv:1605.01977*, 2016. [Online]. Available: https://arxiv.org/abs/1605.01977.

[8] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, and F. Huici, "FlowBlaze: Stateful Packet Processing in Hardware," in *USENIX NSDI 19*, 2019, pp. 531–548.

[9] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful Network-Wide Abstractions for Packet Processing," in *ACM SIGCOMM*, 2016, ISBN: 978-1-4503-4193-6.

[10] G. Sviridov, M. Bonola, A. Tulumello, P. Giaccone, A. Bianco, and G. Bianchi, "LOcAl DEcisions on Replicated States (LOADER) in programmable data planes: programming abstraction and experimental evaluation," *arXiv preprint arXiv:2001.07670*, 2020.

[11] ——, "LODGE: LOcal Decisions on Global statEs in programmable data planes," in *IEEE NetSoft*, 2018, pp. 257–261.

[12] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, *et al.*, "ONOS: towards an open, distributed SDN OS," in *ACM SIGCOMM HotNets*, ACM, 2014, pp. 1–6.

[13] 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, 2014, pp. 503–514.

[14] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren, "Cloud control with distributed rate limiting," in *ACM SIGCOMM Computer Communication Review*, vol. 37, 2007, pp. 337–348.

[15] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches," in *Proceedings of the Symposium on SDN Research*, 2018.

[16] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," in *USENIX NSDI 15*, 2015, pp. 59–72.

[17] Y. Yuan, R. Alur, and B. T. Loo, "NetEgg: Programming network policies by examples," in *ACM SIGCOMM HotNets*, 2014, p. 20.

[18] R. Beckett, M. Greenberg, and D. Walker, "Temporal netkat," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 386–401, 2016.

[19] S. H. Yeganeh and Y. Ganjali, "Beehive: Simple distributed programming in software-defined networks," in *Proceedings of the Symposium on SDN Research*, ACM, 2016.

[20] J. McClurg, H. Hojjat, N. Foster, and P. Černỳ, "Event-driven network programming," in *ACM SIGPLAN Notices*, vol. 51, 2016, pp. 369–385.

[21] A. S. Muqaddas, G. Sviridov, P. Giaccone, and A. Bianco, "Optimal state replication in stateful data planes," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 7, 2020.

[22] M. T. Özsu and P. Valduriez, *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[23] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan, "Measuring control plane latency in SDN-enabled switches," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015, pp. 1–6.

[24] R. Lee and B. Jeng, "Load-balancing tactics in cloud," in *2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, IEEE, 2011.

[25] D. G. Luenberger and Y. Ye, *Linear and Nonlinear Programming*. Springer Publishing Company, Incorporated, 2015.

[26] N. Megiddo, "Linear Programming in Linear Time When the Dimension Is Fixed," *Journal of ACM*, vol. 31, no. 1, pp. 114–127, Jan. 1984, ISSN: 0004-5411.

[27] S. E. Schaeffer, "Survey: Graph Clustering," *Comput. Sci. Rev.*, Aug. 2007, ISSN: 1574-0137.

[28] K. Ruddel and A. Raith, "Graph partitioning for network problems," in *Joint NZSA ORSNZ Conference*, 2013.

[29] *CPLEX Optimizer*. [Online]. Available: https://www.ibm.com/analytics/cplex-optimizer.

[30] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *nature*, vol. 393, no. 6684, p. 440, 1998.

[31] E. Brewer, "CAP twelve years later: How the "rules" have changed," *Computer*, Feb. 2012, ISSN: 0018-9162.

[32] H. Howard and R. Mortier, *A Generalised Solution to Distributed Consensus*, 2019. arXiv: `1902.06776 [cs.DC]`. [Online]. Available: `https://arxiv.org/abs/1902.06776`.

[33] U. Brandes, "On variants of shortest-path betweenness centrality and their generic computation," *Social Networks*, vol. 30, no. 2, 2008, ISSN: 0378-8733. DOI: `http://dx.doi.org/10.1016/j.socnet.2007.11.001`.

[34] *P4 language repository*. [Online]. Available: `https://github.com/p4lang`.

[35] P. L. Consortium *et al.*, "P4 Runtime," *Website, https://github. com/p4lang/PI*, 2017.

[36] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[37] *LOADER repository*. [Online]. Available: `https://github.com/german-sv/loader`.

[38] *Open Packet Processor repository*. [Online]. Available: `https://github.com/netprog-uniroma2/OPP`.

[39] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks," *IEEE communications surveys & tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013.

[40] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *ACM SIGCOMM HotNets*, 2010, p. 19.

[41] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *USENIX NSDI*, 2017.

[42] 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," in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 45, 2015, pp. 183–197.

[43] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, “Fastpass: A centralized zero-queue datacenter network,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 307–318, 2015.

[44] G. Sviridov, A. Bianco, and P. Giaccone, “To Sync or Not to Sync: Why Asynchronous Traffic Control Is Good Enough for Your Data Center,” in *2018 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2018, pp. 1–6.

[45] J. Perry, H. Balakrishnan, and D. Shah, “Flowtune: Flowlet Control for Datacenter Networks.,” in *NSDI*, 2017, pp. 421–435.

[46] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti, “Numfabric: Fast and flexible bandwidth allocation in datacenters,” in *SIGCOMM*, ACM, 2016, pp. 188–201.

[47] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, “Hedera: Dynamic flow scheduling for data center networks..”

[48] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon, “Globally synchronized time via datacenter networks,” in *SIGCOMM*, ACM, 2016, pp. 454–467.

[49] J. Y. Hui, *Switching and traffic theory for integrated broadband networks.* Springer Science & Business Media, 2012.

[50] R. Sinkhorn, “A relationship between arbitrary positive matrices and doubly stochastic matrices,” *The annals of mathematical statistics*, vol. 35, no. 2, pp. 876–879, 1964.

[51] A. Varga and R. Hornig, “An overview of the OMNeT++ simulation environment,” in *Simutools*, ICST, 2008.

[52] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the social network’s (datacenter) network,” in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 45, 2015, pp. 123–137.

[53] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *ACM SIGCOM*, 2010.

[54] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, “pFabric: Minimal near-optimal datacenter transport,” in *ACM SIGCOMM CCR*, 2013.

[55] C.-Y. Hong, M. Caesar, and P. B. Godfrey, “Finishing flows quickly with preemptive scheduling,” *ACM SIGCOMM CCR*, 2012.

[56] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, “Friends, not foes: synthesizing existing transport strategies for data center networks,” in *ACM SIGCOMM*, 2014.

[57] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, “Information-agnostic flow scheduling for commodity data centers,” in *NSDI*, 2015.

[58] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, “Homa: A receiver-driven low-latency transport protocol using network priorities,” in *ACM SIGCOMM*, 2018.

[59] Y. Lu, G. Chen, L. Luo, K. Tan, Y. Xiong, X. Wang, and E. Chen, “One more queue is enough: Minimizing flow completion time with explicit priority notification,” in *IEEE INFOCOM*, 2017.

[60] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, “RDMA over commodity Ethernet at scale,” in *ACM SIGCOMM*, 2016.

[61] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center TCP (DCTCP),” *ACM SIGCOMM CCR*, 2011.

[62] G. Sviridov, P. Giaccone, and A. Bianco, “Low-Complexity Flow Scheduling for Commodity Switches in Data Center Networks,” in *IEEE GLOBECOM*, 2019.

[63] I. A. Rai, E. W. Biersack, and G. Urvoy-Keller, “Size-based scheduling to improve the performance of short TCP flows,” *IEEE Network*, 2005.

[64] L. Chen, J. Lingys, K. Chen, and F. Liu, “Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization,” in *ACM SIGCOMM*, 2018.

[65] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan, “Minimizing flow completion times in data centers,” in *IEEE INFOCOM*, 2013.

[66] 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 CCR*, 2009.

[67] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of scheduling*. Courier Corporation, 2003.

[68] L. Schrage, “A proof of the optimality of the shortest remaining processing time discipline,” *Operations Research*, 1968.

[69] V. Dukic, S. A. Jyothi, B. Karlas, M. Owaida, C. Zhang, and A. Singla, “Is advance knowledge of flow sizes a plausible assumption?” In *NSDI*, 2019.

[70] M. Nuyens and A. Wierman, “The foreground–background queue: a survey,” *Elsevier Performance evaluation*, 2008.

[71] S. Aalto and U. Ayesta, “Recent sojourn time results for multilevel processor-sharing scheduling disciplines,” *Statistica Neerlandica*, 2008.

[72] “IEEE 802.1p standard,” [Online]. Available: https://www.ieee802.org/.

[73] H.-W. Jiao and S.-Y. Liu, “A practicable branch and bound algorithm for sum of linear ratios problem,” *European Journal of Operational Research*, 2015.

[74] B. Stephens, A. Akella, and M. Swift, “Loom: Flexible and Efficient NIC Packet Scheduling,” in *NSDI*, 2019.

[75] K. Avrachenkov, P. Brown, and N. Osipova, “Optimal choice of threshold in two level processor sharing,” *Annals of Operations Research*, 2009.

[76] D. J. Wales and J. P. Doye, “Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms,” *The Journal of Physical Chemistry*, 1997.

[77] G. F. Riley and T. R. Henderson, “The NS-3 network simulator,” in *Modeling and tools for network simulation*, Springer, 2010.

[78] https://github.com/HKUST-SING/PIAS-Software.

[79] https://www.xfinity.com/esports.

[80] https://www.ciena.com/.

[81] https://canopusnet.com.

[82] https://stadia.google.com.

[83] https://www.forbes.com/sites/paultassi/2019/11/18/google-stadia-launch-review-a-technical-conceptual-disaster/.

[84] M. Claypool and K. Claypool, “Latency can kill: precision and deadline in online games,” in *ACM Multimedia Systems Conference*, 2010.

[85] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool, “The effects of loss and latency on user performance in unreal tournament 2003®,” in *ACM SIGCOMM workshop on Network and system support for games*, 2004.

[86] https://www.cnbc.com/2019/07/29/fortnite-world-cup-us-teen-wins-3-million-at-video-game-tournament.html.

[87] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, 2015.

[88] G. Sviridov, C. Beliard, A. Bianco, P. Giaccone, and D. Rossi, “Removing human players from the loop: AI-assisted assessment of Gaming QoE,” in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2020.

[89] G. Sviridov, C. Beliard, G. Simon, A. Bianco, P. Giaccone, and D. Rossi, “Demo abstract: Leveraging AI players for QoE estimation in cloud gaming,” in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2020.

[90] R. Amin, F. Jackson, J. E. Gilbert, J. Martin, and T. Shaw, “Assessing the Impact of Latency and Jitter on the Perceived Quality of Call of Duty Modern Warfare 2,” in *Int. Conf. on Human-Comp. Inter. (HCI)*, 2013.

[91] L. Pantel and L. C. Wolf, “On the impact of delay on real-time multiplayer games,” in *International workshop on Network and operating systems support for digital audio and video*, ACM, 2002.

[92] K.-T. Chen, P. Huang, and C.-L. Lei, “How sensitive are online gamers to network quality?” *Communications of the ACM*, 2006.

[93] S. Zander, I. Leeder, and G. Armitage, “Achieving fairness in multiplayer network games through automated latency balancing,” in *International Conference on Advances in computer entertainment technology*, ACM, 2005.

[94] N. Justesen, P. Bontrager, J. Togelius, and S. Risi, “Deep learning for video game playing,” *IEEE Transactions on Games*, 2019.

[95] M. Hausknecht and P. Stone, “Deep recurrent q-learning for partially observable mdps,” in *AAAI Fall Symposium Series*, 2015.

[96] G. Lample and D. S. Chaplot, “Playing FPS Games with Deep Reinforcement Learning,” in *Conf. on Artificial Intelligence (AAAI)*, 2017.

[97] M. Wydmuch, M. Kempka, and W. Jaśkowski, “ViZDoom competitions: playing doom from pixels,” *IEEE Transactions on Games*, 2018.

[98] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, *et al.*, “Relational deep reinforcement learning,” *arXiv:1806.01830*, 2018.

[99] W. Cai, R. Shea, C.-Y. Huang, K.-T. Chen, J. Liu, V. C. Leung, and C.-H. Hsu, “A Survey on Cloud Gaming: Future of Computer Games,” *IEEE Access*, 2016.

[100] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “OpenAI Gym,” *arXiv:1606.01540*, 2016.

[101] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The Arcade Learning Environment: An Evaluation Platform for General Agents,” *Journal of Artificial Intelligence Research*, 2013.

[102] M. Claypool and D. Finkel, “The Effects of Latency on Player Performance in Cloud-based Games,” in *IEEE/ACM Workshop on Network and System Support for Games (NetGames)*, 2014.

[103] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski, “Vizdoom: A doom-based ai research platform for visual reinforcement learning,” in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2016, pp. 1–8.

[104] Y. Wu and Y. Tian, “Training agent for first-person shooter game with actor-critic curriculum learning,” *URL: https://openreview. net/pdf*, 2016.

[105] A. Dosovitskiy and V. Koltun, “Learning to act by predicting the future,” *arXiv preprint arXiv:1611.01779*, 2016.

[106] S. Bhatti, A. Desmaison, O. Miksik, N. Nardelli, N. Siddharth, and P. H. Torr, "Playing doom with slam-augmented deep reinforcement learning," *arXiv preprint arXiv:1612.00380*, 2016.

[107] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson, "Learning latent dynamics for planning from pixels," *arXiv:1811.04551*, 2018.

[108] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, *et al.*, "Model-based reinforcement learning for Atari," *arXiv:1903.00374*, 2019.

[109] T. J. Walsh, A. Nouri, L. Li, and M. L. Littman, "Learning and planning in environments with delayed feedback," *Autonomous Agents and Multi-Agent Systems*, 2009.

[110] K. V. Katsikopoulos and S. E. Engelbrecht, "Markov decision processes with delays and asynchronous cost collection," *IEEE Transactions on Automatic Control*, 2003.

[111] M. Jaderberg, W. M. Czarnecki, I. Dunning, L. Marris, G. Lever, A. G. Castaneda, C. Beattie, N. C. Rabinowitz, A. S. Morcos, A. Ruderman, *et al.*, "Human-level performance in first-person multiplayer games with population-based deep reinforcement learning," *arXiv:1807.01281*, 2018.