Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 5507845, 16 pages
https://doi.org/10.1155/2022/5507845

Research Article

In the Direction of Service Guarantees for Virtualized Network Functions

Muhammad Siraj Rathore,[1] Naveed Ahmad,[2] Rashi Kohli,[3] Jawaid Iqbal,[1] Roobaea Alroobaea,[4] Saddam Hussain,[5] Syed Sajid Ullah,[6] and Fazlullah Umar[7]

[1]*Department of Computer Science Capital University of Science and Technology, Islamabad, Pakistan*
[2]*School of Computing, Faculty of Engineering, Universiti Teknologi Malaysia, 81310 Skudai, Johor, Malaysia*
[3]*Senior Member IEEE, USA*
[4]*Department of Computer Science, College of Computers and Information Technology, Taif University, P. O. Box 11099, Taif 21944, Saudi Arabia*
[5]*Department of Information Technology, Hazara University, Mansehra, 21120 KP, Pakistan*
[6]*Department of Information and Communication Technology, University of Agder (UiA), N-4898 Grimstad, Norway*
[7]*Department of Information Technology, Khana-e-Noor University, Pol-e-Mahmood Khan, Shashdarak, 1001 Kabul, Afghanistan*

Correspondence should be addressed to Fazlullah Umar; fazlullahumar@gmail.com

Received 24 February 2022; Accepted 13 April 2022; Published 2 May 2022

Academic Editor: Kuruva Lakshmanna

The trend of consolidating network functions from specialized hardware to software running on virtualization servers brings significant advantages for reducing costs and simplifying service deployment. However, virtualization techniques have significant limitations when it comes to networking as there is no support for guaranteeing that network functions meet their service requirements. In this paper, we present a design for providing service guarantees to virtualized network functions based on rate control. The design is a combination of rate regulation through token bucket filters and the regular scheduling mechanisms in operating systems. It has the attractive property that traffic profiles are maintained throughout a series of network functions, which makes it well suited for service function chaining. We discuss implementation alternatives for the design and demonstrate how it can be implemented on two virtualization platforms: LXC containers and the KVM hypervisor. To evaluate the design, we conduct experiments where we measure throughput and latency using IP forwarders (routers) as examples of virtual network functions. Two significant factors for performance are investigated: the design of token buckets and the packet clustering effect that comes from scheduling. Finally, we demonstrate how performance guarantees are achieved for rate-controlled virtual routers under different scenarios.

1. Introduction

Network function virtualization (NFV) has attained much interest recently for its potential to reduce costs for deployment and operations in telecommunication networks and to facilitate network service deployment [1, 2]. Based on virtualization and commodity hardware, NFV shifts packet processing from specialized hardware devices to software running on regular computer systems. It enables the execution of multiple network functions (such as IP forwarders, firewalls, load balancers, and WAN accelerators) on virtualized computer systems, and such functional blocks can be combined to create advanced network services. For instance, a network operator could create multiple virtual machines on a computer system and let each virtual machine perform a specific packet processing task for a certain customer.

NFV has emerged as a flexible, programmable, and cost-efficient alternative to special-purpose networking hardware. However, there are new challenges in terms of performance and service guarantees that come with the transition of network functions from hardware to software, where network functions are consolidated onto general-purpose servers.

Although general-purpose servers are less costly and more programmable as compared to hardware counterparts, however, on the other hand, it is challenging to achieve high performance and service guarantees for competing network functions on a shared server. In this work, we investigate how service guarantees can be ensured in such a scenario.

In this paper, we consider the scenario where network functions are implemented as virtual machines on general-purpose servers, for instance, as containers or as guest machines on a hypervisor host. This means that each server simultaneously can host many virtualized network functions that belong to different service instances and potentially even to different customers. Therefore, the server needs to ensure that each virtual network function is provided with the resources it needs in order to meet its service requirements.

Network service guarantees are typically provided through packet scheduling, where packets are dispatched according to a certain queuing discipline, such as weighted fair queuing, round robin, and FIFO. However, regular computer systems do not provide support for controlling the scheduling of packets onto virtual machines in such a way. Instead, packets are dispatched to virtual machines in the order they arrive, without any service differentiation. This lack of resource control mechanisms may lead to resource contention where an overloaded network function in one virtual machine starves network functions running in other virtual machines.

The purpose of this work is to investigate how performance guarantees in terms of throughput, and bounded delay can be provided for parallel running network functions (implemented as virtual machines). We put forward a design for service guarantees for packet processing in virtual machines that we argue can be implemented efficiently in current computer systems, and we demonstrate and investigate two different implementations of the design in Linux.

Our design for performance guarantees has its foundation in *rate-controlled service disciplines* [3], where traffic is controlled through two mechanisms, *rate control* and *scheduling*. Rate control ensures that a traffic flow conforms to an agreed-upon *traffic profile*. In this context, a traffic flow means that traffic can be distinguished from other traffic based on certain packet header values. For instance, source and destination IP address/port number might be used to distinguish a traffic flow from another flow. For a flow, there is an associated *traffic profile*, which is used to enforce specific traffic characteristics for a flow. For instance, delay and bandwidth bounds can be configured as an associated traffic profile. In addition to rate control, the other component of this design is scheduling, which is responsible for scheduling traffic flows onto links, i.e., to access the channel in order to reach the desired destination. An attractive property of this design is that the traffic profile is invariant: provided that the input traffic conforms to the traffic profile, the output traffic will also conform. This is advantageous for *service function chaining*, a concept closely related to NFV, where services are composed of a series of network functions. As each network function in a chain preserves the traffic profile, it will be maintained also by the whole chain.

We show how the design can be implemented in both a container-based (LXC) [4] and a hypervisor-based (KVM) [5] Linux virtualized system. The traffic profile is expressed as a *token bucket*, with a long-term rate (average throughput) and a burst size (for bounded packet delay), and we use token bucket filters for rate control. We study how the existing token bucket implementation in Linux traffic control can be used for the traffic control function. Furthermore, we argue that due to inherent interleaving characteristics of executing network functions in software on servers in a time-sharing manner, virtual network functions will process packets in *clusters*. The clustering effect turns out to be an important factor that influences performance and constrains the traffic profiles that can be supported.

Finally, we measure the overhead of adding a token bucket filter in the packet processing path. In addition, we investigate how the two implementations (LXC and KVM) can guarantee throughput and latency in overload situations, and when multiple network functions are running in parallel on the same computer system. Our results indicate that a rate-controlled service design ensures average throughput and bounded delay in overload situations, which is not achievable otherwise (default settings without rate control).

The rest of this paper is organized as follows: Section 2 gives motivation and a problem statement for our work. Section 3 provides a background on classic rate-controlled service discipline, and we analyze how this design can be adapted for a computer system. In Section 4, we discuss the requirements and implications of realizing rate-controlled service disciplines for virtualized network functions. Our implementations of the rate-controlled service discipline in Linux (both for LXC and KVM virtual environments) are detailed in Section 5, and in Section 6, we experimentally evaluate the implementations and discuss the results. Section 7 covers related work. Finally, Section 8 concludes the paper.

2. Motivation and Problem Statement

In a virtualized computer system, different virtual machines share the system resources such as CPU, memory, and I/O devices. However, it is well-known that the resource contention between virtual machines may lead to unpredictable performance penalties [6–8]. For instance, an overloaded virtual machine may consume a resource and starve other virtual machines. In such a situation, it becomes difficult to provide service guarantees to different virtual machines. We demonstrated in our earlier studies how an overloaded virtual router (a virtual machine acting as a packet forwarder) can affect the performance of another virtual router running in parallel [9, 10].

Consider a scenario in a multitenant cloud environment where two virtual routers are administered by two different customers running on a shared platform. The service-level agreement for each virtual router is 300-kilo packets per second (kpps). The desired performance is achieved by both virtual routers when a load of 300 kpps is offered in parallel on each virtual router (Table 1). However, when we overload virtual router 1 (VR1) by increasing the load from 300 kpps

Table 1: Problem of performance isolation.

Offered load (kpps)		Throughput (kpps)	
VR1	VR2	VR1	VR2
300	300	300	300
1000	300	480	55

to 1000 kpps, the performance of VR2 decreases from 300 kpps to only 55 kpps [9]. This is due to the fact that heave packet drop occurs for VR2 due to lack of service which leads to poor performance.

We believe that rate limiting the offered load on a system might be useful to avoid an overloaded system, for instance, rate limiting both VR1 and VR2 to 300 kpps in the above scenario. We hypothesize that if we can avoid an overloaded system, we can achieve performance objectives for multiple virtual routers running in parallel. The idea leads us to the study of classic rate-controlled service discipline where the offered load on a system is rate limited (through token bucket filters) to achieve certain performance guarantees.

Our contributions to this work are as follows:

(1) The adaptation of the theoretical rate-controlled service design for virtualized computer systems

(2) Implementation of rate-controlled service design in Linux for two different virtual environments (i.e., KVM and LXC). We demonstrate that rate-controlled service design guarantees average throughput and bounded delay even in overload situations

(3) We measure the performance overhead of adding rate controllers along the packet processing path

(4) We investigate the phenomenon of packet clustering on a virtualized computer system and how it can affect performance guarantees in certain cases

The topic of NFV performance optimization has been extensively studied in the literature [11–15]. There are studies where mechanisms are suggested to avoid resource contention to significantly improve the performance of virtual machines. For instance, dCat [6] suggests a dynamic CPU cache management scheme that improves performance by avoiding CPU cache contention between multiple virtual machines. Similarly, VM scheduler optimization is proposed by tableau [7] to avoid CPU contention and thus improve performance. However, the focus of our work is a bit different than the performance optimization where we investigate how to provide service guarantees to network functions in overload conditions.

To the best of our knowledge, few studies address the topic of performance guarantees for network functions running as virtual machines. One of the few examples is XTC (Xen Throughput Control), which uses Xen resource control to avoid CPU contention among virtual routers [16]. XTC assigns CPU quotas depending on the target maximum throughput for a virtual router. Our approach advances beyond previous work, as we aim to control latency in addition to throughput. ResQ [8] is another example where a resource manager aims to provide efficient scheduling of network functions while providing service guarantees. The resource manager uses system profiling and collects resource usage information and re-adjusts resource allocation (CPU cache, memory) to achieve certain throughput and latency targets. Both XTC and ResQ aim to provide service guarantees on packets by avoiding system-level resource contention. However, there is no direct control over packet scheduling which makes it difficult to provide end-to-end service guarantees for a chain of network functions in presence of link disturbances. We believe our work is a step forward in that direction since we directly control packets by enabling rate-controlled service discipline and preserving traffic profiles across chains of virtual network functions.

3. Related Work

Significant efforts have been made to improve the performance of virtualized network functions executing on commodity computer hardware. Anderson et al. [11] argue that virtualization with Docker containers [17] is well-suited for NFV, thanks to the lightweight nature of containers, upon which Docker is based. The work presented in [18] introduces a new socket type (AF_GRAFT, i.e., socket grafting) which enables bypassing the regular networking stack for containers and hence improves network performance. ClickOS [12] is an example of a Xen-based [19] virtualized platform that has been highly optimized to improve networking performance. NetVM [13] is a virtualization architecture, which uses the Intel data plane development kit (DPDK [20]) together with KVM virtualization to make network functions. Moreover, hardware-assisted virtualization on network interfaces through the use of SR-IOV has been demonstrated to improve the performance of virtual machines in a server setting [21, 22]. XDP [23] is yet another example of high-performance networking where authors investigate how packet forwarding performance can be improved while using the host kernel, contrary to the I/O bypass technologies (such as DPDK and SR-IOV) where the host kernel is bypassed to improve performance. In recent years, software-based packet processing solutions are proposed using Intel DPDK while aiming for a line rate of 40 and even 100 Gb/s [24, 25]. Indeed, packet rates reported in these studies are much higher than the rate reported in our study. It is because the focus of our work is different since we aim to achieve performance guarantees in overloaded conditions whereas the aforementioned studies target high-speed networks. We believe that our proposed idea applies to high-speed networks as well since the principles of mitigating overloaded conditions (i.e., rate-controlled design) remain the same for the any-speed network. We plan to evaluate our proposed solution for high-speed networks in future work.

Some studies explore how delay guarantees can be accomplished for a chain of network functions using network calculus [26] and the mixed-integer linear programming model [27]. Our work is different from these studies since we are considering both throughput and delay as a

performance metric and also the expected effects of clustering on the performance of network functions.

Some other studies explore how the performance of VNFs can be predicted in advance on given network infrastructure. Profiling-based solutions collect the data (system resources, workload, etc.) and use learning algorithms to make predictions about the performance of VNFs. The work presented in [28] suggests a profiling framework to accurately model the performance of VNF. Similarly, the work presented in [29] proposes a profiling-based solution that can model the performance of a network functions chain (multiple VNFS connected in a series). However, in profiling-based solutions, there are chances of inaccurate predictions. There is a continuous learning mechanism to improve the accuracy of the performance model. In contrast, we follow a different approach (rate-controlled service design) where the given performance parameters are guaranteed through token buckets which provide direct control over packets.

There are several studies of architectures aimed to achieve networking performance guarantees for virtual machines in data center environments. Gatekeeper [30] is an example of a rate-limit approach, where Linux hierarchical token buckets are used to enforce rate limits. SENIC [31] uses hardware-assisted rate limits at traffic sources to improve performance of latency-sensitive applications. Along the same lines, Silo [32] dynamically enforces rate limits on virtual machines in order to achieve performance guarantees. The work presented in [6] argues that CPU cache sharing among multiple workloads in a multitenant cloud environment may lead to poor performance isolation. The authors propose a dynamic cache management technique (dCat) that aims to improve performance isolation by avoiding CPU cache contention. Similarly, Tableau [7] proposes a VM scheduling algorithm in order to avoid CPU contention among multiple virtual machines. However, in the aforementioned studies, the virtual machines are acting as traffic sources, which is different from the problem scenario we are addressing, namely, how to provide performance guarantees for virtual network functions.

Admission control and resource allocation in NFV is an active research area [33]. An important issue here is the mapping of virtualized network function onto a physical network of servers. There are several studies in that direction [34–40]. For instance, Cohen et al. [34] argue that NFV mapping introduces a new type of optimization problem. An algorithm is proposed to map virtualized network functions onto a physical platform. Similarly, Mijumbi et al. [35] formulate the problem of virtual network function mapping and compare the performance of greedy and tabu search-based algorithms. Moens and De Turck put forward a formal model for Virtual Network Function Placement (VNF-P) [36], which is evaluated for a small service provider scenario. The work presented in [37] presents an online preemptive algorithm to map a chain of network functions on an NFV platform. Similarly, Auto-3P [38] is yet another solution based on machine learning models that suggests the best placement for VNF. We consider these studies to be complementary to our work, where we assume that there exists a mechanism for mapping network functions onto virtual machines.

4. Rate-Controlled Service Disciplines

In this section, we give a brief background to classic rate-controlled service disciplines [3] and discuss how this theoretical model can be applied in the context of computer communication.

4.1. Rate-Controlled Service Disciplines. Rate-controlled service disciplines use rate control to provide guarantees on delay and bandwidth to *connections*. A connection in this context is a stream of traffic with an associated *traffic profile* (for instance, delay and bandwidth bounds). For instance, a communication session (e.g., video call) between two users A and B over the network can be considered a connection. Traffic control in rate-controlled service disciplines has two parts, a *rate controller* and a *scheduler*, as shown in Figure 1. The rate controller has several *regulators*, one for each connection.

When an incoming packet arrives, the corresponding connection is identified, and the packet is passed to the regulator for that connection. The regulator ensures that the packet conforms to the connection's traffic profile and places the packet on an outbound queue. It is then the job of the scheduler to decide the order of packet transmissions on the egress link.

Rate-controlled service disciplines are a framework that allows for a wide range of regulators and schedulers, and the exact service properties depend on the choice of regulator and scheduler. For our purposes, where we want to strike a balance between implementation efficiency and service guarantees, a *token bucket* regulator combined with an ordinary *first come, first serve* (FCFS) scheduler turns out to be (close to) a perfect match. Both components have implementation counterparts in Linux, and the resulting rate controller and the scheduler can guarantee per-connection bandwidth and delay bounds.

4.2. Token Buckets. A token bucket [41] controls a traffic stream according to a traffic profile (r, b), where r represents long-time bit rate and b is bucket size (also called burst size). The token bucket regulator puts a constraint on the traffic so that over any time interval t, the number of bits that can be transmitted is bounded by $rt + b$.

The time it takes for a packet to pass through a token bucket regulator combined with an FCFS scheduler is bounded by the delay imposed by the regulator plus the processing delay in the scheduler. Furthermore, for a chain of such rate-control schedulers, the end-to-end delay is bounded by the sum of the delays of the individual rate-control schedulers, and the bandwidth guarantee is maintained across the chain.

The token bucket regulator ensures that traffic arriving at the scheduler conforms to the traffic profile. This could be done in two ways: traffic that does not conform to the traffic profile could be discarded, or it could be delayed in a queue until it is eligible for scheduling. We refer to the

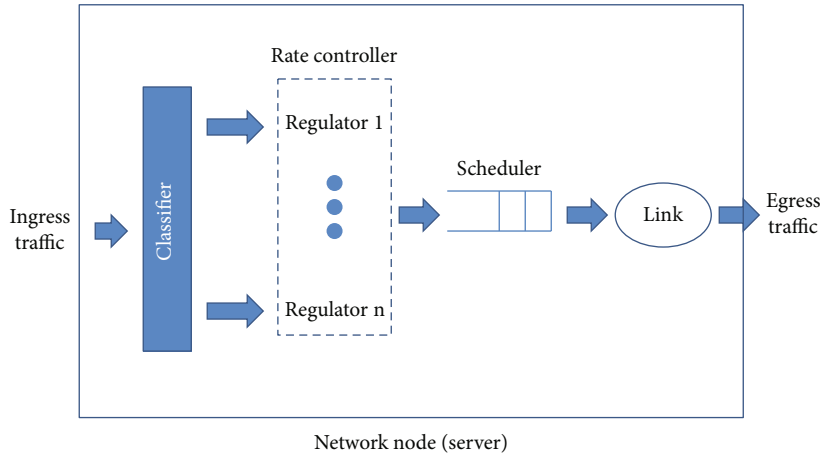

Figure 1: Rate-controlled server design.

former as *policing*, and the latter as *shaping*. In our architecture (which is discussed in Section 4), we use both.

4.3. Resource and Admission Control. The rate-controlled service discipline guarantees throughput and latency for data transmitted as a stream of bits. For computer communication, this is not directly suitable; instead, we need to express traffic profiles in units of bytes and packets. Our reasoning is as follows: the process of admitting a new connection involves asserting that there are sufficient resources to grant the connection's service demands. In the theoretical model of the rate-controlled service discipline, the only critical resource is link capacity. Thus, the admission decision consists of determining that the new connection can be accepted without oversubscribing link capacity.

In a computer system, link capacity may not be the only critical resource. Processing and storage capacity could also be limited, for instance. In particular, there is a significant per-packet cost in packet processing. As we will see in our experiments, for small packets, the limiting factor is packet processing capacity, rather than link capacity. Thus, the packet rate should also be part of the traffic profile. Of course, depending on the application, it might also be necessary to consider other resources in admission decisions. However, for the purposes of this paper, we limit our study to traffic profiles expressed in terms of packet rates (i.e., kilo packets per second). To be more specific, we use long-time packet rate (i.e., average throughput) and packets burst size (for the bounded delay) as traffic profiles. A token bucket is configured to rate the limit according to this traffic profile. For instance, a traffic profile of 100,5 for a flow means that 100 kpps to be configured as token bucket rate limit whereas bucket size should be configured to 5 means the maximum allowed burst size is 5 packets. The bounded packet delay and burst size are related in this context. Since the bursty traffic leads to unbounded packet delays, therefore, we bound burst size (thorough token bucket size) which in turn leads to bounded packet delay. The details of our rate-controlled service discipline with token buckets are presented in the next section.

5. Adaptation of Rate-Controlled Service Discipline for Virtualized Network Functions

In this section, we present how rate-controlled service discipline can be adapted for a virtualized computer system (VS). We take a virtual router (packet forwarding through a virtual machine) as an example of a virtual network function and explore how a rate-controlled service can be provisioned in this case.

5.1. Architecture of a Virtualization System. Our aim is to design a VS that hosts a number of virtual network functions, as illustrated in Figure 2, where multiple virtual network functions (exemplified by virtual routers, VRs) share ingress and egress network links. When a packet arrives on an ingress link, the packet is processed by the classifier. The classifier determines, for example, based on the destination MAC address, the VR for which the packet is intended, and places the packet on the corresponding ingress queue. When the VR is ready to process the packet, the VR gets the packet from the ingress queue, performs the packet processing operations (such as packet forwarding), and finally places the packet on the output queue of an egress link.

5.2. Performance Guarantees. Our objective is to provide performance guarantees for the VRs. We start with a *service agreement* between the user and the provider of the VR service. The agreement specifies a traffic profile for a connection; the provider agrees to provide resources for processing traffic according to the profile, while the user commits to generating traffic that conforms to the profile. Hence, for each VR, there is a corresponding traffic profile, and consequently, we organize our rate-control service design so that each VR serves one connection.

Once the classifier has identified the VR to which an incoming packet belongs, the next step is to verify that the arrival of the packet complies with the traffic profile for that VR. We choose token buckets to express traffic profile characteristics, so we use a token bucket filter to check that the packet can be accepted for further processing. This is shown as the policer function in Figure 3.

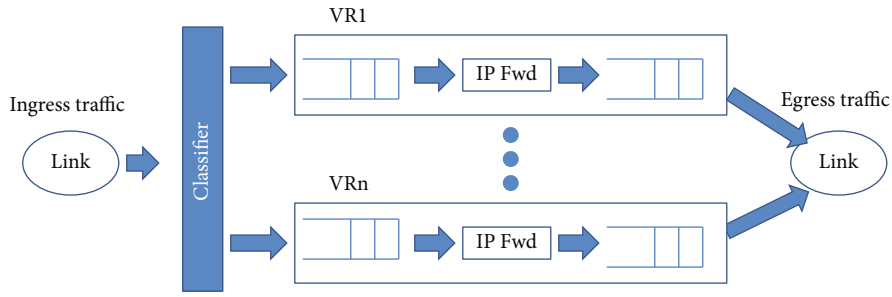

Figure 2: Architecture of a virtualized system.

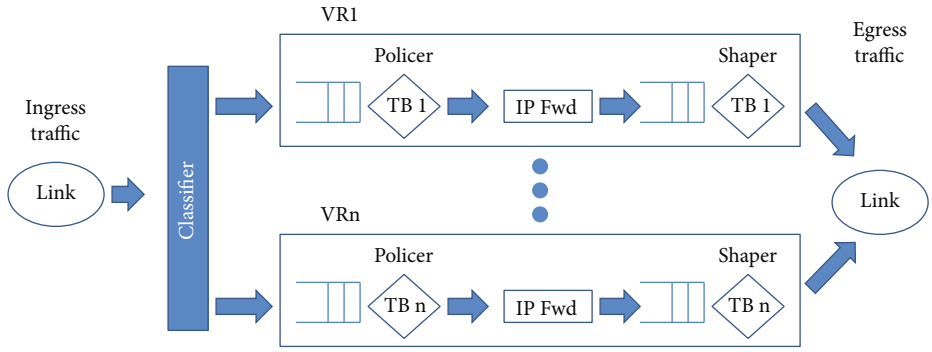

Figure 3: Rate-controlled virtual routers.

Having determined that the packet is within the traffic profile, the packet is then dispatched to the corresponding VR for processing. Since CPU resources are shared among all VRs, this involves a CPU scheduling decision to select the VR to execute next. How this scheduling is performed depends on the virtualization platform and has consequences for the performance of our design.

The way in which VRs are scheduled onto CPUs is important for performance in two ways. First, there is a certain overhead involved, so there is a cost associated with switching contexts from one VR to another. This influences the total performance of the VS. Second, due to this cost, process context switches cannot take place too often. Typically, the context switching rate is lower than the packet arrival rate. The result of this is that VRs process packets in *clusters*. When a VR is scheduled for execution, it is likely that there is a cluster of packets waiting in the VR's input queue. The VR will process all those packets right after each other (or as many packets as CPU scheduling permits). This packet clustering can improve overall processing efficiency, but it comes at the expense of increased average packet delay.

Another possible cause of packet clustering is *interrupt mitigation* (also known as interrupt coalescing or interrupt moderation). Interrupt processing is expensive, and when the packet rate goes up, it can be more efficient to disable interrupts in the network driver and use polling instead. The NAPI interface in the Linux networking subsystem dynamically switches between interrupt and polling mode depending on packet rate [42]. This increases performance significantly, in particular at high packet rates, but also leads to clustering.

Clustering not only increases latency but also distorts traffic profiles, since the burstiness increases. In fact, on the egress side of a VR, the traffic may no longer conform to the agreed-upon traffic profile. This could have serious consequences for our rate-control service design since the end-to-end service guarantees rely on the traffic profile to be maintained at each intermediate step. In other words, if we want to provide service guarantees through a chain of VRs, we need to ensure that the output from each VR conforms to the traffic profile.

To make sure that a VR maintains a certain traffic profile, we can use a token bucket again. At the egress side of a VR, the traffic passes through a token bucket shaper (Figure 3)—a queue where the departures are controlled by a token bucket. This implies a *non-work-conserving* design: the link may sometimes be idle even when there is a packet to send, as transmitting the packet would be in violation of the traffic profile.

Expressing the performance guarantees in terms of worst-case delay is straightforward. Assume we have n rate-controlled VRs mapped onto a CPU according to Figure 3. There are two components contributing to the delay: the waiting time T_W, which is the time a packet is waiting in the ingress queue of a VR, and the service time T_S, which is the time it takes for a VR to process a cluster of packets. The waiting time for VR n depends on the service time for VRs 1 to $(n-1)$:

$$\mathrm{Tw} = \sum_{i=1}^{n-1} T_{\mathrm{S(i)}}. \quad (1)$$

The worst-case delay, $d_{\max}$, experienced by a packet in VR n is obtained by adding the service time of VR n, so that

$$d_{\max} \leq \sum_{i=1}^{n} T_{S(i)}. \quad (2)$$

Furthermore, T_S is determined by the size of the packet cluster, b (packets), and the processing rate g (packets per second):

$$T_S = \frac{b}{g}. \quad (3)$$

Accordingly, the worst-case delay experienced by a packet in VR n is

$$d_{\max} \leq \frac{\sum_{i=1}^{n} b_i}{g_i}. \quad (4)$$

5.3. Policing and Shaping. In our design, we use token buckets in two ways (see Figure 3). At the ingress, a token bucket is used as a policing function to make sure that the traffic profile is maintained. Noncompliant packets are discarded. The main purpose of our token-bucket policing function is to protect against misbehaving sources and against potential disturbances on the links so that resources in the virtualization system are not overutilized. On the egress side, we use a token bucket to compensate for distortions to the traffic caused by imperfections of the platform where the VR executes. Here, packets are not dropped. Instead, they are queued until they are eligible for transmission on the egress link according to the traffic profile.

So how does our architecture relate to the regulators and schedulers of the rate-controlled service design in Figure 1?

The purpose of the regulator in the rate-controlled service design is to regulate the traffic to the scheduler. In our architecture, the token bucket policing function at the ingress *is not* intended for the regulator function (it is to protect against misbehaving sources). This regulator function is actually performed by the token bucket shaper at the egress. This means that the regulator and scheduler functions are split over two systems: the regulator is located at the egress of the first system whereas the scheduler is located at the ingress of the next (subsequent/second) system.

To elaborate on this further, it would be possible to shift the regulator function to the ingress side, by turning the token bucket policer at the ingress into a token bucket shaper. However, considering that the packet processing may distort the traffic profile and add burstiness (i.e., clustering), we still need the shaping function on the egress side to reshape the traffic back into the proper profile. Therefore, we prefer to keep the regulator function on the egress side and thereby make it the responsibility of the (first) system to make sure that traffic on the output conforms to the profile.

6. Implementing Rate-Controlled Virtualized Network Functions in Linux

In this section, we discuss virtualization techniques for the VS, in the areas of computer system virtualization and I/O virtualization. We analyze how token buckets can be added along the forwarding path of a VR, by evaluating an existing implementation of token bucket filters in Linux.

6.1. Computer System Virtualization. Computer virtualization allows multiple operating systems instances to run at the same time, as virtual machines, on the same computer. With *hypervisor-based* virtualization, this is accomplished through the virtualization of hardware resources while with *container-based* virtualization, operating system resources are virtualized (such as files, system libraries, and routing tables).

There are large differences between the two virtualization technologies in how packet processing is performed, which has consequences for networking performance. Therefore, in this paper, we compare the two technologies, with KVM as a representative for hypervisor-based virtualization [5] and LXC for containers [4]. Both KVM and LXC are part of the main Linux kernel distribution. We analyze how packets are scheduled in the two cases and the effects this has on performance.

In KVM, a virtual machine runs as a regular user-space process, which is scheduled using the "Completely Fair Scheduler" (CFS) process [43]. The CFS uses a configurable weight parameter to determine the CPU share for a process [44, 45].

One possible approach for achieving service guarantees would be to use the CFS scheduler to mimic a *weighted fair queuing* packet scheduler [46], by assigning process weights in proportion to the rates of the connections served by the processes. On average, this could be expected to give the desired long-time distribution of capacity among connections. However, in order to also guarantee end-to-end delay bounds, the connection still needs token bucket control [46]. Thus, using the support for weighted scheduling in Linux would not eliminate the need for token buckets. In fact, as we shall see, with token bucket rate control, there is no need for weighted process scheduling in order to achieve our objectives; it suffices to serve packets on a first come, first serve basis.

LXC virtualization is done at the operating system level, where the operating system provides different views of system resources to different containers. For instance, separate containers could have distinct routing tables. In contrast to KVM, where packet processing takes place in user processes, LXC packet processing is done by the operating system kernel. The operating system kernel is shared by all containers, so the kernel does packet processing on behalf of the containers. Therefore, packet processing in LXC can be expected to be more efficient, as there is no involvement in user process scheduling and context switching.

6.2. Network I/O Virtualization. In addition to computer virtualization techniques as represented by KVM and LXC, network I/O virtualization is another important component

for virtual network functions. We will use it for directing network traffic between network interface cards (NICs) and VRs (the classifier function in Figure 3). The Linux software bridge [47] and Open vSwitch [48] are examples of network virtualization in Linux. Furthermore, single root I/O virtualization (SR-IOV [49, 50]) is a hardware technique to offload packet switching functionality onto NICs. With SR-IOV, several input queues share the same NIC. An incoming packet is classified on the NIC and then dispatched to the corresponding input queue. Such hardware assistance has been shown to significantly improve packet forwarding performance [10]. Therefore, we use SR-IOV to direct incoming packets to VRs.

6.3. Adding Token Bucket on the Packet Forwarding Path of a Virtual Router. The first step in designing our rate-control scheduler for Linux is to consider the token bucket policing function, which verifies that packets conform to traffic profiles. First, we review the packet processing path in the operating system kernel. The path is illustrated in Figure 4. Note that this applies to both kinds of virtualization—with LXC, the processing takes place in the operating system kernel of the VS (host kernel), while with KVM, it takes place in the VR's (guest machine) kernels.

When a packet is received on a NIC, the packet is transferred through direct memory access (DMA) to memory and made available on an input queue organized as a ring buffer (RX Ring) (Figure 4), and a hardware interrupt is generated. The interrupt handler does not perform the actual packet processing; instead, a software interrupt (*SoftIRQ*) is posted. When the SoftIRQ occurs, the packet is processed by the network device driver (we use the Intel ixgbe driver for Ethernet NICs [51]), and the packet is then passed to the IP forwarding module to determine the egress network interface. Finally, the packet is placed on the egress NIC's queue for transmission.

Linux has support for token buckets as a traffic control queuing discipline, "tcqdisc" [52], which we denote "TcTB," i.e., traffic control through the token bucket. The corresponding packet processing path is illustrated in Figure 5. It can be seen that a packet is placed on a second queue ("Ingress qdisc") and another SoftIRQ is posted. When this second SoftIRQ is processed, the packet is processed by TcTB (which is configured as a traffic policer). Later in the forwarding path, when a packet is available on the egress queue, it is processed by another TcTB which acts as a shaper according to our proposed architecture.

We notice in Figure 5 that TcTB policer is located at a later point in the processing chain. The packet first goes through a couple of queuing operations and SoftIRQs, before it is policed. In general, if a packet should be dropped, it is better to drop it early in the processing chain. In a sense, CPU cycles spent processing a packet (which is dropped later on) are wasted cycles. Our hypothesis is that implementing a token bucket policing at an earlier stage of the forwarding path can be more efficient. As the goal of this paper is to investigate the properties of our rate-controlled design and its possible implementations, rather than to seek its maximum speed, we continue our study with existing implementation of token bucket filter in Linux (i.e., TcTB policer and shaper). However, we argue that for practical use, both the policing and the shaping functions are better suited for implementation in hardware on network interface cards than for software implementations.

7. Experimental Evaluation

In this section, we evaluate the performance of two implementations of our design of rate-controlled service: one based on KVM and the other based on LXC. As a first step, we evaluate the overhead of policer and shaper modules that are added to the forwarding path of a virtual router. After that, we analyze average throughput and latency for rate-controlled VRs under different scenarios while varying offered load, the number of virtual routers, and token bucket settings.

7.1. Experimental Setup. The configuration for the experimental evaluation is shown in Figure 6. We use three Linux machines connected in a sequence. The first machine is a traffic generator that runs the pktgen [53, 54] Linux packet generator to provide traffic load. The next machine is the device under test (DUT), where the virtual routers are running. The DUT is configured with two network interfaces, eth0, and eth1. The ingress traffic is received on eth0 and forwarded towards eth1. The traffic then arrives at the third machine, the traffic sink, which runs the receiver side of pktgen gathering performance statistics.

The DUT consists of an Intel i7 Quad Core 3.4 GHz processor with directed I/O support (Intel Q67 Express chipset) and 4 GB of RAM. The machine is equipped with an Intel 82599 chipset 10 Gb/s dual-port network adapter and runs Linux kernel net-next v. 4.6.1. For each experiment, five iterations of the test are performed, and average results are taken.

In all experiments, unless stated otherwise, we use a single CPU core and 256-byte packets as offered load. The reason for using a single CPU (although the system contains multiple CPU cores) and small-size packets is to offer maximum stress to the system since our objective is to evaluate rate-controlled service discipline in overload conditions. It is obvious that more CPU cores can be added to improve packet processing rates. However, it is important to mention that the focus of our work is to evaluate rate-controlled service discipline which is different than the topic of performance optimization. For instance, there are studies [24, 25] on high-performance networking where the targets are 40 and 100 Gb/s line rates. Indeed, the results reported in these studies are far superior to the rates reported in our study. We believe that performance optimization techniques are complementary to the work of rate-controlled service discipline.

7.2. Experiments. We have performed three different experiments to validate our rate-controlled design. The objective and details of each experiment are given below.

7.2.1. Processing Overhead of Token Bucket Policing and Shaping. In this experiment, we wish to assess the overhead

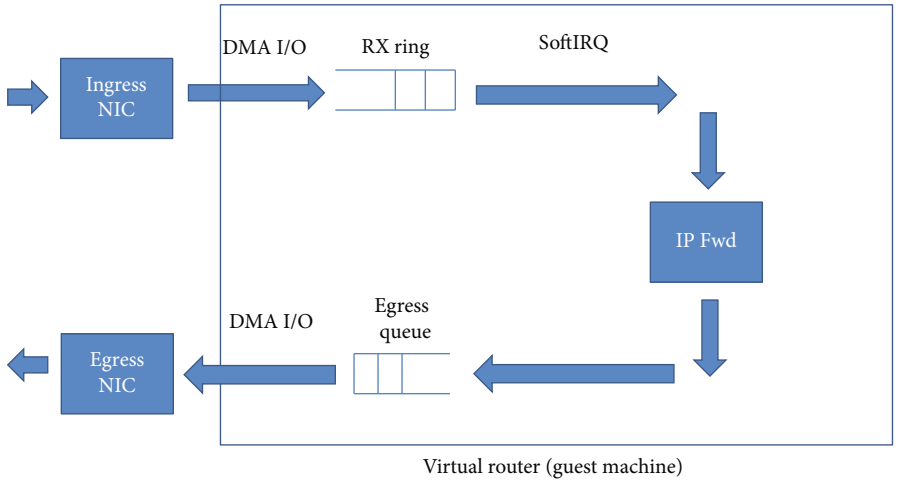

Figure 4: Forwarding path of the hardware-assisted virtual router in Linux.

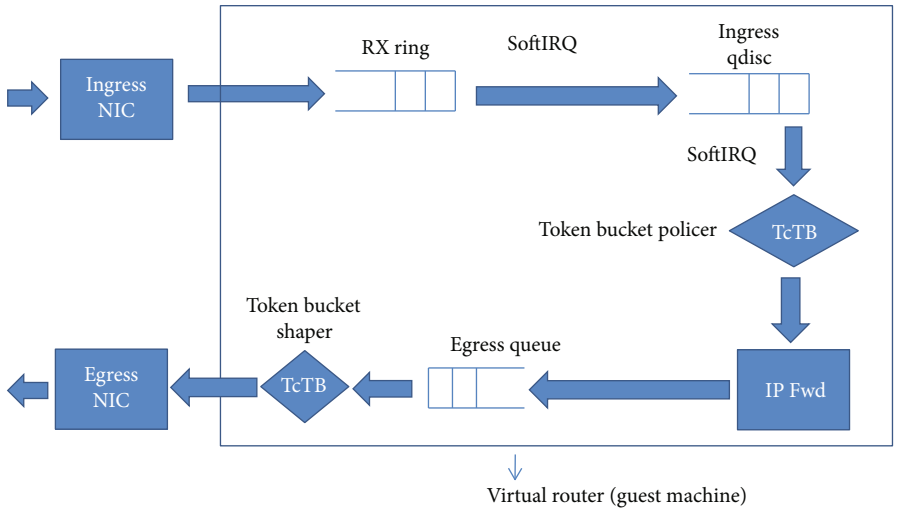

Figure 5: Forwarding path with a token bucket as a rate-controlled service discipline (TcTB for policing and shaping).

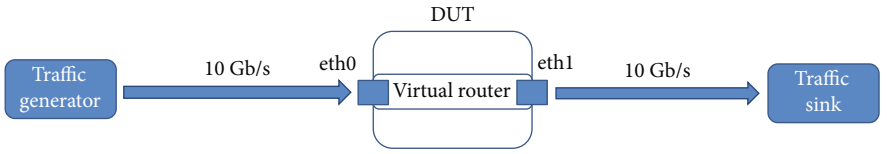

Figure 6: Experimental setup.

of adding token bucket functionality to the packet processing path. We do this by using average throughput (packet rate) measurements where we generate traffic load and observe the effect on maximum throughput.

We run a VR on DUT (running alone without any parallel load) and measure the packet forwarding performance (i.e., throughput) of VR against the offered load. As a baseline case, we measure the throughput of a VR without adding any token bucket filter. Then we add a token bucket filter along the forwarding path and measure the throughput again. The performance difference between the two cases indicates the overhead of adding a token bucket. For the token bucket, we first add TcTB shaper only without any policer on the ingress path (i.e., TcTB shaper only). Then we introduce TcTB policer, so both TcTB policer and TcTB shaper are added in this case (i.e., TcTB policer and shaper).

It is important to mention that we want token buckets processing to take place for each packet, but without dropping any packets (since packet drop results in throughput degradation which is not desirable in this scenario). Therefore, we need to use a token bucket rate that is above the highest possible rate when buckets are enabled. To determine this rate, we first measure the throughput of the baseline system without traffic control (as shown in Figure 4),

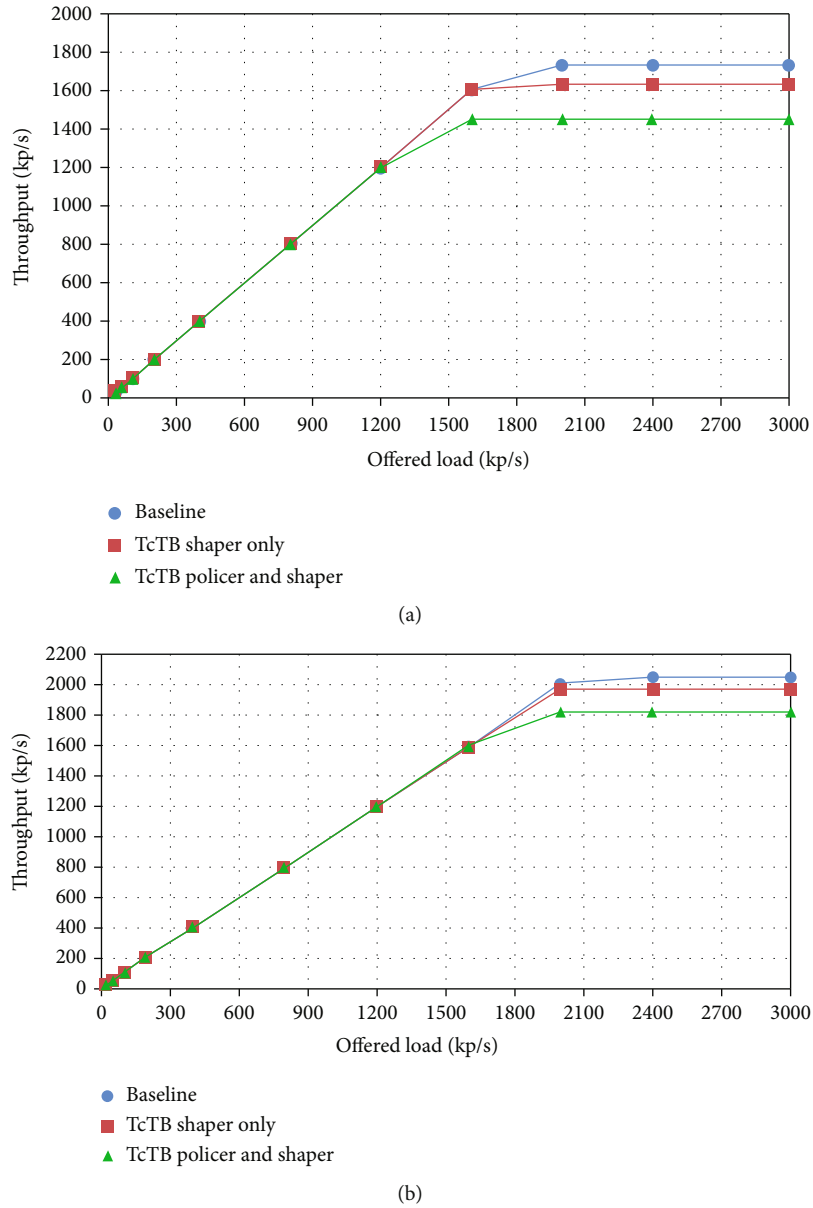

Figure 7: Throughput of a KVM VR (a) and an LXC VR (b) running on a single CPU core for three cases: without token bucket (Baseline), with TcTB traffic shaper only, and with TcTB traffic policer and shaper. The rate limit is configured to baseline throughput (1730 kp/s for KVM and 2052 kp/s for LXC), and the bucket size is 1000 packets.

and we take this throughput as the token bucket rate limit for two token bucket cases in this experiment. The results are shown in Figure 7(b).

For KVM, by adding the token bucket shaper on the egress side of VR, the maximum throughput drops to 1635 kp/s as compared to the baseline throughput of 1730 kp/s. Throughput further reduces to 1451 kp/s when we add TcTB policer as well. It shows 16% performance drop as compared to the baseline case.

Similar curve patterns can be found in the case of LXC but indeed with overall superior performance as compared to KVM (as expected due to the lightweight nature of containers). For LXC, performance drops from 2052 kp/s to 1827 kp/s while moving from baseline to token bucket (policer and shaper). It means performance drop is 11% when both policer and shaper are added. This result indicates that the performance overhead of adding token bucket filters can be minimized to some extent by using lightweight container virtualization. Our hypothesis is that overhead can be further reduced by placing token bucket policer at an early stage of the packet processing path, for instance, implementing token buckets as a part of the network device driver or even completely offloaded to NIC hardware.

7.2.2. Throughput and Latency for Varying Token Bucket Sizes with a Single VR. The packet clustering that is an effect of scheduling mechanisms in the operating system, such as interrupt mitigation and process scheduling, will have an

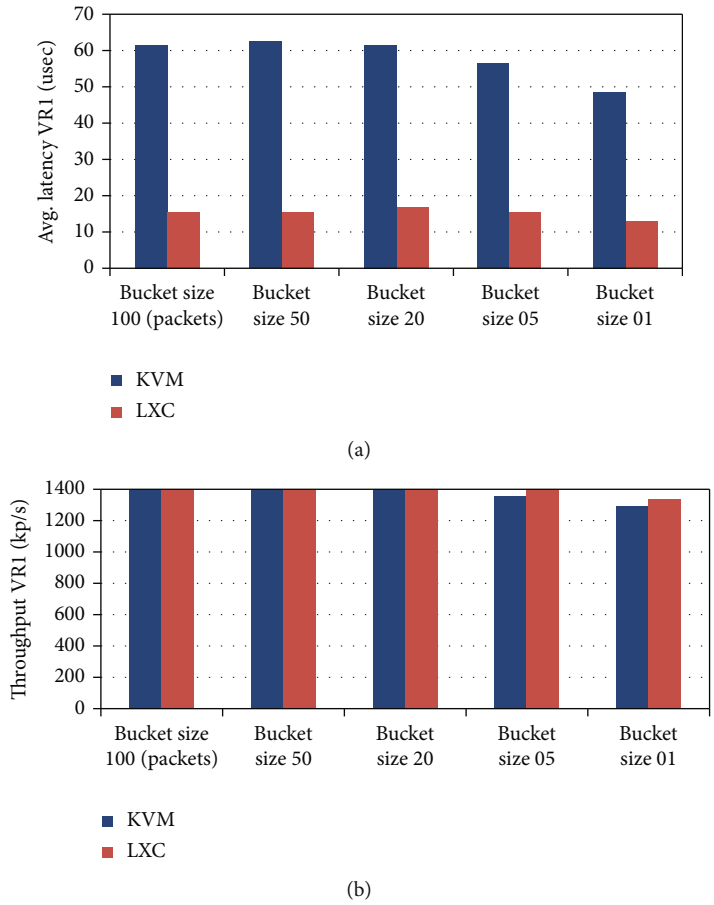

FIGURE 8: Average latency (a) and throughput (b) for a virtualization system with a single router for varying token bucket sizes. The offered load is 1400 kp/s, and the token bucket rate limit is 1400 kp/s. The policing and shaping token buckets have the same configuration (rate limit and bucket size).

influence on traffic, and our objective in this experiment is to study the relationship between token bucket filters and packet clustering.

We start with a basic scenario with a single virtual router (VR) configured with TcTB traffic policing and shaping, where we again set the token bucket fill rates so high that no packet drops should occur. We run the experiments on both LXC and KVM, chose an offered load that is slightly below the maximum throughput for the KVM configuration (1400 kilo packets per second), and set the token bucket rate to be the same as the offered load. We then measure throughput and latency for five different token bucket sizes: 1, 5, 20, 50, and 100 packets. The results are shown in Figure 8(b).

There is a significant difference in latency between LXC and KVM, where the latency for KVM is considerably higher. This is mainly an effect of scheduling: With KVM, the VR runs as a user process under the control of the operating system scheduler. Scheduling decisions are made at a lower rate compared to the packet arrival rate. Therefore, packets will queue up in the VR's input buffer before they are processed by the VR, with the effect that the VR will process packets in clusters. This leads to an increase in average latency. In contrast, LXC packets are processed immediately when they arrive, since LXC packet processing takes place in the kernel of the VS. Therefore, there is no corresponding queue build-up for LXC, and the average latency is kept down.

For KVM, throughput and latency decrease for token buckets of five packets and smaller. The explanation for this is that there is a mismatch between the size of the token bucket and the packet clustering caused by the Linux scheduling. Clusters are larger than the token bucket, and therefore, clusters are truncated to the size of the token bucket, with the result that packets are discarded and the throughput goes down. As the token bucket gets smaller, more packets are dropped, which shortens the packet queues and thereby lowers the average delay.

LXC, in contrast, does not exhibit such performance characteristics, so there are no noticeable clustering effects for LXC. For a token bucket size of one, which corresponds to a perfect constant rate, there is a certain performance degradation also for LXC, indicating that there are small variations in packet processing rate.

We proceed with a separate investigation of LXC closer to its performance limits and measure throughput and latency as the load is gradually increased from 100 kp/s to 1800 kp/s. The results are shown in Fig. 9(b). For bucket size

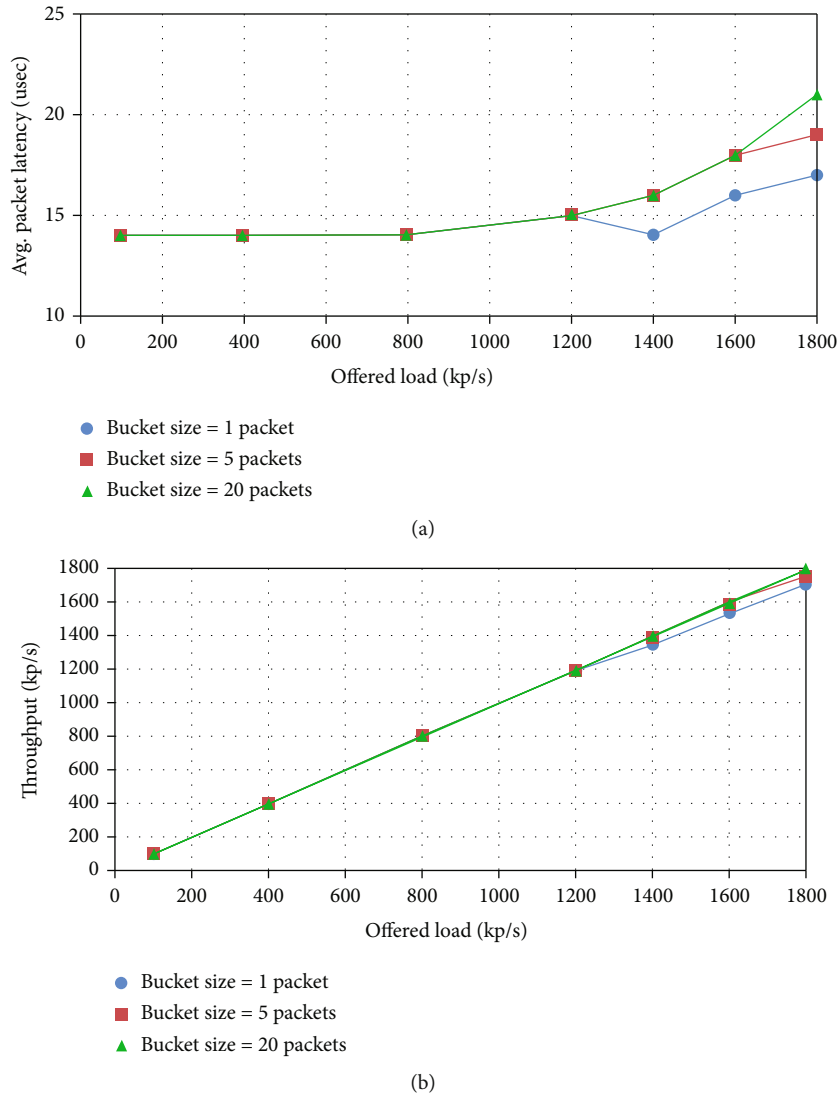

FIGURE 9: Average latency (a) and throughput (b) for an LXC-based virtual router for varying token bucket sizes. The offered load is gradually increased up to 1800 kp/s. The token bucket rate limit is the same as the offered load. The policing and shaping token buckets have the same values for their configuration parameters (rate limit and bucket size).

20, there are no indications of packet drop, but latency increases as the load increases. This indicates that there are queues building up as a result of clustering, but the clusters are small enough that they fit in the buckets. With smaller buckets (size one and five), throughput drops as well, so then the clusters are becoming too large for the buckets. This is expected: at high load, NAPI operates in polling mode, and the number of packets processed in a single poll operation increases when the load increases, and therefore, the size of clusters goes up. Furthermore, latency goes down for bucket size one at an offered load of 1400 kp/s (Figure 9(b) top), indicating that packet drops start to occur at this load. This result indicates that bucket size should not be too small since it may result in throughput degradation and violation of service level agreement between customer and service provider.

7.2.3. Throughput and Latency for Varying Token Bucket Sizes with Multiple Virtual Routers. In Section 7.2.2, we investigated how token bucket and packet cluster sizes affect the throughput and latency of a virtual router. This experiment was performed with a single virtual router without any parallel load on the system. As we know from the system analysis in Section 4.2, increasing the number of virtual routers increases the waiting for time T_W (and hence cluster size) for the virtual routers, which has implications for performance. We, therefore, proceed by studying latency and throughput for a configuration with multiple virtual routers while varying bucket size, in a manner similar to the experiments in Section 7.2.2.

We consider a scenario with eight virtual routers VR_1 to VR_8. The load generated for VR_1 is 900 kp/s, and its token bucket rate limit (and desired throughput) is also 900 kp/s. We measure throughput and latency for VR_1 while varying its bucket size between 1 and 100, in the presence of parallel load on VR_2 to VR_8. A total load of 1000 kp/s is uniformly distributed across VR_2 to VR_8. Each VR_2 to VR_8 has a token

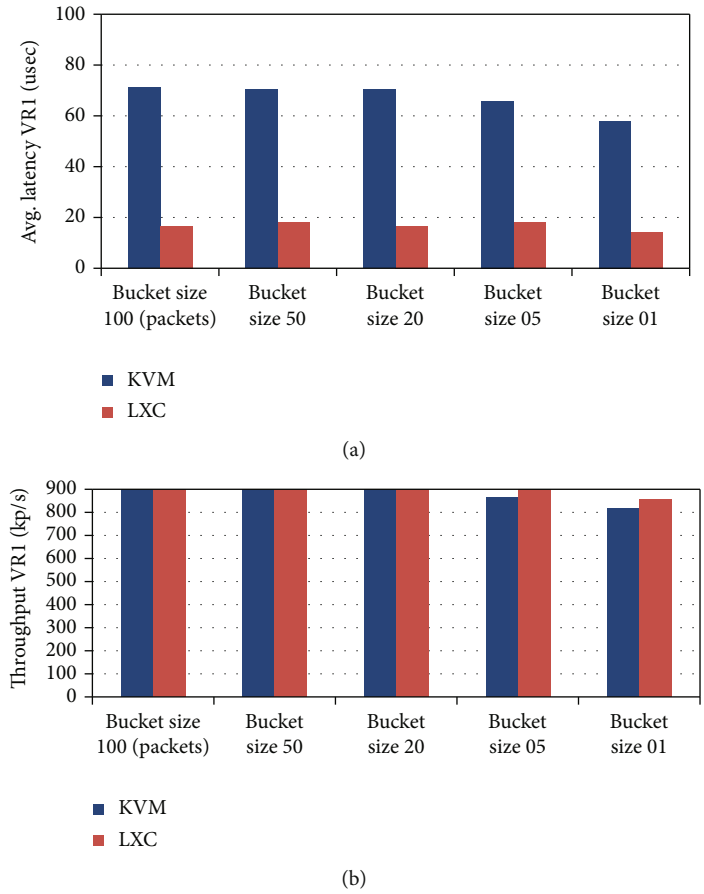

FIGURE 10: Average latency (a) and throughput (b) for VR_1 in presence of 7 rate-controlled *VRs*. The offered load is 900 kp/s at VR_1 and the token bucket rate limit is also configured to 900 kp/s. The configuration is the same both at traffic policer and shaper. The load of 1000 kp/s is uniformly distributed across VR_2 to VR_8. The token bucket rate limit is 50 kp/s and the bucket size is 1000 packets on each *VR* from VR_2 to VR_8.

bucket filter with a rate limit of 50 kp/s and bucket size of 1000 packets.

The results are shown in Figure 10. We notice that target throughput is achieved for VR_1 when larger token bucket sizes are used (20, 50, and 100 packets). However, for smaller token buckets, we again see the effect of packet clustering due to process scheduling, where the token bucket truncates clusters so that the throughput goes down when the bucket size is decreased. To be more precise, it is the traffic policer at VR_1 that drops packets beyond the bucket size. Then the question is, can we increase the bucket size of the traffic policer to avoid packet drops, and what are the performance implications of such a change?

In all experiments until now, we have the same token bucket parameters for the traffic policer and the shaper (we assume that these parameters are set according to a given traffic profile). We could, however, have a larger bucket size at traffic policer to allow packets beyond the traffic profile while still maintaining the traffic profile at traffic shaper. This means that we adjust the traffic policer to allow for traffic distortions caused by the virtualization system and use the traffic shaper to ensure that egress traffic still conforms to the traffic profile. These settings are in accordance with our design (Section 4.3), where the shaper is responsible for regulating traffic, whereas the purpose of policer is to protect against misbehaving sources and interfering activities. In other words, we sacrifice a small degree of resource efficiency by configuring a larger bucket size at the traffic policer in order to improve the packet forwarding performance of VR_1. For this purpose, we reconfigure the traffic policer at VR_1 to a rate limit of 900 kp/s and bucket size of 20 packets. The rate limit of the traffic shaper at VR_1 is also 900 kp/s. We measure throughput and latency for VR_1 while varying its shaper bucket size from 1 to 5 packets. The rest of the virtual routers (VR_2 to VR_8) are configured in the same way as described above, and the distribution of offered load is also the same.

Both KVM and LXC reach the target throughput of 900 kp/s at VR_1 with these revised configurations. The latency for VR1 is shown in Figure 11. We notice that the latency is increased in comparison to Figure 10. This is expected, due to the fact that excessive packets now are queued at the shaper (instead of being dropped at the policer). It results in long queues at the traffic shaper and hence higher latency, but target throughput is achieved on the other hand. This result indicates a tradeoff between average throughput and latency.

It is also important to mention that the other virtual routers (VR_2 to $VR_{8)}$ are rate-limited successfully to the

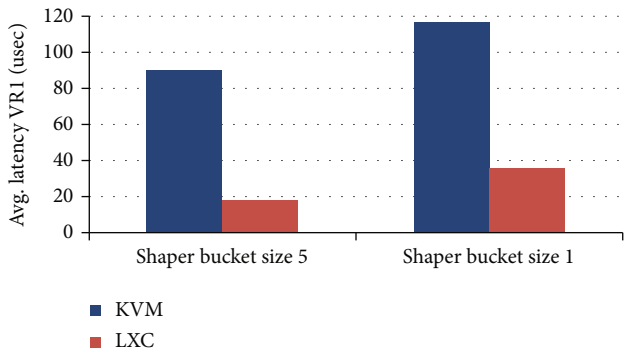

FIGURE 11: Average latency for VR_1 in the presence of 7 rate-controlled *VRs*. The offered load is 900 kp/s at VR_1, and the token bucket rate limit is also configured to 900 kp/s (both at traffic policer and shaper). The bucket size at traffic policer is 20 packets. The load of 1000 kp/s is uniformly distributed across VR_2 to VR_8. The token bucket rate limit is 50 kp/s, and the bucket size is 1000 packets on each *VR* from VR_2 to VR_8.

desired throughput of 50 kp/s. We notice that even though the aggregate offered load on the system (1900 kp/s on all virtual routers) is higher than the processing capacity of the system, the system still achieves performance guarantees for all competing virtual routers. It is due to the fact that resource contention is avoided through rate limits and excessive offered load is dropped. As a result, we see strong performance isolation between virtual routers which was not the case for the scenario presented earlier without rate limits in Section 2 (Table 1).

7.3. Summary and Discussion. The purpose of our experiments is to investigate the performance properties of the rate-controlled service discipline, where we wish to study the trade-offs involved in adding rate control functionality to the packet processing path. Certainly, there is some performance overhead of adding token bucket filters along the packet processing path. However, on the other hand, we guarantee performance according to configured traffic profiles.

When we compare LXC and KVM implementation of our design, we find that LXC has considerably lower latency and higher throughput. This is not surprising, since there are fundamental differences between KVM and LXC that have far-reaching implications when it comes to realizing a rate-controlled service discipline. With KVM, as with most other hypervisor-based virtualization technologies, packet processing takes place in user processes. That means that before a packet can be processed, the corresponding user process needs to be scheduled for execution. Process context switching occurs with a frequency that is much lower than the packet arrival rate, which leads to queue build-up so that packets are processed in clusters. LXC, as a representative of container-based virtualization, does not add latency in this way. With LXC, packets are processed immediately when they arrive on the network interface; packet processing takes place in the operating system kernel, not in user-space processes, so there is no process scheduling involved. Overall, our results indicate that it is possible to maintain a given target throughput and low (bounded) average packet latency both for KVM and LXC. The results support our hypothesis that a rate-controlled server can offer performance guarantees in a virtual environment.

8. Conclusions and Future Work

In this paper, we investigate how throughput guarantees and bounded delay can be provided for virtualized network functions running as virtual machines on a regular computer system. Our work is based on design, implementation, and experimental evaluation. We put forward an architectural design of a virtualization system using a rate-controlled service discipline based on token bucket regulators combined with first come, first served scheduling. We demonstrate that our solution can provide service guarantees in a context with multiple virtual machines competing for resources, as well as under overload conditions. Moreover, we investigate the effects of packet clustering that stem from executing virtualized network functions in software. Finally, we present experimental evaluations of our design, implemented using two different virtualization techniques in Linux: the KVM hypervisor and LXC containers. Our results show that our design provides high rate limit accuracy and that it behaves well in overload situations, in particular when using LXC virtualization.

In this work, we have evaluated rate-controlled service design for an IP forwarder which is one example of virtualized network function. In future, we plan to add more network functions (such as firewall, NAT) and evaluate our design for a chain of network functions. More complex scenarios can also be envisioned by considering parallel running network function chains on a multicore platform. It would be interesting to examine the scalability of the solution. For instance, one interesting question is how many parallel network function chains can be enabled on a single server? Furthermore, how the proposed architecture can be extended to a large NFV setup where the underlying infrastructure is based on a cluster of commodity servers and switches?

Data Availability

The data used in this research can be obtained upon request to the corresponding authors.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

"The authors are grateful to the Taif University Researchers Supporting Project number (TURSP-2020/36), Taif, Saudi Arabia".

References

[1] ETSI portal, "Network functions virtualization: an introduction, benefits, enablers, challenges and call for action," 2022, http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[2] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: state-of-the-art and research challenges," *IEEE Communication Surveys and Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[3] H. Zhang and D. Ferrari, "Rate-controlled service disciplines," *Journal of High Speed Networks*, vol. 3, no. 4, pp. 389–412, 1994.

[4] "Linux containers," 2022, https://linuxcontainers.org/.

[5] A. Kivity, Y. Kamay, and D. Laor, "KVM: the Linux virtual machine monitor," in *Proceedings of Linux Symposium*, pp. 225–230, Canada, Ottawa, June 2007.

[6] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, "dcat: dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service," in *Proceedings of ACM EuroSys Conference (EuroSys'18)*, pp. 1–13, Porto, Portugal, April 2018.

[7] M. Vanga, A. Gujarati, and B. Brandenburg, "Tableau: a high-throughput and predictable VM scheduler for high-density workloads," in *Proceedings of ACM EuroSys Conference (EuroSys'18)*, pp. 1–16, Porto, Portugal, April 2018.

[8] A. Tootoonchian, A. Panda, C. Lan et al., "ResQ: enabling SLOs in network function virtualization," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pp. 283–297, Renton, WA, USA, April 2018.

[9] S. Rathore, M. Hidell, and P. Sjödin, "Data plane optimization in open virtual router," in *Proceedings of IFIP Networking Conference*, pp. 379–392, Valencia, Spain, May 2011.

[10] S. Rathore, M. Hidell, and P. Sjödin, "KVM vs. LXC: comparing performance and isolation of hardware-assisted virtual routers," *American Journal of Networks and Communications*, vol. 2, no. 4, pp. 88–96, 2013.

[11] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *Proceedings of International Conference on Computing, Networking and Communications (ICNC'16)*, pp. 1–7, Kauai, Hawaii, March 2016.

[12] J. Martins, M. Ahmed, C. Raiciu et al., "ClickOS and the art of network function virtualization," in *Proceedings of 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pp. 459–473, Seattle, Washington, April 2014.

[13] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: high performance and flexible networking using virtualization on commodity platforms," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, Seattle, Washington, April 2014.

[14] G. Wang and T. E. Ng, "The Impact of virtualization on network performance of Amazon EC2 Datacenter," in *Proceedings of 29th IEEE Conference on Information Communications (INFOCOM'10)*, pp. 1–9, San Diego, California, March 2010.

[15] J. Whiteaker, F. Schneider, and R. Teixeira, "Explaining packet delays under virtualization," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 38–44, 2011.

[16] R. S. Couto, M. E. Campista, and L. H. Costa, "XTC: a throughput control mechanism for Xen-based virtualized software routers," in *Proceeding of 54th IEEE Global Communications Conference (GLOBECOM'11)*, pp. 1–6, Houston, Texas, December 2011.

[17] "Docker containers," 2022, https://www.docker.com.

[18] R. Nakamura, Y. Sekiya, and H. Tazaki, "Grafting sockets for fast container networking," in *Proceedings of ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS'18)*, pp. 15–27, Lthaca, NY, USA, July 2018.

[19] Xen virtualization, "The Xen Project," 2022, http://www.xenproject.org/.

[20] "Intel data plane development kit (DPDK)," 2022, http://dpdk.org.

[21] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," in *Proceedings of 16th IEEE International Symposium on High Performance Computer Architecture*, pp. 1471–1480, Bangalore, India, January 2010.

[22] J. Liu, "Evaluating standard-based self-virtualizing devices: a performance study on 10 GbE NICs with SR-IOV support," in *Proceedings of 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*, pp. 1–12, Atlanta, Georgia, April 2010.

[23] T. Hoiland-Jorgensen, J. D. Brouer, D. Borkmann et al., "The eXpress data path: fast programmable packet processing in the operating system kernel," in *Proceedings of ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT'18)*, pp. 54–66, Crete, Greece, December 2018.

[24] R. McGuinness and G. Porter, "Evaluating the performance of software NICs for 100-gb/s datacenter traffic control," in *Proceedings of ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS'18)*, pp. 74–88, Lthaca, NY, USA, July 2018.

[25] P. Li, X. Wu, Y. Ran, and Y. Luo, "Designing virtual network functions for 100 GbE network using multicore processors," in *Proceedings of ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS'17)*, pp. 49–59, Beijing, China, May 2017.

[26] H. Yao, M. Xiong, H. Li, L. Gu, and D. Zeng, "Joint optimization of function mapping and preemptive scheduling for service chains in network function virtualization," *ELSEVIER Journal of Future Generation Computer Systems*, vol. 108, pp. 1112–1118, 2020.

[27] M. Bunyakitanon, A. P. da Silva, X. Vasilakos, R. Nejabati, and D. Simeonidou, "Auto-3P: an autonomous VNF performance prediction & placement framework based on machine learning," *Elsevier Journalof Computer Networks*, vol. 181, p. 107433, 2020.

[28] S. Asgari, S. Jamali, R. Fotohi, and M. Nooshyar, "Performance-aware placement and chaining scheme for virtualized network functions: a particle swarm optimization approach," *The Journal of Supercomputing*, vol. 77, no. 11, pp. 12209–12229, 2021.

[29] Y. Yue, B. Cheng, X. Liu, M. Wang, B. Li, and J. Chen, "Resource optimization and delay guarantee virtual network function placement for mapping SFC requests in cloud networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1508–1523, 2021.

[30] Z. Wang, J. Zhang, and T. Huang, "Determining delay bounds for a chain of virtual network functions using network calculus," *IEEE Communications Letters*, vol. 25, no. 8, pp. 2550–2553, 2021.

[31] T. -M. Pham, "Traffic engineering based on reinforcement learning for service function chaining with delay guarantee," *IEEE Access*, vol. 9, pp. 121583–121592, 2021.

[32] S. V. Rossem, W. Tavernier, D. Colle, M. Pickavet, and P. Demeester, "Optimized sampling strategies to model the performance of virtualized network functions," *Springer Journal of Network and System Managements*, vol. 28, no. 4, pp. 1482–1521, 2020.

[33] S. V. Rossem, W. Tavernier, D. Colle, M. Pickavet, and P. Demeester, "VNF performance modelling: from standalone to chained topologies," *Elsevier Journal of Computer Networks*, vol. 181, p. 107428, 2020.

[34] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes, "Gatekeeper: supporting bandwidth guarantees for multitenant datacenter networks," in *Proceedings of USENIX 3rd Workshop on I/O Virtualization (WIOV'11)*, Portland, Oregon, June 2011.

[35] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "SENIC: scalable NIC for end-host rate limiting," in *Proceeding of 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pp. 475–488, Seattle, Washington, April 2014.

[36] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: predictable message completion time in the cloud," in *Proceedings of 2015 ACM Conference on SIGCOMM*, pp. 435–448, London, UK, August 2015.

[37] J. Gil Herrera and J. F. Botero, "Resource allocation in NFV: a comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[38] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *Proceedings of 34th IEEE Conference on Information Communications (INFOCOM'15)*, pp. 1346–1354, Hong Kong, April 2015.

[39] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. De Turck, and S. Davy, "Design and evaluation of algorithms for mapping and scheduling of virtual network functions," in *Proceedings of 1st IEEE Conference on Network Softwarization (NetSoft'15)*, pp. 1–9, London, U.K., April 2015.

[40] H. Moens and F. De Turck, "VNF-P: a model for efficient placement of virtualized network functions," in *Proceedings of 10th International Conference on Network and Service Management*, pp. 418–423, Rio de Janeiro, Brazil, November 2014.

[41] J. Turner, "New directions in communications (or which way to the information age?)," *IEEE Communications Magazine*, vol. 24, no. 10, pp. 8–15, 1986.

[42] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond Softnet," in *Proceedings of 5th Annual Linux Showcase & Conference*, Oakland, California, November 2001.

[43] "Linux process scheduler," 2022, https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[44] S. Wang, Y. Chen, W. Jiang, P. Li, T. Dai, and Y. Cui, "Fairness and interactivity of three CPU schedulers in Linux," in *Proceedings of 15th IEEE International Conference on Embeded and Real-Time Computing Systems and Applications (RTCSA'09)*, pp. 172–177, Beijing, China, August 2009.

[45] C. S. Pabla, "Completely fair scheduler," *Linux Journal*, vol. 2009, no. 184, 2009.

[46] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 344–357, 1993.

[47] "Linux bridge module," 2022, http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge.

[48] "Open vSwitch project," 2022, http://openvswitch.org/.

[49] "Intel Single root I/O virtualization support," 2022, https://www.intel.com/content/dam/doc/white-paper/pci-sig-single-root-io-virtualization-support-in-virtualization-technology-for-connectivity-paper.pdf.

[50] "Intel virtualization technology," 2022, https://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices.

[51] "Intel ixgbe network device drivers," 2022, http://www.intel.com/content/www/us/en/support/network-and-i-o/ethernet-products/000005688.html.

[52] "Linux Advanced Traffic Control," 2022, https://linux.die.net/man/8/tc.

[53] R. Olsson, "Pktgen the Linux packet generator," in *Proceedings of Linux Symposium*, vol. 2, pp. 11–24, Ottawa, Canada, July 2005.

[54] D. Turull, P. Sjödin, and R. Olsson, "Pktgen: measuring performance on high speed networks," *Journal of Computer Communications*, vol. 82, pp. 39–48, 2016.