Computer Networks 127 (2017) 109–125

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Achieving minimum bandwidth guarantees and work-conservation in large-scale, SDN-based datacenter networks

Daniel S. Marcon[a,b,*], Fabrício M. Mazzola[b], Marinho P. Barcellos[b]

[a] University of Vale do Rio dos Sinos (UNISINOS), RS, Brazil
[b] Federal University of Rio Grande do Sul (UFRGS), RS, Brazil

ARTICLE INFO

Article history:
Received 31 January 2017
Revised 8 July 2017
Accepted 14 August 2017
Available online 16 August 2017

Keywords:
Datacenter networks
Software-Defined Networking
Network sharing
Performance interference
Bandwidth guarantees
Work-conservation

ABSTRACT

Performance interference has been a well-known problem in datacenters and one that remains a constant topic of discussion in the literature. Software-Defined Networking (SDN) may enable the development of a robust solution for interference, as it allows dynamic control over resources through programmable interfaces and flow-based management. However, to date, the scalability of existing SDN-based approaches is limited, because of the number of entries required in flow tables and delays introduced. In this paper, we propose Predictor, a scheme to scalably address performance interference in SDN-based datacenter networks (DCNs), providing minimum bandwidth guarantees for applications and work-conservation for providers. Two novel SDN-based algorithms are proposed to address performance interference. Scalability is improved in Predictor as follows: first, it minimizes flow table size by controlling flows at *application-level*; second, it reduces flow setup time by proactively installing rules in switches. We conducted an extensive evaluation, in which we verify that Predictor provides (*i*) guaranteed and predictable network performance for applications and their tenants; (*ii*) work-conserving sharing for providers; and (*iii*) significant improvements over DevoFlow (the state-of-the-art SDN-based proposal for DCNs), reducing flow table size (up to 94%) and having similar controller load and flow setup time.

1. Introduction

Cloud providers lack practical and efficient mechanisms to offer bandwidth guarantees for applications [1–4]. The datacenter network (DCN) is typically oversubscribed and shared in a best-effort manner, relying on TCP to achieve high utilization and scalability. TCP, nonetheless, does not provide robust isolation among flows in the network [5–8]; in fact, long-lived flows with a large number of packets are privileged over small ones [9], a problem called *performance interference* [10–12]. The problem is a long-term challenge, and previous work on the field [1–3,11,13–16] has allowed important advances. In this context, we study how to address the problem in large-scale, SDN-based datacenter networks (DCNs). We aim at achieving minimum bandwidth guarantees for applications and their tenants while maintaining high utilization (i.e., providing work-conserving capabilities) in large DCNs.

Software-Defined Networking (SDN) [17] may enable the development of a robust solution to deal with performance interference, as it allows dynamic control over resources through programmable interfaces and flow-based management [18]. However, to date, the scalability of existing SDN-based approaches is limited, because of the number of entries required in flow tables and delays introduced (mostly related to flow setup time) [18–20]. The number of entries required in flow tables can be significantly higher than the amount of resources available in commodity switches used in DCNs [19,21], as such networks typically have very large flow rates (e.g., over 16 million/s [22]). Flow setup time, in turn, is associated with the transition between the data and control planes whenever a new flow arrives at a switch [1] (latency for communication between switches and the controller), and the high frequency at which flows arrive and demands change in DCNs restricts controller scalability [23]. As a result, the lack of scalability hinders the use of SDN to address interference in large DCNs.

The scalability of SDN-based datacenters could be improved by devolving the control to the data plane, such as proposed by DevoFlow [19] and Difane [24], but deployability is limited since they require switches with customized hardware. Another approach would be using a logically distributed controller, such as proposed

* Corresponding author.
E-mail addresses: daniel.stefani@inf.ufrgs.br, daniel.stefani@gmail.com (D.S. Marcon), fmmazzola@inf.ufrgs.br (F.M. Mazzola), marinho@inf.ufrgs.br (M.P. Barcellos).

[1] We use the terms “switches” and “forwarding devices” to refer to the same set of SDN-enabled network devices, that is, data plane devices that forward packets based on a set of flow rules.

http://dx.doi.org/10.1016/j.comnet.2017.08.008
1389-1286/

by Kandoo [25]. However, it does not scale for large DCNs where communications occur between virtual machines (VMs) connected by different top-of-rack (ToR) switches. This happens because the distributed set of controllers needs to maintain synchronized information (strong consistency) for the whole network. This is necessary in order to route traffic through less congested paths and to reserve resources for applications.

We rely on two key observations to address performance interference and scalability of SDN in DCNs: (*i*) providers do not need to control each flow individually, since they charge tenants based on the amount of resources consumed by applications [2]; and (*ii*) congestion control in the network is expected to be proportional to the tenant's payment (defined in their Service Level Agreements – SLAs) [12,13]. Therefore, we adopt a broader definition of flow, considering it at application-level,[3] and introduce Predictor, a scheme for large-scale datacenters. Predictor deals with the two aforementioned challenges (namely, performance interference and scalability of SDN/OpenFlow in DCNs) in the following manner.

Performance interference is addressed by employing two SDN-based algorithms (described in Section 5.3) to dynamically program the network, improving resource sharing. By doing so, both tenants and providers have benefits. Tenants achieve predictable network performance by receiving minimum bandwidth guarantees for their applications (using Algorithm 1). Providers, in turn, maintain high network utilization (due to work-conservation provided by Algorithm 2), essential to achieve economies of scale.

Scalability is improved in two ways. First, as we show through measurements (Section 3), reducing flow table size also decreases the time taken to install rules in flow tables (stored in Ternary Content-Addressable Memory – TCAM) of switches. In the proposed approach, flow table size is minimized by managing flows at application-level and by using wildcards (when possible). This setting allows providers to control traffic and gather statistics at application-level for each link and device in the network.

Second, we propose to proactively install rules for intra-application communication, guaranteeing bandwidth between VMs of the same application. By proactively installing rules at the moment applications are allocated, flow setup time is reduced (which is important especially for latency-sensitive flows). Inter-application rules, in turn, may be either proactively installed in switches (if tenants know other applications that their applications will communicate with [11] or if the provider employs some predictive technique [26,27]) or reactively installed according to demands. Proactively installing rules has both benefits and drawbacks: while flow setup time is eliminated, some flow table entries may take longer to expire (they might be removed only when their respective applications conclude and are deallocated). Our decision is motivated by the fact that intra-application traffic volume is expected to be the highest type of traffic [12].

In general, Predictor's strategy to address scalability of SDN/OpenFlow in large-scale datacenters presents a trade-off. The benefits are related to reducing flow table size and flow setup time in datacenter networks. Reducing flow table size enables (*i*) providers to use cheaper forwarding devices (i.e., with smaller flow tables); and (*ii*) forwarding devices to install rules in a shorter amount of time (as shown in Section 3.2). Reducing flow setup time greatly benefits latency-sensitive applications. The drawbacks are related to the time rules remain installed in forwarding devices and the ability to perform fine-grained load balancing. First, rules for intra-application communication (i.e., communication between VMs of the same application) are installed when applications are allocated and are removed when applications conclude their execution and are deallocated. Hence, some rules may remain installed longer than in other proposals. Second, since rules are installed at application-level, the ability to perform fine-grained load balancing in the network (e.g., for a flow or for a restricted set of flows) may be reduced. Note that Predictor can also install and manage rules at lower levels (for instance, by matching source and destination MAC and IP fields), since it uses the OpenFlow protocol. Nonetheless, given the amount of resources available in commodity switches and the number of active flows in the network, low-level rules need to be kept to a minimum.

Contributions. In comparison to our previous work [28], in this paper we present a substantially improved version of Predictor, in terms of both efficiency and resource usage. We highlight five main contributions. First, we run experiments to motivate Predictor and show that the operation of inserting rules at the TCAM takes more time and is more variable according to flow table occupancy. Thereby, the lower the number of rules in TCAMs, the better. Second, we extend Predictor to proactively provide inter-application communication guarantees (rather than only reactively providing it), which can further reduce flow setup time. Third, we develop improved versions of the allocation and work-conserving rate enforcement algorithms to provide better utilization of available resources (without adding significant complexity to the algorithms). More specifically, we improved (*i*) the allocation logic in Algorithm 1, so that resources can be better utilized without adding significant complexity; and (*ii*) rate allocation for VMs in Algorithm 2, so that all bandwidth available can be utilized if there are demands. In our previous paper [28], there could be occasions when some bandwidth would not be used even if there were demands. Fourth, we address the design of the control plane for Predictor, as it is an essential part of SDN to provide efficient and dynamic control of resources. Fifth, we conduct a more extensive evaluation, comparing Predictor with different modes of operation of DevoFlow [19] and considering several factors to analyze its benefits, overheads and technical feasibility. Predictor reduces flow table size up to 94%, offers low average flow setup time and presents low controller load, while providing minimum bandwidth guarantees for tenants and work-conserving sharing for providers.

The remainder of this paper is organized as follows. Section 2 discusses related work, and Section 3 examines the challenges of performance interference and scalability of SDN in DCNs. Section 4 provides an overview of Predictor, while Section 5 presents the details of the proposal (specifically regarding application requests, bandwidth guarantees, resource sharing and control plane design). Section 6 presents the evaluation, and Section 7 discusses generality and limitations. Finally, Section 8 concludes the paper with final remarks and perspectives for future work.

2. Related work

Researchers have proposed several schemes to address scalability in large-scale, SDN-based DCNs and performance interference among applications. Proposals related to Predictor can be divided into three classes: OpenFlow controllers (related to scalability in SDN-based DCNs), and deterministic and non-deterministic bandwidth guarantees (related to performance interference).

OpenFlow controllers. DevoFlow [19] and DIFANE [24] propose to devolve control to the data plane. The first one introduces new mechanisms to make routing decisions at forwarding devices for small flows and to detect large flows (to request controller assistance to route them), while the second keeps all packets in the data plane. These schemes, however, require more complex, customized hardware at forwarding devices. Kandoo [25] provides a logically distributed control plane for large networks. Nonetheless,

[2] Without loss of generality, we assume one application per tenant.

[3] An application is represented by a set of VMs that consume computing and network resources (see Section 5.1 for more details).

it does not scale when most communications occur between VMs located in different racks. It is so because the distributed set of controller instances needs to maintain synchronized information (strong consistency) for the whole DCN. This is necessary in order to route traffic through less congested paths and to reserve resources for applications. Lastly, Hedera [29] and Mahout [30] require precise statistics from the network with at most 500 ms of interval between them to efficiently route large flows and utilize available resources [31]. However, obtaining statistics with such frequency is impractical in large DCNs [19]. In contrast to these proposals, Predictor requires neither customized hardware nor precise statistics from the network with at most 500 ms of interval between them. Furthermore, Predictor scales to the high dynamic traffic patterns of DCNs, independently of the source and destination of communications.

Deterministic bandwidth guarantees. Silo [13], CloudMirror [1] and Oktopus [32] provide strict bandwidth guarantees for tenants by isolating applications in virtual networks. Unlike Predictor, these approaches (*i*) do not provide complete work-conservation (which may result in underutilization of resources); and (*ii*) address only intra-application communication.

Proteus [26], in contrast to Predictor, needs to profile temporal network demands of applications before allocation, since it requires such information for allocating applications in the infrastructure. Such requirement may be unrealistic for some types of applications (e.g., ones that consume an excessive amount of resources or that have requirements which depend on external factors). EyeQ [33] attempts to provide bandwidth guarantees with work-conservation. However, different from Predictor, it cannot provide guarantees upon core-link congestion [34] (and congestion is not rare in DCNs [8]). Finally, Hadrian [12] introduces a strategy that considers inter-application communication, but it (*i*) needs a larger, custom packet header (hindering its deployment); (*ii*) does not ensure complete work-conservation, as the maximum allowed bandwidth is limited according to the tenant's payment; and (*iii*) requires switches to dynamically perform rate calculation (and enforce such rate) for each flow in the network. Unlike Hadrian, Predictor provides complete work-conservation and performs rate calculation at server hypervisors (freeing switches from this burden).

Li et al. [35] propose a model to ensure bandwidth guarantees, to minimize network cost and to avoid congestion on low-cost links for inter-datacenter traffic. Since Predictor focuses on intra-datacenter traffic, the proposal is mostly orthogonal to ours. As a matter of fact, Predictor could be used inside datacenters to provide bandwidth guarantees with work-conservation while Li et al.'s proposal could be employed on the set of inter-datacenter links belonging to the path used for communications between VMs allocated inside different datacenters.

Non-deterministic bandwidth guarantees. Seawall [10] and NetShare [36] share the network proportionally according to weights assigned to VMs and tenants. Thus, they allocate bandwidth at flow- and link-level. FairCloud [37] explores the trade-off among network proportionality, minimum guarantees and high utilization. These proposals, however, may result in substantial management overhead (since bandwidth consumed by each flow at each link is dynamically calculated according to the flow weight, and large DCNs can have millions of flows per second [22]). Predictor, in contrast, reduces management overhead by considering flows at application-level.

Varys [3], Baraat [2] and PIAS [14] seek to improve application performance by minimizing average and tail flow completion time (FCT). Karuna [15] minimizes FCT for non-deadline flows while ensuring that deadline flows just meet their deadlines. Unlike Predictor, none of them provide minimum bandwidth guarantees.

QJUMP [11] explores the trade-off between throughput and latency. While being able to provide low latency for selected flows, it may significantly reduce network utilization (as opposed to Predictor, which can achieve high network utilization).

Finally, Active Window Management (AWM) [38], AC/DC TCP [8] and vCC [7] address congestion control. Specifically, AWM seeks to minimize network delay and to keep goodput close to the network capacity, while AC/DC TCP [8] and vCC [7] address unfairness by performing congestion control at the virtual switch in the hypervisor. Consequently, these three proposals are mostly orthogonal to Predictor, since a DCN needs both congestion control (e.g., AWM, AC/DC TCP or vCC) and bandwidth allocation (e.g., Predictor).

3. Motivation and research challenges

In this section, we review performance interference (Section 3.1) and discuss the challenges of using SDN in large-scale DCNs to build a solution for interference (Section 3.2). In the context of SDN, we adopt an OpenFlow [39] view, since it is the most accepted SDN implementation by Academia and Industry. Through OpenFlow switch measurements, we quantify flow setup time and show it can hinder scalability of SDN as a solution to the performance interference problem.

3.1. Datacenter network sharing

Several recent measurement studies [11,13,32,40–43] concluded that, due to performance interference, the network throughput achieved by VMs can vary by a factor of five or more. For instance, Grosvenor et al. [11] show that variability can worsen tail performance (the worst performance in a set of executions) by 50× for clock synchronization (PTPd) and 85× for key-value stores (Memcached). As the computation typically depends on the data received from the network [26] and the network is agnostic to application-level requirements [3], such variability often results in poor and unpredictable application performance [44]. In this situation, tenants end up spending more money.

Performance variability is usually associated with two factors: type of traffic and congestion control. The type of traffic in DCNs is remarkably different from other networks [5]. Furthermore, the heterogeneous set of applications generates flows that are sensitive to either latency or throughput [30]; throughput-intensive flows are larger, creating contention in some links, which results in packets from latency-sensitive flows being discarded (adding significant latency) [9,45]. TCP congestion control (used in such networks), in turn, cannot ensure performance isolation among applications [8]; it only guarantees fairness among flows. Judd and Stanley [40] show through measurements that many TCP design assumptions do not hold in datacenter networks, leading to inadequate performance. While TCP can provide high utilization, it does so very inefficiently. They conclude that the overall median throughput of the network is low and that there is a large variation among flow throughput.

Popa et al. [37] examine two main requirements for network sharing: (*i*) bandwidth guarantees for tenants and their applications; and (*ii*) work-conserving sharing to achieve high network utilization for providers. In particular, these two requirements present a trade-off: strict bandwidth guarantees may reduce utilization, since applications have variable network demands over time [26]; and a work-conserving approach means that, if there is residual bandwidth, applications should use it as needed (even if the available bandwidth belongs to the guarantees of another application) [10].

In this context, SDN/OpenFlow can enable dynamic, fine-grained network management in order to develop a robust strategy to explore this trade-off and achieve predictable network performance with bandwidth guarantees and work-conserving sharing.

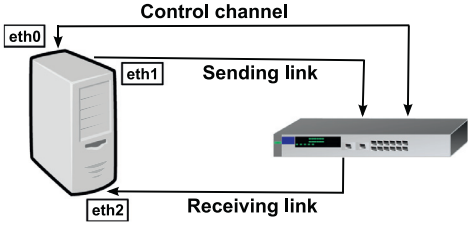


Fig. 1. Measurement setup.

3.2. Scalability challenges of SDN/openflow in DCNs

SDN-based networks involve the control plane more frequently than traditional networking [19]. In the context of large-scale DCNs, this aspect leads to two scalability issues: flow setup time (the time taken to install new flow rules in forwarding devices) and flow table size in switches.

Flow setup time. It may add impractical delay for flows, especially for latency-sensitive ones [13] (as adding even 1 ms of latency to these flows is intolerable [46]). As SDN relies on the communication between network devices (data plane) and a logically centralized controller (control plane), it increases (*i*) control plane load and (*ii*) latency (sources for augmented delay). Control plane load is increased because a typical ToR switch will have to request rules to the controller for approximately more than 1,500 new flows per second [47] and the controller is expected to process and reply to all requests in, at most, a few milliseconds. Consequently, this may end up making both the communication with the controller and the controller itself bottlenecks. Latency is augmented because new flows are delayed at least two RTTs (i.e., communication between the ASIC and the management CPU and between that CPU and the controller) [19], so that the controller can install the appropriate rules at forwarding devices.

Experiments to measure flow setup time. We evaluated the time taken to perform the operation of inserting rules at a switch's TCAM. Our measurement setup (shown in Fig. 1) consists of one host with three 1 Gbps interfaces connected to an OpenFlow switch (Centec v350): eth0 interface is connected to the control port and eth1 and eth2 are connected to data ports on the switch. The switch uses OpenFlow 1.3 and has a TCAM that stores at most 2,000 rules. The host runs OpenFlow controller Ryu, which listens for control packets on eth0.

The experiment works as follows. The switch begins with a given number of rules installed in the TCAM (which represents its table occupancy). The host runs a packet generator to send a single UDP flow on eth1. This flow generates a table-miss event in the switch (i.e., the switch does not have an appropriate rule to handle the flow sent by the packet generator). Consequently, the switch sends a packet_in message to the controller. Upon receiving the packet_in, the controller processes the request and sends back a flow_mod message with the appropriate rule to be installed in the switch TCAM to handle the flow. Once the switch installs the rule, it forwards the matching packets to the link connected on the host eth2 interface. Like He et al. [48], the latency of the operation is calculated as follows: (*i*) timestamp1 is recorded when the controller sends the flow_mod to the switch on eth0; and (*ii*) timestamp2 is recorded when the first packet of the flow arrives on the host eth2 interface. Since the round-trip time (RTT) between the switch and host is negligible in our experiments, [4] the latency is calculated by subtracting timestamp1 from timestamp2.

[4] While the RTT is negligible in our experiments (as the switch is directly connected to the controller), it may not be the case in large-scale datacenters with hundreds of switches.

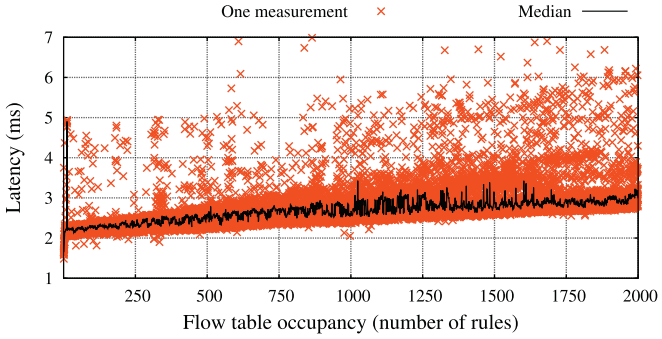


Fig. 2. Latency of inserting new rules according to flow table occupancy.

Fig. 2 shows the latency of inserting new rules at the TCAM (y-axis) according to the number of rules already installed in the table (x-axis). The experiment was executed in a baseline SDN setting (without employing our proposal, Predictor); it was repeated 10 times; and each point in the plot represents one measured value of one repetition and the line depicts the median value. Results show that median latency and variability increase according to flow table occupancy. These results are in line with previous measurements in the literature, such as He et al. [48]. Since adding even 1ms may be intolerable for some applications (e.g., latency-sensitive ones) [46], reduced flow table occupancy is highly desirable in DCNs because of flow setup time.

Flow table size. Flow tables are a restricted resource in commodity switches, as TCAMs are typically expensive and power-hungry [20,21,49]. Such devices usually have a limited number of entries available for OpenFlow rules, which may not be enough when considering that large-scale datacenter networks have an elevated number of active flows per second [22].

Therefore, the design of Predictor takes both flow setup time and flow table size in switches into account. More specifically, Predictor proactively installs rules for intra-application communication at allocation time (thereby eliminating the latency of flow setup for most traffic in DCNs) and considers flows at *application-level* (reducing the number of flow table entries and, consequently, the time taken to install new rules in forwarding devices). We detail Predictor and justify the decisions in the next sections.

4. Predictor overview

We first present an overview of Predictor, including its components and the interactions between them in order to address the challenge of performance interference in large-scale, SDN-based DCNs.

Predictor is designed taking four requirements into consideration: (*i*) scalability, (*ii*) resiliency, (*iii*) predictable and guaranteed network performance, and (*iv*) high network utilization. First, any design for network sharing must scale to hundreds of thousands of VMs and deal with heterogeneous workloads of applications (typically with bursty traffic). Second, it needs to be resilient to churn both at flow-level (because of the rate of new flows/s [22]) and at application-level (given the rates of application allocation/deallocation observed in datacenters [10]). Third, it needs to provide predictable and guaranteed network performance, allowing applications to maintain a base-level of performance even when the network is congested. Finally, any design should achieve high network utilization, so that spare bandwidth can be used by applications with more demands than their guarantees.

Predictor is designed to fulfill the above requirements. While providers can reduce operational costs and achieve economies of scale, tenants can run their applications predictably (possibly faster, reducing costs). Fig. 3 shows an overview of Predictor, which

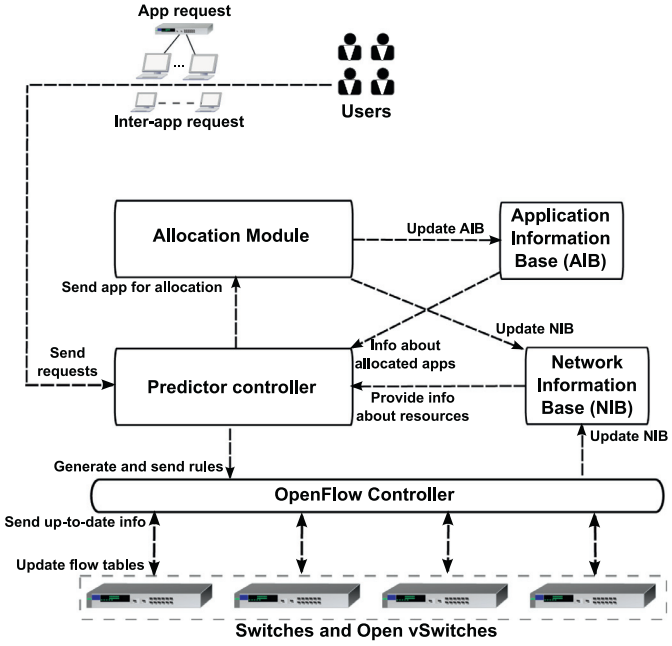


Fig. 3. Predictor overview.

is composed of five components: Predictor controller, allocation module, application information base (AIB), network information base (NIB) and OpenFlow controller. They are discussed next.

Predictor controller. It receives requests from tenants. A request can be either an application to be allocated (whose resources to be used are determined by the allocation module) or a solicitation for inter-application bandwidth guarantees (detailed in Sections 5.1 and 5.3). In case of an incoming application, it sends the request to the allocation module. Once the allocation is completed (or if the request is for inter-application communication), the Predictor controller generates and sends appropriate flow rules to the OpenFlow controller. The OpenFlow controller, then, updates the tables (of forwarding devices) that need to be modified.

Note that the controller installs rules to identify flows at application-level. This granularity increases neither flow setup time nor overhead in forwarding devices. Rules for intra-application communication are installed when the respective applications are allocated and removed when those applications conclude and are deallocated. Rules for inter-application communication are installed when VMs need to exchange data with VMs of other applications. This behavior, in fact, reduces overhead compared to a traditional SDN/OpenFlow setting, where rules must be installed for each new flow in the network. This happens because large-scale datacenters typically have millions of active flows per second. Furthermore, it hinders neither network controllability for providers nor network sharing among tenants and their applications because, as discussed in Section 1, (*a*) providers charge tenants based on the amount of resources consumed by applications; and (*b*) congestion control in the network is expected to be performed at application-level [12]. We provide more details in Sections 6 and 7.

When necessary, Predictor can also take advantage of flow management at lower levels (for instance, by matching other fields in the OpenFlow protocol. Nonetheless, given the amount of resources available in commodity switches and the number of flows that come and go in a small period of time, low-level rules need to be kept to a minimum.

Allocation Module. This component is responsible for allocating incoming applications in the datacenter infrastructure, according to available resources. It receives requests from the Predictor controller, determines the set of resources to be allocated for each new request and updates the AIB and NIB. We detail the allocation logic in Section 5.3.1.

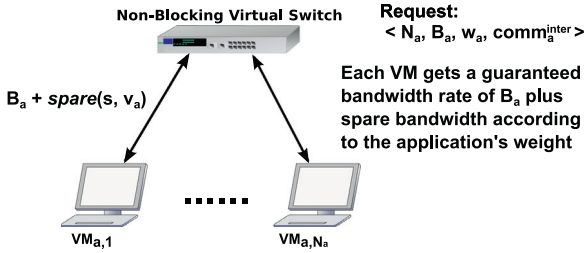


Fig. 4. Virtual network topology of a given application.

Application Information Base (AIB). It keeps detailed information regarding each allocated application, including its identifier (ID), VM-to-server mapping, IP addresses, bandwidth guarantees, network weight (for work-conserving sharing), links being used and other applications it communicates with. It provides information for the Predictor controller to compute flow rules that need to be installed in switches.

Network Information Base (NIB). It is composed of a database of resources, including hosts, switches, links and their capabilities (such as link capacity and latency). In general, it keeps information about computing and network state, received from the OpenFlow controller (current state) and the allocation module (resources used for newly allocated applications). The Predictor controller uses information stored in the NIB to map logical actions (e.g., intra- or inter-application communication) into the physical network. While the AIB maintains information at application granularity, the NIB keeps information at network layer. The design of the NIB was inspired by Onix [17] and PANE [50].

OpenFlow controller. It is responsible for communication to/from forwarding devices and Open vSwitches [51] in hypervisors, in order to update network state and get information from the network (e.g., congested links and failed resources). It receives information from the Predictor controller to modify flow tables in forwarding devices and updates the NIB upon getting information from the network.

Each component is installed in one or multiple dedicated servers, as follows. The Predictor controller is a logically centralized system (i.e., multiple controller instances, one instance per server), as discussed in Section 5.4. The allocation module is installed in a server that implements the allocation logic explained in Section 5.3. Like the Predictor controller, AIB and NIB are implemented in a logically centralized system. Finally, the OpenFlow controller can be implemented in the same set of servers as the Predictor controller (as it interacts frequently with the Predictor controller) or in different server(s).

5. System description

Section 5.1 presents application requests. Section 5.2 examines how bandwidth guarantees are provided for applications. Section 5.3 describes the mechanisms employed for resource sharing (namely, resource allocation and work-conserving rate enforcement). Finally, Section 5.4 details the control plane design.

5.1. Application requests

Tenants request applications to be allocated in the cloud datacenter using the hose model (similarly to past proposals [12,13,23,26,32]). The hose model allows to capture the semantics of guarantees being offered, as shown in Fig. 4. In this model, all VMs of an application are connected to a non-blocking virtual

switch through dedicated bidirectional links. This represents how tenants perceive their applications (i.e., as if they were the only ones executing in the cloud). Each application a is represented by its resource demands and network weight, or more formally, $\langle N_a, B_a, w_a, comm_a^{inter} \rangle$. Its terms are: $N_a \in \mathbb{N}^*$ specifies the number of VMs; $B_a \in \mathbb{R}^+$ represents the bandwidth guarantees required by each VM; $w_a \in [0, 1]$ indicates the network weight; and $comm_a^{inter}$ is an optional field that contains information about inter-application communication for application a. This specification is taken as input in the resource allocation algorithm (Algorithm 1) to map the requested resources for the application in the datacenter infrastructure. This is similar to Virtual Network Embedding [52].

The network weight enables residual bandwidth (unallocated or reserved bandwidth for an application and not currently being used) to be proportionally shared among applications with more demands than their guarantees (work-conservation). Therefore, the total amount of bandwidth available for each VM of application a at a given period of time, following the hose model, is denoted by $B_a + spare(s, v_a)$, where $spare(s, v_a)$ identifies the share of spare bandwidth assigned to VM v of application a (i.e., v_a) located at server s:

$$\texttt{spare}(s, v_a) = \frac{w_a}{\sum_{v \uparrow V_s | v \in V_s} w_v} * \texttt{SpareCapacity} \quad (1)$$

where V_s denotes the set of all co-resident VMs (i.e., VMs placed at server s), $v \uparrow V_s | v \in V_s$ represents the subset of VMs at server s that need to use more bandwidth than their guarantees and `SpareCapacity` indicates the residual capacity of the link that connects server s to the ToR switch.

The term $comm_a^{inter}$ is optional and allows tenants to proactively request guarantees for inter-application communication (since it would be infeasible to provide all-to-all communication guarantees between VMs in large-scale datacenters [12]). This field represents a set composed of elements in the form of $\langle \texttt{srcVM}_\texttt{a}, \texttt{dstVMs}, \texttt{begTime}, \texttt{endTime}, \texttt{reqRate} \rangle$, where: $\texttt{srcVM}_\texttt{a}$ denotes the source VM of application a; `dstVMs` is the set of destination VMs (i.e., unicast or multicast communication from the source VM to each destination VM); `begTime` and `endTime` represent, respectively, the time that the communication starts and ends; and `reqRate` indicates the total amount of bandwidth per second needed by flows belonging to traffic from this (these) communication(s). This information will be used by Algorithm 1 to reserve bandwidth for the specified inter-application communications when allocating the application.

By providing optional specification of inter-application communication, Predictor allows requests from tenants with or without knowledge of application communication patterns and desired resources. Application traffic patterns are often known [3,11] or can be estimated by employing the techniques described by Lee et al. [1], Xie et al. [26] and LaCurts et al. [27]. Note that, even if tenants do not proactively request resources for communication with other applications/services (i.e., they do not use $comm_a^{inter}$), their applications will still be allowed to reactively receive guarantees for communication with others (as detailed in Section 5.2).

In line with past proposals [12,13,26,32], two assumptions are made. First, we abstract away non-network resources and consider all VMs with the same amount of CPU, memory and storage. Second, we consider that all VMs of a given application receive the same bandwidth guarantees (B_a).

5.2. Bandwidth guarantees

Predictor provides bandwidth guarantees for both intra- and inter-application communication. We discuss each one next.

Intra-application network guarantees. Typically, this type of communication represents most of the traffic in DCNs [12]. Thus,

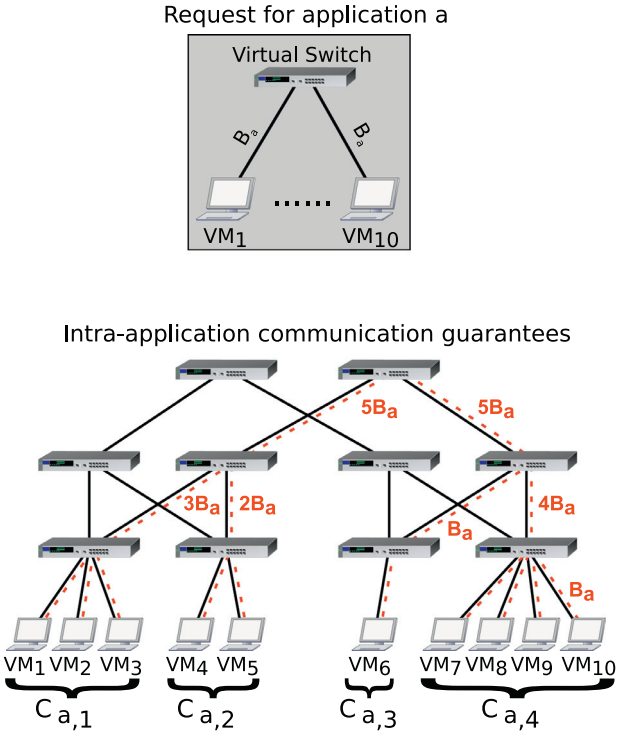


Fig. 5. Example of intra-application bandwidth guarantees.

Predictor allocates and ensures bandwidth guarantees at application allocation time[5] (i.e., the moment the application is allocated in the datacenter infrastructure) by proactively installing flow rules and rate-limiters in the network through OpenFlow.

Each VM of a given application a is assigned a bidirectional rate of B_a (as detailed in Section 5.1). Limiting the communication between VMs located in the same server or in the same rack is straightforward, since it can be done locally by the Open vSwitch at each hypervisor.

In contrast, for inter-rack communication, bandwidth must be guaranteed throughout the network, along the path used for such communication. Predictor provides guarantees for this traffic by employing the concept of *VM clusters*. [6] To illustrate this concept, Fig. 5 shows a simplified scenario where a given application a has four clusters: $c_{a,1}$, $c_{a,2}$, $c_{a,3}$ and $c_{a,4}$. Since each VM of a cannot send or receive data at a rate higher than B_a, traffic between a pair of clusters $c_{a,x}$ and $c_{a,y}$ is limited by the smallest cluster: $rate_{c_{a,x},c_{a,y}} = \min(|c_{a,x}|, |c_{a,y}|) * B_a$, where $rate_{c_{a,x},c_{a,y}}$ represents the calculated bandwidth for communication between clusters $c_{a,x}$ and $c_{a,y}$ (for $x, y \in \{1, 2, 3, 4\}$ and $x \neq y$), and $|c_{a,i}|$ denotes the number of VMs in cluster i of application a. In this case, $rate_{c_{a,x},c_{a,y}}$ is guaranteed along the path used for communication between these two clusters by rules and rate-limiters configured in forwarding devices through OpenFlow.

We apply this strategy at each level up the topology (reserving the minimum rate required for the communication among clusters). In general, the bandwidth required by one VM cluster to communicate with all other clusters of the same application is

[5] While Predictor may overprovision bandwidth at the moment applications are allocated, it does not waste bandwidth because of its work-conserving strategy (explained in Section 5.3.2). Without overprovisioning bandwidth at first, it would not be feasible to provide bandwidth guarantees for applications (as DCNs are typically oversubscribed).

[6] A VM cluster is a set of VMs of the same application located in the same rack.

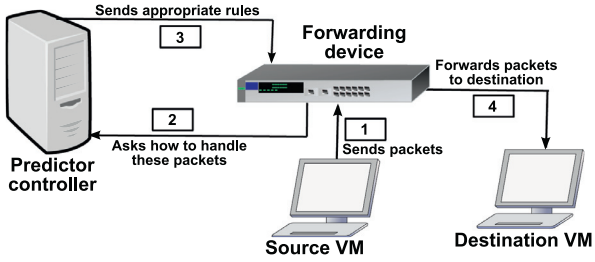


Fig. 6. Detailed process of reacting to new flows in the network for inter-application communication.

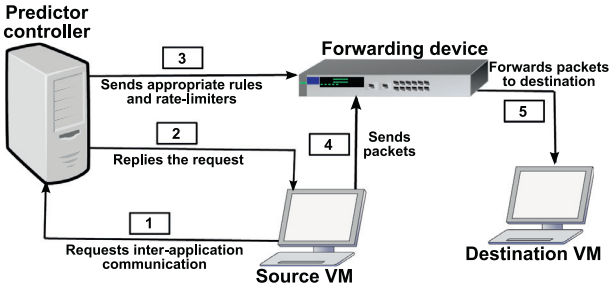


Fig. 7. Detailed process of receiving communication requests from applications for inter-application communication.

given by the following expression:

$$rate_{c_{a,x}} = \min\left(|c_{a,x}| * B_a, \sum_{c \in C_a, c \neq c_{a,x}} |c| * B_a\right) \qquad \forall c_{a,x} \in C_a \qquad (2)$$

where $rate_{c_{a,x}}$ denotes the bandwidth required by cluster x to communicate with other clusters associated with application a and C_a indicates the set of all clusters of application a.

Inter-application communication. Applications in datacenters may exhibit complex communication patterns. However, providing them with static hose guarantees does not scale for DCNs [12], since bandwidth guarantees would have to be enforced between all pairs of VMs. Furthermore, tenants may not know in advance all applications/services that their applications will communicate with.

Predictor dynamically sets up guarantees for inter-application communication according to the needs of applications and residual bandwidth in the network. In case guarantees were not requested using the field $comm_a^{inter}$ (as described in Section 5.1), the Predictor controller provides two ways of establishing guarantees for communication between VMs of distinct applications and services: reacting to new flows in the network and receiving communication requests from applications.

Reacting to new flows in the network. Fig. 6 shows the process of reacting to new flows in the network. When a VM (source VM) needs to exchange data with one or more VMs of another application, it can simply send packets to those VMs (first step in the figure). The hypervisor (through its Open vSwitch, the forwarding device) of the server hosting the source VM receives such packets and, since they do not match any rule, sends a request to the controller asking how to handle packets from this new flow (second step in the figure). The Predictor controller, then, determines the rules needed by the new flow and installs the set of rules along the appropriate path in the network (third step in the figure). The forwarding device, then, forwards the packets to the destination VM along the path determined by the controller (fourth step in the figure). In this case, the communication will have no guarantees (i.e., it will be best-effort) and the maximum allowed rate will be up to the guarantees of the VM (B_a), except when there is available (unused) bandwidth in the network (i.e., Predictor provides work-conservation). The allowed rate for the VM is determined by Algorithm 2.

Receiving communication requests from applications. Fig. 7 shows the process of receiving communication requests from applications. Prior to initiating the communication with destination VM(s) belonging to other application(s), the source VM sends a request to the Predictor controller for communication with VMs from other applications (first step in the figure). This request is composed of the set of destination VMs, the bandwidth needed and the expected amount of time the communication will last. Upon receiving the request, the Predictor controller verifies residual resources in the network, sends a reply to the source VM (second step in the figure) and, in case there are enough available resources, generates and installs the appropriate set of rules and rate-limiters for this communication (third step in the figure). The source VM, then, initiates the communication by sending packets to the destination VMs (fourth step in the figure), and the forwarding devices (with appropriate rules and rate-limiters already installed by the Predictor controller) forward these packets to the destination VMs along the path(s) determined by the controller (fifth step in the figure). In this case, the bandwidth rate for the communication will be guaranteed. This approach is similar to providing an API for applications to request network resources, like PANE [50].

5.3. Resource sharing

In this section, we discuss how resources are shared among applications. In particular, we first examine the process of resource allocation and, then, present the logic behind the work-conserving mechanism employed by Predictor.

5.3.1. Resource allocation

The allocation process is responsible for performing admission control and mapping application requests in the datacenter infrastructure. An allocation can only be made if there are enough computing and network resources available [53]. That is, VMs must only be mapped to servers with available resources, and there must be enough residual bandwidth for communication between VMs (as specified in the request). For simplicity, we follow related work [13,26,32] and discuss Predictor and its allocation component in the context of traditional tree-based topologies implemented in current datacenters.

We design a location-aware heuristic to efficiently allocate tenant applications in the infrastructure. The key principle is minimizing bandwidth for intra-application communication (thus allocating VMs of the same application as close as possible to each other), since this type of communication generates most of the traffic in the network (as discussed before) and DCNs typically have scarce resources [26].

Algorithm 1 employs an online allocation strategy, allocating one application at a time, as they arrive. It receives as input the physical infrastructure P (composed of servers, racks, switches and links) and the incoming application request a (formally defined in Section 5.1 as $\langle N_a, B_a, w_a, comm_a^{inter}\rangle$), and works as follows. First, it searches for the best placement in the infrastructure for the incoming application via dynamic programming (lines 1 – 12). To this end, $N_s^p(l-1)$ represents the set of neighbors (directly connected switches) of switch s at level $l-1$. Furthermore, three auxiliary data structures are defined and dynamically initialized for each request: (i) set R^a stores subgraphs with enough computing resources for application a; (ii) V_s^a stores the total number of VMs of application a the s-rooted subgraph can hold; and (iii) C_s^a stores the number of VM clusters that can be formed in subgraph s. The algorithm traverses the topology starting at rack level (level 1), up

Algorithm 1: Location-aware algorithm.

Input : Physical infrastructure P (composed of servers, racks, switches and links), Application a=$\langle N_a, B_a, w_a, comm_a^{inter}\rangle$
Output: Success/Failure code *allocated*

```
   // Search for the best placement in the infrastructure
1  R^a ← ∅;
2  foreach level l of P do
3  |  if l == 1 then // Top-of-Rack switches
4  |  |  foreach ToR r do
5  |  |  |  V_r^a ← num. available VMs in the rack;
6  |  |  |  C_r^a ← 1;
7  |  |  |  if V_r^a ≥ N^a then R^a ← R^a ∪ {r};
8  |  else // Aggregation and core switches
9  |  |  foreach Switch s at level l do
10 |  |  |  V_s^a ← Σ_{w∈{N_s^P(l−1)}} V_w^a;
11 |  |  |  C_s^a ← Σ_{w∈{N_s^P(l−1)}} C_w^a;
12 |  |  |  if V_s^a ≥ N^a then R^a ← R^a ∪ {s};
   // Proceed to the allocation
13 allocated ← failure code;
14 while Application a not allocated and R_a not empty do
15 |  r ← Select subgraph from R^a;
16 |  R^a ← R^a \ {r};
   |  // VM placement
17 |  Allocate VMs of application a at r;
   |  // Bandwidth allocation
18 |  foreach Level l from 0 to Height(r) do
19 |  |  Allocate bandwidth at l according to Section 5.2 and Equation 2;
20 |  foreach Inter-application communication c ∈ comm_a^inter do
21 |  |  Allocate bandwidth for inter-application communication c specified at allocation time (as defined in Section 5.1);
22 |  if Application was successfully allocated at r then
23 |  |  allocated ← success code
24 |  else allocated ← failure code;
25 return allocated;
```

to the core, and determines subgraphs with enough available resources to allocate the incoming request.

After verifying the physical infrastructure and determining possible placements, the algorithm starts the allocation phase (lines 13 – 24). First, it selects one subgraph r at a time from the set R^a to allocate the application (line 15). The selection of a candidate subgraph takes into account the number of VM clusters. Therefore, the selected subgraph is the one with the minimum number of VM clusters (i.e., the best candidate), so that VMs of the same application are allocated close to each other, reducing the amount of bandwidth needed for communication between them (as the network often represents the bottleneck when compared to computing resources [54]).

When a subgraph is selected, the algorithm allocates the application with a coordinated node (VM-to-server, in line 17) and link (bandwidth reservation, in lines 18 – 21) mapping, similarly to the virtual network embedding problem [55]. In particular, bandwidth for intra-application communication (lines 18 – 19) is allocated through a bottom-up strategy, as follows. First, it is reserved at servers (level 0). Then, it is reserved, in order, for each subsequent level of the topology, according to the bandwidth needed by communication between VMs from distinct racks that belong to the same application (as explained in Section 5.2 and in Eq. 2, and exemplified in Fig. 5). After that, bandwidth for inter-application communication (that was specified at allocation time in field $comm_a^{inter}$) is allocated in lines 20 – 21 (recall that $comm_a^{inter}$ was defined in Section 5.1).

Finally, the algorithm returns a success code if application a was allocated or a failure code otherwise (line 25). Applications that could not be allocated upon arrival are discarded, similarly to Amazon EC2 [56].

5.3.2. Work-Conserving rate enforcement

Predictor provides bandwidth guarantees with work-conserving sharing. This is because only enforcing guarantees through static provisioning leads to underutilization and fragmentation [23], while offering work-conserving sharing only does not provide strict guarantees for tenants [12]. Therefore, in addition to ensuring a base-level of guaranteed rate, Predictor proportionally shares available bandwidth among applications with more demands than their guarantees (work-conservation), as defined in Eq. (1).

We design an algorithm to periodically set the allowed rate for each co-resident VM on a server. In order to provide smooth interaction with TCP, we follow ElasticSwitch [23] and execute the work-conserving algorithm between periods of time one order of magnitude larger than the network round-trip time (RTT), e.g., 10 ms instead of 1 ms.

Algorithm 2 aims at enabling smooth response to bursty traffic (since traffic in DCNs may be highly variable over short periods of time [4,9]). It receives as input the list of VMs (V_s) hosted on server s, their current demands (which are determined by monitoring VM socket buffers, similarly to Mahout [30]), their bandwidth guarantees and their network weight (specified in the application request and defined in Section 5.1).

Algorithm 2: Work-conserving rate allocation.

Input : Set of VMs V_s allocated on server s, Current demands of VMs *demand*, Bandwidth guarantees B for each VM, Network weight w for each VM
Output: Rate *nRate* for each co-resident VM

```
1  foreach v ∈ V_s do
2  |  if v ↓ V_s then nRate[v] ← demand[v];
3  |  else nRate[v] ← B[v];
4  residual ← LinkCapacity − (Σ_{v↑V_s} B[v] + Σ_{v↓V_s} demand[v]);
5  hungryVMs ← v ↑ V_s | v ∈ V_s;
6  while residual > 0 and hungryVMs not empty do
7  |  foreach v ∈ hungryVMs do
8  |  |  nRate[v] ←
   |  |  nRate[v] + min(demand[v] − nRate[v], (w[v] / Σ_{u↑V_s} w[u]) × residual));
9  |  |  if nRate[v] == demand[v] then
10 |  |  |  hungryVMs ← hungryVMs \ {v};
11 return nRate;
```

First, the rate for each VM is calculated based on their demands and the guaranteed bandwidth $B[v]$ (lines 1 – 3). In case the demand of a VM is equal or lower than its bandwidth guarantees (represented by $v{\downarrow}V_s | v \in V_s$), the rate is assigned and enforced (line 2), so that the exact amount of bandwidth needed for communication is used (wasting no network resources). In contrast, the guarantee $B[v]$ is initially assigned to $nRate[v]$ for each VM $v \in V_s$ with higher demands than its guarantees (represented by $v{\uparrow}V_s | v \in V_s$), in line 3. Then, the algorithm calculates the residual bandwidth of the link connecting the server to the ToR switch (line 4). The residual bandwidth is calculated by subtracting from the link capacity the guarantees of VMs with higher demands than their guarantees and the rate of VMs with equal or lower demands than their guarantees.

The last step establishes the bandwidth for VMs with higher demands than their guarantees (line 5 – 10). The rate (line 8) is determined by adding $nRate[v]$ (initialized in line 3) and the minimum bandwidth between (*i*) the difference of the current demand ($demand[v]$) and the rate ($nRate[v]$); and (*ii*) the proportional share of residual bandwidth the VM would be able to receive according to its weight $w[v]$. Note that there is a "while" loop (lines 6 – 10) to guarantee that all residual bandwidth is used or all demands are satisfied. If this loop were not used, there could be occasions when there would be unsatisfied demands even though some bandwidth would be available.

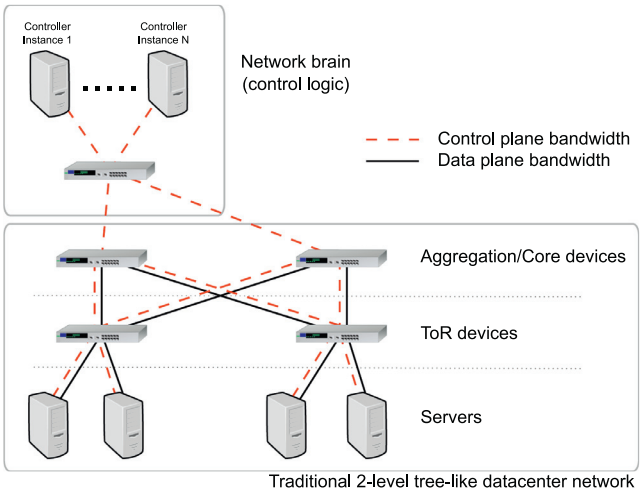


Fig. 8. Design of Predictor's control plane.

With this algorithm, Predictor guarantees that VMs will not receive more bandwidth than they need (which would waste network resources) and bandwidth will be fully utilized if there are demands (work-conservation). Moreover, the algorithm has fast convergence on bandwidth allocation and can adapt to the significant variable communication demands of cloud applications. Therefore, if there is available bandwidth, VMs can send traffic bursts at a higher rate (unlike Silo [13], Predictor allows traffic bursts with complete work-conservation).

In summary, if the demand of a VM exceeds its guaranteed rate, data can be sent and received at least at the guaranteed rate. Otherwise, if it does not, the unutilized bandwidth will be shared among co-resident VMs whose traffic demands exceed their guarantees. We provide an extensive evaluation in Section 6 to verify the benefits of the algorithm.

5.4. Control plane design

The control plane design of the network is an essential part of software-defined networks, as disconnection between the control and data planes may lead to severe packet loss and performance degradation (forwarding devices can only operate correctly while connected to a controller) [57,58]. Berde et al. [59] define four requirements for the control plane: (*i*) high availability (usually five nines [60]); (*ii*) global network state, as the control plane must be aware of the entire network state to provide guarantees for tenants and their applications; (*iii*) high throughput, to guarantee performance in terms of satisfying requests even at periods of high demands; and (*iv*) low latency, so that end-to-end latency for control plane communication (i.e., updating network state in response to events) is small.

Based on these requirements, Fig. 8 shows the control plane design for Predictor. In this figure, we show, as a basic example, a typical 2-layer tree-like topology with decoupled control and data planes. We can see two major aspects: (*i*) the placement of controller instances (control plane logic) as a cluster in one location of the network (connected to all core switches); and (*ii*) the separation between resources for both planes (represented by different line styles and colors for link bandwidth), indicating out-of-band control plane communication. We discuss them next.

Cluster of controller instances. Following WL2 [61], the control plane logic is composed of a cluster of controller instances. There are two reasons for this. First and most important, Predictor needs strong consistency among the state of its controllers to provide network guarantees for tenants and their applications. If instances were placed at different locations of the topology, the amount of synchronization traffic would be unaffordable, since DCNs typically have highly dynamic traffic patterns with variable demands [4,5,9]. Moreover, DCNs are typically oversubscribed with scarce bandwidth [26].

Second, the control plane is expected to scale-out (periodically grow or shrink the number of active controller instances) according to its load, needed for high availability and throughput. Since DCNs usually count with multiple paths [45], one controller location is often sufficient to meet existing requirements [62]. Furthermore, if controllers were placed at several locations, a controller placement algorithm (e.g., Survivor [57]) would have to be executed each time the number of instances were adjusted, which would delay the response to data plane requests (as this is a NP-Hard problem [62]).

Out-of-band control. Predictor uses out-of-band control to manage the network. As the network load may change significantly over small periods of time [22] and some links may get congested [9] (due to the high oversubscription factor [63]), the control and data planes must be kept isolated from one another, so that traffic from one plane does not interfere [7] with the other. In other words, control plane traffic should not be impacted by rapid changes in data plane traffic patterns (e.g., bursty traffic). Using out-of-band control, some bandwidth of each link shared with the data plane (or all bandwidth from links dedicated to control functions) is reserved for the control plane (represented in Fig. 8 as red dotted lines). In the next section, we show how the amount of bandwidth reserved for the control plane affects efficiency of Predictor.

6. Evaluation

Below, we evaluate the benefits and overheads of Predictor. We focus on showing that Predictor (*i*) can scale to large SDN-based DCNs; (*ii*) provides both predictable network performance (with bandwidth guarantees) and work-conserving sharing; and (*iii*) outperforms existing schemes for DCNs (the baseline SDN/OpenFlow controller and DevoFlow [19]). Towards this end, we first describe the environment and workload used (in Section 6.1). Then, we examine the main aspects of the implementation of Predictor (in Section 6.2). Finally, we present the results in Section 6.3.

6.1. Setup

Environment. To show the benefits of Predictor in large-scale cloud platforms, we developed, in Java, a discrete-time simulator that models a multi-tenant, SDN-based shared datacenter. In our measurements, we focus on tree-like topologies used in current datacenters [13]. More specifically, the network topology consists of a three-level tree topology, with 16000 machines at level 0. We follow current schedulers and related work [64] and divide computing resources of servers (corresponding to some amount of CPU, memory and storage) into slots for hosting VMs; each server is divided into 4 slots, resulting in a total amount of 64000 available VMs in the datacenter. Each rack is composed of 40 machines linked to a ToR switch. Every 10 ToR switches are connected to an aggregation switch, which, in turn, is connected to the datacenter core switch. The capacity of each link is defined as follows: servers are connected to ToR switches with access links of 1 Gbps; links from racks up to aggregation switches are 10 Gbps; and aggregation switches are attached to a core switch with links of 50 Gbps. Therefore, there is no oversubscription within each rack and 1:4

[7] In oversubscribed networks, such as DCNs, where traffic may exceed link capacities in some occasions, in-band control may result in network inconsistencies, as control packets may not (or take a long time to) reach the destination.

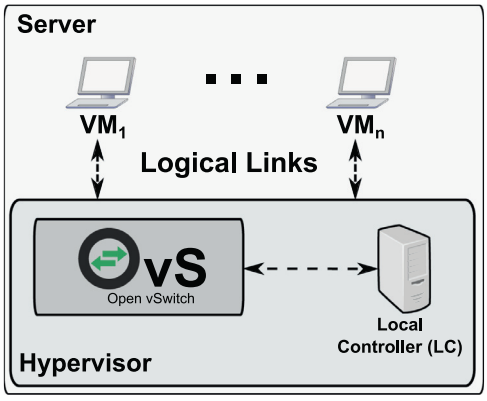


Fig. 9. Architecture of server-level implementation of Predictor.

oversubscription at the aggregation and core layers. This setup is in line with prior work [12,13,26].

Workload. The workload is composed of incoming applications to be allocated in the datacenter. They arrive over time, which means Predictor employs an online allocation strategy (as defined in Section 5.3.1). For pragmatic reasons and lack of publicly available traces, we could not obtain realistic workloads for large-scale cloud datacenters. Therefore, we generated the workload according to the measurements of related work [10,12,22,26]. Following the results of prior studies in datacenters [14,15], we consider a heterogeneous set of applications, including MapReduce and Web Services. Each application a is represented as a tuple $\langle N_a, B_a, w_a, comm_a^{inter} \rangle$, defined in Section 5.1. The values were generated as follows. The number of VMs in application a (N_a) is exponentially distributed around a mean of 49 VMs (following measurements from Shieh et al. [10]). The maximum guaranteed bandwidth rate of each VM in application a (B_a) was generated by reverse engineering the traces used by Benson et al. [22] and Kandula et al. [47]. More specifically, we used their measurements related to inter-arrival flow-time and flow-size at servers to generate and simulate network demands of applications. Unless otherwise specified, of all traffic in the network, 20% of flows are destined to other applications (following measurements from Ballani et al. [12]) and 1% is classified as large flows (following information from Abts et al. [9]). We pick the destination of each flow by first determining whether it is an intra- or inter-application flow and then uniformly selecting a destination. Finally, the weight w_a of each application a is uniformly distributed in the interval [0, 1].

6.2. Implementation aspects of predictor

Fig. 9 shows the architecture of the server-level implementation of Predictor. As described by Pfaff et al. [51], the virtual machines allocated on the server send and receive packets to/from the network through the hypervisor, using an Open vSwitch. Using the architecture proposed by Pfaff et al. as a basis, we implemented a local controller which directly communicates with the Open vSwitch. Together, the Open vSwitch and the local controller are responsible for handling all traffic to/from local virtual machines.

This architecture leverages the relatively large amount of processing power at end-hosts [65] in the datacenter to implement two key aspects of Predictor (following the description presented in the previous sections): (*i*) identifying flows at application-level; and (*ii*) providing network guarantees and dynamically enforcing rates for VMs. Both aspects are discussed next.

First, to perform application-level flow identification, Predictor utilizes Multiprotocol Label Switching (MPLS). More specifically, applications are identified in OpenFlow rules (at the Open vSwitch) through the label field in the MPLS header. The MPLS label is composed of 20 bits, which allows Predictor to identify 1,048,576 different applications. The complete operation of identifying and routing packets at application-level works as follows. For each packet received from the source VM, the Open vSwitch (controlled via the OpenFlow protocol) in the source hypervisor pushes a MPLS header (four bytes) with an ID in the label field (the application ID of the source VM for intra-application communication or a composite ID for inter-application communication). Subsequent switches in the network use MPLS label and IP source and destination addresses (which may be wildcarded, depending on the possibilities of routing) matching fields to choose the correct output port to forward incoming packets. When packets arrive at the destination hypervisor, the Open vSwitch pops the MPLS header and forwards the packet to the correct VM.

Second, the local controller at each server performs rate-limiting of VMs. More precisely, the local controller dynamically sets the allowed rate for each hosted VM by installing the appropriate rules and rate-limiters at the Open vSwitch. The rate is calculated by Algorithm 2, discussed in Section 5.3.2. Using this strategy, Predictor can reduce rate-limiting overhead when compared to previous schemes (e.g., Silo [13], Hadrian [12], CloudMirror [1] and ElasticSwitch [23]), for it only rate-limits the source VM while other schemes rate-limit each pair of source-destination VMs.

6.3. Results

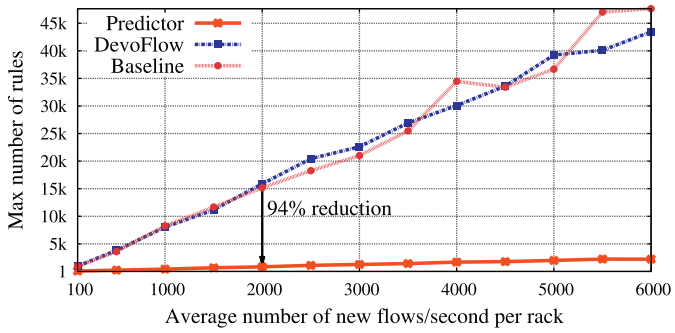
Next, we explain the behavior of the three schemes we are comparing against each other (Predictor, DevoFlow and the baseline). Then, we show the results of the evaluation: (*i*) we examine the scalability of employing Predictor on large SDN-based DCNs; and (*ii*) we verify bandwidth guarantees and predictability.

Comparison. We compare Predictor with the baseline SDN/OpenFlow controller and the state-of-the-art controller for DCNs (DevoFlow [19]). Before showing the results, we briefly explain the behavior of Predictor, the baseline and DevoFlow.

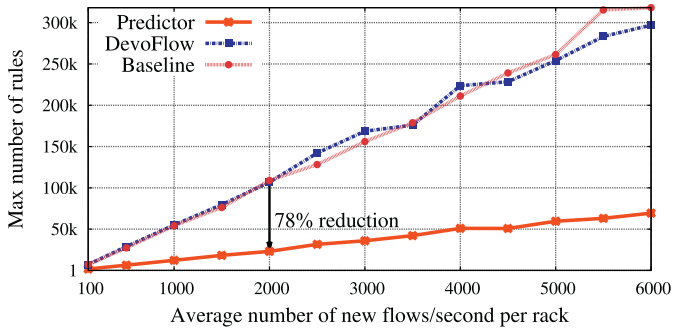
In Predictor, bandwidth for intra-application communication is guaranteed at allocation time. For inter-application communication guarantees, we consider two modes of operation, as follows. The first one is called Proactive Inter-Application Communication (PIAC), in which tenants specify in the request all other applications that their applications will communicate with (by using the field $comm_a^{inter}$, as explained in Section 5.1). The second one is called Reactive Inter-Application Communication (RIAC), in which rules for inter-application traffic are installed by the controller by either reacting to new flows in the network or receiving communication requests from applications, as defined in Section 5.2. Note that both modes correspond to the extremes for inter-application communication: while PIAC considers that all inter-application communication is specified at allocation time, RIAC considers the opposite. Furthermore, we highlight that both modes result in the same number of rules in devices, but differ in controller load and flow setup time (results are shown below).

In the baseline, switches forward to the controller packets that do not match any rule in the flow table (we consider the default behavior of OpenFlow versions 1.3 and 1.4 upon a table-miss event). The controller, then, responds with the appropriate set of rules specifically designed to handle the new flow.

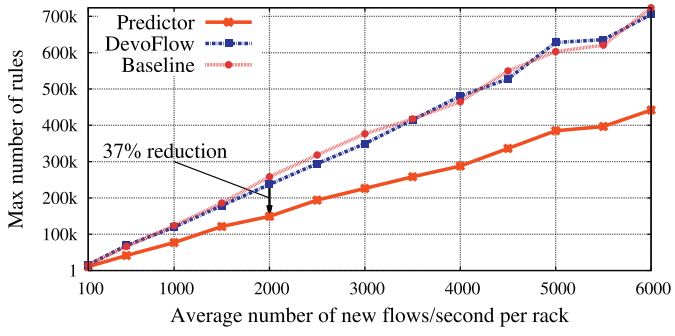
DevoFlow considers flows at the same granularity than the baseline, thus generating a similar number of rules in forwarding devices. However, forwarding devices rely on more powerful hardware and templates to generate rules for small flows without involving the controller. For large flows, DevoFlow has two modes



(a) In hypervisors.



(b) In ToRs.



(c) In aggregation switches.

Fig. 10. Maximum number of rules (that were observed in the experiments) in forwarding devices.

of operation. DevoFlow Triggers requires switches to identify large flows and ask the controller for appropriate rules for these flows (i.e., only packets of large flows are forwarded to the controller). DevoFlow Statistics, in turn, requires forwarding devices to send the controller uniformly chosen samples (packets), typically at a rate of 1/1000 packets, so that the controller itself identifies and generates rules for large flows. In summary, both DevoFlow modes generate the same number of rules in devices, but differ in controller load and flow setup time.

Scalability metrics. We use four metrics to verify the scalability of Predictor in SDN-based datacenter networks: number of rules in flow tables, controller load, impact of reserved control plane bandwidth and flow setup time. These are typically the factors that restrict scalability the most [18,23].

Reduced number of flow table entries. Fig. 10 shows how network load (measured in new flows/second per rack) affects flow table occupancy in forwarding devices. More precisely, the plots in Fig. 10(a), (b) and (c) show, respectively, the maximum number of entries observed in our experiments [8] that are required in any hypervisor, ToR and aggregation switch for a given average rate of new flows at each rack (results for core devices are not shown, as they are similar for all three schemes).

In all three plots, we see that the average number of arriving flows during an experiment affects directly the number of rules needed in devices. These results are explained by the fact that the number of different flows that pass through forwarding devices is large and may quickly increase due to the elevated number of end-hosts (VMs) and arriving flows in the network. Overall, the increase of the total number of flows requires more rules for the correct operation of the network (according to the needs of tenants) and enables richer communication patterns (representative of cloud datacenters [12]). Note that the number of rules for the baseline and DevoFlow is similar because (*i*) they consider flows at the same granularity; and (*ii*) the same default timeout for rules was adopted for all three schemes.

The results show that Predictor substantially outperforms DevoFlow and the baseline (especially for realistic numbers of new flows in large-scale DCNs, i.e., higher than 1,500 new flows/second per rack [23]). More importantly, the curves representing Predictor have a smaller growing factor than the ones for DevoFlow and the baseline. The observed improvement happens because Predictor manages flows at application-level and also wildcards the source and destination addresses in rules when possible (as explained in Section 6.2). Considering an average number of new flows/second per rack between 100 and 6,000, the improvements provided by Predictor are between 92–95% for hypervisors, 76–79% for racks and 32–40% for aggregation switches. More specifically, for a realistic number between 1,500 and 2,000 new flows/second per rack, Predictor reduces the number of rules up to 94% in hypervisors, 78% in ToRs and 37% in aggregation devices. In core devices, the reduction is negligible (around 1%), because (*a*) a high number of flows does not need to traverse core links to reach their destinations, thus the baseline and DevoFlow do not install many rules in core devices, while Predictor installs application-level rules; and (*b*) Predictor proactively installs rules for intra-application traffic (while other schemes install rules reactively).

Since Predictor considers flows at application-level and inter-application flows may require rules at a lower granularity (e.g., by matching MAC and IP fields), we now analyze how the number of inter-application flows affects the number of rules in forwarding devices for Predictor (previous results considered 20% of inter-application flows, a realistic percentage according to the literature [12]). Note that we only show results for Predictor because the percentage of inter-application flows does not impact the number of rules in forwarding devices for the baseline and DevoFlow.

Fig. 11 shows the maximum number of entries in hypervisors, ToR, aggregation and core devices observed in our experiments (y-axis) for a given percentage of inter-application flows (x-axis), considering an average of 1,500 new flows/second per rack. As expected, we see that the number of rules in devices increases according to the number of inter-application flows. This happens because this type of communication often involves a restricted subset of VMs from different applications. Therefore, Predictor may not install application-level rules for these flows and may end up installing lower-granularity ones (e.g., by matching the IP field). Nonetheless, application-level rules address most of the traffic in the DCN.

Moreover, the number of rules in aggregation and, in particular, in core switches is higher than in ToR devices and in hyper-

[8] Since flow table capacity of current available OpenFlow-enabled switches ranges from one thousand [66] to around one million entries [67], the observed values during the experiments are within acceptable ranges.

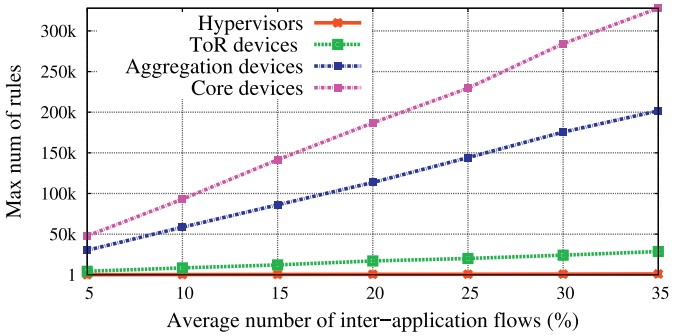


Fig. 11. Maximum number of rules in forwarding devices for different percentages of inter-application flows for Predictor.

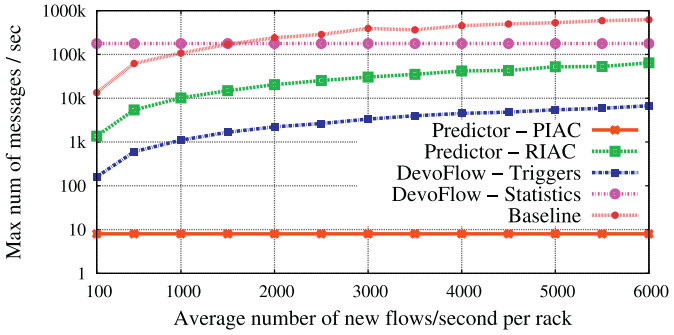


Fig. 12. Controller load.

visors. It is so because core switches interconnect several aggregation switches and, as time passes, the arrival and departure of applications lead to dispersion of available resources in the infrastructure. In this context, VMs from different applications (allocated in distinct ToRs) communicate with each other through paths that use aggregation and core switches.

In general, Predictor reduces the number of rules installed in forwarding devices, which can (*i*) improve hypervisor performance (as measured by LaCurts et al. [27]); (*ii*) minimize the amount of TCAM occupied by rules in switches (TCAMs are a very expensive resource [21] and consume a high amount of power [68]); and (*iii*) minimize the time needed to install new rules in TCAMs, as measured in Section 3.

Low controller load. As DCNs typically have high load, the controller should handle flow setups efficiently. Fig. 12 shows the required capacity in number of messages/s for the controller. For better visualization, the y-axis is represented in logarithmic scale, as the values differ significantly for different schemes. As expected, the number of messages sent to the controller increases according to the average number of new flows/s per rack (except for Predictor PIAC and DevoFlow Statistics). The controller must set up network paths and allocate resources according to arriving flows (flows without matching rules in forwarding devices).

The baseline imposes a higher load to its controller than other schemes. DevoFlow Statistics requires a regular load to its controller, independently of the number of flows, as the number of messages sent by forwarding devices to the controller depends only on the amount of traffic in the network DevoFlow Triggers, in turn, only needs controller intervention to install rules for large flows (at the cost of more powerful hardware at forwarding devices). Thus, it significantly reduces controller load, but may also reduce controller knowledge of (*i*) network load and (*ii*) flow table state in switches. Predictor RIAC proactively installs application-level rules for intra-application communication at allocation time and reactively sends rules for inter-application traffic upon receiving communication requests, which reduces the number of flow requests when compared to the baseline ($\approx$91%) but increases it in comparison to DevoFlow Triggers ($\approx$8%). Finally, Predictor PIAC receives fine-grained information about intra- and inter-application communication at application allocation time, proactively installing the respective rules when needed. Therefore, controller load can be significantly reduced (i.e., the controller receives requests only when applications are allocated) without hurting knowledge of network state, but at the cost of some burden on tenants (as they need to specify inter-application communication at allocation time for Predictor PIAC).

Recall that the Predictor modes under evaluation correspond to extremes. Therefore, in practice, we expect that Predictor controller load will be between the results shown for PIAC and RIAC. Moreover, we do not show results for controller load varying the number of large flows because results are the same for both modes (and also for the baseline and DevoFlow Statistics). DevoFlow Triggers, however, imposes a higher load to its controller as the number of large flows increases (Fig. 12 depicted results for a realistic value of 1% of large flows).

So, in both modes, the Predictor controller is aware of most of the traffic (at application-level) and performs fine-grained control. In contrast, DevoFlow Triggers has knowledge of only large flows (approximately 50% of the total traffic volume [22]) and DevoFlow Statistics has partial knowledge of network traffic with a high number of messages sent to the controller. Note that, despite these benefits, there are some drawbacks related to the time rules remain installed in forwarding devices and the ability to perform fine-grained load balancing. First, rules for intra-application communication are installed when applications are allocated and are removed when applications conclude their execution and are deallocated. Hence, some rules may remain installed more time than in a baseline setting. Second, since rules are installed at application-level, the ability to perform fine-grained load balancing in the network (e.g., for a flow or for a restricted set of flows) may be reduced.

Impact of control plane bandwidth. SDN separates the control and data planes. Ideally, control plane communication is expected to be isolated from data plane traffic, avoiding cross-interference. In this context, the bandwidth it requires varies: the more dynamic the network, the more control plane traffic may be required for updating network state and getting information from forwarding devices. We evaluate the impact of reserving some amount of bandwidth (5%, 10%, 15%, 20%, 25% and 30%) on data plane links to the control plane and compare it with a baseline value of 0% (which represents no bandwidth reservation for the control plane). In other words, we want to verify how the acceptance ratio of applications (y-axis) is affected according to the amount of bandwidth reserved for the control plane (x-axis), since the network is the bottleneck in comparison to computing resources [26,54]. Fig. 13 confirms that acceptance ratio of requests decreases according to the amount of bandwidth available for the control plane (clearly, more bandwidth for the control plane means less bandwidth for the data plane). Nonetheless, this reduction is small, even for a worst-case scenario: reserving 30% of bandwidth on data plane links for the control plane results in accepting around 9% fewer requests.

Therefore, depending on the configuration, SDN may affect DCN resource utilization and, consequently, provider revenue. There are two main reasons: (*i*) it involves the control plane more frequently [19]; and (*ii*) switches are constantly exchanging data with the controller (for both flow setup and the controller to get updated information about network state). In this context, the amount of bandwidth required for the control plane for flow setup is directly proportional to the number of requests to the controller (Fig. 12). In our experiments, switches were configured to send the first

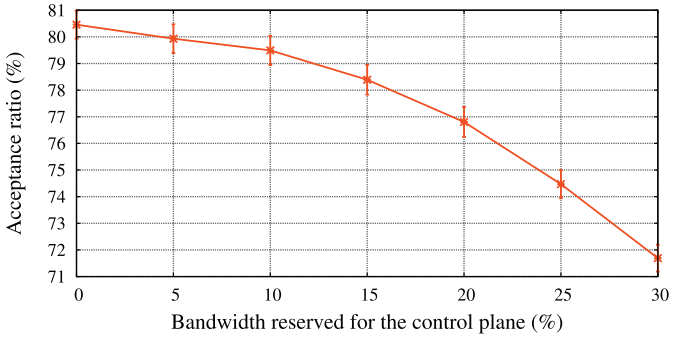


Fig. 13. Impact of reserved bandwidth for the control plane on acceptance ratio of requests (error bars show 95% confidence interval).

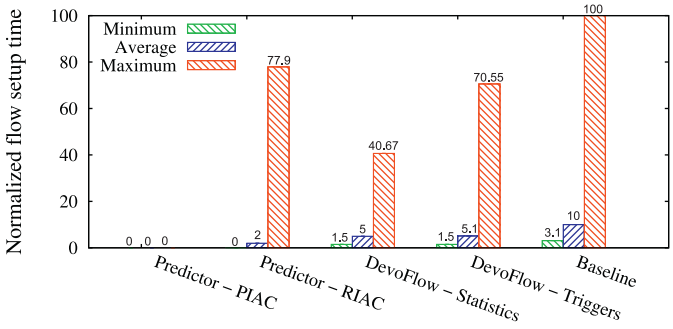


Fig. 14. Flow setup time (normalized by the maximum flow setup time of the baseline) introduced by the SDN paradigm for new flows.

128 bytes of the first packet of new flows to the controller (instead of sending the whole packet). With this configuration and for a realistic number of new flows/s per rack (i.e., 1,500 new flows), the bandwidth required by the controller for flow setup for each scheme was at most the following: 1 Mbps for Predictor PIAC, 15 Mbps for Predictor RIAC, 2 Mbps for DevoFlow Triggers, 173 Mbps for DevoFlow Statistics and 166 Mbps for the baseline. Even though Predictor may require more bandwidth for its control plane than DevoFlow in some occasions, it has better knowledge of current network state and does not need customized hardware at forwarding devices.

Reduced flow setup time. The SDN paradigm typically introduces additional latency for the first packet of new flows; traditional SDN implementations (e.g., baseline) delay new flows for at least two RTTs in forwarding devices (communication between the ASIC and the management CPU and between that CPU and the controller)[9] [19].

The results in Fig. 14 show the minimum, average and maximum flow setup time (additional latency) for the schemes being compared, during the execution of the experiments. For a better comparison, latency values are normalized (and shown in a scale from 0 to 100) according to the maximum value of the baseline (i.e., the highest value in our measurements). Predictor PIAC proactively installs rules for (*a*) intra-application traffic at allocation time and (*b*) for inter-application traffic before the communication starts (due to the information provided in the application request). Thus, it has no additional latency for new flows. Predictor RIAC, in turn, presents no additional latency for most of the flows (due to the proactive installation of rules for intra-application communication). However, it introduces some delay for inter-application flows (maximum measured latency was around 80% of the maximum for

[9] For a detailed study of latency in SDN, the interested reader may refer to Phemius et al. [69]..

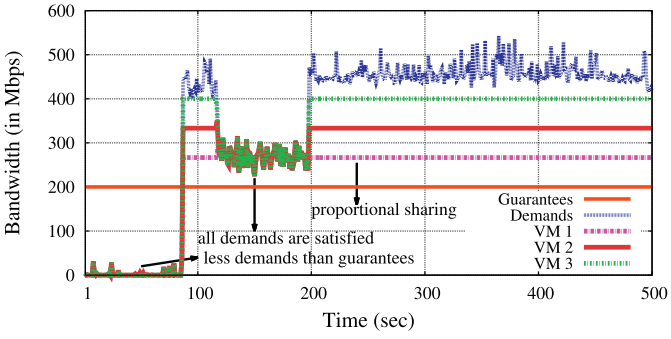


Fig. 15. Proportional sharing according to weights (VM 1: 0.2; VM 2: 0.4; and VM 3: 0.6), considering the same guarantees (200 Mbps) and the same demands for all three VMs allocated on a given server connected through a link of 1 Gbps.

the baseline). As both Predictor modes correspond to extremes, results will actually be somewhere between RIAC and PIAC. That is, the level of information regarding inter-application communication in application requests will vary, thus eliminating flow setup time for some inter-application flows.

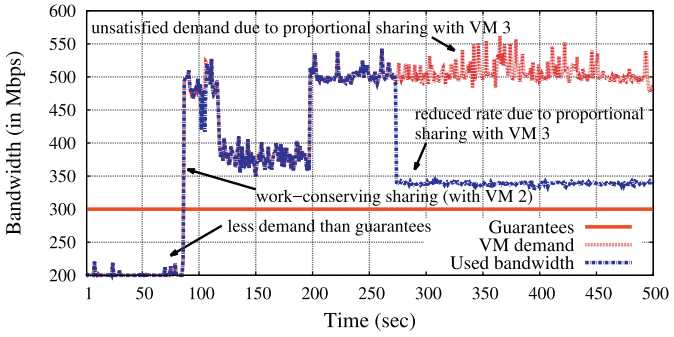
In DevoFlow Statistics, forwarding devices generate rules for all flows in the network. In other words, most of the latency is composed of the time taken for communication between the ASIC and the management CPU when a new flow is detected. Later on, the controller installs specific rules for large flows when it identifies such flows. Thus, this scheme typically introduces low additional latency. In the case of DevoFlow Triggers, large flows require controller assistance (thereby increasing additional latency), while small flows are handled by the forwarding devices themselves (low additional latency).

Finally, the baseline requires that forwarding devices always ask for controller assistance to handle new flows, resulting in increased control traffic and flow setup time in comparison to Predictor and DevoFlow.

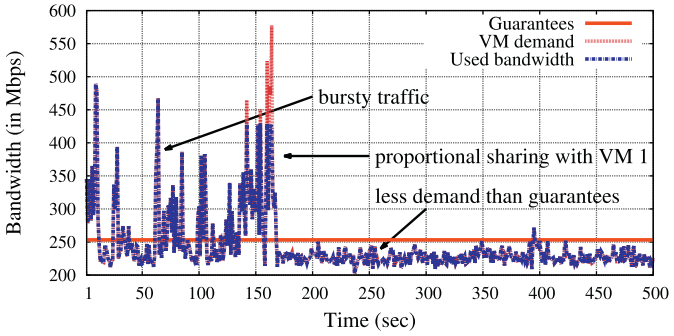
After verifying the feasibility of employing Predictor on large-scale, SDN-based DCNs (i.e., the benefits provided by Predictor, as well as the overheads), we turn our focus to the challenge of bandwidth sharing unfairness. In particular, we show that Predictor (*i*) proportionally shares available bandwidth; (*ii*) provides minimum bandwidth guarantees for applications; and (*iii*) provides work-conserving sharing under worst-case scenarios, achieving both predictability for tenants and high utilization for providers.

Impact of weights on proportional sharing. Before demonstrating that Predictor provides minimum guarantees with work-conserving sharing, we evaluate the impact of weights when proportionally sharing available bandwidth. More specifically, we first want to confirm that available bandwidth is proportionally shared according to the weights assigned to applications and their VMs.

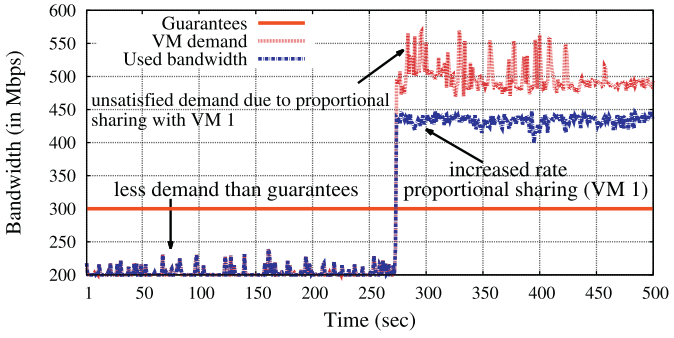
Toward this end, Fig. 15 shows, during a predefined period of time, three VMs from different applications allocated on a given server. These VMs have the same demands and guarantees (blue and red lines, respectively), but different weights (0.2, 0.4 and 0.6, respectively). Note that at instants 87 s, 118 s and 197 s, there is rapid change in bandwidth demands (blue line) of these VMs. This change reflects in the allocated bandwidth rate for each VM (pink, brown and green lines). We verify that, in case the sum of all three VM demands do not exceed the link capacity (1 Gbps), all VMs have their demands satisfied (e.g., between 1 s – 86 s and 118 s – 197 s), independently of their guarantees. In contrast, if the sum of demands exceed the link capacity, each VM gets a share of available bandwidth (i.e., more than its guarantees) according to its weight (the higher the weight, the more bandwidth it gets). Note that, in this case, the rate of each VM stabilizes (between 87 s – 117 s and 197 s – 500 s) because, as the sum of demands exceed



(a) VM 1.



(b) VM 2.



(c) VM 3.

Fig. 16. Bandwidth rate achieved by the set of VMs allocated on a given server during a predefined period of time.

the link capacity (and VMs have the same demands and guarantees), the only factor that impacts available bandwidth sharing is the weight. In general, the results show that the use of weights enables proportional sharing.

Minimum bandwidth guarantees for VMs. We define it as follows: the VM rate should be (*a*) at least the guaranteed rate if the demand is equal or higher than the guarantees; or (*b*) equal to the demand if it is lower than the guarantees. To illustrate this point, we show, in Fig. 16, the set of VMs (in this case, three VMs from different applications) allocated on a given server during a predefined time period of an experiment. Note that VM 1 [Fig. 16(a)] and VM 3 [Fig. 16(c)] have similar guarantees, but receive different rates ("used bandwidth") when their demands exceed the guarantees (e.g., after 273 s). This happens because they have different network weights (0.17 and 0.59, respectively), and the rate is calculated considering the demands, bandwidth guarantees, network weight and residual bandwidth. Moreover, we see (from Figs. 15 and 16) that VMs may not get the desired rate to satisfy all of their demands instantaneously (when their demands

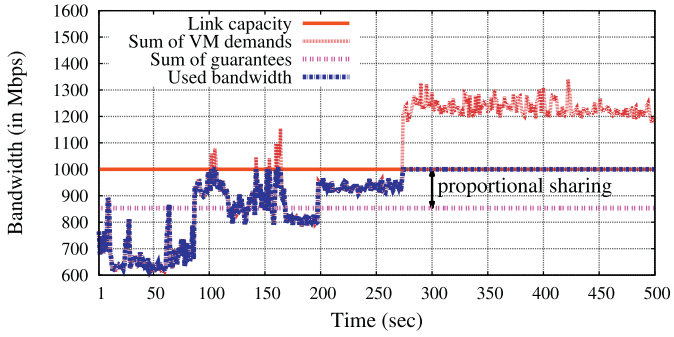


Fig. 17. Work-conserving sharing on the server holding the set of VMs from Fig. 16.

exceed their guarantees) because (*i*) the link capacity is limited; and (*ii*) available bandwidth is proportionally shared among VMs.

In summary, we see that Predictor provides minimum bandwidth guarantees for VMs, since the actual rate of each VM is always equal or higher than the minimum between the demands and the guarantees. Therefore, applications have minimum bandwidth guarantees and, thus, can achieve predictable network performance.

Work-conserving sharing. Bandwidth which is not allocated, or allocated but not currently used, should be proportionally shared among other VMs with more demands than their guarantees (according to the weights of each application, using Algorithm 2). Fig. 17 shows the aggregate bandwidth [10] on the server holding the set of VMs in Fig. 16. In these two figures, we verify that Predictor provides work-conserving sharing in the network, as VMs can receive more bandwidth (if their demands are higher than their guarantees) when there is spare bandwidth. Thus, providers can achieve high network utilization. Furthermore, by providing work-conserving sharing, Predictor offers high responsiveness [11] to changes in bandwidth requirements of applications.

In general, Predictor provides significant improvements over DevoFlow, as it allows high utilization and fine-grained management in the network for providers and predictability with guarantees for tenants and their applications. As a side effect, Predictor may have higher controller load than DevoFlow (the cost of providing fine-grained management in the network without imposing to tenants the burden of specifying inter-application communication at allocation time).

7. Discussion

After evaluating Predictor, we discuss its generality and limitations.

Application-level flow identification. In our proof-of-concept implementation, Predictor identifies flows at application-level through the MPLS label (application ID with 20 bits). Therefore, it needs a MPLS header in each packet (adding four bytes of overhead). In practice, there are at least two other options to provide such functionality. First, when considering the matching fields defined by OpenFlow, application-level flows could also be identified by utilizing IEEE standard 802.1ad (Q-in-Q) with double VLAN tagging. The advantage of double tagging is a higher number of IDs available (24 bits), while the drawback is an overhead of eight

[10] Note that Predictor considers only bandwidth guarantees when allocating VMs (i.e., it does not take into account temporal demands). Therefore, even though the sum of temporal demands of all VMs allocated on a given server may exceed the server link capacity, the sum of bandwidth guarantees of these VMs will not exceed the link capacity.

[11] Responsiveness is a critical aspect of cloud guarantees [70].

bytes (two VLAN headers) per packet. Second, application-level flows could be identified by using OpenFlow Extensible Match (OXM)[12] to define a unique match field for this purpose. Nonetheless, this method is less flexible, as it requires (*i*) switch support for OXM; and (*ii*) programming to add a new matching field in forwarding devices.

Topology-awareness. Even though Algorithm 1 was specifically designed for tree-like topologies, the proposed strategy is topology-agnostic. Therefore, we simply need to replace Algorithm 1 to employ Predictor in DCNs with other types of interconnections. We used a tree-like placement algorithm in this paper for three reasons. First, currently most providers implement DCNs as (oversubscribed) trees, since they can control the oversubscription factor more easily with this type of structure (in order to achieve economies of scale). Second, by using an algorithm specially developed for a particular structure, we can enable better use of resources. Thus, we show more clearly the benefits and overheads of the proposed strategy. Third, we used tree topologies for the sake of explanation, as it is easier to explain and to understand how bandwidth is allocated and shared among VMs of the same application in this kind of topology (e.g., in Fig. 5) than, for example, in random graphs.

Dynamic rate allocation with feedback from the network. The designed work-conserving algorithm does not take into account network feedback provided by the OpenFlow module. This design choice was deliberately made; we aim at reducing management traffic in the network, since DCNs are typically oversubscribed networks with scarce resources [26]. Nonetheless, the algorithm could be extended to consider feedback, which would further help controlling the bandwidth used by flows traversing congested links.

Application ID management. Predictor controller assigns IDs for applications (in order to identify flows at application-level) upon allocation and releases IDs upon deallocation. Therefore, ID management is straightforward, as Predictor has full control over which IDs are in use at each period of time.

Application request abstraction. Currently, Predictor only supports the hose model [71]. Nonetheless, it can use extra control applications (one for each abstraction) (*i*) to parse requests specified with other models (e.g., TAG [1] and hierarchical hose model [12]); and (*ii*) to install rules accordingly. With other abstractions, Predictor would employ the same sharing mechanism (Section 5.3). Thus, it would provide the same level of guarantees.

8. Conclusion

Datacenter networks are typically shared in a best-effort manner, resulting in interference among applications. SDN may enable the development of a robust solution for interference. However, the scalability of SDN-based proposals is limited, because of flow setup time and the number of entries required in flow tables.

We have introduced Predictor in order to scalably provide predictable and guaranteed performance for applications in SDN-based DCNs. Performance interference is addressed by using two novel SDN-based algorithms. Scalability is tackled as follows: (*i*) flow setup time is reduced by proactively installing rules for intra-application communication at allocation time (since this type of communication represents most of the traffic in DCNs); and (*ii*) the number of rules in forwarding devices is minimized by managing flows at application-level. Evaluation results show the benefits of Predictor. First, it provides minimum bandwidth guarantees with work-conserving sharing (successfully solving performance interference). Second, it eliminates flow setup time for most traffic in the network and significantly reduces flow table size (up to 94%), while keeping low controller load (successfully dealing with scalability of SDN-based DCNs). In future work, we intend to evaluate Predictor on Flexplane [72] and on a testbed (such as CloudLab [73]).

[12] OXM was introduced in OpenFlow version 1.2 and currently is supported by several commercial forwarding devices.

Acknowledgments

This work has been supported by the following grants: MCTI/CNPq/Universal (Project Phoenix, 460322/2014-1) and Microsoft Azure for Research grant award.

References

[1] J. Lee, Y. Turner, M. Lee, L. Popa, J.-M. Kang, S. Banerjee, P. Sharma, Application-driven bandwidth guarantees in datacenters, ACM SIGCOMM, 2014.
[2] F.R. Dogar, et al., Decentralized task-aware scheduling for data center networks, ACM SIGCOMM, 2014.
[3] M. Chowdhury, et al., Efficient coflow scheduling with Varys, ACM SIGCOMM, 2014.
[4] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, S. Katti, NUMFabric: fast and flexible bandwidth allocation in datacenters, in: Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference, SIGCOMM '16, ACM, New York, NY, USA, 2016, pp. 188–201, doi:10.1145/2934872.2934890.
[5] J. Guo, F. Liu, X. Huang, J.C. Lui, M. Hu, Q. Gao, H. Jin, On efficient bandwidth allocation for traffic variability in datacenters, IEEE INFOCOM, 2014.
[6] Q. Li, M. Dong, P.B. Godfrey, Halfback: running short flows quickly and safely, in: Conference on Emerging Networking EXperiments and Technologies, in: CoNEXT '15, ACM, New York, NY, USA, 2015.
[7] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, I. Keslassy, Virtualized congestion control, in: Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference, SIGCOMM '16, ACM, New York, NY, USA, 2016, pp. 230–243, doi:10.1145/2934872.2934889.
[8] K. He, E. Rozner, K. Agarwal, Y.J. Gu, W. Felter, J. Carter, A. Akella, AC/DC TCP: virtual congestion control enforcement for datacenter networks, in: Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference, SIGCOMM '16, ACM, New York, NY, USA, 2016, pp. 244–257, doi:10.1145/2934872.2934903.
[9] D. Abts, B. Felderman, A guided tour of data-center networking, Commun. ACM 55 (6) (2012) 44–51, doi:10.1145/2184319.2184335.
[10] A. Shieh, S. Kandula, A. Greenberg, C. Kim, B. Saha, Sharing the data center network, in: USENIX NSDI, 2011.
[11] M.P. Grosvenor, M. Schwarzkopf, I. Gog, R.N.M. Watson, A.W. Moore, S. Hand, J. Crowcroft, Queues don't matter when you can JUMP them!, in: USENIX NSDI, 2015.
[12] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, G. O'Shea, Chatty tenants and the cloud network sharing problem, in: USENIX NSDI, 2013.
[13] K. Jang, J. Sherry, H. Ballani, T. Moncaster, Silo: predictable message latency in the cloud, in: ACM SIGCOMM 2015 Conference, SIGCOMM '15, ACM, New York, NY, USA, 2015.
[14] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, C. Tian, Information-agnostic flow scheduling for commodity data centers, in: USENIX NSDI, 2015.
[15] L. Chen, K. Chen, W. Bai, M. Alizadeh, Scheduling mix-flows in commodity datacenters with Karuna, in: Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference, SIGCOMM '16, ACM, New York, NY, USA, 2016, pp. 174–187, doi:10.1145/2934872.2934888.
[16] M. Chowdhury, Z. Liu, A. Ghodsi, I. Stoica, HUG: multi-resource fairness for correlated and elastic demands, USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), USENIX Association, Santa Clara, CA, 2016.
[17] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, S. Shenker, Onix: a distributed control platform for large-scale production networks, in: USENIX OSDI, 2010.
[18] Y. Jarraya, T. Madi, M. Debbabi, A survey and a layered taxonomy of software-Defined networking, IEEE Commun. Surv.Tutorials PP (99) (2014) 1–29, doi:10.1109/COMST.2014.2320094.
[19] A.R. Curtis, J.C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, S. Banerjee, DevoFlow: scaling flow management for high-performance networks, in: ACM SIGCOMM, 2011.
[20] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao, C. Guo, Explicit path control in commodity data centers: design and applications, in: USENIX NSDI, 2015.
[21] R. Cohen, L. Lewin-Eytan, J.S. Naor, D. Raz, On the effect of forwarding table size on SDN network utilization, in: IEEE INFOCOM, 2014.
[22] T. Benson, A. Akella, D.A. Maltz, Network traffic characteristics of data centers in the wild, in: ACM IMC, 2010.
[23] L. Popa, P. Yalagandula, S. Banerjee, J.C. Mogul, Y. Turner, J.R. Santos, ElasticSwitch: practical work-conserving bandwidth guarantees for cloud computing, in: ACM SIGCOMM, 2013.
[24] M. Yu, J. Rexford, M.J. Freedman, J. Wang, Scalable flow-based networking with DIFANE, in: ACM SIGCOMM, 2010.
[25] S. Hassas Yeganeh, Y. Ganjali, Kandoo: a framework for efficient and scalable offloading of control applications, in: ACM HotSDN, 2012.

[26] D. Xie, N. Ding, Y.C. Hu, R. Kompella, The only constant is change: incorporating time-varying network reservations in data centers, in: ACM SIGCOMM, 2012.
[27] K. LaCurts, S. Deng, A. Goyal, H. Balakrishnan, Choreo: network-aware task placement for cloud applications, in: ACM IMC, 2013.
[28] D.S. Marcon, M.P. Barcellos, Predictor: providing fine-grained management and predictability in multi-tenant datacenter networks, in: IFIP/IEEE IM, 2015.
[29] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, Hedera: dynamic flow scheduling for data center networks, in: USENIX NSDI, 2010.
[30] A.R. Curtis, W. Kim, P. Yalagandula, Mahout: low-overhead datacenter traffic management using end-host-based elephant detection, in: IEEE INFOCOM, 2011.
[31] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, M. Handley, Data center networking with multipath TCP, in: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets-IX, ACM, New York, NY, USA, 2010, pp. 10:1–10:6, doi:10.1145/1868447.1868457.
[32] H. Ballani, P. Costa, T. Karagiannis, A. Rowstron, Towards predictable datacenter networks, in: ACM SIGCOMM, 2011.
[33] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, A. Greenberg, EyeQ: practical network performance isolation at the edge, in: USENIX NSDI, 2013.
[34] J. Guo, F. Liu, D. Zeng, J. Lui, H. Jin, A cooperative game based allocation for sharing data center networks, in: IEEE INFOCOM, 2013.
[35] W. Li, K. Li, D. Guo, G. Min, H. Qi, J. Zhang, Cost-minimizing bandwidth guarantee for inter-datacenter traffic, IEEE Trans. Cloud Comput. PP (99) (2016), doi:10.1109/TCC.2016.2629506. 1–1.
[36] V.T. Lam, S. Radhakrishnan, R. Pan, A. Vahdat, G. Varghese, Netshare and stochastic netshare: predictable bandwidth allocation for data centers, SIGCOMM Comput. Commun. Rev. 42 (3) (2012) 5–11, doi:10.1145/2317307.2317309.
[37] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, I. Stoica, FairCloud: sharing the network in cloud computing, in: ACM SIGCOMM, 2012.
[38] M. Barbera, A. Lombardo, C. Panarello, G. Schembra, Active window management: an efficient gateway mechanism for TCP traffic control, in: 2007 IEEE International Conference on Communications, 2007, pp. 6141–6148, doi:10.1109/ICC.2007.1017.
[39] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74.
[40] G. Judd, M. Stanley, Attaining the promise and avoiding the pitfalls of TCP in the datacenter, in: USENIX NSDI, 2015.
[41] J. Schad, J. Dittrich, J.-A. Quiané-Ruiz, Runtime measurements in the cloud: observing, analyzing, and reducing variance, Proc. VLDB Endow. 3 (1–2) (2010) 460–471.
[42] G. Wang, T.S.E. Ng, The impact of virtualization on network performance of amazon EC2 data center, in: IEEE INFOCOM, 2010.
[43] R. Shea, F. Wang, H. Wang, J. Liu, A deep investigation into network performance in virtual machine based cloud environment, in: IEEE INFOCOM, 2014.
[44] H. Shen, Z. Li, New bandwidth sharing and pricing policies to achieve awin-win situation for cloud provider and tenants, in: IEEE INFOCOM, 2014.
[45] P. Gill, N. Jain, N. Nagappan, Understanding network failures in data centers: measurement, analysis, and implications, in: ACM SIGCOMM, 2011.
[46] M. Alizadeh, A. Greenberg, D.A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, Data center TCP (DCTCP), in: ACM SIGCOMM, 2010.
[47] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, R. Chaiken, The nature of data center traffic: measurements & analysis, in: ACM IMC, 2009.
[48] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L.E. Li, M. Thottan, Measuring control plane latency in SDN-enabled switches, in: ACM SOSR, 2015.
[49] S.A. Jyothi, M. Dong, P.B. Godfrey, Towards a flexible data center fabric with source routing, in: USENIX NSDI, 2015.
[50] A.D. Ferguson, A. Guha, C. Liang, R. Fonseca, S. Krishnamurthi, Participatory networking: an API for application control of SDNs, in: ACM SIGCOMM, 2013.
[51] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, M. Casado, The design and implementation of open vSwitch, in: USENIX NSDI, 2015.
[52] M. Chowdhury, M.R. Rahman, R. Boutaba, Vineyard: virtual network embedding algorithms with coordinated node and link mapping, IEEE/ACM Trans. Netw. 20 (1) (2012) 206–219, doi:10.1109/TNET.2011.2159308.
[53] H. Moens, B. Hanssens, B. Dhoedt, F. De Turck, Hierarchical network-aware placement of service oriented applications in clouds, in: IEEE/IFIP NOMS, 2014.
[54] L. Chen, Y. Feng, B. Li, B. Li, Towards performance-centric fairness in datacenter networks, in: IEEE INFOCOM, 2014.
[55] N. Chowdhury, M. Rahman, R. Boutaba, Virtual network embedding with coordinated node and link mapping, in: IEEE INFOCOM, 2009.
[56] Amazon EC2, 2014, Available at: http://www.goo.gl/Fa90nC.
[57] L. Muller, R. Oliveira, M. Luizelli, L. Gaspary, M. Barcellos, Survivor: an enhanced controller placement strategy for improving SDN survivability, in: IEEE GLOBECOM, 2014.
[58] Y. Hu, W. Wendong, X. Gong, X. Que, C. Shiduan, Reliability-aware controller placement for software-defined networks, in: IFIP/IEEE IM, 2013.
[59] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, G. Parulkar, Onos: towards an open, distributed SDN os, in: ACM HotSDN, 2014.
[60] F.J. Ros, P.M. Ruiz, Five nines of southbound reliability in software-defined networks, in: ACM HotSDN, 2014.
[61] C. Chen, C. Liu, P. Liu, B.T. Loo, L. Ding, A scalable multi-datacenter layer-2 network architecture, in: ACM SOSR, 2015.
[62] B. Heller, R. Sherwood, N. McKeown, The controller placement problem, in: ACM HotSDN, 2012.
[63] D. Adami, B. Martini, M. Gharbaoui, P. Castoldi, G. Antichi, S. Giordano, Effective resource control strategies using openflow in cloud data center, in: IFIP/IEEE IM, 2013.
[64] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, A. Akella, Multi-resource packing for cluster schedulers, in: ACM SIGCOMM, 2014.
[65] M. Moshref, M. Yu, R. Govindan, A. Vahdat, Trumpet: timely and precise triggers in data centers, in: Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference, SIGCOMM '16, ACM, New York, NY, USA, 2016, pp. 129–143, doi:10.1145/2934872.2934879.
[66] Y. Nakagawa, K. Hyoudou, C. Lee, S. Kobayashi, O. Shiraki, T. Shimizu, Domainflow: practical flow management method using multiple flow tables in commodity switches, in: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '13, ACM, New York, NY, USA, 2013, pp. 399–404, doi:10.1145/2535372.2535406.
[67] Switching Made Smarter, 2015, Available at: http://www.noviflow.com/products/noviswitch/.
[68] D. Kreutz, F.M.V. Ramos, P. Veríssimo, C.E. Rothenberg, S. Azodolmolky, S. Uhlig, Software-defined networking: a comprehensive survey, CoRR (2014).
[69] K. Phemius, M. Bouet, Openflow: why latency does matter, in: IFIP/IEEE IM, 2013.
[70] J.C. Mogul, L. Popa, What we talk about when we talk about cloud network performance, SIGCOMM Comput. Commun. Rev. 42 (5) (2012) 44–48, doi:10.1145/2378956.2378964.
[71] N.G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K.K. Ramakrishnan, J.E. van der Merive, A flexible model for resource management in virtual private networks, in: ACM SIGCOMM, 1999.
[72] A. Ousterhout, J. Perry, H. Balakrishnan, P. Lapukhov, Flexplane: an experimentation platform for resource management in datacenters, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), USENIX Association, Boston, MA, 2017, pp. 438–451.
[73] CloudLab, 2016, Available at : https://www.cloudlab.us.

Daniel S. Marcon is a professor at University of Vale do Rio dos Sinos – UNISINOS, Brazil. He received his Ph.D. in Computer Science from Federal University of Rio Grande do Sul – UFRGS, Brazil. He also holds a B.Sc. degree in Computer Science from UNISINOS (2011) and a M.Sc. degree in Computer Science from UFRGS (2013). His research interests include datacenter networks, cloud computing, network virtualization and software-defined networking (SDN).

Fabrício M. Mazzola is an undergraduate Computer Science student at the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS), Brazil. His research interests include software-defined networking (SDN), network virtualization and datacenter networks. More information can be found at http://www.lattes.cnpq.br/2243053098009957.

Marinho P. Barcellos received a PhD degree in Computer Science from University of Newcastle Upon Tyne (1998). Since 2008 Prof. Barcellos has been with the Federal University of Rio Grande do Sul (UFRGS), where he is an Associate Professor. He has authored many papers in leading journals and conferences related to computer networks, network and service management, and computer security, also serving as TPC member and chair. He has authored book chapters and delivered several tutorials and invited talks. His work as a speaker has been consistently distinguished by graduating students. Prof. Barcellos was the elected chair of the Special Interest Group on Computer Security of the Brazilian Computer Society (CESeg/SBC) 2011- 2012. He is a member of SBC and ACM. His current research interests are datacenter networks, software-defined networking, information-centric networks and security aspects of those networks. He was the General Co-Chair of ACM SIGCOMM 2016 and TPC Co-Chair of SBRC 2016. More information can be found at http://www.inf.ufrgs.br/~marinho.