Virtual Network Flavors: Differentiated Traffic Forwarding for Cloud Tenants

Aryan TaheriMonfared(✉) and Chunming Rong

IDE Department, University of Stavanger, Stavanger, Norway
{aryan.taherimonfared,chunming.rong}@uis.no

Abstract. Today, a cloud system user can select a specific type of virtual machine for deployment based on needs, for instance memory or storage size. In terms of networking, however, no similar mechanism exists which allows users to select a virtual network based on characteristics such as link speed and QoS. The lack of such a mechanism makes it difficult for users to manage VM instances along their associated networks. This limits the efficacy and scalability of cloud computing suppliers.

This paper presents a novel approach for defining *virtual network flavors* and differentiated forwarding of traffic across the underlay networks. The flavors enable tenants to select network properties including maximum rate, maximum number of hops between two VMs, and priority. Measures such as metering, prioritizing, and shaping facilitate steering traffic through a set of paths to satisfy tenants' requirements. These measures are designed such that the legacy parts of the underlay network can also benefit from them. Software Defined Networking (SDN) mechanisms are an essential part of the solution, where the underlay and overlay networks are managed by a network operating system. The implementation and evaluation data are available for further development [2].

Keywords: Software defined networking · QoS · Virtual network

1 Introduction

Today's cloud infrastructure as a service provider supports a variety of virtual machine types. These types are frequently referred to as *flavors*, which defines a virtual machine's specifications, such as the number of vCPU, memory, and storage size. Flavors aid users by simplifying the selection and specification of VMs. However, there exists no similar mechanism for the virtual networks connecting these VMs. Virtual networks are mostly created with similar specifications, which are also limited. They lack quality of service (QoS) support, path calculation options, etc. Per-tenant virtual network building blocks, as presented in [13], propose an extreme approach where the new architecture delegates all network functions to tenants. While this solution can be beneficial for an enterprise customer, it is a burden for the average customer. Moreover, a tenant is not in control of the underlay network and cannot influence the forwarding decisions. Thus, a provider should deliver more functionality for virtual networks.

M.C. Aguayo-Torres et al. (Eds.): WWIC 2015, LNCS 9071, pp. 165–179, 2015.
DOI: 10.1007/978-3-319-22572-2_12

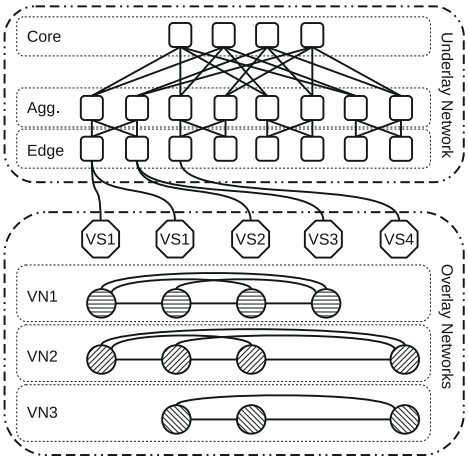


Fig. 1. High level overlay and underlay architecture

Tunnelling is a common technique for creating virtual networks and isolating tenants' traffic in the underlay. However, this technique gives rise to several issues in adapting legacy network functions to handle tenants' traffic. In particular, identifying, classifying, and engineering tenants' traffic in the core network, without decapsulating, are challenging tasks.

Moreover, typical data center network topologies consist of a multi-rooted tree [4]. Therefore, there are multiple paths between any two nodes. Virtual network traffic between two endpoints of a tenant should be forwarded through the path which fulfils the tenant's QoS requirements. A central controller with a unified view of the network can perform significantly better than hash-based Equal-Cost Multi-Path (ECMP) load-balancing [4].

This paper proposes an approach for differentiated forwarding of overlay traffic across the underlay network (Fig. 1), such that it satisfies the virtual network flavor constraints. Initially, the virtual network flavor should be defined as a structure that presents a tenant traffic class, maximum rate, priority, etc. (Sect. 3). Then, the following steps are necessary to enforce the flavor requirements. The first step is to discover the underlay network topology. The underlay topology is later used to find a set of flavor-compliant paths between VMs which are connected to the same virtual network (Sect. 4). Therefore, the tenant's points of presence in the data center should be identified and the topology of its virtual network should be discovered (Sect. 5). Then, tenants' traffic are classified according to their network flavors (Sect. 6). The classification is performed only once, at the endpoint virtual switch (Sect. 7). The overlay traffic is marked such that a legacy network node can also benefit from the classification. Finally, the aforementioned end-to-end paths, for each class, are programmed in the overlay and underlay networks (Sect. 8). In addition, an extensive evaluation framework is developed to study the effectiveness and accuracy of the flavor implementation (Sect. 9).

2 Related Work

Plug-n-Serve [6] is a load-balancing system for web services. It uses the network state and servers load to customise flow routing and update service capacity (e.g. number of hosts).

Wang et al. [14] proposes a proactive load-balancing approach for online services using OpenFlow. It creates a minimal set of wildcard rules to aggregate clients' requests and update the set to reflect the system dynamics.

Hedera [4] focuses on scheduling large flows in a data center with a topology inspired by the fat tree topology. It shows that dynamic flow scheduling can substantially improve network bandwidth in multi-rooted trees, while the deployment cost is moderated. Simulated Annealing and Global First Fit algorithms are compared to ECMP, and the results show that the former outperforms the rest in most cases.

Khan et al. [7] introduces a mechanism for virtual network QoS support in the Multi-Protocol Label Switching (MPLS) transport domain. OpenFlow domains are the ingress and admission points to the MPLS transport network, and have their own disjoint controllers. However, the transport network is configured by legacy means. Each virtual network session admission, at an OpenFlow domain, requires communication with the domain's controller. This requirement imposes an overhead on the session establishment process, which is caused by the communication round-trip time and the decision-making logic in the controller.

3 Virtual Network Flavors

Customers' traffic in a cloud infrastructure can be isolated using different techniques, such as IP block splitting, VLAN tagging, and tunnelling. Maintaining IP blocks is a tedious task – the VLAN field has a limited size (up to 4094 under IEEE 802.1Q), and tunnels introduce additional overheads, in terms of forwarding capacity and traffic volume. However, tunnelling is the most flexible approach and can be widely used if the overheads are reduced by employing hardware acceleration (e.g. Virtual eXtensible Local Area Network (VXLAN) [8]) or hardware-friendly protocols (e.g. Stateless Transport Tunnelling (STT) [5]).

This paper focuses on virtual networks which use tunnelling techniques for traffic isolation and end-to-end forwarding. However, these mechanisms can be expanded, with minor extensions, to cover other virtualization techniques.

Virtual network flavor specifies the quality of end-to-end paths between any two VMs which are connected to a virtual network. Although our proposed mechanisms support varying flavors for the paths in a virtual network (i.e. virtual network path flavor), for the sake of simplicity the case is presented where paths in a virtual network use the same flavor (i.e. virtual network flavor). Path flavor plays an essential role in providing fine-grained QoS for provisioned resources by a tenant. For instance, resources involved in latency-sensitive tasks utilize the shortest path, while high traffic volume operations are forwarded through the least utilized path.

Each flavor has a class that is used for coarse-grained traffic classification and facilitates the legacy network support. Since legacy networks have a limited capacity for traffic classes (e.g. 6-bit Differentiated Services Code Point (DSCP)), multiple flavors may have the same class. In addition to the class, a flavor determines the end-to-end priority and maximum rate of virtual network traffic in the underlay network. In the following text, the underlay refers to the infrastructure's network that is controlled by the infrastructure provider; while overlay refers to the tenant's virtual network that is established on top of the infrastructure network, Fig. 1.

As the virtual switch inside a compute node has high throughput, the inter-VM communication inside a single hypervisor is not shaped. Therefore, traffic engineering takes place for the inter-hypervisor VM's communication.

4 Underlay Topology Discovery and Path Calculation

The data center network has a tree-like topology with redundant paths between any two nodes. Therefore, it can be represented as a weighted directed sparse graph, where an edge has a REFERENCE_SPEED/LINK_SPEED weight. The underlay topology is constructed by processing node and link updates, which are sent from the controller platform.

For a different set of tenants with varying network demands, shortest path routing is not flexible enough [3]. The aim is to calculate the first k-shortest loopless paths (KSP) between any given node pair, where k is a positive integer. A modified version of Yen's algorithm [15] is implemented for path calculation. Moreover, a set of constraints can be used to further limit the result. For instance, the constraints can ensure that packets visit a set of nodes before reaching the destination to apply middle-box functions, or avoid another set to bypass congestion. Modularity of the approach makes it flexible to update the algorithm with another group of policies.

5 Overlay Topology Discovery

Tunnels are established on demand between interested endpoints. Tunnel endpoints are responsible for marking tenants' traffic with their tunnel keys, prior to encapsulation. Two types of tunnel discovery are performed – proactive and reactive. The proactive approach is suitable for OpenFlow 1.0 and 1.3 protocols. It listens to the management plane events which are dispatched from the OVSDB southbound plug-in. When a new tunnel interface is identified, the key, IP address, interface ID, and the corresponding tenant ID are added to a data store. The data store is further processed to build a matrix of tunnel endpoints for each tenant, that represent the tenant's virtual network. The data store reflects the current status of tenants' overlays, and is updated by relevant events (e.g. add, remove, modify). This approach is applicable where tunnels are "port-based" and created with explicit flow keys. In the port-based model each tenant has a separate tunnel and the port fully specifies the tunnel headers [11].

The reactive approach takes advantage of the cloud platform APIs for retrieving the tenant network properties (e.g. tunnel key, provisioning hosts, virtual ports, etc.). By correlating the properties with the information from the SDN controller, tunnel endpoints, which are involved in the tenant overlay, are discovered. This approach is necessary for "flow-based" tunnels, introduced in OpenFlow 1.3. Flow-based tunnels do not strictly specify the flow key in the tunnel interface and one OpenFlow port can represent multiple tunnels. Therefore, the matching flow rule should set the key and other required fields, such as source/destination IP addresses.

6 Tenant Traffic Classification in Underlay Network

Topology discovery for the overlay and underlay networks was explained in the previous sections, as well as path calculation between two endpoints. However, this is not enough for classifying tenants' traffic in the underlay, when it is encapsulated. Classes are essential for differentiated forwarding of overlay traffic.

Irrespective of the isolation technique chosen for the network virtualization, classifying tenant traffic in the underlay network is a challenging task. If IP block splitting or VLAN tagging are used for isolation, source/destination IP addresses or VLAN IDs suffice for tenant traffic identification, respectively. However, distinguishing tunnelled traffic is not trivial. The tenant IP packet is encapsulated in the tunnel packet, and the tunnel ID (64 bits) is inside the tunnel header. OpenFlow 1.3 [9] proposes 40 match fields, and can not match on the tunnel ID in a transit node.

Therefore, either the first packet of a flow should be sent to the controller, for decapsulation, or another field should be used for the classification. The former approach is not practical, since the overhead is significant. For the latter approach, the IPv6 header has a "flow label" field (20 bits), which can be used for this purpose. However, there is no equivalent field in the IPv4 header. In IPv4, the DSCP field is used. This approach has two major benefits: proactive flow programming (i.e. efficient forwarding) and seamless integration with the legacy network DiffServ domains.

A third approach is to use an IP addressing scheme in the underlay which reflects the virtual network tenant and forwarding classes in addition to the location. Further implementation and evaluation are parts of the future work.

7 Endpoint Virtual Switch Architecture

Tenants' tunnelled traffic should be marked, according to its class, before leaving a host. However, when a packet is sent to a tunnel interface, the Linux routing table determines the egress interface. This decision cannot be overridden by the virtual switch. Therefore, the OpenFlow LOCAL port on the switch is configured such that the routing table will choose it as the egress for all tunnelled traffic (Fig. 2). The configuration consists of assigning an IP address from the provider underlay subnet to the LOCAL port.

Since, the LOCAL port is chosen as the egress interface, external interfaces should be added to the virtual switch. The external interfaces connect the host to the underlay network. In addition, the virtual switch is supplied with a set of rules to forward the traffic between LOCAL and external interfaces, and avoid loops in case of persistent broadcast packets.

Moreover, tunnels are created such that they copy the inner IP header ToS to the outer one. This provides visibility to the DSCP value of a packet in transit.

8 Differentiated Forwarding

This section explains the end-to-end network programming steps for implementing the overlay QoS. It consists of controlling endpoint switches of a traffic exchange and underlay transit nodes along the path.

8.1 Programming Endpoints

The tenant's egress IP traffic from VMs is marked with the DSCP value derived from the endpoint pair's flavor (Sect. 6). The encapsulated packets are further processed in the switch and an external interface is chosen according to the provided policy for the tenant and the tunnel destination. The detailed steps are as follows (Fig. 2): (1) Marking the tenant's traffic with the chosen DSCP value. (2) Sending the marked traffic to the tunnel. (3) Resolving the egress port using the Linux routing table. (4) Capturing the tunnelled traffic on the switch LOCAL port. (5) Checking the tunnel packets and marking those without DSCP by the biggest value for the given endpoint pair. (6) Forwarding the traffic through an external interface.

The encapsulated traffic destined for a tenant VM(s) is forwarded through the underlay and received on an external interface of the destination host. It is decapsulated in the tunnel interface and forwarded to the VM(s) or another switch: (7) Receiving traffic on the external interface. (8) Forwarding ingress

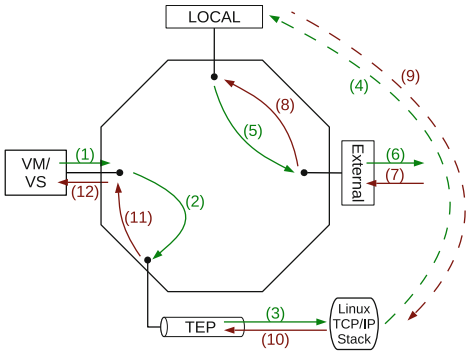


Fig. 2. Virtual switch configuration

traffic from an external interface to the LOCAL port. (9) Resolving the tunnel port. (10) Capturing the traffic on the tunnel port. (11) Decapsulating the packet. (12) Forwarding the packet to the destination.

8.2 Programming Underlay

Underlay (aka transit) nodes are networking devices in the edge, aggregation, and core networks, that are responsible for forwarding tenants' traffic between compute nodes. Tenant traffic identification is explained in Sects. 6 and 8.1 has discussed overlay traffic marking in each endpoint. Transit datapaths should be programmed such that they honour the classification made at the endpoints. The *(in_port, ip_src, ip_dst, mark)* tuple determines the class of a tenant flow in transit. However, efficiently programming the tuple across the network is a challenging task. This section proposes a few approaches for the differentiated forwarding of tenants' traffic in transit nodes.

Programming Transit Nodes on a Given Path Between Two Endpoints. Endpoints are compute nodes which are hosting resources provisioned by a tenant. This approach finds a path between two given endpoints for a tenant, such that it complies with the tenant's virtual network flavor. Then, it programs intermediate nodes to forward the overlay traffic.

It is proactive and utilizes the forwarding table space efficiently. The proactive mechanism reduces the runtime overhead, and responds well to small flows by avoiding further communication with the controller (i.e. slow path).

Programming Complete Reachability Matrix Between All Endpoints for All Classes. This approach calculates routes for all network classes between all endpoints. If provisioned resources by all tenants are uniformly distributed over endpoints, it has less computational cost compared to the previous one, while the storage cost is of the same order. Like the previous one, this is proactive and has a minimal impact on the forwarding speed.

However, if the tenants distribution is not uniform, the forwarding table space is not used efficiently and may cause a state explosion.

Programming on PacketIn Messages. A third approach is to wait for PacketIn messages from intermediate nodes, when they do not have a matching flow rule for a packet. The switch sends a PacketIn message to the controller, which contains the packet and its arrival port. Then, the controller finds the corresponding tenant for the flow and calculates the path for it, according to the network flavor. Finally, it programs all the nodes on the path for that flow. Therefore, other packets will have a matching flow entry, and would be forwarded in the datapath. This is a reactive mechanism for programming the network. Although it uses the table space efficiently, the overhead is significant, specifically for short-lived flows.

The evaluation (Sect. 9) focuses on the first approach, which provides a balance between the number of installed flow entries and the runtime computational complexity. In addition, the path between two endpoints can be updated dynamically,

with least disruption. The new path flow entries have lower priorities compared to the old ones. Therefore, the target traffic uses the old path, while the new path is programmed in the underlay. When all flow tables are updated, the old path is removed and the traffic will match the new one. This approach ensures that packets are not forwarded to switches without matching flow entries.

8.3 Traffic Engineering Methods

Implementing traffic engineering using SDN mechanisms can be more flexible and effective compared to IP and MPLS based TEs [3]. Unified views of the networking substrates and applications, through southbound and northbound APIs, provide visibility into the network status and the applications' requirements. The SDN specification abstraction [12] makes the proactive programming and dynamic reprogramming of the network efficient, as devices are not handled individually. Therefore, the controller can efficiently react to the network status and application demand.

In addition to the chosen path between two endpoints, meters and queues of the OpenFlow switches are exploited for traffic engineering and QoS differentiation in the underlay.

Meter. A meter is a part of a flow entry instruction set. It measures the rate of all the packets which are matched to the flows to which it is attached. The meter counters are updated when packets are processed. Each meter has a set of bands which specify the minimum rate at which their action can be applied. A band can support "drop" or "dscp remark" operations [9]. Meter creation, configuration, and flow assignment are performed by the OpenFlow wire protocol version 1.3.0.

In our design, each traffic class has a set of meters, which are attached to the flows mapped to it. The minimum rate of a meter band is specified in the network flavor, otherwise it is REFERENCE_BW/CLASS_NUM, where the REFERENCE_BW represents the physical port speed (i.e. 1 Gbps for our test-bed) and CLASS_NUM is the traffic class specified in the flavor. When the packet rate is over the band rate, the band applies and packets are dropped.

Queue. A port in an OpenFlow switch can have one or more queues, where flow entries are mapped to them. A queue determines how flows are treated, using a set of properties: a minimum rate, a maximum rate, and an experimenter property [9]. The maximum rate of a queue is set to REFERENCE_BW/(MQ-QP), where MQ and QP are the maximum number of queues and the queue priority, respectively. In contrast to the meter, queue configuration is handled by the OVSDB protocol, while flow mapping is done by the OpenFlow wire protocol.

Flows of a tenant are forwarded to a queue according to the virtual network flavor priority. The queue priority determines which one is served first. Therefore, in addition to the rate limiting function, overlay traffic with a higher priority class is forwarded before lower priority ones.

Meters are more useful for fine-grained rate limiting, since a switch can support more meters than queues, and meters are per-flow while queues are per-port.

Therefore, a meters' configuration can be more granular. Although the number of queues in hardware is limited, they provide better guarantees for traffic prioritization and are more effective for coarse-grained policies. A combination of meters and queues can deliver complex QoS frameworks [9].

For the evaluation, minimum rates of meter bands and maximum rates of queues are chosen with significant differences for each flavor to clearly visualize the impact on the network QoS (Table 1).

9 Evaluation

Flavors should create a clear distinction between virtual networks' QoS. An extensive evaluation framework is developed to measure a variety of QoS parameters and study the system behaviour under different scenarios.

9.1 Evaluation Framework

The framework has a modular architecture and allows new experiments to be added as plug-ins. The experiment covers all scenarios that should be assessed. It requires a set of parameters, which are used to create a group of sub-experiments. A sub-experiment is executed according to the specified execution type and the results are comparable. Some of the parameters used for the virtual network throughput analysis are as follows:

– *Instances range*: The minimum and maximum number of virtual machine instances which are involved in the measurements.

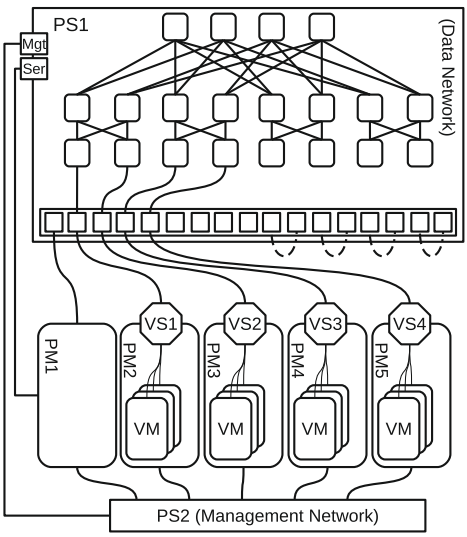


Fig. 3. Evaluation test-bed

- *Networks range*: The minimum and maximum number of virtual networks, created in this experiment.
- *Classes*: The traffic classes (specified in the flavors), assigned to virtual networks, where lower class numbers have higher priorities and better QoS.
- *Instance distribution*: The distribution of instances over networks.
- *Network distribution*: The distribution of networks over classes.
- *Path length*: The maximum number of hops between two instances of a class.
- *Instances execution type*: Determines whether instances should perform the task simultaneously or sequentially (i.e. `i:{true,false}`).
- *Classes execution type*: Determines whether networks of a class should execute the task in parallel or in series (i.e. `c:{true,false}`).
- *TE strategy*: Specifies the traffic engineering method(s) in the underlay network (i.e. `none, meter, queue, meter_queue`).
- *Failure threshold*: The upper threshold for the permitted number of failures before a sub-experiment is terminated.

Before executing a sub-experiment, all the configurations are set to their default values and the cloud platform is reinitialized. This will avoid propagation of side-effects from previous sub-experiments and increase precision. After the reinitialization, virtual networks and instances are created according to the aforementioned parameters. Then, the SDN controller programs the network and the framework waits for the instances to become reachable. Once enough instances are reachable, the framework instructs instances to perform the measurement and report back. If the number of reachable instances is less than the failure threshold, the process is terminated with an error report and the next sub-experiment is started.

When all reachable instances finish their tasks, the reports are stored and processed. Finally, all instances and networks are deleted.

In addition to the input parameters, each report contains a set of task-specific results. For instance, the throughput measurement reports on TCP/UDP rate, Rx/Tx CPU utilization, Round Trip Time (RTT), retransmission occurrence, number of report errors, number of missing values, start/end time and hosting hypervisors. The processing phase consists of report classification based on common parameters, data aggregation, statistical analysis and plotting. The outcomes are presented in two forms, TE strategy comparison (Sect. 9.4) and execution type comparison (Sect. 9.5).

9.2 Execution Types

Evaluating virtual network performance is not a trivial task [13]. Tenants of a cloud service provision and release resources dynamically; they also have workloads with varying traffic characteristics throughout the day. Therefore, virtual network QoS should be studied under different scenarios where a realistic workload is simulated in the network.

An execution type defines the scheduling method of a sub-experiment. It determines how competing networks with different classes and their instances

should perform the measurement. To limit the scope of the evaluation, two parameters are considered: network classes and instances execution types. The network class execution type specifies whether throughput of competing networks should be measured concurrently. Whereas, the instance execution type controls the parallel or series measurement between two instances in a network.

The number of concurrent individual measurements (CIM) depends on the execution type, and estimated values are presented in Table 2. However, it might not be the exact number in an arbitrary given point of time, because a strict mechanism is not used for the experiment scheduling. As an example, some instances of a network class may become reachable sooner than others, when all networks and their instances have been requested simultaneously. Therefore, it may show some stochastic behaviours and noises in the results.

Although concurrent execution of the measurements decreases each class throughput, higher priority classes perform better than lower priority ones.

Table 1. TE method properties

	Meter	Queue	
Class	Min rate	Priority	Max rate
1	1 Gbps	6	1 Gbps
2	500 Mbps	5	500 Mbps
3	333 Mbps	4	333 Mbps
4	250 Mbps	3	250 Mbps

Table 2. Scheduling method properties

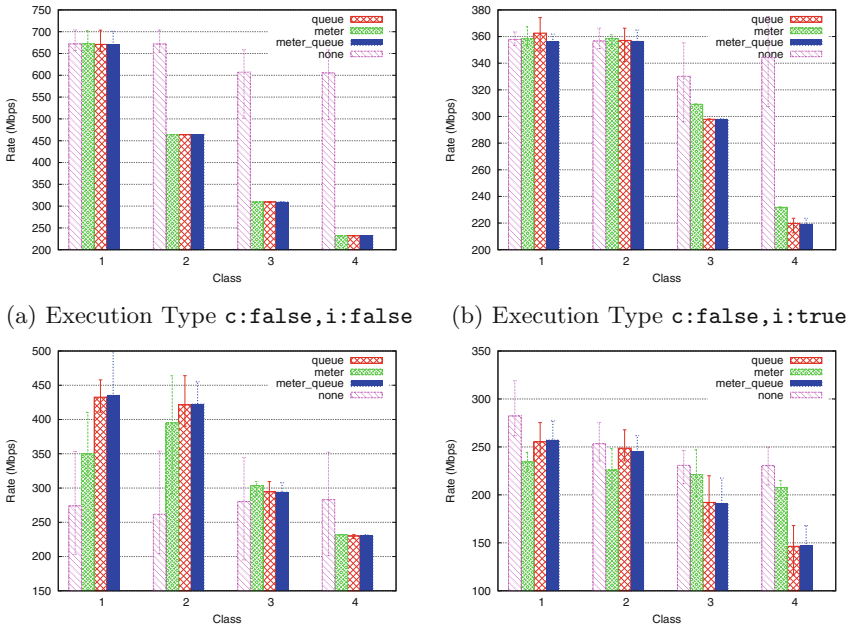
Type	# CIM
`c:false,i:false`	*1*
`c:false,i:true`	*(#instances choose 2)*
`c:true,i:false`	*#classes*
`c:true,i:true`	*#classes* × *(#instances choose 2)*

9.3 Evaluation Setup

The platform is deployed on 5 Intel NUCs, with Core i5 processors, 16 GByte memory, 140 GByte SSD storage, and two Gigabit Ethernets. Each node is connected to the management and data networks. The management network uses an 8-port Gigabit switch, and the data network uses an OpenFlow capable Pica-8 P-3290 switch. The whitebox switch has 8 physical queues (0–7), and the highest priority queue is dedicated to the switch-controller communication.

As shown in Fig. 3, the SDN (PM1) and cloud (PM2) controllers have dedicated nodes and the rest are compute nodes (PM3,4,5). A Fat-Tree topology with K-port commodity switch is emulated in the whitebox switch (PS1), where K is 4. As the switch does not support Open vSwitch patch-port, the remaining physical ports are connected in pairs to patch the emulated switches.

The SDN controller modules are developed as part of the OpenDaylight project, and the cloud platform is OpenStack. The implementation has 10,613 lines of code and is available for the community, along with the raw evaluation data and additional analysis [2]. The virtual machine image is built using Buildroot [1] and the VM flavor is m1.nano with one vCPU and 256 MB memory.



(a) Execution Type `c:false,i:false` (b) Execution Type `c:false,i:true`

(c) Execution Type `c:true,i:false` (d) Execution Type `c:true,i:true`

Fig. 4. Comparing the impact of QoS strategies on TCP throughput for different execution types

9.4 Traffic Engineering Strategy Analysis

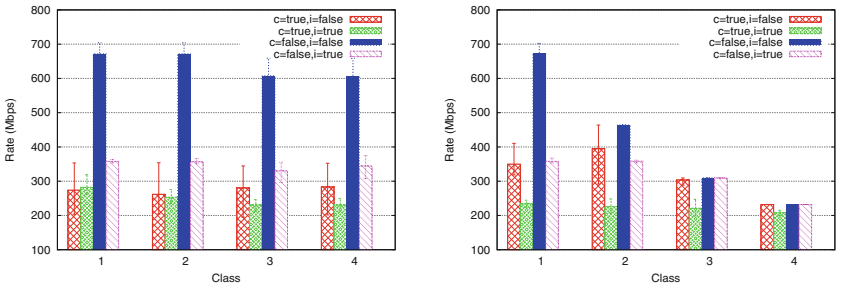
Figure 4 represents the mean TCP throughput between any two instances on a network, with a distinct class, that are hosted on different hypervisors. Each sub-figure depicts a specific execution type, and data series are different strategies.

Figure 4a plots the TCP throughput when at any point of time measurement is performed between two instances of a single class. As explained in Sect. 9.2, since this scheduling method offers the least concurrency, each class achieves the maximum throughput under different TE strategies. When the only class enforcement method is the chosen path (i.e. strategy is `none`), the differences between classes' throughput are not significant. However, other strategies make considerable differences between classes.

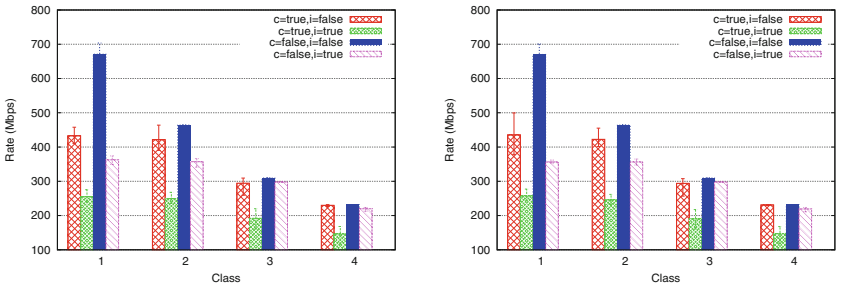
As shown in Figs. 4b, c and d, when instances or networks execute the measurements concurrently, the TCP throughput is decreased.

9.5 Execution Type Analysis

Figure 5 also represents the mean TCP throughput between any two instances, where each sub-figure depicts a specific strategy, and data series represent execution types. It can be observed that the throughput is mandated by the class priority, and it is independent of the scheduling method.

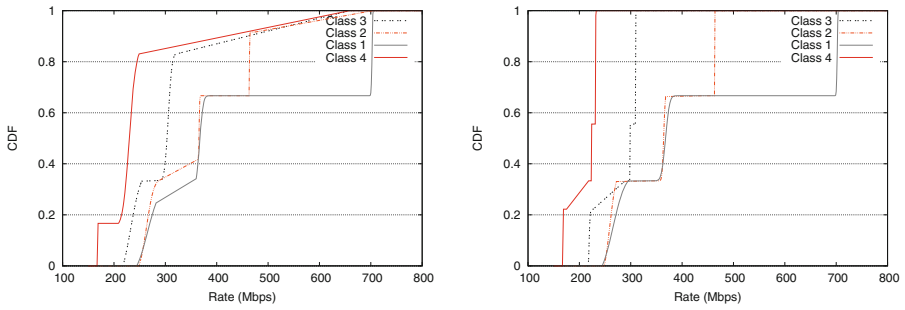


(a) Strategy `none` (b) Strategy `meter`



(c) Strategy `queue` (d) Strategy `meter_queue`

Fig. 5. Comparing the impact of execution types on TCP throughput for different QoS strategies



(a) TE methods: `none, meter, queue, meter_queue` (b) TE methods: `meter, queue, meter_queue`

Fig. 6. CDF of the 90th percentile TCP throughput for each class independent of the experiment scheduling approach.

In Fig. 5c, each queue has a maximum rate of REFERENCE_BW/CLASS_NUM; the TCP throughputs of all classes are limited by this constraint. However, if the maximum rate is set to REFERENCE_BW, when classes are not executed concurrently (i.e. `c:false,i:*`), lower priority classes will perform as good as

the strategy `none` case in Fig. 5a; whereas the throughputs in other execution types remain the same. It is due to the queue scheduling mechanisms of the switch, where the higher priority queues are served first.

Figure 6 depicts the CDF of the 90th percentile TCP throughput for each class, independent of the scheduling method. Figure 6a presents the CDF with and without traffic engineering mechanisms explained in Sect. 8.3, while Fig. 6b only includes the 90th percentile throughput when at least one TE method is employed. It can be deduced that higher priority classes (lower class numbers) perform better than the lower priority ones.

10 Conclusion

Virtual network flavor is a crucial feature, but missing in cloud platforms design. The proposed approach utilizes SDN mechanisms for delivering flavors, and providing QoS for overlay virtual networks. SDN abstractions make overlay traffic classification and underlay traffic engineering much more efficient. The logically centralized SDN controller not only provides a unified view of the network through southbound APIs, but also has visibility into applications' demands through northbound APIs. Standard (i.e. OpenFlow 1.3 [9]) and open (i.e. OVSDB [10]) protocols are used for the control and management planes, and all operations are performed using them. Therefore, there is no manual configuration involved for network management and programming.

Acknowledgements. This work is done in collaboration with Norwegian NREN (UNINETT).

References

1. Buildroot. http://buildroot.uclibc.org/
2. Vnet-flavors. http://www.ux.uis.no/~aryan/docs/dc/vnet-flavor/
3. Akyildiz, I.F., Lee, A., Wang, P., Luo, M., Chou, W.: A roadmap for traffic engineering in SDN-OpenFlow networks. Comput. Netw. **71**, 1–30 (2014)
4. Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., Vahdat, A.: Hedera: dynamic flow scheduling for data center networks. In: Proceedings of the 7th USENIX NSDI, NSDI 2010, pp. 19–19. USENIX Association (2010)
5. Davie, B., Gross, J.: A Stateless Transport Tunneling Protocol for Network Virtualization. Internet Draft (Informational), October 2014
6. Handigol, N., Seetharaman, S., Flajslik, M., McKeown, N., Johari, R.: Plug-n-serve: Load-balancing web traffic using OpenFlow. ACM SIGCOMM Demo **4**(5), 6 (2009)
7. Khan, A., Kiess, W., Perez-Caparros, D., Triay, J.: Quality-of-service (QoS) for virtual networks in OpenFlow MPLS transport networks. IEEE, November 2013
8. Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., Wright, C.: Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348
9. Open Networking Foundation: OpenFlow switch specification version 1.3.0

10. Pfaff, B., Davie, B.: The Open vSwitch Database Management Protocol. No. 7047 in Request for Comments, IETF, published: RFC 7047 (Informational), December 2013
11. Pfaff, B.: Open vSwitch manual. http://benpfaff.org/~blp/ovs-fields.pdf
12. Shenker, S., Casado, M., Koponen, T., McKeown, N.: The future of networking, and the past of protocols. Open Networking Summit (2011). http://opennetsummit.org/archives/oct11/shenker-tue.pdf
13. TaheriMonfared, A., Rong, C.: Flexible building blocks for software defined network function virtualization. In: 10th International Conference on QShine (2014)
14. Wang, R., Butnariu, D., Rexford, J.: OpenFlow-based server load balancing gone wild. In: Proceedings of the 11th USENIX Hot-ICE. USENIX Association (2011)
15. Yen, J.Y.: Finding the k shortest loopless paths in a network. Manage. Sci. **17**(11), 712–716 (1971)