Implementation and Evaluation of a Carrier-grade OpenFlow Virtualization Scheme

Pontus Sköldström
Acreo Swedish ICT AB
Kista, Sweden
pontus.skoldstrom@acreo.se

Wolfgang John
Ericsson Research
Kista, Sweden
wolfgang.john@ericsson.com

***Abstract*— Software Defined Networking (SDN) concepts are seen as suitable enablers for network virtualization, especially in the Data Center Network domain. However, also carrier network operators can benefit from network virtualization, since it allows new business models, promising economical benefits through sharing the cost of network infrastructure in e.g. multi-tenancy or service-isolation scenarios. Such use-cases pose additional requirements on virtualization schemes, including strict performance and information isolation, transparency of the virtualization system, high availability, as well as low CAPEX and OPEX demands. In order to fulfill these requirements, we previously proposed a flexible virtualization scheme for OpenFlow. In this paper we discuss the implementation of the proposed scheme and point out relevant lessons learned during the process, leading to architectural and technological updates. We then evaluate the system in terms of data path performance: the impact on forwarding latency is negligible, while the impact on network throughput is depending on the type of traffic and the choice of encapsulation technology. In summary, the overhead can be kept small and would not significantly affect a production network. Thus, we conclude that the minor performance degradations are outweighed by the benefits of the virtualization system.**

I. Introduction

Software Defined Networking (SDN) describes a networking concept which decouples network control from forwarding - two critical networking functions, that have traditionally been interleaved in monolithic network elements with distributed control planes. In SDN, network state and intelligence is moved to a logically centralized control plane, and the actual per-hop forwarding decisions are programmed on the data plane elements by the remote control plane via open interfaces, most prominently OpenFlow. At the same time, the centralized control plane presents an abstracted view of the network to any type of control program or application using the network. By exposing a common API to abstract the underlying networking details, SDN enables easy programmability, automation, and network control. This opens up opportunities for network innovation through creation of new applications and services in software, i.e. decoupled from hardware restrictions.

OpenFlow, originally developed by researchers and Stanford and now marshalled by the Open Networking Foundation (ONF), emerged as the most prominent interface between data plane elements and SDN controllers [1]. OpenFlow switches consist of flow tables containing flow entries that are used to identify the incoming packets and make forwarding decisions based on matching the incoming traffic against the predefined rules. Flow tables are statically or dynamically configured by the controller via OpenFlow messages transported over a secure channel. These OpenFlow messages are specified by the OpenFlow protocol, being used for configuration of OpenFlow switches, management of OpenFlow tables, and retrieval of the flow table state and capabilities of the switches.

Besides the promised benefits of SDN itself (i.e. ease of operation, increased service and innovation velocity), carrier network operators also recognized the possibilities of SDN to enable network virtualization [2], thereby following a trend that has been ongoing in data center networks for a while already. Virtualization of operator networks opens up for new business model promising economical effects by sharing the cost of network infrastructure, used for e.g. multi-tenancy or service-isolation scenarios [3]. However, existing techniques such as VLANs, Virtual Routing Forwarding (VRF) and various Virtual Private Network (VPN) approaches solve network virtualization only partially. Advanced use cases, such as a multi-tenancy scenario enabling virtual network operators on top of a single physical networking infrastructure, pose at least the following requirements on a virtualization scheme: (i) strict *isolation* between virtual networks, in terms of a) network performance (e.g. bandwidth); and b) information, thus preventing explicit information leakage (e.g. eavesdropping of packets, etc.) and implicit, indirect information leakage (such as topology or policy information); (ii) operational isolation by providing a *transparent* virtualization system, allowing tenants unrestricted choice of protocols, labels, address spaces, etc.; (iii) minimal additional OPEX and CAPEX demands, such as the possibility to realize the system in software, thus avoiding additional or specialized hardware requirements; and (iv) full integration to the existing architecture and software realization of SDN without the need to introduce additional elements (e.g. middleboxes) in order to avoid single-points of failures and to limit additional administrative burdens.

In order to fulfill these requirements, we proposed a flexible virtualization scheme for OpenFlow in [4], [5]. After a short recap of this scheme in section II, we will discuss the implementation of the system and point out relevant lessons learned during this process in section III. We evaluate the system in terms of performance impact in section IV, and finally conclude the paper in section V.

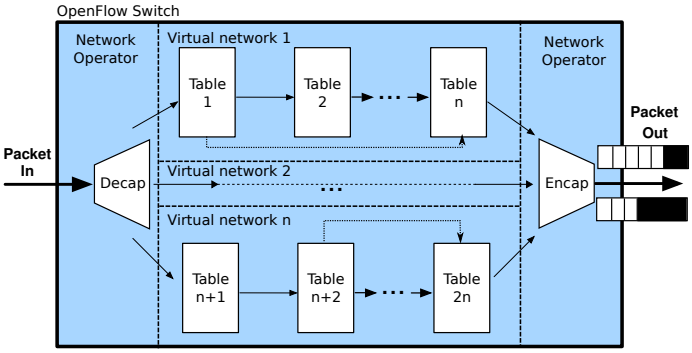

Fig. 1: Conceptual model [4]. Packets are de- and encapsulated in the first and last table, resp. In between, they pass through a set of flow tables reserved for a particular virtual network.

II. Conceptual model

We introduced the conceptual model (see Fig. 1) in [4], where we proposed to create virtual networks through two means: packet encapsulation on the links; and partitioning of flow tables in the switches. On the links some type of encapsulation (we suggested MPLS tunnels) is used to differentiate packets belonging to different networks, without setting any restrictions on what address spaces are supported by a particular virtual network. In the switches, a number of flow tables would be reserved for a single virtual network. Once a packet enters a switch, the encapsulation would determine to which set of flow tables in the OpenFlow pipeline the packet should be sent. Before forwarding the packet to those tables, the encapsulation would be removed. A virtual network customer would be limited to only be able to view and access the tables corresponding to his virtual network, the packets seen by the customer (and the assigned flow tables) would seem as if they had not been virtualized (since the encapsulation has been removed). The customer could then manipulate the packets in his virtual network using the assigned tables. Before the packets are sent to the next hop they are encapsulated once again. The network operator would configure the switches, in particular the first and last table, using an OpenFlow controller. Once configured, customers can connect to the switches using their own OpenFlow controllers which would only see the tables and ports reserved for a particular virtual network. This model is particularly suited for multi-tenancy use-cases by fulfilling the requirements defined in section I:

i) Isolation in terms of strict bandwidth limits can be ensured using normal QoS tools. Information isolation, i.e. the risk for information leakage or packet injection into another virtual network is minimized by encapsulation.
ii) Transparency is ensured by encapsulating packets between switches. There are no restrictions on the address spaces assigned to the virtual network customers, they may for example use the same IP or VLAN address space without colliding. This is crucial since it removes any need to negotiate between customers concerning who uses which identifier or adapt control software.
iii) The model can be implemented through simple software/-firmware upgrades to the switches, no additional devices or hardware modifications are necessary.
iv) As the translation functionality is distributed to all switches, the impact of failures is reduced. At the same time, the main intelligence of the system resides in the virtualization manager as controller application, in line with the SDN/Openflow paradigm.

Our approach fulfills all requirements posed by the multi-operator use-case, while FlowVisor [6] schemes, common in current literature, typically cannot fulfill req. ii and iv.

III. Implementation

During the SPARC project[1] we developed a proof-of-concept implementation of the model based on OpenFlow version 1.1 [7]. While successful, we had to update the model in certain aspects as we faced some issues during the implementation.

A. Data plane model

A detailed view of the implemented data plane model can be seen in Fig. 2. The first table is used to assign traffic to virtual networks depending on the type of port the packet packet arrives on: a customer port or a shared port[2]. In the case of a customer port some filtering may be performed before forwarding the packet, on a shared port the VLAN encapsulation will always be removed before forwarding. Once forwarded to a set of customer tables the customer may apply any matches or actions it wishes, there are no restrictions, even the use of groups and packet in/packet out via the controller is unrestricted. Once the final customer flow table or group has been passed the packet is forced to go through a per-port per-customer group that reapplies the encapsulation and assigns a queue depending on the outgoing port and virtual network. If the outgoing port is a customer port, this final group is skipped and the packet is forwarded directly to the port. At the port, QoS queues are used to apply per-customer QoS restrictions that prevents any customer from using more that its assigned share of bandwidth. Queuing here may be skipped if it is not demanded by the use-case, e.g. multi-service scenarios at a single operator might not require strict bandwidth allocation.

The flow tables and groups that form the virtualization system, or belong to other customers, are completely hidden from a connected customer controller. The customer only "sees" the assigned flow tables and ports, as well as the groups that the customer himself creates. Responsibility of enforcing this view as well as making sure that de-/encapsulation is performed correctly, etc., is split between the *translation unit* located on the switch and an application at the network owners controller, what we call the *virtualization manager*.

B. Translation unit

The translation unit is configured by a virtualization manager application, using the experimenter extensions of OpenFlow

[1] www.fp7-sparc.eu

[2] A customer port is a port directly connected to a customer network, while a shared port is a link carrying packets belonging to multiple customers (similar to access and trunk ports in VLAN terminology).

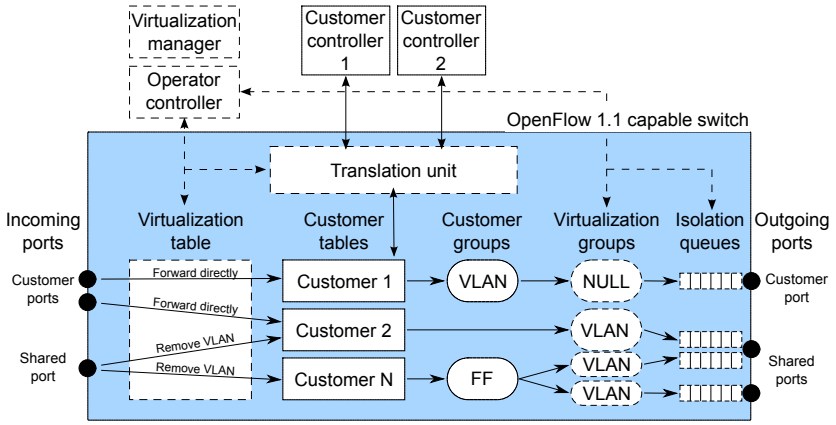

Fig. 2: The implemented data plane model: Elements in solid lines represent a standard OpenFlow 1.1 switch; Elements in dashed lines represent additions needed to implement the virtualization scheme.

1.1. It is responsible for translating incoming and outgoing OpenFlow messages (think of it as a single switch FlowVisor [6]). This is done in order to provide a restricted virtual view, only showing the correct tables, groups, and statistics as well as enforcing that the connected controllers are only able to program the intended tables, etc. The virtualization manager adds virtual networks to a translation unit by sending a `ADD_VN` message, see Fig. 3. The `ADD_VN` message configures:

- A virtual network identifier, an encapsulation identifier, and a queue identifier.
- Ranges of the flow- and group tables that are assigned to the virtual network.
- Lists of assigned customer- and shared ports.

With the `ADD_VN` information available it is fairly straightforward to implement a translation unit that filters and modifies incoming and outgoing OpenFlow messages.

For example, when a customer controller requests a list of all available ports using a `OFPT_FEATURES_REQUEST` message the translation unit examines the assigned port information for the virtual network assigned to the asking controller and discovers that the actual ports, e.g. {3,6,8}, are shared ports that are assigned to the particular virtual network. The port information for {3,6,8} is fetched and translated to virtual identifiers {1',2',3'} before replying to the request. Simple translation like this can quickly be applied to most of the OpenFlow messages with little programming effort in order to show only the appropriate information whether it concerns translating to/from actual/virtual identifiers for ports, flow- or group tables.

Similarly, commands from customer controllers need to be modified. For example a `OFPT_FLOW_MOD` command requires translation of the virtual flow table identifier and modification of both the Match and Action list. In the Match, e.g. the Incoming port may have to be updated in order to refer to a physical instead of virtual port identifier. Any Actions in the Action list has to be inspected, any group identifiers must be modified as well as outgoing ports. In order to enforce that the encapsulation is reapplied, any customer actions sending packets to a port are converted into actions forwarding the packet the appropriate group that has been setup for encapsulating and forwarding the packet to the physical port.

Simple translation of the relevant identifiers (ports, flow tables, groups, etc.) is enough in most cases. This simplicity result in our proof-of-concept implementation being a patch of only about 2000 lines to the original OpenFlow 1.1 switch implementation, a fairly minor modification (the original size is near 50000 lines). However, there are some fields that are more complicated to deal with appropriately, for example the *buffer identifier* in the `OFPT_FLOW_MOD` message. Our implementation handles these in a very naïve fashion, so a more robust implementation would need to be slightly larger.

C. Virtualization manager

The virtualization manager runs as an application on top of an OpenFlow 1.1 NOX controller, representing the controller of the operator owning the physical network. It not only configures the translation unit on all switches, but also configures their first table (table 0, the virtualization table), the encapsulation groups, and the per-port per-customer QoS queues. It obtains the information about which virtual networks should be configured, and how, by reading a configuration file containing the networks topologies and per virtual network configuration. In the future, advanced configuration management such as datapath configuration synchronization, global coherence checks, etc. would reside here as well.

```
struct ofl_exp_add_vn {
    struct ofl_exp_msg_header header;
    uint32_t virtual_network_id;
    uint16_t encapsulation_id;
    uint32_t queue_id;
    uint16_t flow_table_range[2];
    uint32_t group_table_range[2];
    uint32_t n_shared_ports;
    uint32_t n_customer_ports;
    uint32_t shared_ports[];
    uint32_t customer_ports[];
};
```

Fig. 3: Experimental message ADD_VN for adding a virtual network to the translation unit.

Configuration of the virtualization table currently includes setting up rules for either forwarding packets directly from a port to a customer table, or decapsulating the packet before forwarding it. The virtualization groups are created with a unique group identifier derived from the virtual network identifier and virtual identifier of the physical port they are forwarding to after re-encapsulating packets. The particular group identifiers are necessary for the translation unit; the translation unit needs to derive the same group identifiers whenever it replaces a customers `OFPAT_OUTPUT` action with a `OFPAT_GROUP` action. This is done in order to forward packets via a virtualization group responsible for reapplying the encapsulation before the packet is forwarded to the actual physical outgoing port.

Finally, we need to be able to authenticate and associate connecting controllers to corresponding virtual networks. In a more mature implementation, this could be done with PKI certificates also used for setting up SSL connections. We instead simply added a secret value to the OpenFlow handshake process. If this value matches a virtual network identifier, the controller is assigned to that network. For the Operator controller, which has no virtual network associated, a specific secret value is used to grant full access.

D. Lessons learned

In our original design we focused on MPLS tunnels as the encapsulation format. They seemed to be a suitable encapsulation technology, primarily because of their ability to be stacked. However, if the packet that you want to encapsulate does not already have an MPLS label, you would have to modify the EtherType of that packet to indicate that the packet is now an MPLS packet. This requires that if MPLS labels are used, we must use multiple labels per virtual network, since we need one label per virtual network/EtherType combination in order to be able to set the correct, original, EtherType on the packets when we remove the MPLS label.

The extra complexity of managing multiple labels, potentially one for each defined EtherType per virtual network, led us to reconsider VLAN tagging instead of MPLS labelling. Since VLANs can be stacked without limit in OpenFlow 1.1 without modifying the original packet, they turn out to be a good replacement. However, VLANs have limitations, primarily the amount of available identifiers (12 bits) might make VLANs tricky to use in an environment where they are already used for other purposes, e.g. identifying services.

Since any encapsulation format that does not modify the original packet is suitable, we also investigate encapsulation using Ethernet Pseudowires over MPLS (PWE), which we had implemented earlier in OpenFlow 1.1 [5]. While Pseudowires add significantly larger overhead, they can be used together with the MPLS OAM that we adapted to OpenFlow during the SPARC project. This allows us to set up end-to-end protected LSPs that can carry our PWE encapsulated virtual links, providing sub-50ms rerouting of virtual networks.

The original design also contained a encapsulation table responsible for re-applying the encapsulation. This was changed to groups since it is much simpler to implement, you only need to rewrite the output actions. In the encapsulation table case you are required to track where packets have been before arriving at the final table, e.g. by using the metadata fields. This becomes quite complicated and requires an extra table lookup, something avoided when using groups.

IV. Evaluation

After verifying that the implementation works properly we set out to evaluate its performance. We expect that the introduction of the virtualization system will impact two aspects of the OpenFlow switch, the control channel as well as data plane forwarding performance.

The control channel performance (i.e. latency of OpenFlow commands) is affected by the addition of a translation unit in the switch, since it has to inspect and modify OpenFlow messages. However, due to the simplicity of these operations we expect the increased processing time to be on a timescale not measurable at a client controller, especially when compared to latencies in the control network itself[3].

Performance loss in the data plane on the other hand is critical and any performance losses here could be detrimental to the viability of the virtualization solution. Therefore, our performance evaluation focuses on the performance of the data plane implementation and we do not evaluate the performance of the OpenFlow control channel or the virtualization manager.

On the data plane level we are primarily concerned about two things. First, the penalty induced on forwarding latency introduced by decapsulation/encapsulation at each hop combined with the extra steps taken within the switch, traversing the virtualization table and groups. The second aspect is the throughput penalties induced by adding encapsulation, which we analyze based on typical packet size distributions in Internet traffic.

A. Latency impact of data plane encapsulation

In order to measure per-hop data plane processing costs of the virtualization system (using multiple tables and packet tagging/encapsulation), we constructed a setup in which packets are sent over a large number of hops before having their RTT measured. By comparing the difference in RTT for different configurations we hope to be able to detect differences in their performances. However, in order to reduce the noise in delay variations (i.e. jitter) caused by network interfaces, etc., we run each OpenFlow switch as a process on a single server and connect the switches using virtual interfaces. This, combined with applying a Linux kernel optimized for low latency (Linux-RT) dramatically reduces measured jitter.

However, as a consequence of this setup we expected that the difference in processing time for the different scenarios would be too small to detect if we only send our probe packets through a single node. Therefore we devised a looping setup

[3]Note that this is not necessarily the case for other virtualization methods, esp. the FlowVisor approach. A separate FlowVisor adds not only extra processing time, but also transmission delay to due at least one extra hop in the control network before the OpenFlow message reaches the switch.

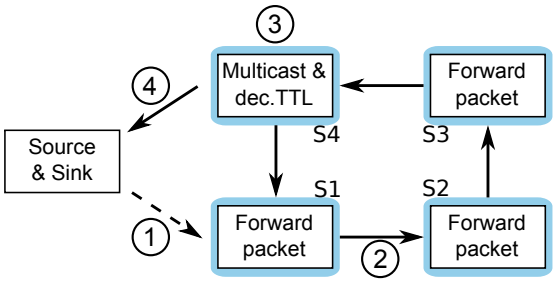

Fig. 4: Experimental setup, consisting of a ring of OpenFlow switches circulating the packet injected by the host. 1) Packet is injected by the host, 2) packet is forwarded until it reaches S4, 3) S4 multicasts the packet to the host and to switch S1 after decrementing the TTL, 4) the host captures the packet.

that allows us to flexibly scale the number of hops in order to increase the cumulative processing time, hopefully making it large enough to be measurable.

Our forwarding loop, which can be seen in Fig. 4, is made up of four OpenFlow switches (S1 to S4) controlled by a NOX controller running the virtualization manager responsible for configuring the switches for the different scenarios. The experiment was executed on an 8-core server running a Linux-RT kernel. We further pin each OpenFlow switch process to a dedicated core in order reduce jitter even more. Once the virtualization scheme in the switches has been configured appropriately, a customer controller connects to the virtual switches. The customer controller configures the virtual switches to forward incoming packets through the ring.

To initialize the experiment, the server injects a single ICMP ping packet to S1, which starts forwarding the packet through the ring. In order to allow multiple observations of the packet at source/sink, S4 is configured with an OpenFlow multicast group, not only forwarding a copy of the packet onward through the ring (i.e. back to S1), but also forwarding a copy to the second interface to the source/sink. To stop the packet from looping endlessly the TTL is decremented at S4, causing the packet to be dropped after a configurable number of loops.

For our measurements, this setup ensures that the source/sink is responsible for both injecting and receiving the packet, thus avoiding any possible time-synchronization issues between two separate source and sink hosts. The inter-arrival times of packets observed at the host thus represent the time it takes to traverse four data path elements and five links.

We ran the experiment with five configurations in order to test two different encapsulation formats (VLAN/PWE) and the effects of queuing for bandwidth isolation (-Q). We compare the results with a baseline setup without virtualization, i.e. with native OpenFlow 1.1 switches (Base). In the experiments, the operator controller configures the virtualization scheme, after which a customer controller connects and configures the loop forwarding/multicasting which is identical regardless of any configuration of the virtualization. For each of the configurations we collected roughly half a million inter-arrival samples, which are summarized in Table I and Fig. 5.

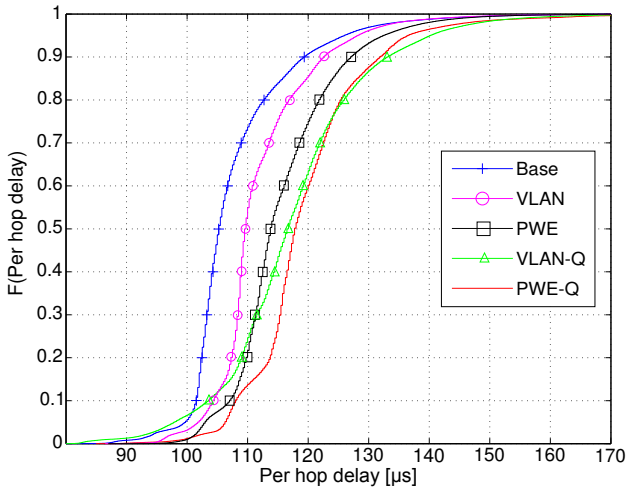

Fig. 5: CDF plots of the per-hop latency for the different measured configurations, Base is without virtualization, -Q is with encapsulation and QoS queuing.

Statistic		Base	VLAN	PWE	VLAN-Q	PWE-Q
Minimum	(μs)	77	87	87	74	85
Maximum	(μs)	344	727	698	504	871
Median	(μs)	105	110	114	117	118
Median	(rel. inc.)	—	+5%	+9%	+11%	+12%

TABLE I: Statistics of the per-hop latency for the configurations, the last line shows the relative increase of the median.

We can see that the per hop latency is increased by a few microseconds as we use more complex virtualization with VLAN and PWE alone adding 5 and 9 microseconds respectively. When QoS queuing is included the added latency compared to without QoS is almost doubled (4-7 microseconds extra). The absolute difference in latency, e.g. 12 microseconds in the case of the baseline (Base) compared to VLAN with QoS queuing (VLAN-Q), indicates an upper limit to the latency overhead, assuming that actual hardware implementations would perform better than our software implementation. The relative difference, roughly a 11% increase in per hop delay in the VLAN-Q case, seems to be a significant cost. However, this is in an ideal scenario without any link delay, no queuing on the interface, or other sources of latency. The extra latency is in the scale of a few microseconds per hop, which in the end makes is negligible even for paths with many hops, given that one-way delays usually are in the *millisecond* range (i.e. three orders of magnitude larger).

We conclude that while there is a small per-hop delay overhead due to the virtualization scheme (including encapsulation, virtualization- tables and groups, and QoS queuing), it does not add significant latencies to end-to-end delay, which is typically dominated by transmission delays orders of magnitude larger.

B. Throughput impact of data path encapsulation

Encapsulation does not only impact the processing delay, but also effective throughput in the virtual networks due to additional header overhead. Traffic analysis on data from backbone and enterprise networks [8], [9] shows that Internet

Average Packet Size	Encapsulation Header Overhead	
	VLAN (4 Byte)	PWE (26 Byte)
(min) 40 bytes	10.0%	65.0%
(mixed) 700 bytes	0.6%	3.7%
(max) 1500 bytes	0.3%	1.7%

TABLE II: Overhead introduced by VLAN and PWE encapsulation for minimal (40 bytes), full (1500 bytes), and typical average packet sizes (700 bytes).

traffic consists mainly of minimal sized packets just at or above 40 bytes (such as TCP ACK packets, consisting of headers only) or maximum sized packets according to the standard Ethernet MTU of 1500 bytes. This bimodality in the packet size distribution typically yields an average packet size between 500 and 900 bytes on links with a diverse traffic mix, often around 700 bytes [10].

In Table II we summarize the relative overhead caused by the proposed encapsulation formats (VLAN tagging and PWE) on traffic mixes with varying packet size distributions. For traffic consisting of small packets, PWE encapsulation obviously introduces an unacceptable high header overhead, resulting in reduced effective throughput for virtual networks. Even for typical Internet traffic, the extra virtualization overhead by using PWE is non-negligible. We conclude that VLAN tagging is the preferable choice of encapsulation, especially for traffic scenarios including small and medium sized packets. However, in some cases simple VLAN tagging is not sufficient, either due to scalability reasons[4] or the need for advanced features such as protection, a common requirement in carrier network. Regardless of the choice of encapsulation technology, further potential effects on end-user performance need to be considered, such as packet fragmentation or packet drops caused by exceeding the network MTU due to the extra header overhead. However, these considerations concern all encapsulation techniques, and can be mitigated by appropriate MTU configuration and mechanisms such as Path MTU discovery.

V. Conclusions and Future work

We have described the implementation of an OpenFlow virtualization scheme designed to support carrier-grade network environments, in particular, multi-tenancy use-cases. By placing the hypervisor/translation functionality inside the switches, our virtualization scheme achieves inherently better availability compared to centralized solutions. The scheme can be implemented by two simple modifications: firstly, a minor software update to the OpenFlow message processing part of a switch, thus keeping the hardware unchanged; secondly, a simple experimental extension to the protocol itself. For virtual network customers, the virtualization system is completely transparent and no changes are required to either control software or addressing schemes. Not only were we successful in implementing the system, but our evaluation shows that the benefits introduced by the systems outweigh the minor performance degradation. Data path performance in terms of additional forwarding latency is negligible with both implemented encapsulation mechanisms. However, depending on packet size distribution and the encapsulation mechanism, throughput can be impacted significantly.

As future work, we believe it could be benefitial to apply the scheme to the latest OpenFlow version (currenlty 1.3 [11]), especially to investigate the possibilities of even stricter bandwidth isolation using new QoS tools, such as meters, in the virtualization table. Results could show whether current QoS mechanisms are sufficient or not [6]. Another future study item would be scalability considerations of the system in terms of flow tables, group tables, and meters. These discussions depend heavily on the use-case in question as well as upcoming Openflow hardware implementations. As discussed in [4], there are various possibilities when it comes to the placement of the translation unit in an SDN architecture. In our implementation, it is located between the OpenFlow protocol and the OpenFlow software instance on the switch. It would be interesting to investigate the impact of moving the translation unit closer to the forwarding hardware, placing it between multiple OpenFlow software instances running on a switch and the forwarding hardware. This may further simplify the implementation and provide stricter isolation on the control channel level in a switch.

[4]VLAN supports 4096 virtual links, which could potentially be a limit.

Acknowledgment

This work was funded by the EU FP7 project SPARC. The authors would like to thank all SPARC partners for discussions and comments, in particular Manxing Du and Alisa Devlic.

References

[1] Open Networking Foundation (ONF). [Online]. Available: http://www.opennetworking.org/

[2] The SPARC consortium. EU FP7 Project SPARC: Split Architecture for carrier grade networks. [Online]. Available: http://www.fp7-sparc.eu/

[3] M. Forzati, C. Larsen, and C. Mattsson, "Open access networks, the Swedish experience," in *Transparent Optical Networks (ICTON), 2010 12th International Conference on.* IEEE, 2010, pp. 1–4.

[4] P. Skoldstrom and K. Yedavalli, "Network virtualization and resource allocation in openflow-based wide area networks," in *Proceedings of SDN'12: Workshop on Software Defined Networks*. IEEE ICC, 2012.

[5] The SPARC consortium, "Deliverable D3.3: Split Architecture of Large Scale Wide Area Networks," 2012. [Online]. Available: http://www.fp7-sparc.eu/assets/deliverables/SPARC_D3.3_Split_Architecture_for_Large_Scale_Wide_Area_Networks.pdf

[6] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[7] Open Networking Foundation (ONF), "Openflow switch specication version 1.1.0," 2011. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf

[8] W. John, *Characterization and classification of internet backbone traffic*. Disseration, Chalmers University of Technology, 2010.

[9] D. Murray and T. Koziniec, "The state of enterprise network traffic in 2012," in *Asia-Pacific Conference on Communications (APCC)*, 2012.

[10] CAIDA: The Cooperative Association for Internet Data Analysis. Trace statistics for caida passive oc48 and oc192 traces. [Online]. Available: http://www.caida.org/data/passive/trace_stats/

[11] Open Networking Foundation (ONF), "Openflow switch specication version 1.3.1," 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf