Virtual Switching in Windows

Maximilian Endraß
Betreuer: Daniel Raumer, Sebastian Gallenmüller
Seminar Future Internet SS2016
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: endrass@in.tum.de

ABSTRACT
In the field of virtualization, network virtualization tries to ensure mobility and flexibility of virtual machines. One of the key concepts of network virtualization is *virtual switching*, i.e. emulating the behavior of hardware switches in software for general purpose servers or computers; two of those virtual switches are Microsoft's Hyper-V virtual switch, used in the Hyper-V hypervisor and Open vSwitch which also supports Hyper-V (among other hypervisors). In this paper we are going to analyze and compare them with focus on aspects relevant for performance and evaluate the functionality and performance aspects of running Open vSwitch as a virtual switch for Hyper-V.

Keywords
Virtual switch, Open vSwitch, Hyper-V Virtual Switch, SDN, Network virtualization, OpenFlow

1. INTRODUCTION
Network virtualization is meant for datacenters to consolidate their networks to make them more manageable and enable ”cloud” technologies. Virtual switches are a centerpiece of this approach as they begin to include more and more functionality that was previously reserved to higher-level devices or software and unify network hardware across entire datacenters into ”one big switch” [9].

Both for network virtualization and ”traditional” single machine virtualized environments, the feature set and the performance of virtual switches integrated into those physical hypervisors is a crucial point of the entire setup, especially for VMs communicating internally (e.g. as web server and database server). Two virtual switches that have undergone important changes during the last years to increase their virtual network and on-host functionality and performance, are Microsoft's Hyper-V virtual switch and the open-source Open vSwitch. They are arguably common software switches on Windows and both take seemingly different approaches on several aspects of their virtual switch functionality, so in the rest of the paper we are going to take closer a look at them

In Section 2.1, we are going to define the concepts of virtual switches in general and then look at them included in virtualized networks in Section 2.2. In Sections 2.3 and 2.4, we are going to analyze Open vSwitch and Hyper-V virtual switch respectively and conclude by comparing their features and performance in a small benchmark in Section 3 and have a look at Open vSwitch running on Hyper-V and (partly) replacing the integrated Hyper-V virtual switch in Section 3.3.

2. VIRTUAL SWITCHES
Before we compare the virtual switch solutions on Windows, we will explain what virtual switches are and in which particular environments they are used.

2.1 Concept
In virtualized environments, virtual machines created and managed by hypervisors (like VMware ESXi, Microsoft Hyper-V, Xen or KVM) share the same physical environment (i.e. they reside on one physical machine) [20]. To be able to communicate with each other (and with the network outside of the hypervisor) with the same protocols that are used in physical networks, the VMs usually have one or more virtual network interface cards (NICs) with their own MACs and IPs which are connected to a port on a *virtual switch* running on the hypervisor (usually on commodity hardware [9]) which provides connectivity to the outside physical network [20]. These integrated software switches were the first forms of virtual switches, which, in their early days, usually only provided very basic functionality and served no purpose other than extending the physical layer 2 network to the VMs [19].

However, with these early virtual switches, there was still a tight coupling between the traditional physical network, the virtual machines, and the hypervisor, which made the provisioning of new virtual machines cumbersome: In order to provide proper network connectivity to the VMs, the configuration had to be done both on the hypervisor virtual switch and the actual physical network the hypervisor was connected to [19]. Furthermore, coupling VMs with physical network segments renders some of the (theoretical) advantages of VMs, like easy scalability and mobility useless because of the effort required to reconfigure the network to redirect the traffic to a virtual machine's new physical location. From these circumstances the concept of *virtual networking* emerged and with it came more sophisticated virtual switches [19].

In virtual networks, virtual switches provide a greater part of the network connectivity for the VMs; actual physical networks are reduced to carrying tunneled packets between hypervisors or to the internet [19]. This allows for a decoupling of virtual networks from physical networks, while still being able to provide the same feature set as physical networks and a high level of flexibility and modularity [19]. By leveraging the capabilities of modern virtual switches, complex networks can be designed both inside a single hypervisor without needing dedicated network hardware at all, and across multiple hypervisors [21].

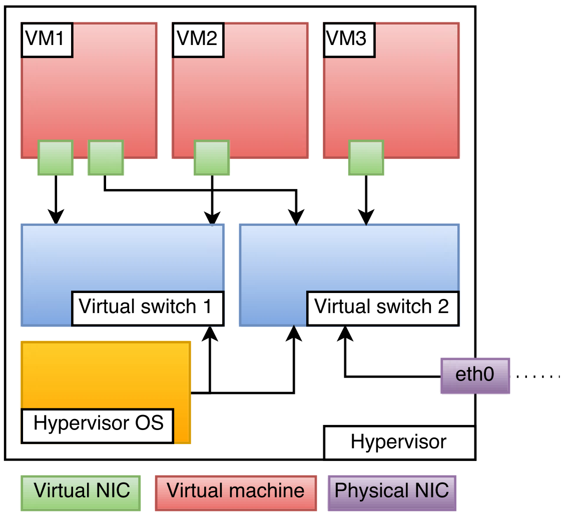

Figure 1: Illustration of a generic virtual switch.

The main task of virtual switches is still very similar to the main task of modern Ethernet switches, i.e. processing Ethernet frames, looking up the destination MAC address of each frame, forward the frames to one or more designated ports, all while learning and maintaining a MAC-to-port forwarding table and avoiding unnecessary transmissions (i.e. not acting as a hub) [21]. But modern virtual switches like Open vSwitch and Hyper-V Virtual Switch are capable of providing features way beyond simple layer 2 packet switching (e.g. router-like and firewall-like L3 and L4 packet filtering, network isolation through VLANs and more) [19]. Fine-grained centralized control over the virtual switches' rules even across different hypervisors is also possible for administrators [19], to be able to e.g. transparently move a virtual machine to a different physical host without network traffic getting lost, by simply assigning the VM to a different virtual port.

Figure 1 shows an example layout of a physical hypervisor running a hypervisor operating system and two virtual switches. Virtual switch 1 is an *internal* switch that connects VM1 and VM2 and the hypervisor OS, while virtual switch 2 is an *external* switch that connects VM1, VM3, the hypervisor OS and the physical NIC which serves as an uplink port to the external physical network. In this example, VM2 has no access to the physical network and VM3 cannot communicate with VM2.

2.2 Software Defined Networking (SDN)

The Open Networking Foundation describes *Software Defined Networking* as a network architecture where the "*control* and *data plane* are decoupled", effectively centralizing the network intelligence and abstracting the network infrastructure from the applications [16]. Virtual switches are a vital part of SDNs: Whereas in a conventional network there is a hierarchical tree structure of Ethernet switches, in an SDN the network "appears to applications [...] as a single, logical switch" [16], greatly simplifying even big network layouts.

The *OpenFlow* protocol "structures communication between the control and data plane" [16]. More specifically, the use of OpenFlow in an SDN allows "direct access to and manipulation of the forwarding (data) plane of network devices", like switches (both virtual and physical) and routers [16], therefore reducing a significant amount of protocols to a single one. Devices only have to understand and process OpenFlow instructions [16]. Therefore, OpenFlow can be compared to "the instruction set of a CPU" [16] and enables centralized and platform-agnostic programmability of networks. This way, OpenFlow can be used to configure simple L2 switching rules as well as more complex L3 or L4 routing and firewall rules.

This centralized networking approach enables a more scalable and flexible network design. For example, in order to transparently move a virtual machine from one physical host to another (as mentioned in Section 2.1), an administrator can simply assign the virtual machine to a different port on the virtual switch in the *control plane*, and the subsequent traffic (in the *data plane*) travelling through the actual physical network will reach its new destination without manually having to adjust different network hardware.

2.3 Open vSwitch

Open vSwitch (OVS) is a "multi-layer, open source virtual switch for all major hypervisor platforms" [19] that can be used both in commodity hardware and in switch hardware [13] and was designed from the ground up with virtual networking in mind. Its design departs from traditional hardware switches and early software switch implementations like the Linux bridge which only provided basic L2 connectivity between VMs and the physical network. Being an OpenFlow virtual switch, OVS was designed for "flexibility and general-purpose usage" [19] to work on many platforms and hypervisors while possibly having to share system resources with the hypervisor operating system and several VMs running on the physical machine. While OVS was originally designed for UNIX-like operating systems and hypervisors running on them (like Xen or KVM) [14], there have been efforts to port it to Microsoft Windows to run with the Microsoft Hyper-V hypervisor, which we cover in more detail in Section 3.3.

2.3.1 Architecture

The three main components of Open vSwitch are the *data-path* in the kernel, the userspace daemon `ovs-vswitchd` and the flow database `ovsdb-server`, also residing in userspace. The userspace daemon is mostly the same across all platforms, while the kernel module is specifically written for the host OS for performance and compatibility reasons [19]. Switch-level configuration, like bridge and interface layouts, are stored by `ovsdb-server` [13] (from which `ovs-vswitchd` pulls its configuration, cf. figure 2).

A basic overview of the functionality of OVS can be seen in Figure 2: A packet from a physical or virtual NIC arrives at the kernel module, then the datapath module checks whether a flow for this packet has already been cached. If the userspace daemon has already instructed the kernel module what to do with the packet (i.e. the respective flow has been cached), the kernel module will simply carry out the specified action (i.e. mostly forwarding the packet to the specified port(s), but other actions like dropping or modifying the packet are also possible). If no flow has been cached, the packet will be forwarded to the userspace daemon, which will then either process the packet locally (e.g. by getting the flow from `ovsdb-server` and then processing it) or outsource the request to a remote OpenFlow controller. Subsequently, the userspace daemon will forward the resulting flow to the kernel datapath, where the specified action can now be carried out and the flow can be cached [19]. In Figure 2 the green line (i.e. the "fast" path) represents the traversal of a packet through OVS, where a corresponding flow has already been cached, which is faster than the traversal path of a packet represented by the red line (i.e. the "slow" path) where the corresponding flow has not yet been cached [19].

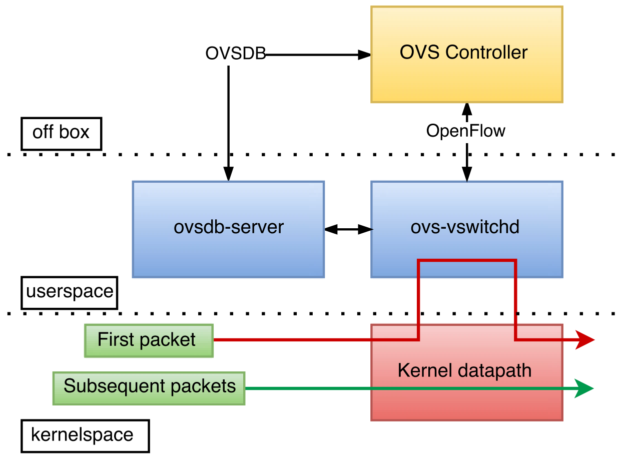

Figure 2: Open vSwitch architecture based on [19].

2.3.2 *Performance optimizations*

Traditional hardware switches are specified to achieve a certain worst-case line-rate performance with dedicated hardware. For OVS however, resource conservation is more critical, since the worst-case performance is secondary to the workload that is running on the user VMs on the hypervisor, and therefore OVS is optimized for common case usage [19] while still being able to gracefully handle worst-case scenarios (like port-scans) e.g. by caching rules.

Caching. Caching takes place in the kernel datapath module, which is effectively separated from the OpenFlow processing. This happens transparently for an OpenFlow controller, which just assumes the packet is going to be matched against the highest priority flow given in the flow table [19].

At first, OpenFlow only used so called *microflow caching.* This means the kernel cache is a simple hash table (without a packet classifier, which now resides in userspace) in which "a single entry exact[ly] matches all packet header fields supported by OpenFlow" [19]. Of course, this leads to cache entries being really specific and usually, with these specific entries only packets of a single transport connection were being efficiently forwarded.

Consequently, the microflow caching approach suffers from a loss of performance in scenarios with lots of short-lived connections, where many cache misses are produced, because for every cache miss, the packet has to be sent to the userspace daemon for classification, resulting in increased latency [19]. Therefore, the concept of *megaflow caching* has been introduced to solve this problem. A megaflow cache is a flow lookup (hash) table with *generic* (or "wildcarded") matching [19]. It mostly resembles an OpenFlow table without priorities. This way, the kernel can terminate the lookup as soon as one match has been found. However, to avoid ambiguity, the userspace daemon is only allowed to install disjoint (i.e. non-overlapping) megaflows in order not to apply the wrong actions because of wrong priorities [19]. To be able to match wildcard header fields while still maintaining a decent level of performance compared to a full packet classification in userspace, OVS performs a so called *staged lookup*: The hashes of four groups (stages) of the packet header (in decreasing order of granularity: metadata, like the switch ingress port, and L2, L3, L4 header data) are calculated and matched against the megaflow cache incrementally. The first hash is calculated only over metadata, the second hash over metadata and the L2 header and so forth [19]. This way, not all packets have to be scanned for L4 headers (e.g. TCP or UDP ports), even if some flows require it. In the end, this solution presents a trade-off, because it requires more hash table lookups than a microflow cache, but might avoid an expensive trip to userspace for packet classification. Microflow caching remains as a "first-level" cache to match a limited number of packets to their designated megaflow cache entry [19].

Packet classification. The wide variety of filtering possibilities offered by Open vSwitch in combination with OpenFlow [15] (packets can be checked against any combination, wildcard or range of MAC addresses, IPv4, IPv6, TCP/UDP header fields, etc.) make packet classification in userspace a rather expensive task on general-purpose processors [19]. OVS therefore introduced a concept called *tuple space search classifier* both for kernel and userspace packet classification [19], although in kernel space there are no priorites and several lookup tables are consolidated into a single table. With this approach, several hash tables are created for different combinations of header fields. For one packet classification (in userspace), every hash table has to be searched and the result with the highest priority will be selected as the resulting flow. Tuple space search classifiers offer constant-time updates (a single hash table operation) and their memory usage scales linearly [19].

Further improvements originate from *tuple priority searching*, where the lookup code is modified in such a way that tuples (i.e. rows of a flow table) are always sorted in descending order of their priorities. Thus, a lookup can always terminate once a matching flow has been found, since every other flow has to have a lower priority [19].

Lookup of IPs and IP ranges (for IPv4 and IPv6) is based on a trie lookup approach called *prefix tracking* which avoids having to scan all IPs. When traversing the trie (e.g. in Figure 3) starting from the root, the search can be terminated once the current matching node has no more leaves (e.g. in Figure 3 for an IP in `10.2.0.0/16`, the lookup can terminate once it reaches the node 2) [19].

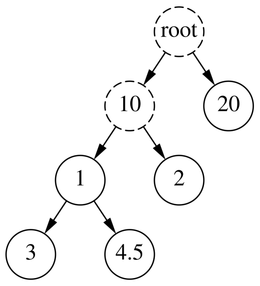

Figure 3: Example prefix tracking tree based on [19].

2.4 Microsoft Hyper-V Virtual Switch

The *Hyper-V role* built into the Windows Server operating system includes a hypervisor, VMbus (a mechanism to enable inter-VM and host-to-VM communication, also used by Hyper-V virtual switch), and, among other components, a "software-based layer-2 Ethernet network switch" [5] called *Hyper-V virtual switch* [10].

2.4.1 Design

Starting from Windows Server 2012, the Hyper-V virtual switch also supports third party extensions [17]. Hyper-V virtual switch (HVS) runs in the management OS (Windows Server or regular desktop editions of Windows starting from Windows 8 [10]) and provides three different kinds of virtual switches [17]:

- An *external switch* which supports a single physical network adapter and an arbitrary number of virtual network adapters, which makes it a comprehensive switch type, connecting all partitions of the host machine as well as the management OS itself (represented by its own virtual network adapter).
- An *internal switch* which supports the same features as an external switch except that it doesn't allow a phyiscal network adapter to be connected.
- A *private switch* which only connects virtual network adapters of Hyper-V child partitions (i.e. virtual machines). That means both physical adapters and the management OS (and applications running on it) can't connect to this switch type.

This design decision mostly serves isolation purposes and thus it is not possible to directly connect two physical network adapters to the same virtual switch. However, VMs can have multiple network adapters, possibly connecting them to multiple different virtual switches.

2.4.2 Packet flow through the virtual switch datapath

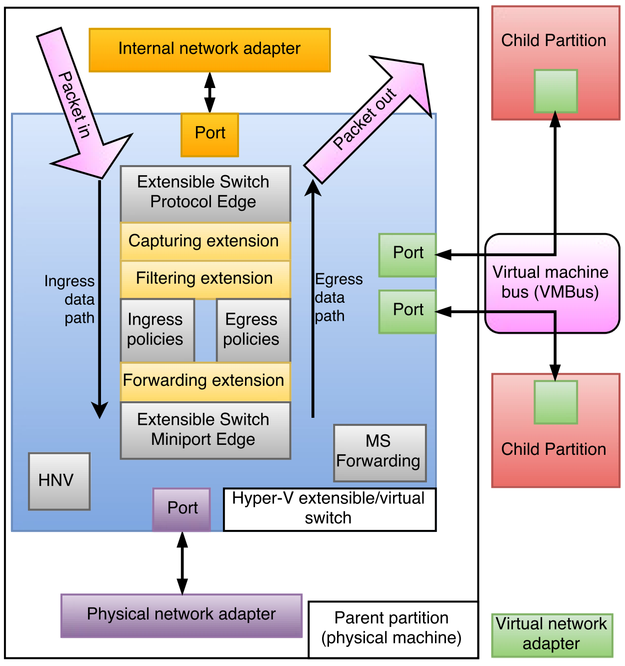

Figure 4: Hyper-V architecture layout based on [17].

The flow of a packet through the Hyper-V virtual switch and its extensions works as follows (cf. Figure 4 for a visual route of a packet) [17, 18]:

1. A packet is first registered at the *protocol edge* which prepares it for the ingress data path by allocating a *context area* that contains out-of-band (OOB) information about the packet, like the source switch port and the origin network adapter.
2. A *capturing extension* can capture and monitor the packet afterwards, but it cannot modify or drop it. It can, however, originate new traffic down the ingress path (e.g. to submit statistics to a monitoring application).
3. The following *filtering extension* is now able to inspect, modify and drop packets.
4. If there is no third-party filtering extension installed or enabled, Hyper-V virtual switch will now apply its built-in ingress policies to the packet before forwarding it to the forwarding extension: Filtering based on Access Control Lists (ACLs), DHCP Guard (to prevent rogue DHCP VM servers), Router Guard (to prevent VMs from pretending to be routers), and more.
5. The *forwarding extension* extends the capabilities of a filtering extension to the process of actually choosing a destination switch port for a packet. If

there is no forwarding extension installed, Hyper-V virtual switch will choose the destination port based on its standard (built-in) settings. This is also the position in the Hyper-V virtual switch data path, where the Windows implementation of Open vSwitch operates [12].

6. Arriving at the *miniport edge*, the Hyper-V virtual switch will apply its built in *port* policies (port based ACLs and Quality of Service, QoS) to the packet and possibly mirror it (if the respective setting has been enabled) by adding additional switch ports to the packet's OOB metadata information. If at this point the packet has not been dropped, it will be forwarded to the *egress* data path.
7. Along the egress data path, the destination switch ports are visible to all extensions. The packet now passes all stations of the ingress data path in reverse order: The forwarding extension can exclude destination switch ports from receiving a packet by removing them from the OOB metadata, then the built-in egress policies are applied to the packet if there are no respective extensions installed.
8. The filtering extension can now drop packets based on the switch destination port. If it modifies the packet, however, it has to clone it and inject it into the ingress data path without destination ports. The subsequent capturing extension can, once again, only analyze the packet and possibly inject new packets on the ingress data path.
9. Then the packet arrives at the overlying protocol edge again and is finally sent to its destination switch ports.

It is possible to install multiple capturing and filtering extensions per instance of Hyper-V virtual switch, but only one forwarding extension [17]. Following the ingress data path, the capturing and filtering extensions cannot see the packet's destination switch ports (because they have not been determined yet). These will later be added to the packet's OOB metadata either by a forwarding extension or by the virtual switch itself, and are visible to extensions along the egress data path.

Other than writing extensions as native NDIS (Network Driver Interface Specification, an API to communicate with network cards on Windows [11]) filter drivers, software vendors can use existing WFP (Windows Filtering Platform, a set of APIs that could previously be utilized by developers to write firewalls or intrusion detection systems for Windows [25]) filter applications to inspect, modify and drop packets along the HVS data path [17]. WFP filter applications operate between filtering and forwarding extensions [26] and usually run in userspace as opposed to NDIS drivers [8]. This possibly enables these kind of network security applications to be transferred from the VMs themselves to a VM-independent level on the underlying network architecture.

2.4.3 Hyper-V network virtualization

Hyper-V network virtualization offers a network virtualization or SDN approach centered around the Hyper-V platform. Hyper-V virtual networks (HVNs, identified by a single *Routing Domain ID*, RDID) ”consist of one or more *virtual subnets*” [6] (identified by *Virtual Subnet IDs*, VSIDs). Virtual subnets represent isolated parts within a virtual network, in which VMs within a virtual subnet have their own client IP addresses (CAs) and use them to communicate with each other, while the traffic is transparently transported over the physical network infrastructure using physical addresses (PAs). This ensures that one virtual subnet can in fact be distributed among multiple hosts and subnets on the physical network infrastructure [6].

Virtual subnets allow inflexible applications, that might rely on specific IP and general network configurations, to be brought to the cloud, without having to heavily modify them, because they can be assigned their own virtual network that is isolated from the actual network infrastructure and can thus have IPs that would otherwise collide with IPs already existing on the datacenter network (customers can ”bring their own IPs” [24]).

This approach is realized by using NVGRE (Network Virtualization using Generic Routing [3]) to encapsulate layer 2 packets into layer 3 packets (cf. Figure 5) and send them across multiple collision domains in the network [3]. Since this process happens inside HVS, it is transparent for the VMs.

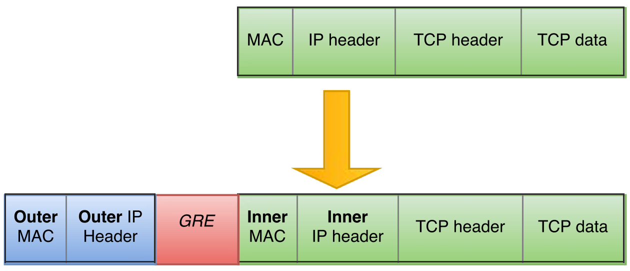

Figure 5: L2 packet encapsulated using NVGRE (based on [3]).

The HNV module inside the Hyper-V virtual switch (cf. Figure 4) now enables extensions to view both the PA and the CA of an encapsulated packet, as it encapsulates and decapsulates NVGRE traffic [6]. That means extensions will receive the encapsulated NVGRE packet on the ingress path and the decapsulated packet on the egress path (because the HNV module operates alongside the miniport edge, cf. Figure 4) or the other way around if the packet is not received from a tunneled connection, but is to be sent to one [18].

3. COMPARISON

Now that we know details about the respective design and datapath layout of both switches, we will compare them in the following Sections.

3.1 Design

Open vSwitch follows a more generic design idea, focusing on the power and programmability of the standardized OpenFlow protocol acting as an "instruction set" [16] of the switch, while the Hyper-V virtual switch enforces a more traditional, pipelined approach of packet processing, although in recent versions it can be extended by filtering and even forwarding extensions (see Section 2.4.2), basically taking control of the switch forwading logic. The current built-in ACLs of Hyper-V virtual switch offer matching for basic header fields (protocol, source & destination MAC, IP, TCP/UDP ports) [23], while OpenFlow basically matches almost all possible packet header fields [15] and offers more actions than the standard "accept"/"deny" (e.g. forwarding to a remote controller). Though it is not as straight forward, functionality like this can be added to HVS through extensions. In fact, in Section 3.3 we are going to discuss an OpenFlow-enabled OVS implementation for HVS. Since OVS is Open Source [14], obviously it can also be extended as such, though it doesn't offer dedicated extension support.

Both virtual switches also offer their own take at the SDN approach, with OVS based around OpenFlow and a centralized OpenFlow controller (see Section 2.2), and HVS based on the concept of "Hyper-V virtual networks" using NVGRE-tunneled packets to communicate (see Section 2.4.3).

For a concise, tabular comparison of the features of both switches, refer to Table 1.

3.2 Performance

The respective design of both virtual switches also impacts their performance which we will analyze in the following Sections.

3.2.1 *Theoretic considerations*

While the caching approach of OVS has already undergone several changes, culminating in a generic megaflow cache with a "first-level" microflow cache (see Section 2.3.2), the caching policies of the Hyper-V virtual switch are not thoroughly documented, although there are approaches to optimize performance for it. The 2012 R2 version of Windows Server introduced virtual RSS (Receive Side Scaling) which "enables network adapters to distribute the kernel-mode network processing load across multiple cores" [7] while maintaining cache locality [9] since in the past there have been problems achieving the full speed of high-throughput network adapters ($\geq$ 10 Gbps) [22]. Another performance improvement presented is *IPsec task offloading* which moves the encryption of network packets from the VMs (i.e. the CPU) to physical NICs [22]. For multiple teamed (virtual) NICs, Hyper-V virtual switch offers load balancing to distribute the load among them as equally as possible [23]. Dynamic VM queue (dVMQ) is another feature introduced in Windows Server 2012, which aims to speed up the HVS performance. It queues up traffic for (virtual) NICs by hashing the destination MAC and putting the traffic "destined for a virtual NIC into a specific queue" [22] mostly tied to a single core to avoid unnecessary CPU interrupts by processing the traffic on random cores.

Overall, OVS has fewer steps in its entire pipeline (cache lookup, possible userspace packet classification, cache update, and applying the action specified through the matched flow, see Section 2.3.1) than HVS, where the packet has to possibly traverse three extensions twice (along the ingress and egress path, see Section 2.4.2) in addition to being analyzed by the switch itself at the miniport edge. OVS has a more generic caching approach (megaflow-cache) to ensure decent processing speed, while HVS introduced a lot of specific, often hardware-related techniques, like vRSS or dVMQ.

3.2.2 *Practical benchmark*

Setup. To compare the performance between Hyper-V virtual switch and Open vSwitch, we tested them to see how they perform under load. The test machine was equipped with an Intel Core i5-2500K CPU (4 cores clocked at 4.1 GHz with no Hyper-Threading). To test Hyper-V virtual switch we used Windows Server 2012 R2 with Hyper-V as a hypervisor, for Open vSwitch we used Debian 8.3 jessie (kernel 3.16) with KVM as a hypervisor (`libvirt` 1.2.9, QEMU 2.1.2). For both tests we used 2 VMs with Debian 8.3 jessie and kernel 3.16 running as guests on the hypervisors and both were assigned 1 CPU core and 1GB RAM each. The virtual network adapters were provided by the respective hypervisors. As a traffic generator we used `trafgen`, and to measure packets on the second virtual machine we used `ifpps`, both part of the `netsniff-ng` package [2]. Trafgen was used to send as many empty identical ICMP packets (42 Bytes per packet, 64 Bytes with padding on wire due to Ethernet frame size requirements) as possible from VM1 to VM2. The reason we used ICMP and not UDP packets was that those did not provoke ICMP "Port Unreachable" replies from VM2, possibly slowing it down.

Measurement. From the result (cf. Figure 6) we can observe that in a direct comparison of 10,000,000 ICMP packets sent with `trafgen` from VM1 to VM2 in each scenario, Open vSwitch achieves an average throughput of 764,696 packets per second (pps) or about 0.76 Mpps, while Hyper-V achieves an average of 554,470 pps or about 0.55 Mpps. Interestingly, HVS's initial throughput is about 0.1 Mpps higher than that of OVS, but the overall throughput of OVS is higher. This can probably be attributed to the caching strategy of OVS, where after about 1 second the megaflow cache has been set up and packets do not have to pass the userspace anymore. HVS's caching strategy is not documented, but it seems to employ one nevertheless, given that the initial performance rises, fluctuates and even slightly drops at 10s, but then settles at about 0.6 Mpps.

Table 1: Feature comparison

Feature	Open vSwitch	Hyper-V virtual switch
Pipeline	Short, generic, programmable with OpenFlow	Long, arbitrary, with 3 extensions
Extensions	Open source extensibility	Extension API with 3 types
UNIX integration	Designed for UNIX-based operating systems	n/a
Windows integration	Beta port with increased managerial complexity	Designed for Windows Server
Caching strategy	Megaflow and Microflow cache	n/a
SDN approach	OpenFlow integration	Hyper-V virtual networks with NVGRE

The performance comparison was meant to be carried out in a very similar environment, and while the hardware, the VM configuration and VM software were the same during both tests, the fact that different operating systems and hypervisors were used (because, unfortunately, as of now, we were not able to test the Windows version of OVS) should be taken into account.

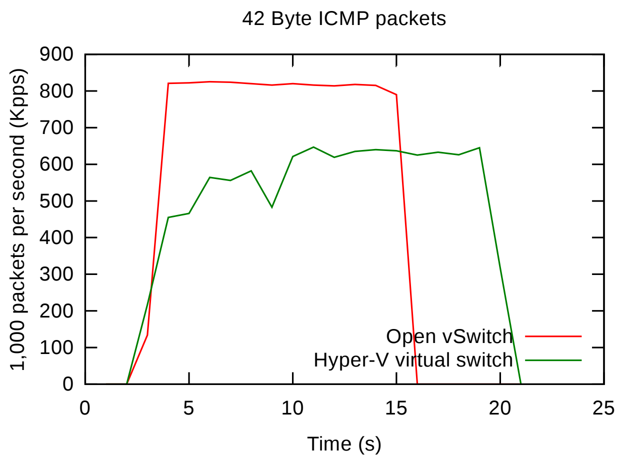

Figure 6: Switch performance comparison

3.3 Open vSwitch running on the Hyper-V hypervisor

Although OVS does not natively support the Hyper-V hypervisor, there is an OVS fork with Windows and Hyper-V support, developed by Cloudbase and VMware [4, 12]. The userspace code was, apart from minor tweaks, mostly left intact, but the kernel module had to be rewritten from scratch due to the fundamentally different architecture of Windows and UNIX-based operating systems. It was designed as an NDIS filter driver (`OVSEXT`) "implemented as a forwarding extension" [12] (cf. Section 2.4.2) so as to leverage the extension framework provided by Hyper-V and replace (almost) the entire built-in switching logic.

In order to make as few changes to the userspace daemon as possible, the developers opted for an emulation of Linux netlink sockets under Windows to let a pseudo device (representing the kernel) communicate with the userspace [12]. Part of the kernel module has also been optimized by using *zero-copy* to only transfer references to the memory location of a packet to userspace instead of deep copying the entire packet [12].

In terms of packet flow, the extension behaves like Open vSwitch: A packet is received by the OVS forwarding extension on the ingress data path, the kernel module of OVS will calculate its hash(es) and look for a cached flow. If there is no such flow, then the packet will be sent to userspace with an upcall, processed there and sent back to the kernel module where the new flow will be installed. Afterwards, the processed packet will be sent "back" onto the HVS egress data path [12]. To keep switch ports in sync between the OVS userspace components, the OVS kernel datapath and HVS, an additional field name has been added to HVS ports which is synchronized between the three components [12].

The inclusion of OVS into Hyper-V as a forwarding extension extends the HVS pipeline even further and thus increases the packet processing overhead and adds the necessity for two basically different virtual switches to stay synchronized. However, since a forwarding extension replaces a substantial amount of default functionality of HVS, we might also see slightly increased performance due to the megaflow cache. Furthermore, it offers datacenter operators a possibility to integrate the Hyper-V hypervisor into their lineup even if their virtual network is reliant on OpenFlow.

4. RELATED WORK

On the topic of virtual switch performance analysis, Emmerich et. al. [9] quantitatively analyze the performance of Open vSwitch compared against other Linux virtual switches or software bridges in a test environment, while Pfaff et. al. [19] look at the real-world performance of Open vSwitch deployed in a commercial Rackspace datacenter. Since we couldn't possibly cover all hardware-offloading features and they are also often subject to change, information on those can be found at the respective virtual switches' websites [5, 14]. A more in-depth look at the SDN features of OVS (also focusing on the integration into OpenStack - a comprehensive software platform for "cloud computing") and HVS is given in talks by Pettit [13] and Williams [24] respectively. The source code and integrated documentation of Open vSwitch [14] and the Windows implementation of OVS [4] is also available. There are, of course, completely different virtual switches with other internals than the two covered in this paper. Commercial solutions by Cisco [1] and VMware [21] might be of particular interest in this case.

5. CONCLUSION

Hyper-V virtual switch has long been the only notable virtual switch for the Hyper-V platform, but its extension support spawned alternatives, deeply integrated into HVS, and Open vSwitch is among them. While OVS (in its native UNIX-like environment) has an edge on HVS in terms of raw performance (as of 03/2016, cf. Section 3.2), both switches perform well even when faced with several hundreds of thousands of packets. Naturally, OVS is more prevalent in Linux-based environments and datacenters, while Hyper-V and HVS provide several Windows-specific capabilities. The OVS implementation for Windows discussed in Section 3.3 tries to bridge the gap between both worlds, albeit with a slight increase in managerial complexity (due to having to manage two different switches). Nevertheless, this approach can help the consolidation of traditional networks into virtual networks, since it extends the availability of a common protocol to another platform.

6. REFERENCES

[1] Cisco nexus 1000v switch for vmware vsphere. `http://www.cisco.com/c/en/us/products/switches/nexus-1000v-switch-vmware-vsphere/index.html`, last visited 2016/05/01.

[2] netsniff-ng homepage. `http://netsniff-ng.org/`, last visited 2016/03/26.

[3] Network virtualization using generic routing encapsulation (nvgre) task offload. `https://msdn.microsoft.com/en-us/library/windows/hardware/dn144775(v=vs.85).aspx`, last visited 2016/03/17.

[4] Open vswitch for hyper-v (fork). `https://github.com/cloudbase/ovs`, last visited 2016/03/21.

[5] Hyper-v virtual switch overview, September 2013. `https://technet.microsoft.com/en-us/library/hh831823.aspx`, last visited 2016/03/16.

[6] Hyper-v network virtualization technical details, June 2014. `https://technet.microsoft.com/en-us/library/jj134174.aspx`, last visited 2016/03/17.

[7] Virtual receive-side scaling in windows server 2012 r2, October 2014. `https://technet.microsoft.com/en-us/library/dn383582.aspx`, last visited 2016/03/18.

[8] B. Combs and L. Hernandez. Extending the hyper-v switch, November 2011. `https://channel9.msdn.com/Events/BUILD/BUILD2011/SAC-559T`, last visited 2016/03/18.

[9] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle. Throughput and latency of virtual switching: A quantitative analysis. 2016.

[10] Hyper-V Overview (Microsoft Technet), March 2015. `https://technet.microsoft.com/en-us/library/hh831531.aspx`, last visited 2016/03/15.

[11] NDIS Drivers. `https://msdn.microsoft.com/en-us/library/windows/hardware/ff565448(v=vs.85).aspx`, last visited 2016/03/18.

[12] Open vSwitch 2014 Fall Conference: Open vSwitch on Hyper-V: Design and Development Status, 2014. `https://www.youtube.com/watch?v=T5gGD2YQqPc`, last visited 2016/03/19.

[13] Open vSwitch Deep Dive - The Virtual Switch for OpenStack, November 2013. `https://www.youtube.com/watch?v=x-F9bDRxjAM`, last visited 2016/03/14.

[14] Open vSwitch website, March 2016. `http://openvswitch.org/`, last visited 2016/03/19.

[15] OpenFlow 1.4.0 specification, October 2013. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf`, last visited 2016/03/19.

[16] OpenFlow Whitepaper, April 2012. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf`, last visited 2016/03/13.

[17] Overview of the Hyper-V Extensible Switch. `https://msdn.microsoft.com/de-de/library/windows/hardware/hh582268(v=vs.85).aspx`, last visited 2016/03/18.

[18] Packet Flow through the Extensible Switch Data Path. `https://msdn.microsoft.com/en-us/library/windows/hardware/hh582269(v=vs.85).aspx`, last visited 2016/03/18.

[19] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[20] H. M. Tseng, H. L. Lee, J. W. Hu, T. L. Liu, J. G. Chang, and W. C. Huang. Network virtualization with cloud virtual switch. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 998–1003, Dec 2011.

[21] VMware Virtual Networking Concepts, 2007. `https://www.vmware.com/files/pdf/virtual_networking_concepts.pdf`, last visited 2016/03/11.

[22] What's New in Hyper-V Virtual Switch in Windows Server 2012, March 2014. `https://technet.microsoft.com/en-us/library/jj679878.aspx`, last visited 2016/03/16.

[23] What's New in Hyper-V Virtual Switch in Windows Server 2012 R2, August 2013. `https://technet.microsoft.com/en-us/library/dn343757.aspx`, last visited 2016/03/16.

[24] C. Williams. Deep dive on hyper-v network virtualization in windows server 2012 r2, June 2013. `https://channel9.msdn.com/events/teched/northamerica/2013/mdc-b380`, last visited 2016/03/17.

[25] Windows Filtering Platform. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366510(v=vs.85).aspx`, last visited 2016/03/18.

[26] Windows Server 2012 Hyper-V Component Architecture Poster. `https://www.microsoft.com/en-us/download/details.aspx?id=29189`, last visited 2016/03/17.