Using libNetVirt to control the virtual network

Daniel Turull, Markus Hidell, Peter Sjödin
KTH Royal Institute of Technology, School of ICT
Kista, Sweden
Email: {danieltt,mahidell,psj}@kth.se

***Abstract*—LibNetVirt proposes an architecture for a network virtualization abstraction using the single node representation model. LibNetVirt is deployed as a library, similar to libvirt in computer virtualization, with a unified interface towards the underlying network specific drivers. The architecture allows management tools to be independent of the underlying technologies. In addition, it enables programmable and on-demand creation of virtual networks. We have evaluated libNetVirt in an OpenFlow-enabled network in three different tests: the setup time of a flow, the behavior of the system under a Denial of Service attack and the packet losses in high rate UDP flows.**

I. Introduction

Network virtualization is a mechanism for sharing a physical network. Virtual networks have been around for several years but nowadays they are one of the main focuses in the networking community. Cloud networking is one of the key drivers for the evolution of virtual networks, since it stresses the requirements for the network.

An important aspect to add flexibility to systems is the abstraction of the underlying technologies. A successful example of such abstraction is machine virtualization, where computers are represented in a straight-forward fashion as CPU, memory, storage, and network interfaces. Another example is disk virtualization, where logical volumes are presented instead of physical disks. Both types of virtualization are fairly well understood and agreed upon. The common denominator here is that the logical view is decoupled from the physical hardware, which allows resources to be shared.

However, virtualizing networks is a different and more complex challenge. Networks are still coupled with the physical infrastructure [1]. There are several technologies for network virtualization, such as, VLANs, VPNs or virtual routers; however they require a significant degree of manual configuration to be done by the network operator, something which makes the adoption to dynamic environment quite impractical.

libNetVirt [2], a previous work from the authors, defines a common set of network abstractions to allow users to easily provide programmable network resources in a similar way to how machine virtualization is done. It provides a unified framework for network virtualization is necessary to expand network functionalities and to provide a single network view. This will lead to environments with shared resources and reduced costs.

LibNetVirt can be used to control different kinds of network, such as, OpenFlow-enabled [3], inside a datacenter to create virtual networks on demand. In a datacenter, several users share the network resources to keep costs down but they still want their communications isolated from the rest of the datacenter users. LibNetVirt offers a tool to simplify the management of the shared network.

In related work, others have focused on the creation of complete networks graphs with virtual links and routers on top of a physical infrastructure [4]–[6]. These require a complete description of the desired topology and elements, which makes the management of the virtual infrastructure unnecessarily complex. Another alternative, proposed by Keller and Rexford [7], is to present the network as a single router, where all in-network functionality is covered. It has advantages for both players: users and network providers. Users do not need to manage the physical network to run their services and network providers can offer their platform with an added value. LibNetVirt uses the single router abstraction to describe a virtual network.

This paper is a continuation of the architecture work presented in [2]. The architecture has been extended, adding capabilities to support L3 networks. In addition, a performance evaluation of the open source [8] prototype on top of an OpenFlow-enabled network has been done.

The rest of the paper is organized as follows. Section II summaries the libNetVirt architecture. Section III briefly explains some of the interiorities of the current implemented drivers. Section IV evaluates the performance of our prototype in three different tests. Section V discusses the related work. Finally, in Section VI we conclude our work.

II. LibNetVirt

LibNetVirt [2] is a C library with Python wrappers, which provides an abstraction for the network where the minimum requirement for the user is to provide the endpoints to interconnect. This section describes the network view, the main elements of the architecture and the basic operations.

A. Virtual Network View

Traditionally, the approach taken when dealing with Virtual Networks (VN) is to map virtual nodes and links (vertices and edges) to physical nodes and links. This provides full management capabilities to the users [7], providing a complete graph of the network view. However, users would need to manage the virtual network in the same way as a regular physical network would be managed, for instance, providing traffic engineering or coping with link failure. We believe that users who want to manage their application in dynamic environments do not need the added complexity of managing

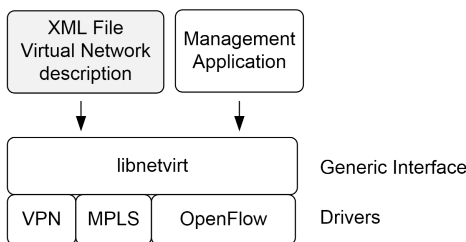


Fig. 1. LibNetVirt architecture

a complete network. The single router abstraction, addresses this problem defining the network as a unique node where all the functionalities are inside. All actions are defined in this single element, where all the complexity is hidden from the user, providing an easy way to interact with network resources.

B. Architecture

LibNetVirt is composed by two different parts: generic interface and drivers (See Fig. 1). The generic interface is a set of functions that allow interacting with the VN and executing the operations. A driver is a technology dependent element which communicates with the required components to manipulate the VN in the physical equipment.

Our view is that libNetVirt is used by the management application to operate the network. The information of a Virtual Network can be provided with an XML description file or invoking directly the API of the library. LibNetVirt processes the request and invokes the specific function of the correct driver. We have identified the following parameters that can describe a VN: *endpoint*, *forwarding* and *path constraint*. However, it is possible to further extend these definitions and define additional parameters.

- **Endpoint**: is the termination of a VN. A resource is connected to an endpoint. An endpoint is described with multiple fields, where some are mandatory and others optional. The mandatory fields are:
  - *uuid*: unique identifier for the endpoint in the VN, which is used to identify the endpoint.
  - *switch id*: identification of the physical switch.
  - *port*: edge port, where the resource is connected.
  - *address*: used to define the interface address when doing L3 forwarding, otherwise not required.

  In addition, other packet fields, such as, VLAN tags or MPLS labels can be used to distinguish traffic from different VNs that use a shared port.
- **Forwarding**: defines the type of packet forwarding. We can have forwarding based on L2 (based on MAC addresses) or L3 (based on IP addresses). It can be extended with new forwarding types.
- **Path constraints**: defines unidirectional constraints between two endpoints. A constraint can be any QoS specification required in the VN, such as, minimal bandwidth between two endpoints. It contains the *uuid* for the source and destination endpoints and the QoS constraint.

C. Basic Operations

LibNetVirt defines some basic operations to manage a VN. They can be executed manually by a network operator or programmable from the user interface. The programmable executions are called from upper layers of the management platform. We have identified the following basic operations, which might be further extended.

- Creation of a VN: the user defines the desired VN in an XML file or through the libNetVirt API.
- Removal of a VN: the user needs to provide the *uuids* of the VN that wants to remove.
- Addition/Modification/Removal of an endpoint: the user adds the description of the endpoints that wants to add or remove.
- Addition/Modification/Removal of a path constraint, in order to manipulate on demand QoS constraints.

III. LIBNETVIRT DRIVERS

Currently, libNetVirt supports two drivers for different technologies and different forwarding methods: an OpenFlow driver for a L2 networks, based on MAC addresses, and an MPLS driver for L3 Virtual Private Networks, where each endpoint belongs to a different IP network.

A. OpenFlow driver

We have implemented a libNetVirt driver for OpenFlow 1.0. OpenFlow [3] is a novel architecture which allows decoupling the data plane from the control plane. Basically, it is a protocol for the communication between a remote controller (control plane) and a switch (data plane). The packet forwarding is based on flows and it can be customized to any set of fields of a packet. The controller is in charge of installing the forwarding rules to the switch.

The OpenFlow driver uses NOX [9] as a platform to control the network. The forwarding rules are not pre-installed in the switches by default. This causes a small delay in the first packet of a flow. The switch needs to send a packet to the controller with the information of the first packet and then the controller sends a response with the forwarding decision. The controller bases the answer on the Virtual Networks (VNs) that are instantiated with libNetVirt. In other words, it will send the forwarding rule only if the ingress endpoint belongs to a VN that is deployed, otherwise it will drop the packet. The controller only sends packets to endpoints that belong to the same VN. Once the controller knows source and destination, it installs the forwarding rules to all in-path switches to minimize the setup delay. Further details can be found in [2].

B. MPLS driver

According to the libNetVirt vision, we also provide support for legacy technologies, such as, MPLS networks, which is currently widely deployed in Wide Area Networks (WAN). Our driver for such networks is a set of scripts that send commands to the involved routers to set up the MPLS network. It configures the different interfaces as well as the protocols, such as OSPF and BGP, involved in the control plane of the

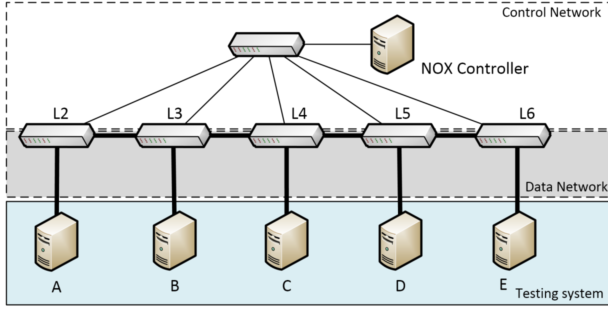


Fig. 2. Experimental setup. Each host is connected to a PC-based switch. The switches are connected in a chain to interconnect all the hosts (Data Network). The switches are connected via a switch to the OpenFlow NOX controller (Control network)

WAN network. It has been tested in a MPLS test bed running a commercial routing stack. A database contains the mapping between the *switch_id* and the access information of all the routers which are controlled by libNetVirt.

IV. Performance evaluation

This section describes some experiments done to evaluate the libNetVirt's OpenFlow driver. We have evaluated libNetVirt in an OpenFlow-enabled network in three different tests: the setup time of a flow, the behavior of the system under a Denial of Service attack and the packet losses in high rate UDP flows.

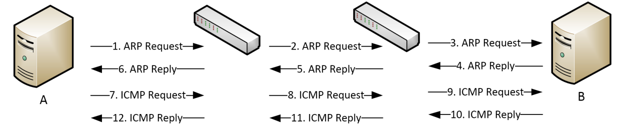
A. Experimental setup

The experimental setup is drawn in Fig. 2, where the purpose of the setup is to create multi-hop paths. The 5 switches and the OF controller are PC-based with Intel® Xeon® CPU X3450, 8 Gigabyte (4x2Gigabyte) of DDR3 (1333MHz) RAM and 8 Copper Port Intel® 82574L Gigabit network. The operating system is Ubuntu Server 11.10 64-bit. We use the Linux bridge module to establish a baseline and OpenVSwitch [10] (OVS) 1.4 when using OpenFlow.

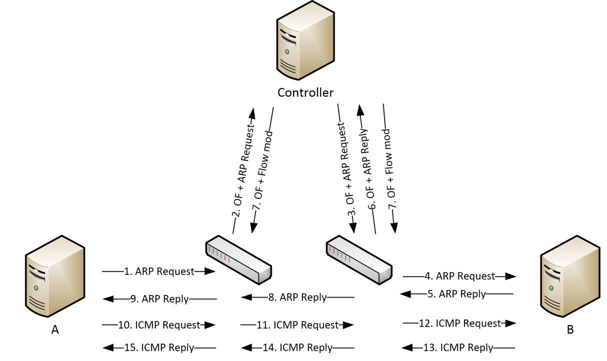
The transmitter and receiver are in the same machine and isolated with Linux namespaces (lxc). Each interface in bundled to a different processor with CPU affinity. This provides higher performance because the interrupts generated by different interfaces are processed independently.

B. Test 1. Setup time

A key difference between legacy switches and OpenFlow switches is the presence of a central controller in the latter case. Our approach with the OpenFlow driver is to configure the forwarding rules when the first packet arrives, introducing a setup delay for the first packet of the flow. If the initial delay is a critical aspect for the application, the OpenFlow driver could install the rules prior to the arrival of the first packet. In this case, the controller should know in advance the location of the hosts. One way to achieve this is to send a single packet from each host, just after the virtual network is



(a) Legacy mode



(b) LibNetVirt mode

Fig. 3. Packets involved in setup time. The main difference between legacy switches and the OpenFlow switches which are controlled by libNetVirt is when the first ARP request arrives (pkt 1) the switch forwards the ARP request encapsulated in an OpenFlow packet to the controller (pkt 2). Then the controller sends to the destination endpoint the same packet (pkt 3). When the host replies with the ARP reply (pkt 5), the packet is stored in a buffer in the switch and forwarded in a OF packet to the controller. The controller learns where the host is and installs the forwarding rules to all the switches in the path (pkt 7). After this, the packets follow the same path, since all the rules have been installed in the switches.

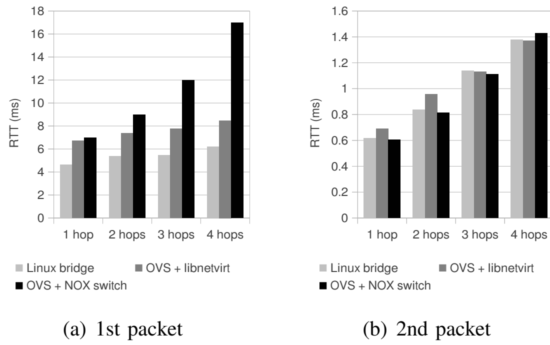
configured. Another option is to have a database with MAC address and endpoint location. Then, when the critical traffic is sent, the path is already installed, reducing the setup time.

The following test quantifies the setup delay between the switches, something which is a key aspect if a low latency in the flows is required. We use ping to measure the RTT for the first ICMP packet. The differences in terms of operation between OF switching and regular switching can be observed in Fig. 3.

We compare the setup time of a flow in three solutions:

- Linux bridge implementation as a baseline to compare the results with legacy switches.
- OVS with the libNetVirt application.
- OVS with the NOX controller running a learning switch application.

We send ICMP request from A to B, to C, to D and to E, with a packet size of 64 byte and 1 req/s. We used low rates because usually the beginning of the flows is slow. We repeated the tests 20 times for each solution and we compare the 90th percentile in Fig. 4(a). We can observe that the Linux Bridge has less delay than the other cases. This is the expected behavior since it does not need to forward the packet to the controller. However, libNetVirt provides the flexibility to create virtual networks on demand with only a slightly increase in packet delay for the first packet of a flow. On the other



(a) 1st packet (b) 2nd packet

Fig. 4. Setup time with ICMP packets. The OpenFlow communication introduces an extra delay in the first packet of a flow. The delay of the first and second packet between the different solutions is compared.

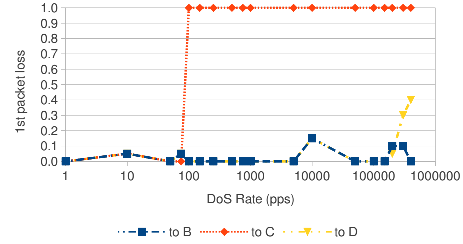


Fig. 5. Packet loss in a DOS attack. Probability of packet loss, when UDP traffic is injected in host C. We use ICMP pings to generate the legitimate traffic: A to B, A to C and A to D

hand, we observe that libNetVirt obtains better results than the learning switch implemented with NOX. The longer delays in the NOX switch can be explained by having all switches send the packet to the controller which introduces an extra delay for each switch in the path.

Also, we can observe in Fig. 4(b) that when the path is configured in the OVS, the RTTs are similar in all cases. Therefore we can conclude that at low rates only the first packet is delayed using OpenFlow. As said, this is an expected result of using OpenFlow. However, OpenFlow gives us the flexibility to modify the forwarding rules to create VNs.

C. Test 2. Denial of Service attack

The resilience against malicious sources is a key aspect in networking. We analyze the response of our system when a Denial of Service (DoS) attack is performed from an endpoint which does not belong to any VN. We want to quantify the effect of sharing the same network infrastructure with different users. This is useful to evaluate the isolation between users.

The traffic is generated with pktgen [11] and we send 64-byte UDP packets at a rate varying between 1 pps and 700 kpps. The traffic is injected in the switch L4, where C is also connected. The legitimate traffic flows from A to B, to C and to D. We measure the number of times that the first packet of a new flow is dropped due to congestion in the controller or switch. The test is repeated 20 times. The results are displayed in Fig 5, where we plot packet loss versus packet rate. We consider that a packet is lost when the delay is more than 1 second. After 1 second, the host retransmits the ARP request if it has not received a response. We observe that the results differ depending on where the endpoints of the flow are. We observe that if the switch is not in the path, the legitimate traffic is only affected when the DoS rate is high. On the other hand, if it is in the path, we start to observe packet loss with rates around 300kpps. The most affected traffic is the one with destination to C, which is where the malicious traffic is injected. The main reasons for losing packets in this switch are:

- OpenFlow packet is lost due to congestion in the switch or controller.
- Packet is lost due to buffer overload caused by the amount of packets waiting for an answer from the controller.
- Forwarding rules are installed too late. The time required to install the forwarding rule in the data path of a congested switch is longer, causing a loss when the packet arrives to the second switch because the incoming port is not an endpoint, and the controller drops the packet. OFlops' authors show in [12] that different forwarding rules have different installation times.

The controller processes all the incoming traffic, which may cause a bottleneck if a malicious client injects traffic. A possible solution to mitigate this attack is to install a drop rule in the ports where there are no endpoints. This will reduce the amount of traffic that the controller needs to process and the switch needs to send to the controller. We can point out that the installation of new flows gets affected by the number of new flows. A better isolation mechanism in the controller and the OVS is needed if we want to archive real isolation between users.

D. Test 3. Packet losses with UDP traffic

Packet loss is a key aspect in networking. It has negative effects on the performance of the network and the applications running on top and needs to be minimized. The setup delay might produce additional packet loss in UDP traffic if the rates are high. The packets need to be stored in the switch buffers while waiting for a response from the controller and there are a limited number of them. TCP traffic will not show similar behavior because it implements congestion control and the transmitter only sends one packet (TCP SYN) and waits for the response of the receiver. The TCP handshake gives time to the switch to install the rules before the bulk traffic is sent.

We want to quantify the packet losses in our system with different packet rates of UDP packets without having the forwarding rules installed. As a baseline, we measured the forwarding rate when the rules are already installed. Linux Bridge and OpenVSwitch can forward around 750 kpps of 64-byte UDP packets without any packet loss.

The test starts with empty ARP tables. Our controller does not need ARP request generated by the transmitter to locate the destination. When the first UDP packet arrives to the controller, it triggers an ARP request to discover where the

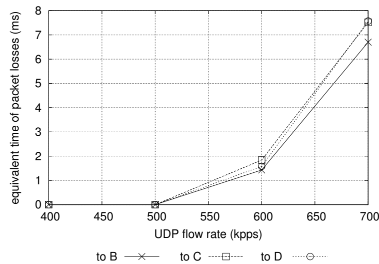


Fig. 6. Packet loss in a UDP burst (90th percentile). We use pktgen to send traffic from A to B, A to C and A to D. The equivalent time in packet losses is the number of packets divided by the generated rate.

destination is. Once the destination is known, the rules are installed. The flow is generated with pktgen and it is 10 seconds long with 64-byte UDP packets. We repeated the test 20 times and we show in Fig. 6 the 90th percentile of the equivalent time of packet losses, which is calculated from the number of packet losses and the generated rate. We should expect a constant time, which is the time to install a forwarding rule. However, we observe a packet loss that increases with the injected rate, which is due to the congestion in the switch.

We observe that the losses increase with the packet rate. All the losses are in the beginning of the flow, before the rules are installed. On the other hand, this is a worst-case scenario, where the client starts sending at full rate without establishing the path before. In addition, the losses only occur during the first milliseconds of the transmission. However, if the packet losses are critical, the path should be installed beforehand or TCP should be used.

V. Related work

OpenStack Quantum [13] is a project inside the OpenStack [14] community to provide network connectivity as a service. At the time of writing, it is work in progress and their authors are focusing on OpenStack platforms to interconnect virtual machines. LibNetVirt has a different vision, namely to be used also in other scenarios, such as, the interconnection between clouds. Furthermore, libNetVirt permits the specification of QoS constraints, such as, minimal bandwidth.

NECs Programmable Flow [15] is a commercial solution to manage virtual networks in an OpenFlow-enabled network. It offers a wide and flexible set of features to manage such networks. LibNetVirt, even though it offers fewer features, enables managing legacy technologies, not only OpenFlow.

Rotsos et al. [12] have developed a framework to evaluate OpenFlow switches. The information provided by their framework can be used to decide which switches to use when using libNetVirt together with OpenFlow.

VI. Conclusions

LibNetVirt is an open source [8] library to simplify the network view towards the user independently of the underlying technologies. It provides a simplified but powerful view towards the network. Its modularity permits to operate different technologies with the same API. We use the concept of a single node to represent the network. Only the definition of endpoints is necessary to create virtual networks.

We have evaluated our implementation of the OpenFlow driver which deploys L2 Virtual Networks. Firstly, we observed that the setup time of a flow increases with the presence of a centralized controller, however this is only in the first packet of the communication. This is inherited from OpenFlow architecture. Secondly, we observed that in a DoS attack, a legitimate client might have packets losses due to misbehavior of a host outside the network. This requires some proactive rules in the switch to minimize the effect and block the misbehaving host. Finally, we observed that UDP traffic at high rates has packet loss which is produced due to the congestion in the switch and controller and it affects the first milliseconds of the transmission.

Acknowledgments

This work is being carried out under the scope of the European FP7 SAIL project under grant number 257448. The authors would like to thank Ericsson Research, Packet Technologies Sweden for the access to their test bed.

References

[1] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, “Virtualizing the network forwarding plane,” in *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, ser. PRESTO ’10. New York, NY, USA: ACM, 2010.
[2] D. Turull, M. Hidell, and P. Sjödin, “libNetVirt: the network virtualization library,” in *Workshop on Clouds, Networks and Data Centers (ICC’12 WS - CloudNetsDataCenters)*, Ottawa, Canada, Jun. 2012.
[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: enabling innovation in campus networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.
[4] N. M. K. Chowdhury and R. Boutaba, “A survey of network virtualization,” *Comput. Netw.*, vol. 54, pp. 862–876, April 2010.
[5] G. Schaffrath *et al.*, “Network virtualization architecture: proposal and initial prototype,” in *VISA ’09: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*. New York, USA: ACM, 2009, pp. 63–72.
[6] T. Anderson, L. Peterson, S. Shenker, and J. Turner, “Overcoming the internet impasse through virtualization,” *Computer*, vol. 38, no. 4, pp. 34 – 41, april 2005.
[7] E. Keller and J. Rexford, “The ”platform as a service” model for networking,” in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, ser. INM/WREN’10. Berkeley, CA, USA: USENIX Association, 2010, p. 4.
[8] “Libnetvirt.” [Online]. Available: https://github.com/danieltt/libnetvirt
[9] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, “NOX: towards an operating system for networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 105–110, Jul. 2008.
[10] U. Pettit, J. Gross, B. Pfaff, M. Casado, and S. Crosby, “Virtual switching in an era of advanced edges,” in *2nd Workshop on Data Center - Converged and Virtual Ethernet Switching (DC CAVES)*, Sep. 2010. [Online]. Available: http://openvswitch.org/papers/dccaves2010.pdf
[11] R. Olsson, “pktgen the linux packet generator,” in *Linux Symposium 2005*, Ottawa, Canada, 2005.
[12] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, “Oflops: An open framework for openflow switch evaluation,” in *Proceedings of Passive and Active Measurements Conference (PAM ’12)*, March 2012.
[13] “OpenStack Quantum.” [Online]. Available: http://wiki.openstack.org/Quantum
[14] “OpenStack.” [Online]. Available: http://www.openstack.org/
[15] “NEC Programmable Flow.” [Online]. Available: http://www.necam.com/pflow/