Network Function Virtualization with Snabb

Karthik Mathiazhagan
Betreuer: Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer
Seminar Innovative Internet Technologies and Mobile Communications
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: karthik.mathiazhagan@in.tum.de

ABSTRACT

Virtualization methodologies are widely used in modern enterprise applications which share computing resources, storage and network. Network Function Virtualization (NFV) takes virtualization techniques one step ahead by migrating the network functions deployed in dedicated hardware infrastructure to software instances running inside a general purpose commodity hardware. On on hand Network Function Virtualization promises huge reduction of cost and deployment effort of network functions by replacing dedicated hardware solutions to software solutions but on the other hand they introduce a series of challenges in the domain of scalability, security and management of network functions.

NFV has entered into telecom market recently and is getting warm welcome by its providers and operators. Taking advantages of modern cloud computing techniques and combining them with NFV offers a wide variety of flexible software solutions for telecom operators. Snabb, a high speed packet processing tool kit has realised the capabilities of Network Function Virtualization and has developed a framework to support NFV infrastructure and allows flexible deployment virtual network function inside separate software containers. In this paper we would analyse different virtualization models present in a VirtIO infrastructure. We shall also discuss about NFV infrastructure implemented in Snabb and evaluate the performance of Snabb apps running inside NFV infrastructure.

Keywords

Snabb, Kernel bypass, NFV, LuaJIT, Lightweight 4-over-6

1. INTRODUCTION

In a traditional network computing platform, the network functions are implemented on a dedicated/proprietary hardware platform. This introduces a high coupling between the network function and the hardware platform. Network Function Virtualization (NFV) decouples these two by providing flexible software solutions for the network functions independent of target hardware. This separation of software from hardware provides a huge flexibility in deploying the network functions in terms of resource sharing, computing power and storage. The NFV infrastructure consist of virtual network functions deployed in so called common off-the-shelf (COTS) resources such as servers, storage, network, CPU, etc. The network service could be mapped to these COTS resources in a time sliced manner or they could also be assigned a fixed COTS resource. A typical view of Network Function Virtualization architecture is shown in Figure 1. The lower layer depicts the COTS resources typically present in a cloud infrastructure, the middle layer is the virtualization layer is responsible for mapping of network services to the physical resources. The top layer depicts the virtual service implemented as different virtual network functions (VNF). The links between these VNFs can also be logically visualized as a forwarding graph showing the functional relation ship between the VNFs. This whole Network Function Virtualization is managed by NFV management and orchestration layer which controls the life cycle management of network functions, also managing the physical resource allocation to the virtual functions.

Network Function Virtualization is gaining huge popularity from telecommunication providers [9] and operators in recent days. Currently the telecommunication equipments are sold in the market in a vertical tightly coupled software with the hardware style. Moving to NFV would promise to reduce huge cost in terms of production, deployment and management. NFV also poses serious challenges in terms of its reliability, scalability, security. For example, the COTS resources on which network functions are implemented could in turn become a bottleneck from performance and reliability point of view. The middle virtualization layer depicted

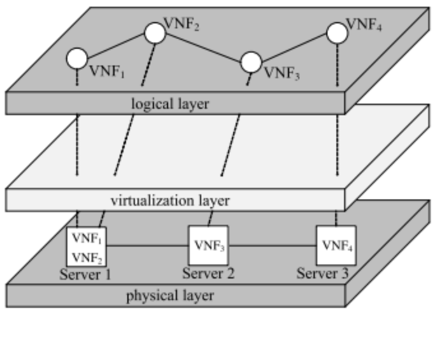

Figure 1: Network Function Virtualization Infrastructure (taken from [6])

above in Figure 1 should guarantee the requirements of Network Function Virtualization such as performance, scalability, flexibility, security and portability with different hypervisors. Snabb a novel high speed networking tool kit exploits the virtualization techniques to provide high performance network computing to virtual machines. In the following sections we shall focus on Snabb, its architecture, some of its built in apps and its compatibility for network function virtualization. Snabb has a dedicated module *'snabbnfv'* for its support on NFV for QEMU/KVM virtual machines.

2. SNABB

Snabb is a high speed networking tool kit developed by Luke Gorrie since 2012. The project is in constant development cycle with new feature releases almost every month. Snabb is build along any modern Linux/x86-64 distribution and could be build as stand alone executable 'Snabb'. Snabb goal is to provide network operators, network service provides, telecommunication providers that have to process huge packet rates about 10 Gbps, a software solution rather than currently available dedicated hardware solutions. Therefore Snabb is designed using some of the novel technologies such as kernel-bypass networking, LuaJIT programming and hardware assisted virtualization. Snabb has moved networking from kernel space to user space gaining almost 10 fold increase in networking performance.

2.1 Architecture

Snabb consist of high level APIs for front end usage and core module for low level system function. These high level APIs could be used to realize your network function. Traditional packet processing frameworks suffer from one huge bottleneck of context switches present in the kernel space. The packets arrive at user space once processed by the kernel. This poses a huge performance issue while process packets at 10 Gbps line rate. Snabb implements kernel-bypass Ethernet I/O which is explained in section 2.1.2 to circumvent this problem. Since Snabb is written using Lua programming Language which is simple and easy to learn, writing a Snabb application is all about writing a Lua script. Every network function written in Snabb can be realized as apps and app chains. The network of apps linked together to implement a network function gives modularity with low coupling and high cohesion design pattern. The following sections illustrate the core architectural concepts of Snabb.

2.1.1 Apps and App Network

Snabb provides with core module `(core.engine)` which executes the network design. Every network design is implemented as either a stand alone app or series of app linked together. An app is a stand alone isolated implementation of a network function. For instance an app could implement a router, switch or a firewall functionality. Every app can be initiated with zero or more number of input and output ports. The packets arrive at input port, the app performs the required processing as designed in the Lua Script and finally the packets leave at the output port. When the network design requires linking of multiple apps together, it could be done using Snabb `(core.link)` module. Links are implemented as circular ring buffers to store and forward packets between apps. Snabb also provides with `(core.config)` module which holds the complete configuration of the network design. It could be realized as a directed graph consisting of apps and links between them. This configuration is finally loaded by Snabb Engine where it initializes the configured apps, links and starts the network function. Once Snabb Engine is loaded, its breath cycle starts to execute. A breath cycle typically consist of inhale and exhale cycle. During inhale the *Pull* routines of all the apps are executed and during Exhale the *Push* routines of all the apps are executed. All Snabb apps should therefore implement the interface `(core.app)` and implement the *Push* and *Pull* method for the above functionality. This way once the Snabb Engine loads the configuration it starts to breath by pumping the traffic though the app chain.

2.1.2 Kernel-bypass Networking

Kernel-bypass networking is gaining huge popularity in recent days. Network stacks implemented currently in Linux does not prove to be efficient when handing huge packet rate at about 10+ Gbps. Kernel-bypass technology enables handing of raw Ethernet packets directly in the user space without the intervention of kernel. The packets no longer pass through the kernel and to user space but directly arrive to user space. This provides a huge gain in performance comparing traditional network stack but at the same time leaves us with huge responsibility in handing packets. This technology is therefore chosen wisely for specific networking domain which involves specific uses cases to handle packets rates above 10 Gbps.

Snabb implements Kernel-bypass networking with two important features provided by modern NICs, memory mapped I/O (MMIO) and Direct Memory Access (DMA). In order to completely get rid of kernel, Snabb has written its own device drivers in Lua for modern Intel 82599 NICs. The process starts by unbinding the PCI device from the kernel to make it available to Snabb. Snabb internally asks kernel to map the device configuration address space to process virtual address space of Snabb. This way once the mapped virtual address is called, it triggers a call back in the PCI device registers. This memory mapped I/O register access is directly implemented by the CPU without the knowledge of kernel. Direct Memory Access (DMA) allows the NIC to read and write to a address space directly visible by the Snabb process. Snabb does this by allocating `Hugepages` which are contiguous allocation of physical memory (memory blocks of 2MB) and mapping the virtual address space of Snabb to resolve to the above allocated physical memory. This way each read and write by the NIC on the physical memory address is directly available by the Snabb user space process.

2.1.3 Lua and LuaJIT

Lua is a simple and easy to learn programming language. It is most widely used in game engines, image processing and also supported in MySQL proxies. The main reasons for choosing Lua for Snabb is that it is very fast compared to traditional C language, its simple Foreign Function Interface (FFI) library which allows Lua to interact with low level APIs written in C and finally LuaJIT.

LuaJIT is a just-in-time compiler for Lua programming language. In contrast to method based JIT compiler, LuaJIT is a trace based JIT compiler. The main difference is that LuaJIT operates on so called traces of a program. During Lua program execution the frequently executed paths, loops of the program are recorded. This is the recording phase of LuaJIT, these recorded byte codes are optimized further by LuaJIT and converted into Intermediate Representation (IR). This optimized Intermediate Representation is called as LuaJIT Traces. These optimized IR code is then translated into native code by LuaJIT so that next time the same loop is executed the interpreter executes this optimized code rather than the byte code again. The CPU executes the native code blindly as long as they belong to the same trace. For the above reasons, LuaJIT traces typically contains a single loop with no unpredictable branches and all function calls inlined.

2.2 Snabb Apps

In previous section we briefly discussed about Snabb architecture and its core modules. In this section we shall discuss about some of the Snabb apps and in later sections we shall analyse their performance. Snabb comes with some of pre compiled apps for certain network functionalities and libraries. App developers could directly include these libraries or apps into their app network to add more functionality.

2.2.1 Snabb Forwarder

This is a simple Snabb app for forwarding packets between two network interfaces. In order to make the login simple, no decision is made regarding forwarding the packets between the interfaces. The app takes in two PCI device address, number of hops and direction as input during program execution. The PCI device address parameter decides the input and the output NIC to do the forwarding. The number of hops basically specifies the number of intermediate forwarders to construct between the start and the end forwarder. Finally the direction argument configures the forwarder either as Single directional or Bi-directional forwarder. This app supports all the Intel 82599 NICs currently supported by Snabb as it uses 'Intel82599' driver for handling traffic from Intel 10 Gbps NICs.

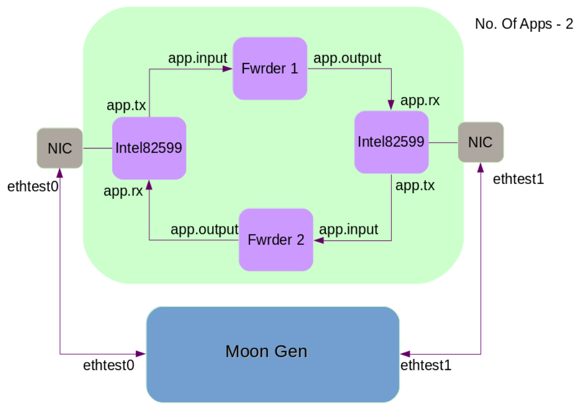

Figure 2: Bi Directional Forwarder Structure

The above Figure 2 better illustrates the Snabb bi-directional forwarder with 2 intermediate hops. Later in section 4.1 we shall analyse the performance of Snabb forwarder when executed directly in our test bed server and under Network Function Virtualization set up.

2.2.2 Lightweight 4-over-6 and Snabb lwaftr

Lightweight 4-over-6 is a IPv6 transition technology as specified in RFC 7596. Lightweight 4-over-6 is comprised of two major components, Lightweight before *'b4'* and Lightweight after *'lwaftr'*. It allows for internet service provides to maintain a pure IPv6 network while still supporting interoperability with IPv4 networks. Pure IPv6 customer premises could be easily made available to access the IPv4 internet and vice-versa. Lightweight before *'b4'* is the customer facing component and Lightweight after *'lwaftr'* is the internet facing component of this architecture.

Snabb *'lwaftr'* implements the internet facing component of Lightweight 4-over-6 and could be deployed inside of black box running in the providers network. The main functionality of this app would be to encapsulate and decapsulate IPv4 packets over IPv6 packets. Snabb does mapping of IPv6 address with the IPv4 addresses by maintaining a binding table. For each IPv4 packets received Snabb references into this binding table to do the right encapsulation. Each of the customer IPv6 address is mapped with a IPv4 address and a port range. This allows for mapping of different customer IPv6 address with same IPv4 address but with different port range. Snabb makes uses of this technique for efficient sharing of IPv4 addresses and also by doing so maintains per customer binding rather than per flow binding. This also means that multiple Snabb *'lwaftr'* apps can concurrently work at different flows from the same customer.

2.2.3 SnabbNFV

SnabbNFV is a Snabb app written for Network Function Virtualization component based out of Snabb. On a very high level it is synonymous with the virtualization layer present in a NFV Infrastructure assigning Virtual network functions to physical networking devices. SnabbNFV works well for QEMU/KVM virtual machines in a VirtIO infrastructure as it implements its own VirtIO back end drivers in user space (VhostUser).

SnabbNFV takes in a Configuration file as input and constructs the app network out of it. The SnabbNFV config is a list of parameters configured during initialization of the virtual infrastructure required to run virtual network functions. Each port in the configuration file is identified by its "port_id" along with other parameters such as MAC address, Vlan id, etc. It also takes in a file socket path and the PCI address of the NIC as input parameters. The file socket would server as a shared path for communication between the Snabb VhosUser VirtIO back end server and QEMU VhostUser VirtIO client. The PCI device address argument passed to snabbnfv would be mapped with the virtual network function implemented inside the QEMU/KVM virtual machine. The Snabbnfv traffic process performs packet processing from the underlying PCI device to all of the virtual machine assigned to that PCI device. Section 3.3 briefly talks about VM infrastructure set up done using Snabbnfv for our test measurements.

3. VIRTUAL MACHINES

Virtual Machine are now a days used in almost all domains in IT infrastructure. Cloud based virtual machines in particular the choice when it comes to flexible deployment of software. In a Network Function Virtualization Infrastructure each network function can be visualized to be instantiated inside one or more virtual machine and the physical hardware resources is decoupled from the network function by the virtualization layer. This virtualization layer should satisfy certain requirements such as abstraction of physical hardware partitions, providing a unified and standard interface to the virtual functions independent of the physical hardware. This ensure portability of virtual network functions to a great level. Software emulated virtualization could be broadly classified into two types namely full virtualization and para virtualization. In full virtualization a layer called as hypervisor sits in between the virtual machine and the physical hardware and emulates all of the physical resource to the virtual machine. The virtual machine has no idea that it runs inside a virtualized environment. However in a para virtualized environment not all of the physical resources are emulated to the virtual machine. The VM and the hypervisor work in a cooperative fashion and is well aware that it is running in a virtual environment. Though both of these methods has its own advantages and disadvantages introduction of hardware assisted virtualization has created another new domain in virtualization. It enabled hypervisors to be fairly simple and also to support full virtualization at the same time.

3.1 QEMU/KVM Virtual Machines

Kernel Virtual Machines (KVM) is one such hardware assisted virtualization technique which takes advantage of virtualization support of underlying hardware. KVM is implemented as a kernel module which could be loaded on demand. Once loaded it turns kernel into a Type 1 hypervisor emulating the hardware resources. The emulation happens with the help of QEMU which emulates devices such as PCI, network cards, Disk, etc. KVM requires the host hardware to be supported by either Intel-VT or AMD SVM virtualization extensions. The addition of KVM as a kernel model has gained a huge advantage in terms of ease of use and flexibility in instantiating virtual machine. Each virtual machine in this mode is just a process and scheduled the same way a normal Linux process is scheduled. In addition to the user and kernel mode of operation in Linux, KVM virtual machines adds another mode called as guest mode. All the linux commands could be executed under guest mode and to kill a virtual machine is as easy as to kill a process in Linux. KVM could also be used along standard paravirtualized VirtIO drivers installed on the host system. The emulation layer of QEMU in that case provides the paravirtualized VirtIO drivers to the guest VMs.

3.2 VirtIO Infrastructure

In a QEMU/KVM virtual machine environment, the virtual machine may acquire network functionality by emulation of paravirtualized VirtIO devices such as VirtIO PCI devices. In a VirtIO Infrastructure the VirtIO PCI device drivers implements the so called 'virtio-ring' data structures which serves as the standard mechanism for communication in a VirtIO environment. The VirtIO drivers are split into two parts, front-end drivers and back-end drivers. The machine specific front-end VirtIO drivers are implemented in the VM while the back-end driver could be implemented independently either by kernel or by QEMU emulation layer. The communication between the drivers is made possible via generic VirtIO queues that are shared between them.

There are many ways to set up a guest in a VirtIO Infrastructure. In a Virtio_Net implementation the guest implements the VirtIO_Net drivers and share a number of virtio queues with the QEMU layer. This is an example, where the VirtIO front-end drivers are implemented by the guest and back-end drivers are implemented by QEMU. This model allows the network traffic to pass through the QEMU process before handling it to guest or vice versa to host network stack. This poses a huge performance issue by invoking additional context switches back and forth to the QEMU layer. The solution to this problem would be to use Vhost_User VirtIO model. This model allows the QEMU/KVM VM to share a number of virtio queues directly with the host kernel. This way the traffic would now bypass the QEMU layer and directly arrive at the guest from the kernel. In this model QEMU would still emulate the virtio PCI devices to the guest but its data plane (i.e the virtio queues) would be directly shared with the kernel back end drivers.

The above mentioned solution of using VirtIO Vhost_Net model would still not be an efficient solution in a Snabb environment as Snabb is a user space process. Snabb completely avoids the kernel context switches by kernel-bypass technology where the packets directly arrive at the user space process. Starting with QEMU 2.1 release features a new model in a VirtIO infrastructure called Vhost_User. The goal of this model is to allow the paravirtualized virtio queues to share between one user space process implementing virtio front-end driver to another user space process implementing the back-end driver. This way the virtio queues of the guest VM could now be directly shared with Snabb process. This model requires to still preserve the VirtIO paradigm of using shared memory and event triggers using 'ioeventfds' and 'irqfds'. VhostUser is a Snabb app which implements the generic back-end driver for VirtIO Vhost_User infrastructure.

3.3 VM Setup with SnabbNFV

In previous section we explained different models of VirtIO Infrastructure and Vhost_User model which is currently used by Snabb for its Network Function Virtualization module using QEMU/KVM Virtual Machines. In this section we shall look into setting up the NFV environment using Snabb. This set up will later be used for performance evaluation of Snabb apps running inside of NFV infrastructure. This requires the host hardware to be Intel VT enabled as QEMU/KVM requires this hardware virtualization feature. Snabbnfv traffic process would need to started first by giving the configuration file, PCI address and file socket path as input parameters.

Once Snabbnfv traffic process starts, it gives the file socket path as input to the Vhost_User app implemented in Snabb as shared memory communication path with the virtio guest drivers. Next, we would need to start a QEMU/KVM virtual machine by passing the same file socket path as before. Once the virtual machine is started, we would be able to see the virtio network interface configured from inside of the VM. This virtio network interface is directly mapped to the PCI device address which was given before as input to Snabbnfv traffic process. Figure 3 shows us the functional set up of QEMU/KVM guest VM running under Snabbnfv infrastructure. Each instance of Snabbnfv takes the supplied PCI NIC device under its control. Our Current set up takes two PCI devices and forwards traffic to two different virtio NICs present in the guest.

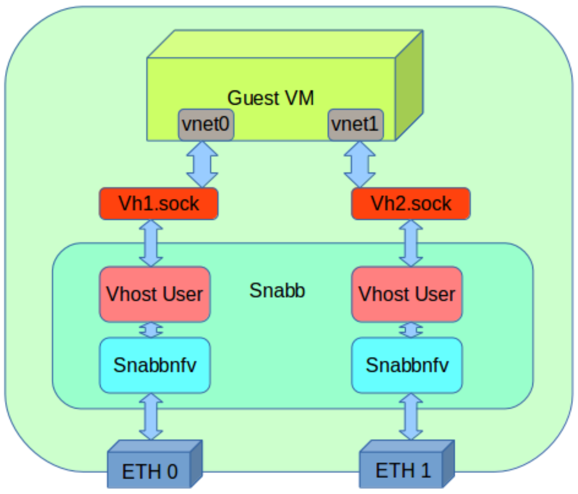

Figure 3: VM with Snabbnfv

3.3.1 *Performance optimization of Snabbnfv*

Since Snabbnfv by itself is an app written in Snabb, it could be optimized in several ways to provide the best performance with lower packet loss when passing traffic through the virtio infrastructure. Some of the optimization steps which could be done to increase the performance of any Snabb app are listed below.

- Assigning Snabb process to a particular CPU.
- Same NUMA node assignment of PCI device and CPU for Snabb process.
- Disabling hyper threading.
- Allocating Huge Pages of size 2 MB during boot time.
- Configuring Ring buffer sizes.

4. PERFORMANCE EVALUATION

In this section we will briefly evaluate the performance of Snabb apps. Multiple Snabb apps were tested in our normal test bed environment and also in Network Function Virtualization environment. The results are then compared to get a brief picture of their differences in performance when executed in these environments. All measurements were executed in baltikum test bed servers equipped with Intel(R) Xeon(R) CPU E3-1230 V2 clocked at maximum of 3.30GHz, supermicro X9SCL/X9SCM motherboard with 16GB DDR3 SDRAM and two Intel 82599EB Network Connections. The test set up consist of two servers connected to each other via 2 Ethernet 10 Gbps links, one server running the Snabb apps and another server used as source of test traffic. MoonGen, a flexible high speed packet generator was used as our source for test traffic. MoonGen could saturate 10 Gbps link with 14.88 Mpps at line rate which makes it an ideal choice for testing Snabb apps with high load traffic. The test bed servers were running on Grml Linux image with kernel version 3.16 and the guest VMs used in NFV environment were running on kernel version 3.13.

4.1 Snabb Forwarder

As already introduced in section 2.2.1 Snabb Forwarder is able to forward packets between the provided input and output network interfaces. It makes use of Intel82599 driver app written for driving Intel 10 Gbps Network interface controllers. It could be initialized to either forward packets in a single direction or in bi-direction. The number of hops parameter supplied during initialization constructs those many number of intermediate hops from the source Intel82599 app to destination Intel82599 app. This app could be used to determine the optimal number of forwarding hops required during design of a network function for certain load traffic.

The measurements were done first by running Snabb forwarder on one of the test bed servers and then running Snabb forwarder again inside of a Virtual Machine infrastructure created by Snabbnfv on the same test bed server. MoonGen was used to blast test traffic at a constant rate and the forwarded traffic back to MoonGen server was measured again to arrive at the average throughput. For the first measurement, MoonGen was made to supply test traffic at around 7.81 Mpps through each of its interfaces and Snabb forwarder was made to run in bi-directional fashion starting initially with 2 hops. After every round of throughput measurement by MoonGen the number of hops was increased by 2 and the measurement was repeated. We observed that Snabb Bi-directional forwarder performed quite well providing a throughput above 75% of the input packet rate even when the number of hops were increased to 8.

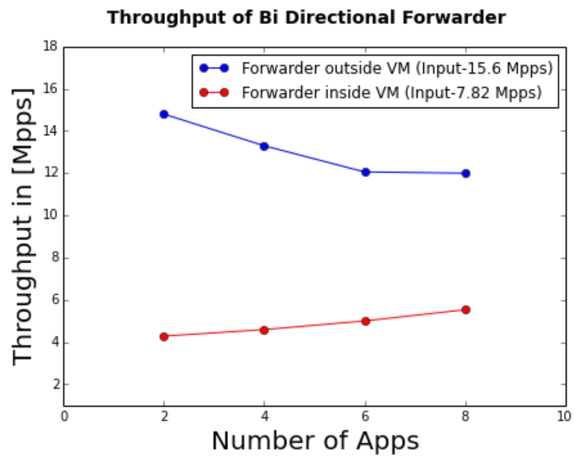

Figure 4: Snabb Forwarder Measurements

For NFV measurement, Snabb NFV module was fist run on the host machine with certain app configuration, socket file path and PCI device address. Since our test would need to forward packets between two different interfaces, two instances of Snabb NFV module was initialized one for each PCI NIC device. QEMU/KVM VM was started by providing the socket path of two PCI NIC device in a VirtIO Vhost User model. Apart from two virtio pci devices, VM was initialized with a third virtIO NIC for connecting with the Management LAN of Baltikum test bed. Measurement was done exactly in similar way as before by running Snabb forwarder inside the VM with increasing number of hops starting from 2. MoonGen was now made to run with test traffic at around 4 Mpps to avoid huge packet losses inside of Snabbnfv app network. We found that Snabb forwarder inside VM was able to provide a throughput of maximum of 70% of input load which was 4 Mpps. Also we found that Snabb forwarder inside the VM was not able to forward more than 3 Mpps out of each of its interfaces even when increasing the input load significantly. Figure 4 shows the results of our test comparing both of the throughput measurements done for Snabb Forwarder.

5. CONCLUSION

Network function virtualization being the recent buzz word in the market of telecommunication industry has made internet service providers to rethink on migrating their existing solution to NFV based solutions. In this paper we discussed about NFV architecture and its features such as flexibility, maintainability and reduction in cost. The main idea of Network Function Virtualization is to decouple the hardware layer from the software running on top of it. This introduces us to the concept of deploying flexible software solution on any generic commodity hardware module. This has huge advantage in terms of hardware vendor dependency when deploying a network functionality. The network functions are no longer a hardware solution but a software solution which could be flexibility deployed in any cloud server infrastructure (COTS). The life cycle management of a network function would be done using NFV management and orchestration module which is part of its infrastructure.

Having introduced to this powerful concept, we then discussed about Snabb, a high speed networking tool kit. Snabb is designed on some of the novel technologies to overcome the bottleneck faced by of current Linux network stacks. The concept of kernel-bypass technology has moved the entire packet handling logic from kernel space to user space giving a lot more flexibility and huge increase in performance. The Direct Memory Access technology enables the NIC to be able to directly read and write into part of the RAM visible by User space process. Access to these memory mapped I/O locations by Snabb process triggers the call back implemented by the NIC hardware registers to read or write data. Implementation using Lua and LuaJIT adds another feather in a cap for Snabb architectural design. LuaJIT being a trace based just-in-time compiler optimizes the Lua byte code during run time by recording traces of frequently executed loops and instructions. Each optimized trace is then further executed every time the loop is executed and CPU could blindly execute instructions as long as they remain in a single trace.

6. REFERENCES

[1] Network Function Virtualization (NFV), Architectural Framework. `http://www-cs-faculty.stanford.edu/~{}uno/abcde.html`

[2] Roberto Riggio, Julius Schulz-Zander, Abbas Bradai. Virtual Network Function Orchestration with Scylla. *SIGCOMM '15: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, ACM, 2015.*

[3] Katerina Barone-Adesi, Andy Wingo. [FOSDEM 2016], SnabbSwitch:Riding the HPC wave to simpler, better network appliances. `https://archive.fosdem.org/2016/schedule/event/snabbswitch/`

[4] Katerina Barone-Adesi, Andy Wingo. [FOSDEM 2016], [*Slides*]. Building fast IPv6 transition mechanisms on Snabb Switch. `https://wingolog.org/pub/fosdem-2016-lwaftr-slides.pdf`

[5] Deutsche Telekom, IPv6 Transition. Deutsche Telekom tests TeraStream, the network of the future, in Croatia. `http://www.telekom.com/media/company/168008`

[6] A. Al-Shuwaili, O. Simeone, J. Kliewer, and P. Popovski. Coded Network Function Virtualization: Fault Tolerance via In-Network Coding. *IEEE WIRELESS COMMUNICATIONS LETTERS, VOL. 5, 2016*

[7] Chengwei Wang, Oliver Spatscheck, Vijay Gopalakrishnan, Yang Xu, David Applegate. Toward High-Performance and Scalable Network Functions Virtualization *IEEE INTERNET COMPUTING, VOL. 20, 2016*

[8] Bruno Chatras; François Frédéric Ozog Network functions virtualization: the portability challenge *IEEE Network, VOL. 30, 2016*

[9] Deutsche Telekom, IPv6 Transition. Deutsche Telekom tests TeraStream, the network of the future, in Croatia. `http://www.telekom.com/media/company/168008`

[10] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. *In Proceedings of the 2015 ACM Conference on Internet Measurement Conference, ACM, 2015.*

[11] L.Gorrie. Snabb Switch: kernel-bypass networking illustrated. `https://github.com/lukego/blog/issues/13`

[12] L.Gorrie. Snabb Switch's LuaJIT Ethernet Device Driver. `http://lukego.github.io/blog/2013/01/03/snabb-switchs-luajit-ethernet-device-driver/`

[13] Irfan Habib. Virtualization with KVM. *Belltown Media, Linux Journal: Volume 2008 Issue 166, February 2008.*

[14] Mike Pall. The LuaJIT Project, [Running LuaJIT]. `http://luajit.org/running.html` `http://www.freelists.org/post/luajit/How-does-LuaJITs-trace-compiler-work,1`

[15] Circular Ring Buffer DataStructure. `https://en.wikipedia.org/wiki/Circular_buffer`

[16] Vhost-User feature for QEMU. *Vhost-User Applied to Snabbswitch Ethernet Switch.* `http://www.virtualopensystems.com/en/solutions/guides/snabbswitch-qemu/?vos=tech`

[17] Snabb Network Function Virtualization.[Code base] `https://github.com/snabbco/snabb/tree/master/src/program/snabbnfv`

[18] Snabb Lightweight 4over6. [Code base] `https://github.com/snabbco/snabb/tree/master/src/program/lwaftr/doc`