Virtualization to Build Large-Scale Networks

University of Oulu
Faculty of Information Technology and Electrical Engineering / Information Processing Science
Master's Thesis
Niko Takkinen
29.4.2019

Abstract

There is not much research concerning network virtualization, even though virtualization has been a hot topic for some time and networks keep growing. Physical routers can be expensive and laborious to setup and manage, not to mention immobile. Network virtualization can be utilized in many ways, such as reducing costs, increasing agility and increasing deployment speed. Virtual routers are easy to create, copy and move. This study will research into the subjects of networks, virtualization solutions and network virtualization. Furthermore, it will show how to build a virtual network consisting of hundreds of nodes, all performing network routing. In addition, the virtual network can be connected to physical routers in the real world to provide benefits, such as performance testing or large-scale deployment. All this will be achieved using only commodity hardware.

Keywords
virtualization, networks, network virtualization

Supervisor
Ph.D., University Teacher Mikko Rajanen

Foreword

I wish to foremostly thank my fiancée Anni for continuous support during the creation of this study and during University studies overall. I also wish to thank my supervisor Mikko Rajanen from the University of Oulu for providing guidance on many aspects. Furthermore, without late-evening, and sometimes weekend, -chats with my supervisor Aki in the organization, as well as advice from Tuomas, this would not have been possible to do in a timely-manner, so thank you.

Niko Takkinen

Oulu, April 29, 2019

Abbreviations

OLSR = Optimized Link State Routing

OSPF = Open Shortest Path First

BGP = Border Gateway Protocol

TCP = Transmission Control Protocol

IP = Internet Protocol

PC = Personal Computer

HW = Hardware

OS = Operating System

VM = Virtual Machine

NIC = Network Interface Card

GUI = Graphical User Interface

Contents

1. Introduction

Networks, fast deployment and versatility are important subjects in the present day's rapid software development cycles and interconnection of things. Not everything can be done in real physical environments, such as building a network of hundreds of nodes, which would be very expensive and time consuming, not to mention hard to operate. Virtualization has been a hot topic for some time now as well (Wang, Iyer, Dutta, Rouskas & Baldine, 2013) and it can be utilized in problems such as creating hundreds of nodes in a network, as it can be more cost effective. Furthermore, virtualization can be used effectively in business environments to e.g. improve resource utilization, lower costs and increase business agility. Virtual environments are easier to setup, modify and replicate, and don't necessarily need its own dedicated hardware, because of advancements in commodity PC's (Personal Computers') hardware. Virtualization has advantages, such as encapsulation, that helps in testing and development processes because a single physical server can host many virtualized development environments (Bănică, Rosca & Stefan, 2007; "Virtualization Overview", n.d.).

The goal of the study is to first research virtualization, virtualization solutions and networking and then build a virtual environment capable of simulating very large networks consisting of hundreds of nodes, all the while using only commodity hardware. To understand these concepts, this study will first look at how computer networks and routing function. The study then looks at virtualization solutions in order to understand how virtualized networks function.

Computers can connect to each other by either wired or wireless media. A system of interconnected computers and computerized peripherals is called computer network. The devices in a computer network can also be called nodes ("Introduction to Computer Networks", n.d.). Nodes then communicate and share data with each other using these networks. A node always selects one path when it has multiple paths to reach a destination. This selection process is termed as Routing and it's done by routers or by means of software processes. There are a few different types of routing, mainly unicast, broadcast and multicast routing. ("Data Communication", n.d.). However, computer networks need more than models and different types of routing. They need routing protocols. Routing protocols are rules or standards that govern communications when computers send data back and forth (Maltz, Xie, Zhan, Zhang, Hjálmtýsson & Greenberg, 2004; Blank, 2006).

Virtualization is not a new concept, as it was first introduced in the 1960s (Bănică et al., 2007; Egi, Greenhalgh, Handley, Hoerdt, Huici, Mathy & Papadimitriou, 2010), however only recently has the hardware (HW) of commodity PC's become powerful enough to make it feasible to run more complex virtualization solutions on a single PC (Egi, Greenhalgh, Handley, Hoerdt, Huici & Mathy, 2008). Virtualization generally refers to the abstraction of fundamental computing resources (Wang, Iyer, Dutta, Rouskas & Baldine, 2013) and it enables running multiple operating systems or software environments on a single hardware. Three distinct approaches to virtualization are discussed in this paper: hosted approach, hypervisor approach and container approach. Containers are somewhat new approach, but their usefulness is proven by the fact that Google uses them extensively ("Containers at Google", n.d.).

Network virtualization is a part of virtualization and it can be done by e.g., virtualizing whole routers or Network Interface Cards (NICs). Virtual router has the same properties as a physical router, i.e. it has routing table, routing protocols, management interface, and so on (Rahore, Hidell & Sjodin, 2012), except it's possible to have multiple virtual routers running on the same hardware platform, sharing the available resources (Wang et al., 2013). The virtualization technology ensures resource isolation among virtual routers, i.e. one virtual router should not be able to see or access resources, such as the routing table, of other virtual routers (Rahore et al, 2012). Rahore et al. (2012) have shown that hardware assisted virtual routers can achieve better aggregate throughput than a non-virtualized router on a multi-core platform.

There isn't much knowledge in the research community in regard to building large-scale virtual networks and with that knowledge in mind, the goal of the study is focused into the following two research questions:

> **RQ1:** What feasible virtualization solutions are available for creating a large-scale virtual environment for network routing?
>
> **RQ2:** How to build a flexible virtual environment that can simulate hundreds of nodes in a network, using only commodity hardware?

This study uses the Design Science Research (DSR) method by Hevner, March, Park and Ram (2004) and the research questions reflect this design science nature in the paper, because the study tries to provide information for the research community while also producing an artifact to answer the research questions. The production of an artifact is important because this study is done for a company, specializing in networks and secure communications, that wishes to have research and a prototype done in this area. Thus, the organization and the DSR method both bring some requirements for the artifact. The requirements and the DSR methodology are explained in Chapter 3.

The main contribution of this study is to show how an artifact has been built to simulate very large networks with an environment that is scalable and flexible, using commodity hardware. Furthermore, the study researches these technologies to provide an overview of how they work and how the artifact works. Lastly, instructions on building and using the artifact is provided in appendix A.

The structure of the study is as follows: next will be discussed prior research related to this topic. Then research methods are shortly discussed, after which the artifacts implementation is shown. After implementation is the evaluation, where the artifact is analysed and evaluated, in addition to evaluation the study. The findings and their implications are then discussed. Lastly a conclusion is drawn. Furthermore, as mentioned, the artifacts instructions are in appendix A.

2. Prior research

This chapter is based on prior literature and it will discuss technologies needed to understand what is included in building virtual networks. It will start with the subjects of networking, network engineering and protocols to get the basic idea of what networks are and what they consist of. Then the subjects of virtualization, network virtualization and, lastly, network traffic simulation are presented and explained.

2.1 Computer Networks

As mentioned in the Introduction, a system of interconnected computers and computerized peripherals such as printers is called computer network. These devices in a computer network can also be called nodes, and the computer network can further be called communication network or more simply as a network. This interconnection among nodes facilitates information sharing among them, because the nodes can send and receive data generated by other nodes on the network. The links connecting the nodes are called Communication channels. ("Introduction to Computer Networks", n.d.). There are five different commonly recognized types of communication networks: Local Area Network (LAN), Metropolitan Area Network (MAN), Wide Area Network (WAN), Wireless Network, and Inter Network (Internet) ("Introduction to Computer Networks", n.d.; "Data Communication", n.d.). All except Wireless Networks are discussed next.

Local Area Network (LAN)

LANs are composed of inexpensive networking and routing equipment. It mostly operates on private IP addresses and does not involve heavy routing. LAN works under its own local domain and is controlled centrally. ("Data Communication", n.d.). LAN is designed for small physical areas such as an office or a group of buildings. LANs are used widely as it is easy to design, with e.g. different topologies such as Star or Ring topology, and to troubleshoot. Personal computers and workstations are connected to each other through LANs. LAN networks are also widely used to share resources like printers or shared hard-drives. ("Introduction to Computer Networks", n.d.).

Metropolitan Area Network (MAN)

MAN was developed in the 1980s. It's basically a bigger version of LAN and uses the similar technology as LAN. It's designed to extend over an entire city. It can be means to connecting a number of LANs into a larger network or it can be a single cable. It is mainly held and operated by single private company, such as Internet Service Providers (ISPs), or a public company. ("Introduction to Computer Networks", n.d.; "Data Communication", n.d.). However, it should be noted that MANs are more complex to design and maintain ("Computer Network", n.d.).

Wide Area Network (WAN)

WAN can be a private network or a public leased network the same way as MAN. WAN is used for a network that covers large distance such as the states of a country. It's

not easy to design and maintain. Communication medium used by WAN are the public switched telephone network (PSTN) or Satellite links. WAN operates on low data rates. (“Introduction to Computer Networks”, n.d.; “Data Communication”, n.d.).

Inter Network (Internet)

Inter Network, or Internet, is a combination of two or more networks. Inter network can be formed by joining two or more individual networks by means of various devices such as routers, gateways and bridges. (“Introduction to Computer Networks”, n.d.). The Internet is the largest network on this planet, and it connects all WANs and can even have connections to LANs. Internet uses TCP/IP (Transmission Control Protocol and Internet Protocol) protocol suite where IP is used as its addressing protocol. Internet enables users to share and access enormous amounts of information worldwide. (“Data Communication”, n.d.).

2.1.1 Network Engineering

Network engineering is the art of planning, implementing and overseeing a computer network that supports certain wanted services (Rouse, 2018). It’s a complicated task involving software, firmware, chip level engineering, hardware, and electric pulses. To ease this task, the whole networking concept is divided into multiple layers, each layer involving some task and being independent of all other layers. Each layer does only specific work, yet almost all networking tasks depend on all these layers. (“Data Communication”, n.d.).

Layers share data and they depend on each other only to take input and send output. In a layered communication system, one layer of a host deals with the task done by or to be done by its peer layer at the same level on the remote host. The task is either initiated by layer at the lowest level or at the top most level. If the task is initiated by the topmost layer, it is passed on to the layer below it for further processing. The lower layer does the same thing, it processes the task and passes on to lower layer. If the task is initiated by lowermost layer, then this is done in reverse. Every layer groups together all procedures, protocols, and methods which it requires to execute its piece of task. (Blank, 2006; “Data Communication”, n.d.).

One of the most famous layered communication architectures is the ISO-OSI model, commonly known as OSI model. ISO stands for International organization of Standardization and OSI stands for Open System Interconnection. The ISO-OSI model is a seven-layer architecture, as can be seen in figure 1, defining seven layers or levels in a complete communication system (Blank, 2006; “Data Communication”, n.d.; “Introduction to Computer Networks”, n.d.). The layers are: (1) Physical Layer, (2) Datalink Layer, (3) Network Layer, (4) Transport Layer, (5) Session Layer, (6) Presentation Layer and (7) Application Layer.

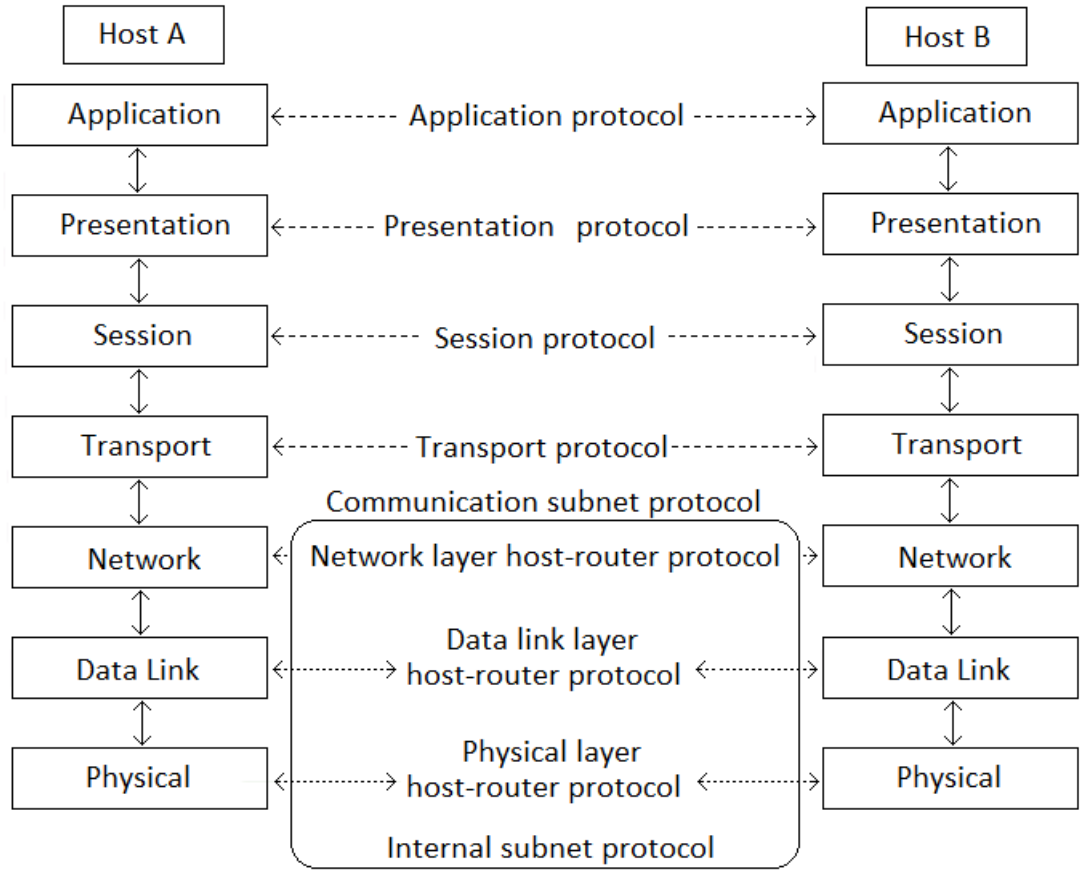

Figure 1. ISO-OSI model.

Layers 2, 3 and 4 are further explained in detail, starting from layer 2, the Data Link layer.

Data Link Layer

Data link layer hides the details of underlying hardware and represents itself to upper layer as the medium to communicate. Data link layer works between two hosts which are directly connected in some sense. Data link layer is responsible for converting data stream to signals bit by bit and to send that over the underlying hardware. At the receiving end, Data link layer picks up data from hardware which are in the form of electrical signals, assembles them in a recognizable frame format, and hands over to upper layer. An important part of Data Link Layer is to provide layer-2 hardware addressing mechanism. Hardware address is assumed to be unique on the link, as the addresses, i.e. Media Access Control (MAC) addresses, are encoded into hardware at the time of manufacturing. (Blank, 2006; “Data Communication”, n.d.; “Introduction to Computer Networks”, n.d.).

Network Layer

As stated earlier, layer 3 in the OSI model is called Network layer. It manages options pertaining to host and network addressing, managing sub-networks, and internetworking. Furthermore, it takes the responsibility for routing packets from source to destination within or outside a subnet. Two different subnets may have different addressing schemes or non-compatible addressing types. Same with protocols, two

different subnets may be operating on different protocols which are not compatible with each other. Network layer has the responsibility to route the packets from source to destination, mapping different addressing schemes and protocols. As such, devices which work on Network Layer mainly focus on routing. (Blank, 2006; "Data Communication", n.d.; "Introduction to Computer Networks", n.d.).

Layer 3 network addressing, or IP addressing, is one of the major tasks of Network Layer. Network addresses are always logical i.e. these are software-based addresses which can be changed. A network address always points to a node or it can represent a whole network and it's always configured on network interface card (NIC) and is generally mapped by system with the MAC address of the machine for Layer-2 communication. IP addressing provides mechanism to differentiate between hosts and network. Because IP addresses are assigned in hierarchical manner, a host always resides under a specific network. A host who needs to communicate outside its subnet, needs to know the destination network address, where the packet or data is to be sent. For this, hosts in different subnets need a mechanism to locate each other. The task can be done by Domain Name Server (DNS). DNS is a server which provides Layer-3 address of remote host mapped with its domain name. When a host acquires the Layer-3 Address (IP Address) of the remote host, it forwards all its packet to its gateway. A gateway is a router equipped with all the information which leads to route packets to the destination host. (Blank, 2006; "Data Communication", n.d.; "Introduction to Computer Networks", n.d.). Network routing is talked more in the next chapter.

Transport Layer

Transport layer is the 4th layer of the OSI model. All modules and procedures pertaining to transportation of data or data stream are categorized into this layer. It offers peer-to-peer and end-to-end connection between two processes on remote hosts. The layer takes data from upper layer, i.e. Application layer, and then breaks it into smaller size segments, numbers each byte, and hands over to lower layer, i.e. Network layer, for delivery. While doing this, it ensures that data is received in the same sequence in which it was sent. A process on one host identifies its peer host on remote network by means of TSAPs (Transport Service Access Points), also known as Port numbers or as ports. Port numbers are well defined and a process which is trying to communicate with its peer knows this in advance. For example, when a DNS client wants to communicate with remote DNS server, it always requests on port number 53. Transport layer has two main protocols, Transmission Control Protocol (TPC) and User Datagram Protocol (UDP) (Blank, 2006; "Data Communication", n.d.; "Introduction to Computer Networks", n.d.), which will be explained further on.

2.1.2 Network Routing

When a device has multiple paths to reach a destination, it always selects one path by preferring it over others. This selection process is termed as Routing. Routing is done by special network devices called routers or it can be done by means of software processes. Routers take help of routing tables, which have an address of destination network and method to reach the network. A router is always configured with some default route. A default route tells the router where to forward a packet if there is no route found for a specific destination. Routers upon receiving a forwarding request, forwards packet to its next hop (adjacent router) towards the destination. The next router on the path follows the same thing and eventually the data packet reaches its destination. ("Data

Communication", n.d.). Network routing has many properties and functionalities from which some major ones are explained next.

Routing Types

There are a few different types of routing, mainly unicast, broadcast and multicast routing. As the name implies, *unicast* is the transmission from a single sender to a single recipient, e.g. point-to-point. *Broadcast* packets on the other hand by default are not forwarded by any routers, meaning routers must be configured to forward broadcasts by e.g. creating broadcast domains. There are a few ways to go about this. One way is for the router to create a data packet and then send it to each host one by one using unicast, creating multiple copies of a single data packet with different destination addresses. This consumes bandwidth and all the destination addresses must be known. Another way is to flood received broadcast packets out of all interfaces. This is easy for the router's CPU (Central Processing Unit) but may cause duplicate packet problems. *Multicast* routing is a special case of broadcast routing, because in multicast routing the data is only sent to nodes which want to receive the packets, i.e. the router must know that there are nodes somewhere in the network which wish to receive the multicast data. ("Data Communication", n.d.).

Traffic Engineering

To enable smooth operation of a network, traffic engineering is needed. Most large IP networks run interior gateway protocols (IGPs), such as Open Shortest Path First (OSPF), that select paths based on static link weights. These weights are typically configured by the network operators. Routers then use these protocols to exchange link weights and construct a complete view of the network topology inside an autonomous system (AS). Then each router computes shortest paths (where the length of a path is the sum of the weights on the links) and creates the routing table, discussed earlier, to allow forwarding of each IP packet to the next hop in its route (Fortz, Rexford & Thorup, 2002).

Tunnelling

If there are two geographically separate networks, such as in Inter Networks (Internet), that want to communicate with each other, they may deploy a dedicated line between them or they can pass their data through intermediate networks. Tunnelling is a mechanism by which two or more networks communicate with each other by going through intermediate networking complexities. It requires configuration at both ends. When data enters from one end of Tunnel, it is tagged. This tagged data is then routed inside the intermediate or transit network to reach the other end of Tunnel. When data exists the Tunnel, its tag is removed and delivered to the other part of the network. Both ends seem as if they are directly connected and tagging makes data travel through transit network without any modifications. ("Data Communication", n.d.).

2.2 Routing Protocols

As mentioned in the Introduction, computer networks need more than models and different types of routing. They need routing protocols. Protocols are rules or standards that govern communications when computers send data back and forth, and when

multiple protocols work together the group is collectively known as a protocol suite or protocol stack (Maltz et al., 2004; Blank, 2006), such as TCP/IP (Blank, 2006). Both the sender and receiver, or simply hosts, involved in data transfer must recognize and observe the same protocol, and when they begin the communication, they first must agree on what protocols to use. (Blank, 2006).

Routing protocols provide the intelligence that takes in physical links and transforms them into a network that enables packets to travel from one host to another. By constructing a collective distributed routing state, routing protocols create a "network-wide intelligence" that transforms a collection of individual links and routers into an IP network. A network's routing design is embodied in the configuration of these protocols. Routing designs are used to establish basic reachability and to deal with the following large and complex objectives and constraints: (1) providing resiliency and predictable behaviour; (2) maintaining stable and efficient internal operations; (3) maintaining contractual or business relationships between different administrative domains; (4) coping with complex interactions between a wide set of protocols, which run concurrently and overlap in functionality (Maltz et al., 2004).

Routing protocols are typically classified as either Interior Gateway Protocols (IGPs) used to exchange information inside a network (such as OSPF and OLSR) or an Exterior Gateway Protocol (EGP) used to exchange information between networks (such as BGP). Both IGPs and EGPs share the common goal of exchanging routing information between routers but differ in the features and performance they provide (Maltz et al., 2004).

Each router can use multiple protocols simultaneously; moreover, multiple instances of the same protocol may exist on a single router. To maintain boundaries on how routing information is shared, each routing protocol runs as a separate process on the router and is identified by a process-id. Furthermore, each router has one or more interfaces and each interface has one or more IP addresses and subnets that identify the set of other IP addresses directly reachable from that interface. Through routing protocols, routes can be learned dynamically. While different protocols exchange different types of routing information to convey routes between adjacent processes, the result is the processes learning routes (Maltz et al., 2004). For example, OSPF and IS-IS (Intermediate System - Intermediate System) use link-state advertisements and BGP uses path-vector records ("Understanding BGP", n.d.).

2.2.1 Network Layer Protocols

As hinted earlier, different layers in the OSI model have different protocols they use. This sub-chapter explores three different Network layer protocols, starting with Internet Protocol Version 4.

Internet Protocol Version 4 (IPv4)

IPv4 is the fourth version of the Internet Protocol and it's a 32-bit addressing scheme used as TCP/IP host addressing mechanism, as mentioned earlier. The main goal of IP addressing is then to enable every host on the TCP/IP network to be uniquely identifiable. IPv4 provides hierarchical addressing scheme which enables it to divide the network into sub-networks, each with well-defined number of hosts. IPv4 also has well-defined address spaces to be used as private addresses (not routable on Internet), and

public addresses (provided by ISPs and are routable on internet). (“Data Communication”, n.d).

TCP/IP Protocol Suite

TCP/IP is a protocol suite that uses TCP and IP protocols. It was born to replace the “language” hosts spoke in ARPAnet to enable networking. The “language” to be replaced was NCP (Network Control Program), which had too many limitations and was not robust enough for the growing super network of ARPAnet. First came TCP in 1974 and later in 1978 it had evolved into a suite of protocols called Transmission Control Protocol/Internet Protocol, i.e. TCP/IP. In 1982 it was decided that TCP/IP would replace NCP, which it did in 1983. This allowed ARPAnet to continue growing and even though in 1990 ARPAnet ceased to exist, from the roots of ARPAnet, the Internet began to rise. Since then TCP/IP has continued to evolve to meet the changing requirements of the Internet. (Blank, 2006).

Surprisingly, TCP/IP doesn’t use the OSI model because TCP/IP was developed before the OSI model was published. Instead it uses a similar reference model developed by the US Department of Defense (DoD). This model has only four layers, but they incorporate the same ideas as the OSI model. This as can be seen from figure 2, where the OSI model is pictured on the left side and the DoD model is pictured on the right side. The layers in the TCP/IP model are:

1. Application Layer
2. Transport Layer
3. Internet Layer
4. Network Interface Layer

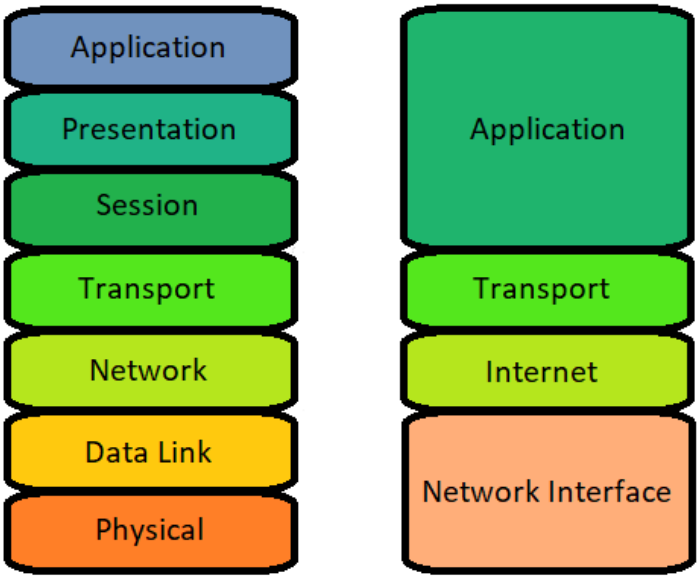

Figure 2. OSI model on the left and DoD model on the right.

TCP/IP’s Application Layer incorporates the same topics as the OSI models Application, Presentation and Session layers combined. The Transport Layer has the same topics as the OSI models Transport layer. The Internet Layer has same topics as the OSI models Network layer. Lastly, the Network Interface Layer has the same topics as the OSI models Data-Link and Physical layers combined.

Worth of note is the Internet Layer of the TCP/IP (or DoD) model. The Internet layer contains several protocols that are responsible for addressing and routing of packets, such as IP, ARP (Address Resolution Protocol), ICMP (Internet Control Message Protocol), and IGMP (Internet Group Message Protocol). (Blank, 2006).

TCP/IP works by IP using ARP to get the hardware address of the destination host. If ARP does not find the IP address in ARP cache, the ARP protocol initiates an ARP request. This request is broadcast on the local network. If the target host is local, IP needs to get the hardware address for the target. If the target host is remote, IP looks in its routing table for an explicit route to that network. Routing table is a table that contains the addresses indicating the best routes to other networks. If there is an explicit route, IP needs to get the hardware address of the gateway listed in the routing table. If there is no explicit route, IP needs to get the hardware address for the default gateway. The default gateway then determines whether the target IP address is on one of its other interfaces or whether the default gateway needs to forward the packet to another router. (Blank, 2006). The process model of TCP/IP is presented in figure 3. Eventually the destination is found.

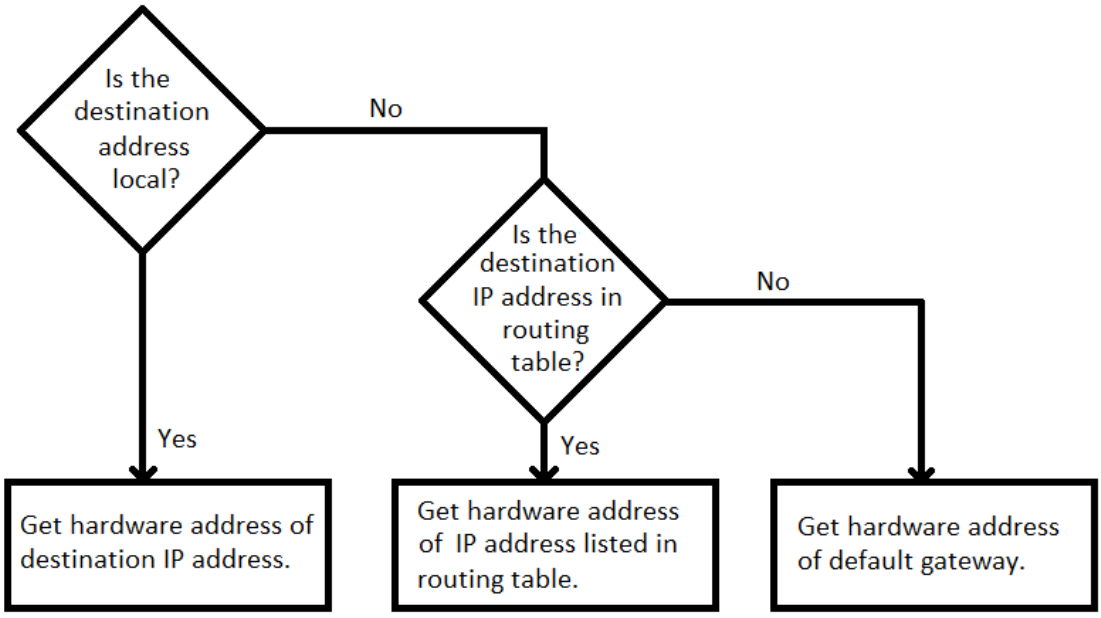

Figure 3. TCP/IP working methodology by Blank (2006).

Even today, it's still the standard for a communications protocol on the Internet. It can be installed and used on virtually every platform. TCP/IP is hardware and software independent; it has failure recovery and can handle high error rates. It's an efficient protocol with low overhead, it can add new networks to the internetwork without service disruption, and furthermore, it has routable data, i.e. a single and meaningful addressing scheme. (Blank, 2006).

Optimized Link State Routing (OLSR) Protocol

Optimized link state routing protocol (OLSR) is used for mobile wireless networks. The protocol is based on the link state algorithm and it's proactive, i.e. table-driven. To explain proactive protocols, let's first discuss reactive protocols. In reactive routing approach, a routing protocol does not take the initiative for finding a route to a destination until it's required. The protocol attempts to discover routes only "on-

demand" by flooding its query in the network. This reduces control traffic overhead at the cost of increased latency. Proactive protocols instead are based on periodic exchange of control messages, causing control traffic overhead. Some messages are sent locally to enable a node to know its local neighbourhood, and some messages are sent in the entire network which permits exchanging the knowledge of topology among all the nodes of the network. Proactive protocols thus immediately provide the required routes when needed, at the cost of bandwidth. ("Data Communication", n.d.; Jacquet, Muhlethaler, Clausen, Laouiti, Qayyum & Viennot, 2001).

OLSR is a link-state protocol and the idea of a pure link state protocol is to flood the entire network with information of links each node has with its neighbour nodes. OLSR is an optimization of this, because instead only a subset of neighbour links are broadcasted through chosen selected nodes, called multipoint relays. Not having all neighbour links in a control message decreases its size and not transmitting these control messages through every node in the network reduces the number of retransmissions. In more detail, this is done by each node in the network selecting a set of nodes in its neighbourhood, which retransmits its packets. Neighbours of node N which are not in node N's multipoint relays (MPRs) set, read and process the packet but do not retransmit the broadcast packet received from node N. This requires each node to maintain a set of neighbours called MPR Selectors. The selection of MPRs happens in such a manner that the set covers all the nodes that are two hops away. ("Data Communication", n.d.; Jacquet et al., 2001).

OLSR keeps the routes for all the destinations in the network, meaning that it's particularly useful for large and dense networks. The protocol is designed to work in a completely distributed manner and thus does not depend upon any central entity. The protocol doesn't require a reliable transmission for its control messages: each node sends its control messages periodically and can therefore sustain a loss of some packets from time to time. It also doesn't need an in-order delivery of its messages: each control message contains a sequence number of most recent information, therefore the re-ordering at the receiving end cannot make the old information interpreted as the recent one. Furthermore, OLSR protocol does hop by hop routing, i.e., each node uses its most recent information to route a packet. Therefore, when a node is moving, its packets can be successfully delivered to it. ("Data Communication", n.d.; Jacquet et al., 2001).

2.2.2 Transport Layer Protocols

As mentioned earlier, the OSI models transport layer uses TCP and UDP protocols. Also mentioned earlier, TCP is one of the most important protocols of Internet Protocols suite. It's used most widely for data transmission. The way TCP works is that the receiver must always send either positive or negative acknowledgement about the data packet to the sender, so that the sender always knows whether the packet has reached its destination or if it needs to be resent. TCP also ensures that the packets reach the end destination in the same order as they were sent. Furthermore, TCP provides error-checking, recovery mechanism, end-to-end communication, flow control, quality of service (QoS) and full duplex server, i.e. it can perform the roles of both receiver and sender. (Blank, 2006; "Data Communication", n.d.).

UDP on the other hand involves the minimum amount of communication mechanisms. In UDP, the receiver of packets does not generate acknowledgements of the received packets and in turn, the sender doesn't wait for any acknowledgements. As such, UDP is said to be an unreliable protocol, yet it provides best effort delivery mechanism

thanks to IP services. Furthermore, UDP is faster and easier to process, causing it to be a suitable protocol for streaming applications such as VoIP (Voice over Internet Protocol). (Blank, 2006; “Data Communication”, n.d.).

2.3 Virtualization

Virtualization is not a new concept, as it was first introduced in the 1960s to allow partitioning of large mainframe hardwares (Bănică et al., 2007; Egi et al., 2010). However, only during the last decade has PC hardware become powerful enough to make running multiple virtual machines (VMs) on one inexpensive box a practical suggestion (Egi et al., 2008).

Virtualization as a term broadly describes the separation of resources for services from the underlying physical delivery of those services. In other words, virtualization refers to the abstraction of fundamental computing resources, giving the users an illusion of sole ownership (Wang et al., 2013). Virtualization allows multiple operating systems (OSs) to share a single physical interface, to maximize the utilization of computer system resources, such as I/O devices (Dong, Yang, Li, Liao, Tian & Guan, 2012). With virtual memory it’s possible, for example, to give software access to more memory than what is physically installed, via background swapping of data to disk storage. Similarly, virtualization can be applied to other IT (Information Technology) infrastructure layers, such as networks and applications (Bănică et al., 2007; “Virtualization Overview”, n.d.). These different virtualization techniques or technologies can be gathered under a term of virtual infrastructure. Virtual infrastructure then provides a layer of abstraction between computing, storage and networking hardware, and the applications, delivering greater IT resource utilization and flexibility (Bănică et al., 2007; “Virtualization Overview”, n.d.). According to Soltesz, Pötzl, Fiuczynski, Bavier and Peterson (2007) VMs are mostly used in software development and testing, but they can be used as part of a software architecture or services as well (“Containers at Google”, n.d) as each VM has their own set of virtual hardware, such as RAM (Random-Access Memory) and CPU, which then run the operating system and applications on top of it (Bănică et al., 2007; Dong et al., 2012).

Virtualization brings some advantages. The virtual machines’ operating system sees a consistent, normalized set of hardware regardless of the actual physical hardware, which is isolated from other virtual machines and the host computer. This is known as isolation (Bănică et al., 2007). Another advantage is the encapsulation of virtual machines, which allows saving, copying and provisioning a virtual machine (Bănică et al., 2007; “Virtualization Overview”, n.d.). Further advantages include partitioning, which means that computing resources are treated as a uniform pool that can be allocated to virtual machines in a controlled manner (Bănică et al., 2007). This brings the ability to run multiple operating systems on a single physical system and share the underlying hardware resources.

Some realizations of these advantages include the possibility for a single machine in a data center to support many different network servers. Isolation ensures that if a virtual server is compromised the damage is limited and the faulty server does not exhaust all OS resources. Another clear advantage is that unused resources from one server can be used by another. Furthermore, perhaps most importantly, different administrators can manage different servers on the same hardware without needing to trust each other, thus enabling many new business models. (Egi et. al., 2008).

There are two typical approaches, or architectures, to virtualization. Hosted approach and hypervisor approach ("Virtualization Overview", n.d.). Recently a third approach, containers, has risen. These three approaches are discussed next.

2.3.1 Hosted Approach

Hosted approach simply put means that the virtualization is done on top of a standard operating system. In hosted approach, the virtualization layer does not have a direct access to the hardware resources, as it has to run on top of an existing OS. Hypervisor, for example, does have direct access to the hardware resources, causing it to be more efficient than hosted approach, enabling greater scalability, robustness and performance. ("Virtualization Overview", n.d.). The hosted approach for virtualization can be seen from figure 4.

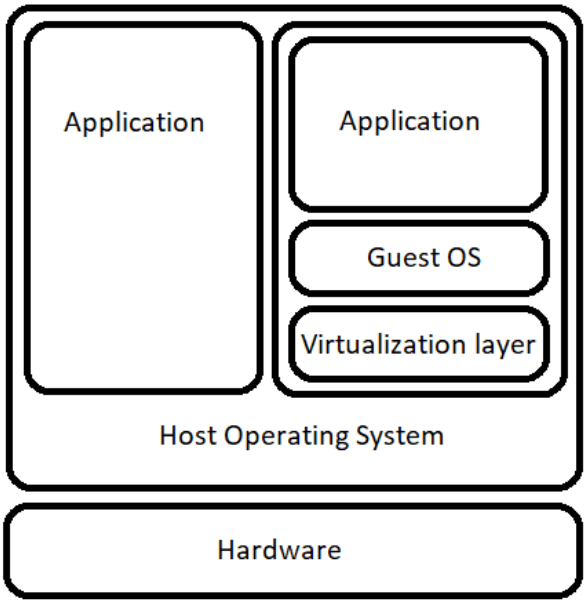

Figure 4. Hosted approach for virtualization.

Other VM architectures range from hardware virtualization up to full software virtualization, including hardware abstraction layer VMs (such as Xen), system call VMs (such as Solaris and VServer), hosted VMs (such as VMWare GSX), emulators (such as QEMU), high-level language VMs (such as Java), and application-level VMs (such as Apache virtual hosting) (Soltesz et al., 2007).

2.3.2 Hypervisor Approach

A hypervisor runs on top of the physical hardware, being the first layer of software installed on a clean x86/x64-based system, which is why hypervisor approach is often also referred as a "bare metal" approach. Hypervisor virtualizes hardware resources to be shared among multiple guest operating systems. Hypervisors incur performance penalties, yet it offers flexibility and isolation among virtual instances. (Rahore et al., 2012). Processors today support virtualization by virtualization hardware assist enhancements. Intel has "Intel VT" while AMD has "AMD-V" (Egi et al., 2008). They enable robust virtualization of the CPU functionality, especially so with multiple cores in the processor (Soltesz et al., 2007). There are some well-known hypervisor

virtualization solutions, mainly KVM (Kernel-based Virtual Machine) and Xen, which are introduced next.

KVM is one example of full virtualization solution for Linux on x86 HW with either Intel VT or AMD-V virtualization extension. KVM consists of a loadable kernel module and a processor specific module. Using KVM, one can run multiple virtual machines running unmodified Linux or Windows images, as is illustrated in figure 5. Each virtual machine has private virtualized hardware: a network card, disk, graphics adapter, etc. KVM is a virtualization module in the Linux kernel that allows the kernel to function as a hypervisor. It was merged into the Linux kernel mainline in kernel version 2.6.20, which was released on February 5, 2007. KVM has also been ported to other operating systems such as FreeBSD and illumos in the form of loadable kernel modules. (Main Page, 2016).

KVM was originally designed for x86 processors and has since been ported to S/390, PowerPC, IA-64, and ARM. It provides hardware-assisted virtualization for a wide variety of guest operating systems including Linux, BSD, Solaris, Windows, Haiku, ReactOS, Plan 9, AROS Research Operating System and OS X. In addition, Android 2.2, GNU/Hurd (Debian K16), Minix 3.1.2a, Solaris 10 U3 and Darwin 8.0.1, together with other operating systems and some newer versions of these listed, are known to work with certain limitations. (Main Page, 2016).

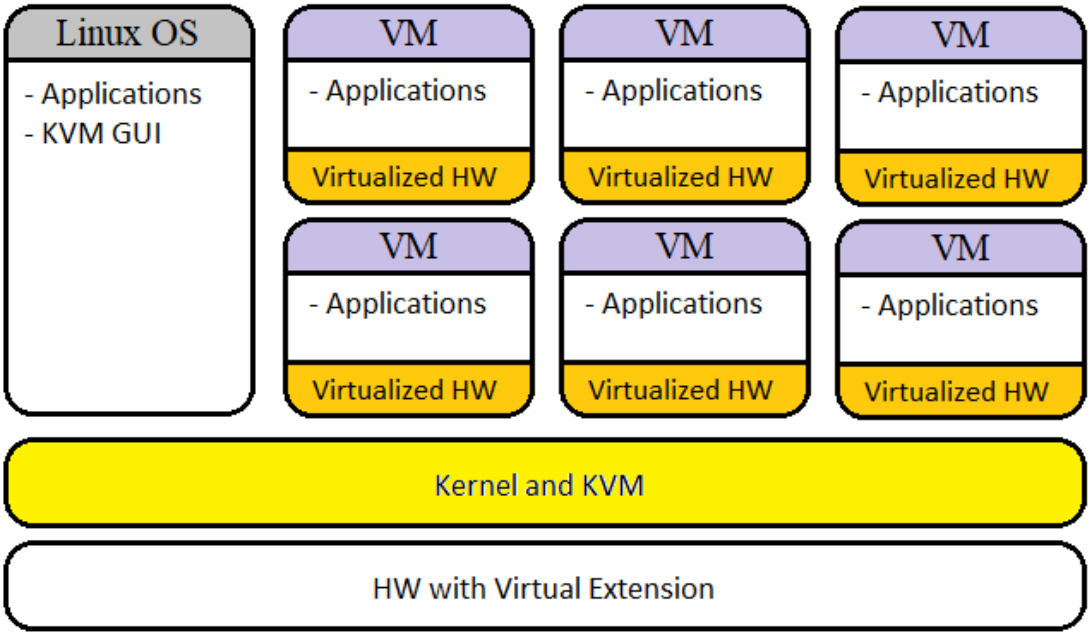

Figure 5. KVM Hypervisor approach to virtualization.

Xen is another popular hypervisor. It allows running many instances of an operating system or different operating systems in parallel on a single machine, or host. In Xen, the hypervisor is designed with multilevel defense protocols as its architecture insulates VMs from each other.

When talking of hypervisor, para-virtualization must be mentioned. Para-virtualization is when operating system compatibility is traded off against performance for certain CPU-bound applications running on system without virtualization hardware assist. Paravirtualized OS's can run on a hypervisor and the guest OS or application is "aware" that it is running within a virtualized environment and has been modified to exploit this

(Dong et al., 2013), offering performance benefits. However, such modified guest OS cannot be migrated back to run on physical HW. (“Virtualization Overview”, n.d.). Both KVM and Xen support para-virtualization. KVM provides paravirtualization support for Linux, OpenBSD, FreeBSD, NetBSD, Plan 9 and Windows guests, using VirtIO API. This includes a paravirtual Ethernet card, disk I/O controller, balloon device, and a VGA graphics interface using SPICE or VMware drivers. (Main Page, 2016).

2.3.3 Container Approach

In a container-based approach, operating system resources (e.g. files, system libraries, routing table) are virtualized to create multiple isolated execution environments on top of same operating system. Each execution environment, i.e. container, may contain its own set of processes, file system, (virtual) network interfaces, routing tables etc. Containers isolate software from its environment and ensure that it works uniformly despite differences, for instance, between development and staging (“What is a Container”, n.d.).

There are two types of containers offered, either application containers or system containers. Docker offers application containers and they say that an application container is a standard unit of software that packages up code and all its dependencies, so that the application can run quickly and reliably from one computing environment to another (“What is a Container”, n.d.). Linux Containers offer system containers, which propose an environment as close as possible as the one you'd get from a VM but with significantly less overhead (“Linux Containers”, n.d.). This is because system containers don’t need virtualized HW or Linux kernel, as all containers use the same Linux kernel that the hosting OS uses. Everything at Google runs in containers, from Gmail to YouTube to Search (“Containers at Google”, n.d.), which speaks volumes. However, according to Xen and Rahore, Hidell and Sjodin (2012), while container-based virtualization is considered more efficient in terms of performance, containers lack flexibility and security. Opening one system container’s kernel affects all the other containers. Furthermore, because system containers share a common kernel, non-Linux operating systems, like Windows, are not supported (Main Page, 2016).

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Docker container technology was launched in 2013 as an open source Docker Engine. It leveraged existing computing concepts around containers and specifically in the Linux world, primitives known as cgroups and namespaces. Docker's technology is unique because it focuses on the requirements of developers and systems operators to separate application dependencies from infrastructure. Success in the Linux world drove a partnership with Microsoft that brought Docker containers and its functionality to Windows Server. (“What is a Container”, n.d.). Figure 6. illustrates how Docker works; there is a Docker platform on top if which application containers can be run, free from the underlying operating system.

Linux Containers, on the other hand, is actually an umbrella project behind LXC, LXD and LXFS, which provide the functionality of system containers. The goal of Linux Containers is to offer a distro and vendor neutral environment for the development of Linux container technologies. Using LXC it’s possible to create and manage system or application containers. It’s a low-level set of tools, templates, libraries and language bindings and it’s very flexible. LXD is built on top of LXC, giving the user an improved user experience in managing system containers, by using a single command line tool. It

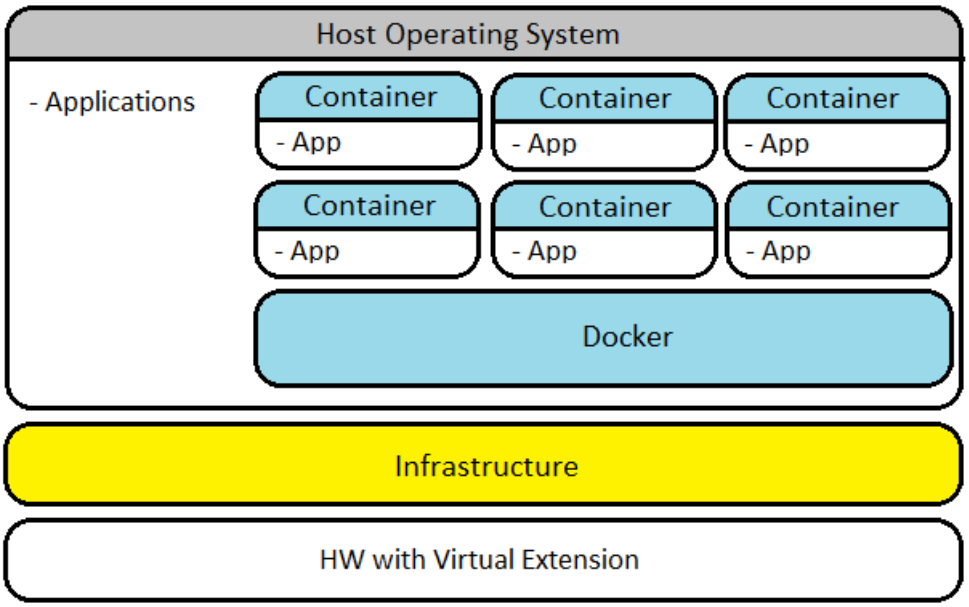

Figure 6. Docker container architecture.

works well over a network in a transparent way. LXFS is a simple userspace solution for overcoming challenges relating to running systemd based containers as a regular unprivileged user while still allowing system inside the container to interact with cgroups. Specifically, it provides (1) a set of files which can be bind-mounted over their /proc originals to provide CGroup-aware values, and (2) a cgroupfs-like tree which is container aware. ("Linux Containers", n.d.).

System container provides a shared, virtualized OS image consisting of a root file system, a set of system libraries and executables. Each container can be booted, shut down, and rebooted just like a regular operating system. Resources such as disk space, CPU guarantees, memory, etc. are assigned to each container when it is created, yet often can be dynamically varied at run time. To applications and the user of a container-based system, the container appears just like a separate host. (Soltesz et al., 2007). Figure 7 illustrates a situation of having multiple containers on a single Linux operating system.

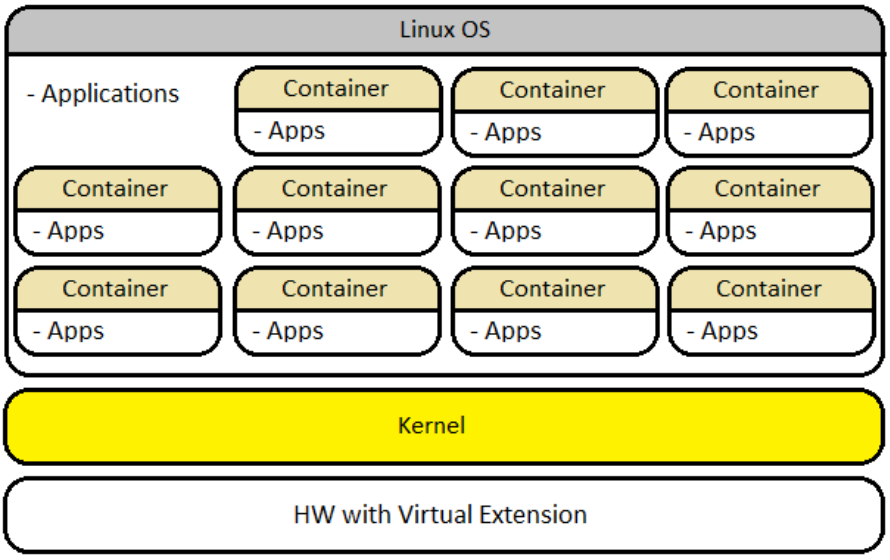

Figure 7. System Container approach.

The main difference between application and system containers is that system containers contain virtually a fully capable operating system, whereas application containers contain only everything needed for running that certain application.

2.3.4 Containers and Virtual Machines

Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize only the operating system or environment instead of also virtualizing the hardware. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require less hardware and operating systems. (“Linux Containers”, n.d.).

Because hypervisors aim to provide full isolation between virtual machines, there’s no support for sharing between VM’s. They are often deployed to let a single machine host multiple, unrelated applications. Hypervisors strongly prefer full isolation over sharing. However, considering that having multiple VMs running the same kernel and similar operating systems, the degree of isolation comes at the cost of efficiency relative to running all applications on a single kernel. Contrast to this, container virtualization techniques enable efficient use of resources either in terms of performance or in terms of scalability (Soltesz et al., 2007).

Containers approach to security isolation directly involves internal operating system objects, such as PIDs, UIDs and IPC, UNIX and so on. The basic techniques used to securely use these objects involve (1) separation of name spaces, i.e., contexts, and (2) access controls, i.e., filters. Access separated name spaces mean that global identifies live in completely different spaces, do not have pointers to objects outside their own name spaces and thus can’t get access to objects outside of its name space. Filters control access to kernel objects with runtime checks to determine whether the container has the appropriate permission. (Soltesz et al., 2007).

The usage for container virtualization is based on the observation that sometimes it’s acceptable in real-world scenario to trade isolation for efficiency. Having only a single underlying kernel image will bring that efficiency (Soltesz et al., 2007). There exists no VM technology that achieves the ideal of maximizing both efficiency and isolation. It’s good to note though, that having both hypervisor and containers is a possibility as they are not mutually exclusive (Soltesz et al., 2007). Then one would have a hypervisor running VMs with containers inside.

2.3.5 Performance of Containers and Hypervisor

Soltesz et al. (2007) did several tests when comparing containers and hypervisors. For hypervisor technology they used Xen version 3.0.4 and for container technology they used VServer version 2.0.3-rc 1, both running on top of Linux Kernel 2.6.16.33. For reference, at the time of this study, the current version for Xen is 4.11, the current version for VServer is 2.6.22 and the current version for latest stable Linux Kernel is 4.20.13. They measured network operations performance, along with disk and CPU performance. For all tests, VServer, i.e. container approach, performance was comparable to an unvirtualized Linux kernel. Yet, the comparison shows that although Xen3 included new features and optimizations, the overhead required by the virtual memory sub-system still introduced an overhead of up to 49% for shell execution. In

terms of absolute performance on server-type workloads, Xen3 lagged an unvirtualized system by up to 40% for network throughput while demanding a greater CPU load and 50% longer for disk intensive workloads. (Soltesz et al., 2007).

In their tests to test network operations performance, Soltesz et al. (2007) used Iperf, which is an established tool for measuring link throughput with TCP or UDP traffic. They used it to measure TCP bandwidth between a pair of systems. Both raw throughput and the CPU utilization were observed on the receiver, in two separate experiments to avoid measurement overhead interfering with throughput. On a multicore processor VServer compared closely to Linux, whereas Xen3 could not achieve line rate. Soltesz et al. (2007) explained that Xen3 saturated the CPU it shared with the host VM.

To test disk performance, Soltesz et al. (2007) used DD benchmark to write a 6GB file to a scratch device. They found Linux and VServer to have identical performance, as the code path for both is basically identical. In contrast, for Xen3 they found significant slowdown for both uniprocessing and symmetric multiprocessing types of CPUs. This was due to additional buffering, copying, and synchronization between the host VM and guest VM to write blocks to disk. (Soltesz et al., 2007).

A virtualization solution with strong isolation would partition the share of CPU time, buffer cache, and memory bandwidth perfectly among all active VMs and maintain the same aggregate throughput as the number of active VMs increased. However, for each additional VM, there is a linear increase in the number of processes and the number of I/O requests. Since it is difficult to perfectly isolate all performance effects, the intensity of the workload adds increasingly more pressure to the system and eventually, total throughput is reduced. Two factors contribute to the higher average performance of VServer: lower overhead imposed by the container approach and a better CPU scheduler for keeping competing VMs progressing at the same rate. As a result, there is simply more CPU time left to serve clients at increasing scale. (Soltesz et al., 2007).

2.4 Network Virtualization

Network virtualization has become a popular topic in recent years (Wang et al., 2013). This chapter will present an overview of network virtualization, what it is and how it's accomplished. Then this chapter will present mature and experimental technologies concerning network virtualization. Lastly, the chapter will present an overview of how to simulate network traffic.

Network virtualization is any form of partitioning or combining a set of network resources, and presenting it to users so that each user, through its set of the partitioned or combined resources, has a unique and separate view of the network (Wang et al., 2013). Virtual networking allows to network virtual machines in the same way that one would network physical machines and it enables building complex networks within a single server host or across multiple server hosts, for production deployments or development and testing purposes ("Virtual Networking", n.d.).

A virtual network comprises three components: (1) virtual hosts, which run software and forward packets; (2) virtual links, which transport packets between virtual hosts; and (3) connectors to connect virtual hosts to virtual links in point-to-point or point-to-multipoint -mode (Bhatia et al., 2008). In network virtualization, as in virtualization overall, there are different options on how to do it. Egi et al. (2008) refer to OS-level

virtualization, which is the same as system containers mentioned earlier, and say that they are an effective solution. Another way to do it is using hypervisors, such as Xen (Bhatia et al., 2008).

Virtual switches allow virtual machines on the same server host to communicate with each other using the same protocols that would be used over physical switches. A virtual machine can be configured with one or more virtual Ethernet adapters, each of which each has its own IP address and MAC address. As a result, virtual machines have the same properties as physical machines from a networking standpoint. ("Virtual Networking", n.d.). This is illustrated in figure 8.

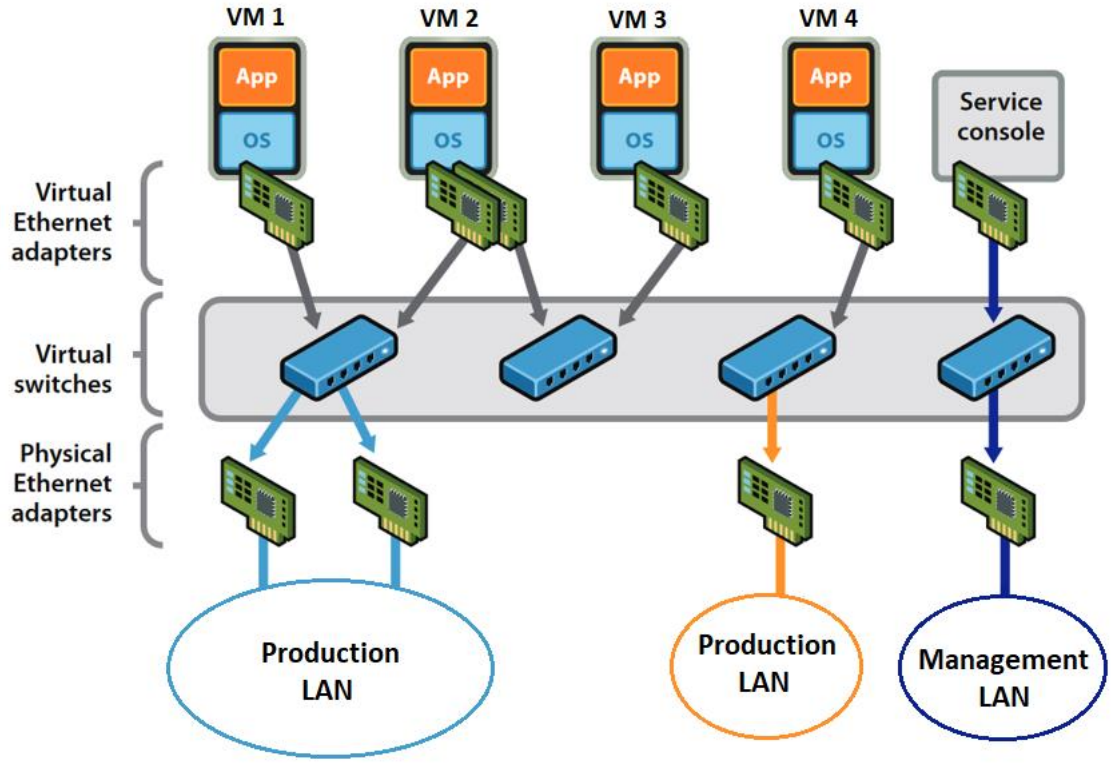

Figure 8. Example virtual network architecture by "Virtual Networking" (n.d.)

The advantages of virtualization in general, e.g. isolation, encapsulation and independent administration, carry over to network virtualisation. Virtual LANs (VLANs) and Virtual Private Networks (VPNs) allow a single network to be subdivided and to have different, isolated users of the network. Extending the idea of true virtualisation to network resources and to routers has many benefits: a single virtual router platform can provide independent routing for multiple networks in a manner that permits independent management of those networks. (Egi et. al., 2008).

Wang et al. (2013) recognize three different virtual network types:

1) Overlay Networks
2) Virtual Private Networks (VPN)
3) Virtual Sharing Network (VSN)

An overlay network is one built upon an existing network, mainly using tunnelling and encapsulation technologies. A VPN is an assembly of private networks that connect to each other but are isolated from public networks such as the Internet. VSN refers to technologies that support the sharing of physical resources among multiple network instances, while providing clear outlining between these instances.

Furthermore, Bhatia et al. (2008) recognized three design goals regarding network virtualization; (1) speed, (2) flexibility, and (3) isolation. Speed translates to the platform being able to forward packets at high rates. The packet forwarding rates should approach that of "native" kernel packet forwarding rates. Flexibility translates to the platform being able to allow experimenters to modify routing protocols, congestion control parameters, forwarding tables and algorithms, and, if possible, the format of the packets themselves. Isolation translates to the platform being able to allow multiple experiments to run simultaneously over a single physical infrastructure without interfering with each other's namespaces or resource allocations. The design goals can be difficult to achieve, but they help in reducing costs, providing flexibility, and keeping up the pace with rapid development cycles (Bhatia et al., 2008).

2.4.1 Mature Technologies

There are three mature technologies used in network virtualization: link virtualization, router virtualization and NIC virtualization, which will be presented next.

Link Virtualization

Link virtualization can be done through multiplexing a physical channel or by virtualizing data paths. In physical channel multiplexing, the multiplexing performs a function very similar to virtualization, i.e. the physical medium is split into channels and the sender and receiver are under an illusion that they own the physical medium (Wang et al., 2013). Data path virtualization on the other hand refer to technologies that manipulate the packets carried on a channel, instead of manipulating the channel. Such a virtual link does not directly depend on the physical properties of the links, rather it is provisioned by nodes. Nodes use various technologies to direct data along these virtual links, i.e. data paths (Wang et al., 2013).

Two popular technologies used in data path virtualization are labels and tunnelling. Labels might also be called tags or IDs and they occupy certain fields in a packet header and serve as identification and sharing mechanisms (Wang et al., 2013). Using these tags, nodes can traffic the packets to the right direction. This enables different VLANs to share a single physical device. Tunnels on the other hand, as mentioned, allow connecting network devices that are not physically near. Popular tunnelling technologies include generic routing encapsulation (GRE) tunnels and Internet Protocol security (IPsec) tunnels. Essentially, tunnels are overlay links and form the fundamental building blocks of overlay networks (Wang et al., 2013).

Router Virtualization

Rahore et al. (2012) explain that a virtual router is just like a physical router: is has a routing table, routing protocols, packet filtering rules, management interface, and so on. Multiple virtual routers can be running on the same hardware platform, sharing the available resources (Wang et al., 2013). This can be achieved using a virtualization technology that divides the system into multiple virtual environments i.e. system virtualization. A host environment is then responsible for allocating and managing resources to the virtual routers. The virtualization technology ensures resource isolation among virtual routers, i.e. one virtual router should not be able to see or access resources (e.g. the routing table) of other virtual routers (Rahore et al, 2012).

Virtual routers are often associated with performance penalties due to overhead from virtualization, but Rahore et al. (2012) propose a forwarding architecture for virtual routers based on multicore hardware where virtual routers can run in parallel on different CPU cores. This reduces resource contention among virtual routers and results in improved performance and isolation. However, hardware based I/O virtualization is essential for this architecture to work. Furthermore, they have shown that hardware assisted virtual routers can achieve better total throughput than a non-virtualized router on a multi-core platform.

The aim when implementing a virtual router should be to keep a packet as deep as possible inside a cache hierarchy (i.e., close to the cores) while distributing the packet processing over as many spare cores as possible within the same cache hierarchy. This ensures that processing is not CPU-limited since multiple cores are in use, while reducing expensive accesses to main memory. A software router's internal organization can be viewed as a graph of interconnected packet processing elements. (Egi et al., 2010).

According to Egi et al. (2008), it's possible to compose a virtual software router in three different configurations where packet forwarding is under taken by using one of the following schemes:

a) Common forwarding plane: Figure 9(a).
b) Interface direct mapping: Figure 9(b).
c) Hybrid forwarding plane: Figure 9(c).

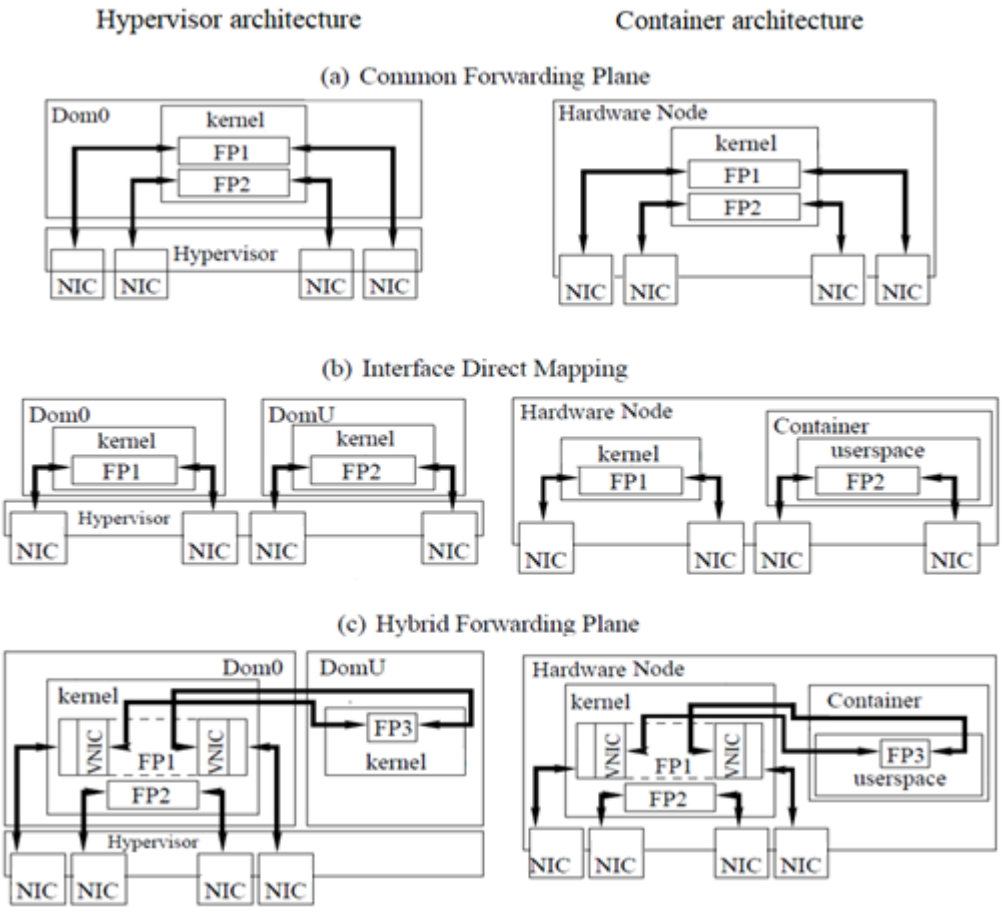

Figure 9. Composing virtual routers using hypervisors or containers, by Egi et al. (2008)

The first configuration, shown in figure 9(a), has a common forwarding domain for all virtual software routers on the box. This is probably best suited to situations where there is no need to isolate custom forwarding paths in separate domains for reasons of security. Using a common forwarding domain for a number of different virtual routers is very efficient and allows a large number of virtual routers to share the same interfaces. The second configuration, figure 9(b), directly maps interfaces into guest domains enabling forwarding planes to safely run untrusted forwarding paths. The third possible configuration, figure 9(c), is where a proportion of the packets from a set of interfaces is filtered in to a guest domain for further processing. (Egi et al., 2008).

A critical issue for a virtual router platform is how to share interfaces between virtualised routers. If an interface needs to be shared between virtualised routers, there are three main options for doing so:

- Use software to demultiplex the flows and process the packets as they arrive.
- Use software to demultiplex the flows, but then requeue them on a per virtual router basis. This allows fairer scheduling between virtual routers.
- Use hardware de-multiplexing in the NIC, and present multiple hardware queues to the OS.

Simple software de-multiplexing has the great advantage of simplicity, and when all the forwarding planes are implemented in the same OS domain, this scales to very large numbers of virtual routers, as shown in Figure 10. Only when the CPU caches start to thrash does performance dip. (Egi et al., 2008).

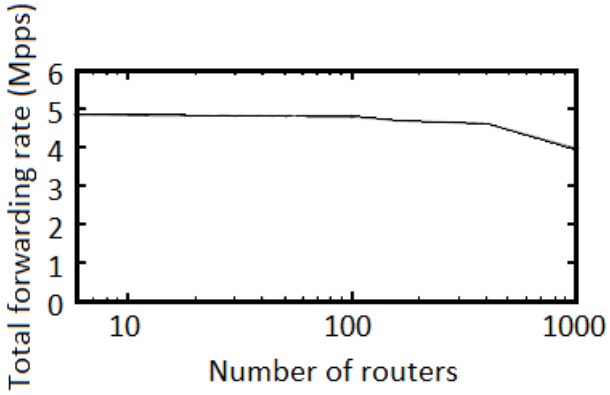

Figure 10. Forwarding rate and number of VMs, by Egi et al. (2008).

The downside of simple software de-multiplexing is fairness. Packets are processed as far as the central queue in the order in which they arrive on the incoming interface, irrespective of the intended prioritization of the virtual routers. If fairness and isolation are required amongst shared forwarding engines, hardware packet classification is needed on the NIC. (Egi et al., 2008).

The forwarding performance of modern software routers is rather good. An inexpensive modern x86 rack-mount server can forward minimum-sized packets at several gigabits per second and larger packets at much higher rates. The fundamental limit on forwarding performance is currently memory latency. Although these systems have huge memory bandwidth, they frequently cannot make use of this due to poor locality of reference in the DMA (Direct Memory Access) controller hardware and PCIe arbitration. Smarter hardware could make much better use of memory bandwidth. With more cores it may be feasible to turn off DMA and to dedicate cores to the role of

transferring data between the NICs and memory in the best way to maintain memory throughput. (Egi et al., 2008).

NIC Virtualization

According to Wang et al. (2013), NIC virtualization can either be software-enabled or hardware-assisted. In software-enabled NIC virtualization there are virtual NICs (vNICs), which are a software emulation of a physical NIC. Their IP and MAC addresses can be controlled. The most common vNIC clients include a virtual OS or an OS-level virtualization instance. A virtual switch (vSwitch) is a software emulation of a physical switch which, however, may not support all features of a physical switch. A vSwitch performs functions such as traffic switching, multiplexing, and scheduling and bridges vNICs with physical NIC(s) if needed.

The links between vNIC and vSwitch are software-emulated links, which are not to be confused with the "virtual links". The bandwidth of these emulated links is only limited by the processing capabilities of the host itself (Wang et al., 2013). Another way to virtualize NIC is with hardware-assisted virtualization, i.e. Single Root I/O virtualization, which is discussed next.

SR-IOV

A regular NIC only provides one peripheral component interconnect express (PCIe) channel, which usually becomes the I/O bottleneck in a VM-centric environment (Wang et al., 2013). To solve this, support for I/O virtualization has been provided by hardware on NICs, and this is known as Single Root I/O Virtualization, i.e. SR-IOV (Rahore et al., 2012). SR-IOV is an industry standard and the main idea is to offload packet handling from the system CPU, as is the case in normal I/O virtualization, to make them directly available inside a virtual router, i.e. packet handling is done by a vNIC running inside the physical NIC. More precisely, it allows a single physical NIC to create multiple NIC instances, known as Virtual Functions (VFs) (Dong et al., 2012; Wang et al., 2013), which provides an Ethernet-like interface (Rahore et al., 2012). Instead of connecting vNIC clients to a vSwitch, every VM can be directly mapped to a Virtual Function for direct access of NICs resources (Wang et al., 2013). SR-IOV can be used with hypervisor-based technologies, such as KVM. Furthermore, it also provides packet classification features, which can help improve in isolation. However, only a limited number of VFs are supported on top of a physical interface. For instance, the Intel 1Gbps NIC supports 8 VFs per port whereas the 10Gbps NIC supports 64 VFs per port. (Rahore et al., 2012).

SR-IOV generally provides performance benefits (Dong et al., 2012; Rahore et al., 2012), such as better throughput, scalability, lower CPU utilization (Wang et al., 2013) and even resource sharing (Dong et al., 2012). Rahore et al. (2012) did a comparison of performance between macvlan and SR-IOV based virtual routers. Macvlan is a Linux network driver that exposes host interfaces directly to VMs or Containers running in the host and it has been integrated in Linux kernel since version 2.6.x (Rahore et al., 2012). Rahore et al. (2012) found that a higher forwarding performance is achieved using SR-IOV and that SR-IOV scales better with the number of CPU cores used. With four CPU cores, SR-IOV achieved almost 32.33% better throughput than macvlan. Furthermore, they found that SR-IOV also results in better throughput than a nonvirtualized IP forwarder.

2.4.2 Experimental Technologies

Wang et al. (2013) have researched experimental technologies that can help in network virtualization. They mentioned four experimental technologies, which are discussed next: PlanetLab, Emulab, VIOLIN and G-Lab.

PlanetLab is a research testbed jointly established in 2002 by Princeton University, Intel, UC Berkeley, and the University of Washington. PlanetLab is composed of nodes, called PlanetNodes, that are dedicated servers running a customized Linux OS. These PlanetNodes are distributed around the world and they can spawn VM slices at users request. The communication between PlanetNodes happens through the Internet, i.e. PlanetLab forms an overlay network over the Internet. Due to the international distribution of nodes, testbeds of all sizes, topologies or geographical coverage may be requested and constructed. There are enhancements, such as VINI and Trellis, build to the PlanetLab to provide some extra functionality, such as tunneling between virtual nodes. (Wang et al., 2013).

Emulab is a similar experimental research network testbed developed at the University of Utah. It mainly emulates network properties based on user requests over a physical network. Each physical site contains a set of Dummynet nodes, ns-Emulates (NSE) and PCs connected through a LAN. Dummynet is an emulation tool used to test networking protocols while NSE generates software-simulated traffic. (Wang et al., 2013).

VIOLIN (internetworking on overlay infrastructure) is a software that is implemented within a user-mode Linux (UML) container, which runs in a PlanetLab slice. A UML container is a virtual host, a virtual switch, or a virtual router. VIOLIN is independent of PlanetLab and it extends UML to enable UDP tunnels between the UML containers. (Wang et al., 2013).

German Lab (G-Lab) is an initiative on the Future Internet sponsored by the German Federal Ministry of Education and Research. It uses PlanetLab-compliant technology and has deployed a testbed in Germany. Based on G-Lab, a topology management tool provides a graphical user interface that allows users to design and use a virtual network by simply dragging and organizing network device and link icons. It's operated under OneLab project, funded by the European Commission. The main goals of OneLab include extending PlanetLab to wireless environments and federating with more testbeds, such as G-Lab. (Wang et al., 2013).

2.4.3 Simulation

When virtualization is not enough, one can try to simulate. One such tool that can be used for network simulation is Scapy. Scapy is a Python-based interactive packet manipulation program that mainly does two things: sending packets and receiving answers. User defines a set of packets, Scapy sends them, receives answers, matches requests with answers and returns a list of packet couples (request, answer) and a list of unmatched packets. This enables the user to send, sniff and dissect and forge network packets. This capability further allows construction of tools that can probe, scan or attack networks. Scapy can handle most classical tasks like scanning, tracerouting, probing, unit tests or network discovery. Lastly, Scapy allows building more high-level functions, such as one that pings a whole network and gives a list of the machines that answer. (Scapy - About Scapy, n.d.).

3. Research Method

This chapter describes the research problem and research method in further detail. The problem area is complicated but the research problem itself is quite simple.

3.1 Research Problem

Creating large-scale networks for testing or development purposes can be hard because of the amount of required hardware. It's not cheap either, as special routers can be quite expensive. Setting up such a physical test environment with hundreds or thousands of nodes would take a long time and would require maintenance and much related supportive hardware, such as cables and power outlets or generators. Adding organizational time constraints to this makes it even less feasible.

There are many things that can be tested regarding routing. One might want to test different network topologies and network types, such as WAN and LAN, in one network. One might want to also test different routing types, e.g. unicast and multicast. Then there are different communication and routing protocols at different levels of the OSI/ISO model, such as OLSR and OSPF. Lastly, there are all the different network routing features, such as tunnelling. Testing all of these features can't be simply tested in a PlanetLab environment as it consists mainly of connections in the Internet. The company this study is done for has their own routing software and they are able to test the aspects of routing they want in small-scale environments, i.e. with physical devices and connections, but a problem arises when considering a use case where there are hundreds of nodes running the company's routing software. It's not possible to test in any normal test environment. However, the purpose of this study is not to create an artifact that supports testing of all the previously mentioned aspects of routing, but to create a platform that enables hosting hundreds of nodes and that can be extended later on to support more features as well.

Furthermore, there is an additional problem in this study. There is no access to the Internet in the development and testing area of the organization, where the implementation was done. To work around this, a mirror has been implemented in the development and testing area, containing some packages for popular Linux OS's such as Debian and Ubuntu. The mirror is not up to date because each packet must be scanned for security reasons by the organization before it can be used. This causes limitations on the study because not everything is available and those packages that are available might not be most recently updated.

To begin to understand how it's possible to build an environment, or a platform, that allows testing of such many nodes and network features, it must be understood what kind of virtualization technologies there are, how they differ and what capabilities they have. That and building the artifact constitute the main research problem in this study.

3.2 Artifacts Requirements

As mentioned earlier, there are some requirements from the company and from a general description of what the artifact should be able to do. The requirements are shown in table 1. The main goal of the requirements is to ensure that the artifact is able to fulfil the main goal of the study, i.e. simulating many nodes. In the Evaluation chapter the artifact is evaluated against these requirements.

Table 1. Requirements of the artifact.

Requirement number and name	**Description**
R1: Large-scale Simulation	The artifact must be able to simulate a network of many hundred nodes.
R2: Encapsulation	The virtual nodes must be encapsulated.
R3: Configurability	The artifact must be configurable.
R4: Adaptability	The artifact must be adaptable.
R5: Outside Connection	The artifact must be able to connect to physical devices outside the artifact.
R6: Replicability of Nodes	The virtual nodes must be easy to replicate.

Furthermore, the company wanted a demonstration and instruction manual on how to create and use the artifact, but they are not part of the requirements for the artifact itself, and thus the demonstration is not take into consideration in this study. However, the instruction manual is in Appendix A, for replicability of this study.

3.3 Design Science Research Method

As mentioned in the Introductions, this study uses a Design Science Research methodology by Hevner et al. (2004) to design and produce an artifact that tries to answer a problem a company has. According Hevner et al. (2004), the design-science paradigm is to create "what is effective", something of utility, and that these innovative creations are to be evaluated. The creation, or artifact, is to enable organizations to address important information-related issues. DSR in information systems has two processes, build and evaluate, and four types of artifacts: constructs, models, methods and instantiations. (Hevner et al., 2004). The artifact type built in this study is a construct.

Hevner et al. (2004) have built a conceptual framework for understanding, executing, and evaluating IS research, which is shown in figure 11. The framework has three stakeholders: Environment, IS Research and Knowledge Base. Environment is where the problem exists, i.e. in this case the company who wants to find a way to simulate large networks. Generally, it consists of people, organizations and the technology they use. IS Research is the next stakeholder and it's where the building and evaluating of the artifact happens. The artifact tries to solve the problem the Environment has. Environment is where from the IS research gets its purpose, or relevance, by giving IS research a problem to solve. Knowledge Base is the last stakeholder and it then provides

the foundations and methodologies for the IS Research. The foundations provide theories, frameworks, etc. for building the artifact, whereas the methodologies provide guidelines for the artifacts evaluation phase. Both the foundations and methodologies must be used with rigor. Lastly, the IS Research provides more information to the Knowledge Base at the completion of the research. (Hevner et al., 2004). This is the way that DSR benefits both the academics and the environment, by creating a solution to problem and also by adding to the existing knowledge base.

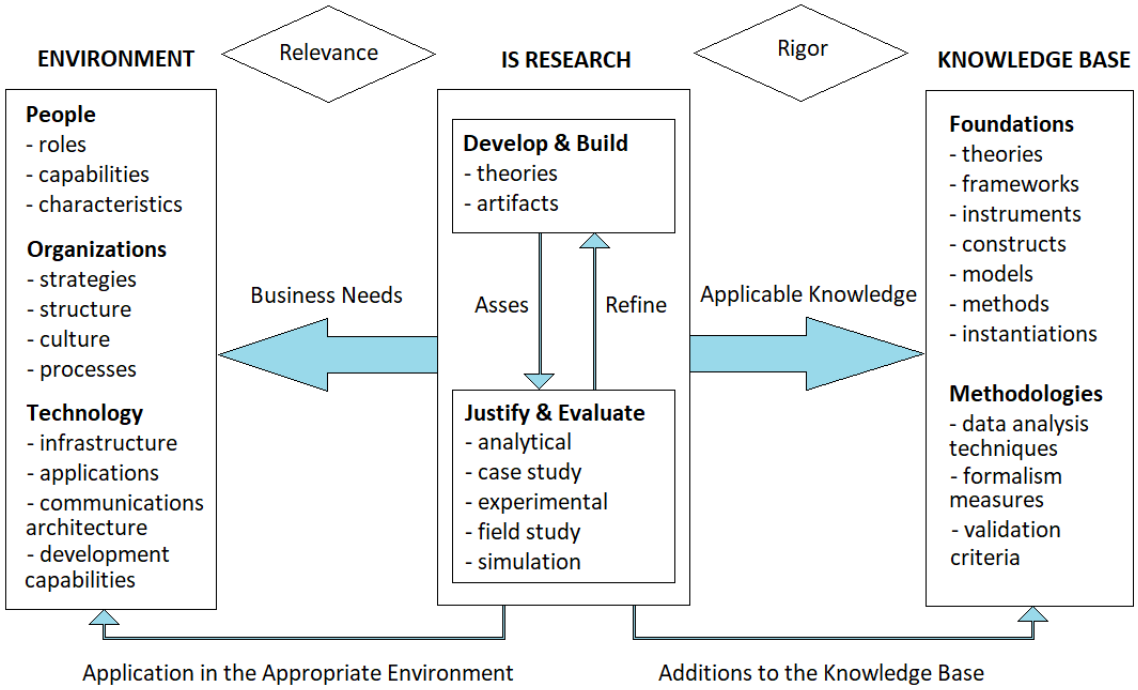

Figure 11. Information Systems Research Framework by Hevner et al. (2004).

The DSR methodology by Hevner et al. (2004) also has seven guidelines for conducting a design science research. The guidelines and their summaries are presented in table 2. The first guideline of a DSR study is to produce a working artifact in the form of a construct, a model, a method, or an instantiation. This ensures that something worthwhile is produced. The second guideline is to ensure that the artifact answers to some important and relevant business problem. In other words, that which is produced, is also useful. The third guideline is to use design evaluation methods to demonstrate the utility, quality and efficacy of the designed artifact. The fourth guideline is to have clear contributions in the research area, as often that is the ultimate assessment for any research, i.e. what new is contributed by the study. The fifth guideline is to use rigorous methods in both the construction and evaluation of the artifact, meaning that there needs to be a methodology to the production and evaluation of the artifact. The sixth guideline is to refine and make the problem real by defining the relevant means, ends and laws of the problem, and then utilize available means to reach desired ends while satisfying laws in the problem environment. The seventh, and last, guideline is to communicate the research in such way that the artifact can be constructed by others with similar organizational resources, and that others can decide themselves whether organization resources should be committed to constructing and using such artifact. In other words, communicate for technology-oriented as well was management-oriented audiences.

Table 2. DSR guidelines and their descriptions.

Guideline	**Summary**
1. Design as an Artifact	Produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
2. Problem Relevance	Develop technology-based solutions to important and relevant business problems.
3. Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods
4. Research Contributions	Provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies
5. Research Rigor	Application of rigorous methods in both the construction and evaluation of the design artifact.
6. Design as a Search Process	Utilizing available means to reach desired ends while satisfying laws in the problem environment.
7. Communication of Research	Present the research effectively both to technology-oriented as well as management-oriented audiences.

Following the seven guidelines ensures an effective design-science research. These guidelines and how they have been accomplished in the study are discussed in Chapter 5, Evaluation.

4. The Artifact

This chapter is divided into two parts: planning and implementation. Planning chapter describes what happened before the actual implementation was started. Implementation chapter then describes how the artifact was implemented and what the artifact consists of. Worth of note is that the words “node” and “container” are used irreplaceable, as each container is supposed to also be a node.

4.1 Planning

Prior to the implementation, there were discussions about how to approach the subject of network virtualization and a few meetings were held within the organization concerning what technologies were available and what the artifact should be able to do. Furthermore, a weekly meeting was held between the author of this paper and three co-workers, in which the progress and design choices were discussed. During the starting discussions it was considered whether to use hypervisor technology or container technology to achieve the first requirement, R1. Pictures of container and hypervisor approach were drawn, which are already shown in the Hypervisor Approach and Container Approach chapters, located in chapter 2.3. The capabilities of hypervisors or containers were not yet fully known, so an idea of simulating a backbone network using Scapy was introduced, as it had been used before in a similar work. The idea was to ensure that the requirement R1 was met. Networking styles and management options were compared between container approach and hypervisor approach because they differ, as can be seen from figure 12.

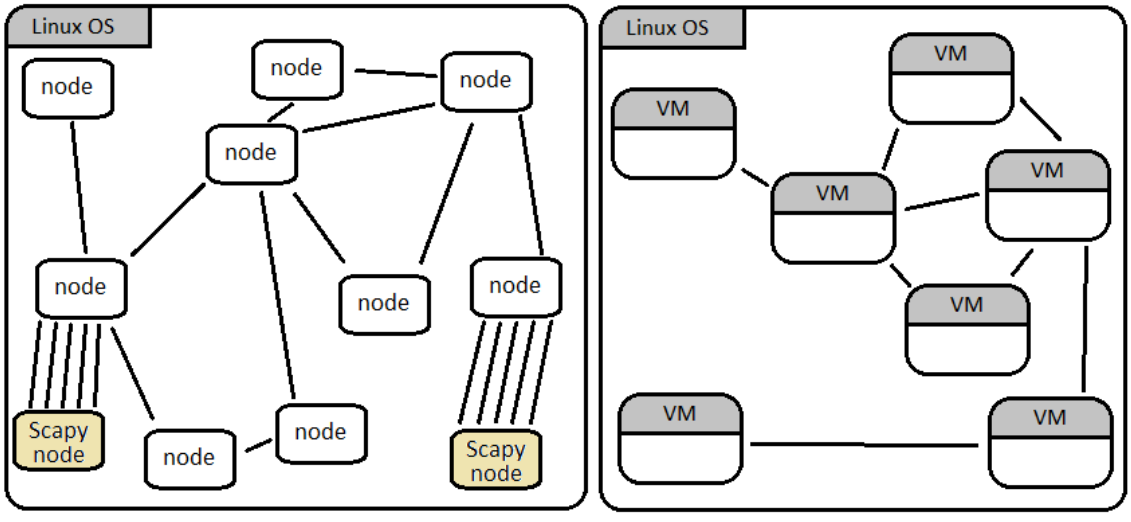

Figure 12. (a) Container approach and (b) hypervisor approach to networking.

From the first discussions the requirements were also drawn, which helped in deciding what technologies to use. The artifact had to be configurable in such a way that the fundamental blocks could be changed if the need arose to e.g. better simulate the router software developed by the organization. Resulting from the discussions, it was decided

to use hypervisor approach together with container approach, because regarding requirements R3 and R4, Configurability and Adaptability respectively, this was the best option for later on e.g. if integrating parts of the organizations router software. Furthermore, having a hypervisor brings better encapsulation (requirement R2) than having only containers. The fusion of hypervisor and container approach is shown in figure 13.

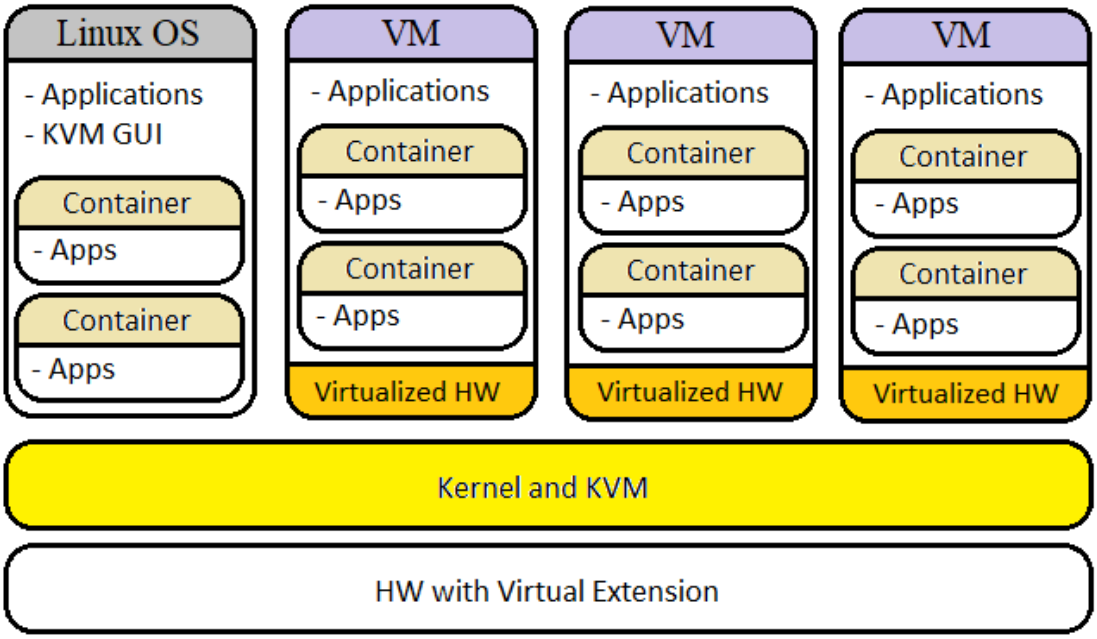

Figure 13. Using hypervisor and container approach together.

Furthermore, with a hybrid approach, the networking is logically clearer, as can be seen from figure 14. In the figure, each node is actually a container. When using VMs and containers within, it can be easier to modify the whole picture or add physical devices (requirement R5), because a VM is a full operating system whereas a container is not. It's also possible to create a certain type of topology within a VM and then copy that VM to create more of that type of topology. Comparing to handling hundreds or thousands of containers in the same space, this seemed like an easier solution to manage.

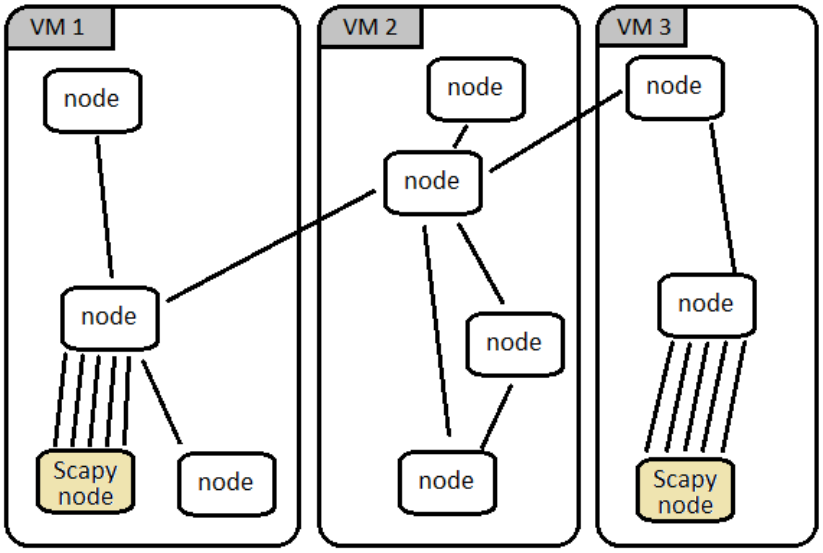

Figure 14. Hybrid approach to networking.

For hypervisor approach, KVM and Xen were initially considered but KVM was chosen because Xen was not available due to security reasons, as mentioned earlier. For container approach LXC and Docker were considered but LXC was chosen due to the need for system containers. System containers are more ready-made for this study's purpose, such as including system architecture, network interfaces and being highly modifiable.

Lastly, resulting from these discussions, a commodity PC was ordered to be used as a development platform for the artifact. The PC, called Virtualization PC from now on, had 1000 Gigabytes of SSD (Solid-State Drive) memory, CPU with 32 Gigabytes of its own memory, consisting of 12 cores. Furthermore, a NIC with four interfaces and SR-IOV support was added to the PC.

4.2 Implementation

After the starting discussion, designs and decisions the implementation started with the Virtualization PC, which had a Linux OS Ubuntu 16.04. KVM was installed through the mirror repository. Using KVM, three host VMs were created, running Debian 9.6., i.e. Debian Stretch, because it was the only image available in the mirror repositories. Each VM was assigned 50 gigabytes of SSD, 1 gigabyte of CPU memory and 1 virtual CPU (vCPU). Then version 2.0.7 of LXC was installed on the host VMs using the mirror repository. This version was used because it was the latest version available for Debian 9.6.

Creating the containers was complicated because LXC required by default to have an internet connection to download modified images of OS's to install, i.e., a packaged rootfs and meta information, rootfs.tar.xz and meta.tar.xz respectively. There was no option to use local images. Container creation was done in the end by creating the directory where LXC caches downloaded images and then inserting Debian rootfs.tar.xz and meta.tar.xz packages there. This caused LXC to think it had already downloaded the image and then proceeded to build a container from that cached image. Installing packages to the containers was not straight forward either, because the containers didn't have access to the Internet. This was resolved by using "apt-get download" option in the host VM, then transferring the downloaded packages directly inside the containers

rootfs directories. Then the installation of packages was done locally in each container using bash scripts.

Using LXC, up to 150 containers were configured and running in one host VM. Creating, configuring and managing that many containers required creation of bash scripts. One of these scripts was done to start a ping from a node to the next node, going through all the nodes except the last one. Depending on the ping configuration, e.g. packet size or transmitting speed, there were cases when the host CPU was running at maximum 400% or under 100%. However, the container-nodes were connected to each other through one bridge in the host, meaning all the traffic went through that single bridge.

Later it was discovered that there was an Ubuntu 16.04, i.e. Ubuntu Xenial, image available. This enabled creating an Ubuntu VM from that image. It was learned from earlier VMs that more resources should be given to a VM, if there was need to run over 100 containers efficiently simultaneously in the VM, which was a likely case. For the first Ubuntu VM it was given 5 virtual CPUs, up to a maximum of 6 if required, 8 gigabytes of CPU memory and 100 gigabytes of SSD. During the implementation of the artifact a total of four VMs were created, all similar to the first one described above. The three VMs created after the first one had a bit less power in terms of performance, as they each had 2 vCPUs, 4 GB of RAM and 100 GB of SSD memory.

Table 3. Artifacts hardware and VM specifications.

Platform	**CPU cores / VCPU**	**RAM (GB)**	**SSD (GB)**
Virtualization PC	12 cores	32	1000
VM 1	5 (up to 6)	8	100
VM 2	2	4	100
VM 3	2	4	100
VM 4	2	4	100

Having Ubuntu VM enabled downloading a newer version of LXC, LXD 3.0.3, from the mirror repository. LXD had more functions and was simpler and quicker to use than LXC. It was thought that having more options would help in achieving the requirements R3 and R4. Furthermore, LXD supported creating own OS images from local rootfs.tar.gz and meta.tar.gz files.

4.2.1 Networking Between Nodes

Using LXD in the new Ubuntu VM, more containers were created and configured, and each container was connected to another one by a bridge. Meaning that each “link” between nodes was done by creating a bridge. A bridge works the same way as a Layer 2 switch, connecting two points together. A single bridge needed two virtual ethernet (veth) interfaces on the host and one interface on both of the two nodes that were to be connected. For example, creating a network of four nodes 1,2,3 and 4, where node 1 has a link to node 2 and node 3, and node 3 also has a link to node 4, would require three

bridges and six veths on the host. For each link, also an interface would be needed in each of the nodes. This example can be seen from a logical viewpoint in figure 15 and from networking viewpoint in figure 16.

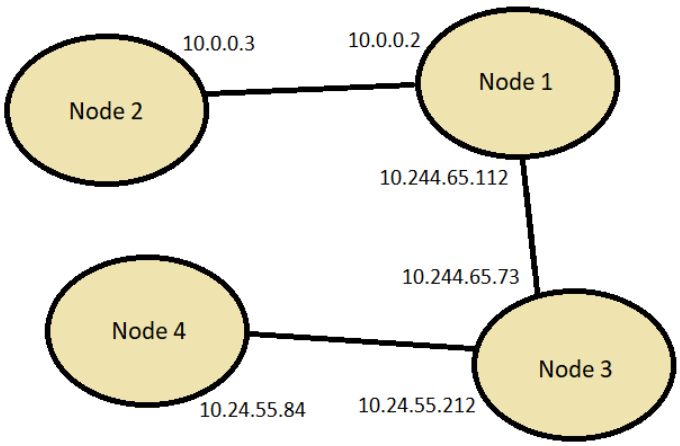

Figure 15. Network of four nodes, logical point of view.

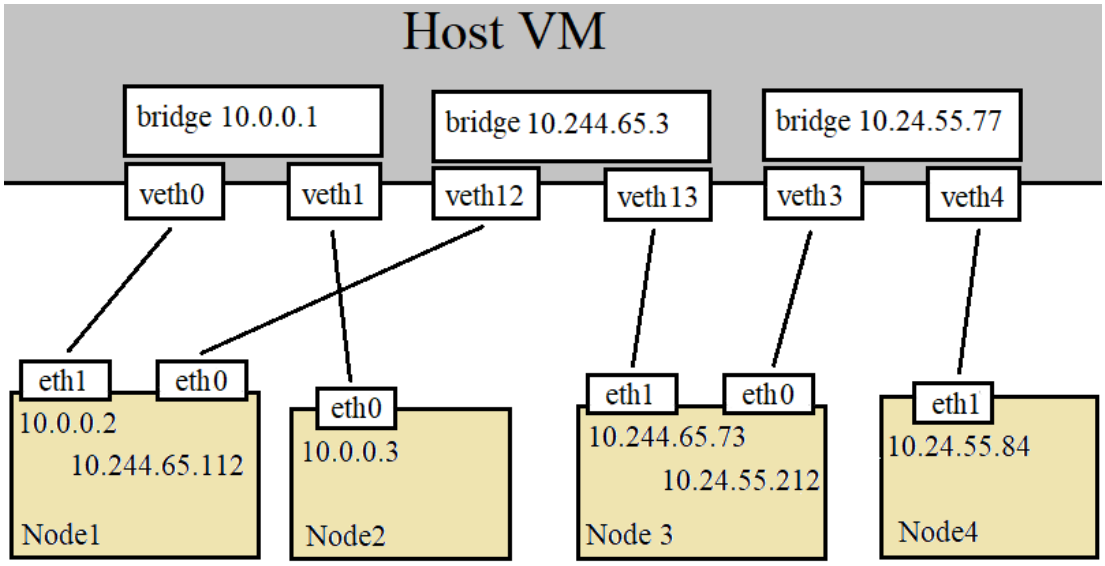

Figure 16. Network of four nodes, connected via bridges and veths.

Using these bridges as links, it was possible to create any kind of network topology. Furthermore, Quagga was installed in each container. Quagga is a network routing software that provides implementations for many routing protocols, such as OSPF and BGP. Without routing functionalities from Quagga, node 3, for example, in the image above can't see or ping node 2 because node 1 does not provide information regarding anything other than the link between it and node 3. Using Quagga, each container was able to perform network routing functions similar to any normal router, allowing node 3 to see and ping node 2 and even more complicated network topologies to work.

This was all in one VM, however, and a VM could only contain a certain number of nodes. This meant that, to enable the creation of very large networks, nodes needed to be able to see other nodes in other VMs as well.

4.2.2 Networking Between Virtual Machines

Connecting VMs together was done by creating a bridge with DHCP (Dynamic Host Configuration Protocol) enabled in the Virtualization PC, by using KVM. This bridge was then attached to each VM, by creating a new interface in them. Through this bridge, each VM could then see each other. Then a profile was created in LXD that would, when taken into use by a container, bridge the containers interface and the VM's interface together. The bridging in containers was done using macvlan. A macvlan driver is a separate Linux kernel driver that makes it possible to create virtual network interfaces, each virtual interface having its own MAC address. Adding this profile to containers across VMs would, together with OSPF, enable network routing between the VMs. This is illustrated in figure 17.

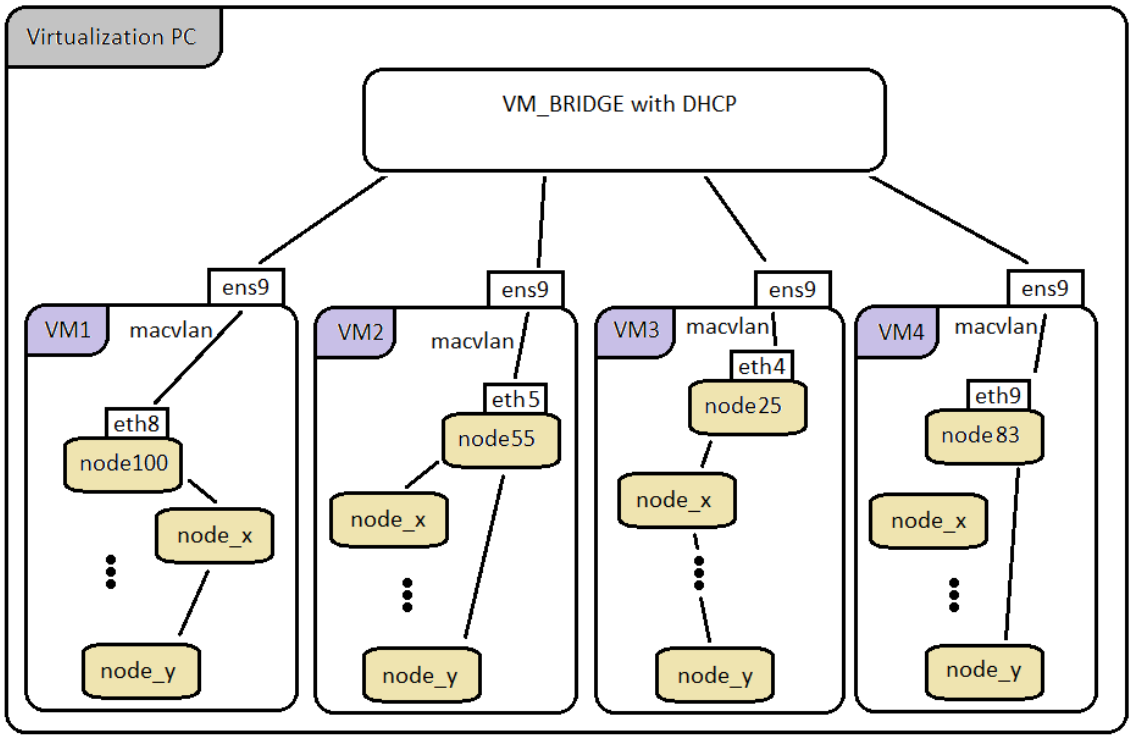

Figure 17. Connecting containers from different VMs together.

Because the bridge has DHCP service and the containers are bridged through the interface in the VM, each container attached to the VM bridge would get an appropriate IP address from the VM bridge to enable fast configuration.

4.2.3 Networking Between Containers and the Real World

Networking between virtual machines and the real world was also done with KVM, by using macvtap driver, as suggested by co-worker within the organization. A macvtap interface is a combination of macvlan and a tap interface. A tap interface is software-only interface, where instead of passing frames to and from a physical Ethernet card, the frames are read and written by a user space program.

Macvtap allowed to attach a VMs vNIC directly to a specified physical NIC on the Virtualization PC. A new interface was created in each VM to enable this, and this new interface was then attached to one interface on the physical NIC in the Virtualization PC. This meant that each VM had their own passthrough line to the physical NIC. Then

the same was done as when networking the VMs together, i.e. a profile was created in LXD that would, when added to a container, create a new interface in the container and bridge it to the interface in the VM. Lastly, each interface in the Virtualization PC's NIC had to be configured without static IP or DHCP and put to manual mode. This would prevent the VM from accessing the passthrough line, allowing only the container attached to the interface to access the outside world. Thus, a route for a node to the outside world was created, fulfilling the requirement R5. This is illustrated in figure 18.

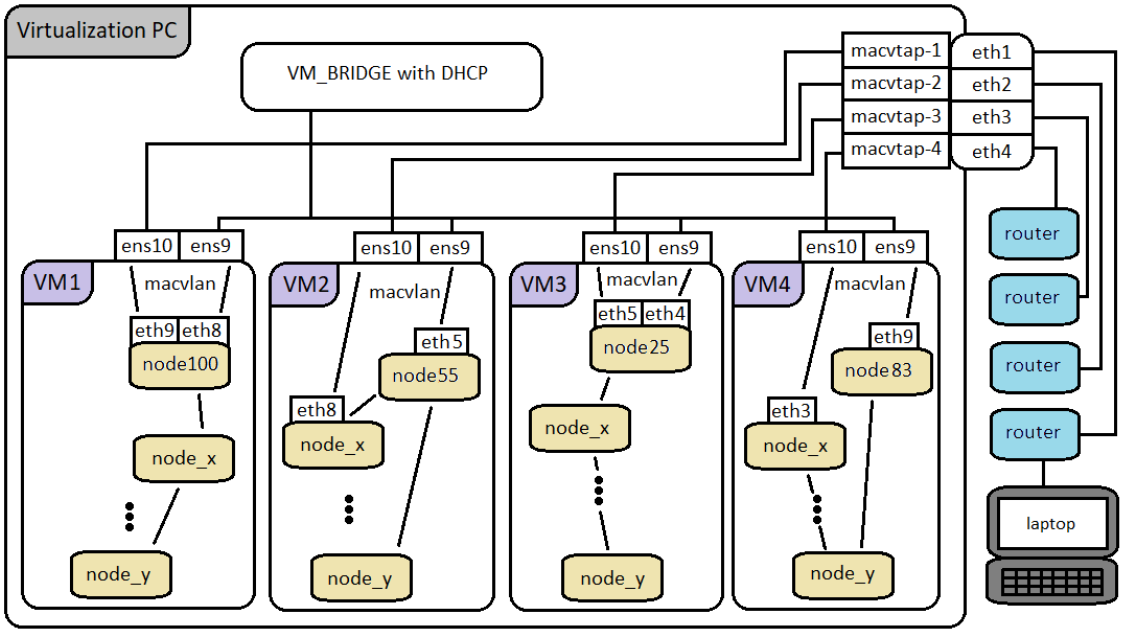

Figure 18. Networking to the real world.

To test the functionalities of OSPF routing, a laptop was attached to a physical router outside the Virtualization PC. This physical router also had OSPF routing enabled. The laptop did not, however, as it did not require it, but it had a static IP outside any IP range used by the containers or VM's. Each node was able to ping the laptop's IP, meaning that the routing was successful. This was further tested with ping's -R parameter, i.e. "Record route" -option, which displays the route buffer on returned packets, to see what path was taken. Traceroute tool was also used to determine that the correct route was taken. With most containers, the route was a long one, because the packets first had to go through long chain topologies to the correct container that had access to the physical NIC. Then the packets would finally travel to the physical router that would route them to the laptop and the laptops replies to the containers.

After this was done, it was time to analyze the artifact. The analyzation and it's results are discussed in the following chapter.

5. Evaluation

In this chapter the artifact is first analysed to provide insights into it and its performance. Then the artifact is evaluated against its requirements gathered from the organization and lastly the study is evaluated against the DSR methodology guidelines.

5.1 Analysis

Creating connections between containers creates many new interfaces in the host VM, as it requires two veths and one bridge per link. For example, creating a mesh network of ten nodes creates 9+8+7+6+5+4+3+2+1=45 links and each link requires three interfaces, making a total of 135 new interfaces in the host VM alone. In this scenario, each node would also have 9 interfaces to connect to of the other 9 nodes, bringing the total interface count to 135+9*10=225. Furthermore, there may be some additional interfaces needed, for example for communication between VM's or to the real world. This is big number of interfaces to manage for only one full mesh topology of 10 nodes. Creating larger networks would bring even further interfaces, creating a problem of controllability. Using LXD, however, the bridges could be named when creating a link, whereas the veths would be given a random name by default. By naming the bridges, it was easier to control the networks. For example, a bridge that connects nodes 45 and 12 would be named br12_45. This was done by creating bash scripts that would automate network creation based on parameters and name the links accordingly. Using the bridge names as parameters, further scripts could be made for controlling the networks. Many other scripts were created to help control the nodes. Furthermore, a python script was created by the study's supervisor from the organization. The python script would output all the links between nodes and the interface the nodes were using for each link.

When looking at the resource usage of the Virtualization PC, CPU power was not deemed to be a problem because with 12 cores each VM could be assigned 2 or 3 cores. Even if a VM had a hundred containers running and performing network routing, the VM's CPU usage would be around 10%. The Virtualization PC's CPU usage would be the same or less, depending whether all the cores were assigned to VM's. Only when doing something operation-heavy, such as creating tens of containers, would the CPU usage rise to the maximum.

What seemed to cause a problem was the amount of SSD storage each VM was given. As mentioned before, each VM was given 100 GB of SSD storage and having 150 containers took 81% of the whole storage. When creating 200 nodes in a VM, it took 97% of the VMs storage even when they had not been started or configured yet, so more storage needed to be added. After adding more storage and configuring the containers, it was deduced that each configured node required a bit less than 500 MB of SSD memory. Ubuntu 16.04 OS itself without graphical UI does not take much memory, a few GB's only. However, this wasn't really a problem, because the Virtualization PC had a total of 1000 GB's of SSD storage, meaning that each VM could easily be given 200 GB's of storage each.

What really became a problem was the amount of RAM memory. As mentioned, the Virtualization PC had a total of 32 GB of RAM and each VM had a minimum of four, later increased to six or seven, GB's of RAM memory. Even with seven GB's of RAM on a VM, the RAM usage in that VM was at 100% all the time when running 150 or 200 containers. The Virtualization PC's RAM usage was very high as well because each VM had a big slice of the Virtualization PC's RAM. For example, when each VM was given seven GB's of RAM and they were at nearly 100% usage, Virtualization PC's RAM usage was slightly over 90% as well. It was noticed that having around 100 containers running caused RAM memory usage to be around 90%, which was an acceptable level. However, this meant that only around 400 containers, 100 in each VM, could be running in the Virtualization PC. It was not possible to lower RAM usage by e.g. increasing the amount of CPU cores given to a VM.

For data gathering from containers, the study used LXD's own commands, on top of which the previously mentioned Python script was built. For gathering data from VMs, the study used Glances software in the Virtualization PC as well as the "top" command in the VMs. Glances and "top" both show and monitor processes and system resource usage in Linux.

5.2 Artifact Evaluation

Next is evaluated and discussed how the artifact fulfils the requirements given to it. The requirements and their fulfilment status are summarized in table 4. Each requirement is discussed separately.

Table 4. Artifacts requirements and their status.

Requirement	**Fulfilment status**
R1: Large-scale simulation	Questionable
R2: Encapsulation of nodes	Questionable
R3: Configurability	Fulfilled
R4: Adaptability	Fulfilled
R5: Outside connection	Fulfilled
R6: Replicability of nodes	Fulfilled

R1: Large-scale simulation – Questionable

This requirements fulfilment is questionable because it depends on exactly how many nodes are required. If a maximum of 400 nodes are required, this requirement is fulfilled as is but if more, e.g. thousands, of nodes are required, then the artifact does not fulfil this requirement. However, there is the option of running Scapy on some containers to simulate the network of thousands of nodes, but container nodes cannot be created more with the system hardware that are present in the artifact.

R2: Encapsulation of nodes – Questionable

This requirement is also questionable because from networking and operating system point of view the containers are encapsulated. However, containers don't have their own kernel or hardware so they can't be classified as entirely encapsulated. If the said kernel or hardware is changed or broken somehow, all the containers are affected. However, this is lessened by the choice of using hypervisor together with containers. Each container runs inside a VM and VMs fulfil encapsulation by default, meaning that the containers are entirely encapsulated from other VM's containers.

R3: Configurability – Fulfilled

Configurability requirement was fulfilled when choosing to use hypervisor and container approach, because together they bring many means to configure the artifact. Each VM can be configured differently and each system container runs a full Linux OS, meaning there is much that can be configured. Many things can be configured in the artifact itself as well, such as the number of VMs and the number of containers in each VM. Using Quagga means that different routing protocols can be used as well as different network topologies. Furthermore, new routing protocols can be installed as each node is a Linux OS environment. Lastly, even the container objects supported configuration through LXD's own commands. For example, through LXD's container configuration, a node could be given new interfaces, or set a static IP address in wanted interfaces.

R4: Adaptability – Fulfilled

This is similar to R3, because the containers use the VM's kernel and hardware enabling each VM's hardware and system settings to be adjusted differently to bring about fundamental differences between two VM's containers, such as different kernel version in each VM. New VM's can be added if new software needs to be tested and old ones can be deleted. Even other PCs running more VMs can be added and linked together, using the outside connection. There is room for growth.

R5: Outside connection – Fulfilled

This requirement was fulfilled with the installation of bridges and routing protocol. It required a macvlan bridge between a containers interface and a VM's interface, and a macvtap bridge between the VM's interface and the Virtualization PC's physical interface on its NIC. Lastly the requirement required a routing software to run in the container and a router outside. Without the routing protocol, however, it's still an IP connection to the physical world.

R6: Replicability of nodes – Fulfilled

Replicability of nodes was fulfilled on two levels: container level and VM level. LXD itself provides a copying mechanism for containers and KVM also provides a copying mechanism for VMs, which also copy the containers within. Furthermore, LXD had separate profiles which could be created, edited, destroyed or attached to containers. A profile could even be attached during containers creation. These profiles supported different degrees of configuration, but more importantly, they enabled easy replication of nodes. Instead of fully copying a node, it was possible to create a node that would use the same configuration settings, but still be a separate node with it's own ID's, IP's, etc.

5.3 DSR Study Evaluation

The Design Science Research methodology guidelines are for conducting and ensuring an effective design science research. They concern the whole study, not just the artifact that is built as a part of the study. The guidelines, their descriptions and fulfilment statuses are summarized in table 5. Each guideline and the fulfilment status are discussed separately.

1. Design as an Artifact – Accomplished

The goal of this guideline was to ensure the study produces a viable artifact and this was accomplished because the produced artifact is a functionable construct. This was verified by the artifacts evaluation.

2. Problem Relevance – Accomplished

The organization that the artifact was developed for had a genuine need of finding a way to test their routers in very large networks. This problem was the starting point of this study and the study tries to fulfil that need by having created the artifact.

Table 5. DSR guidelines, their description and fulfilment status.

Guideline	Description	Status
1. Design as an Artifact	Produce a viable artifact in the form of a construct, a model, a method, or an instantiation.	Accomplished
2. Problem Relevance	Develop technology-based solutions to important and relevant business problems.	Accomplished
3. Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods	Questionable
4. Research Contributions	Provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies	Accomplished
5. Research Rigor	Application of rigorous methods in both the construction and evaluation of the design artifact.	Not accomplished
6. Design as a Search Process	Utilizing available means to reach desired ends while satisfying laws in the problem environment.	Accomplished
7. Communication of Research	Present the research effectively both to technology-oriented as well as management-oriented audiences.	Accomplished

3. Design Evaluation – Questionable

The goal of this guideline was to ensure rigorous demonstration of the utility, quality and efficacy of the artifact by using well-executed evaluation methods. This was accomplished by using requirements for the artifact and then finding out if they have been met. However, not all parts, such as quality, were included in the requirements and it can be questioned whether the evaluation methods were well-executed. Thus, this guidelines status is questionable.

4. Research Contributions – Accomplished

The goal of this guideline was to ensure that the study provides clear and verifiable contributions in the areas of the design artifact. This guideline has been accomplished by providing information how the artifact was built, analysing what the artifact can do and what inhibitors there are for its usage. Furthermore, to enable verifiable contributions, instructions on the artifacts installation is provided in appendix A.

5. Research Rigor – Not accomplished

This is similar to the third guideline. The evaluation of the design artifact was accomplished questionably, however no clearly defined method for the construction of the artifact was used. Thus, this guidelines status is "Not accomplished".

6. Design as a Search Process – Accomplished

This guideline has a notion of reaching the desired end goal while satisfying laws in the problem environment. The problems concerning the artifacts building and usage environment was discussed in Chapter 3.1., where it was mentioned that e.g. there was no Internet connection in the artifacts development environment. This guideline was indeed accomplished, as the artifact was built and was deemed working without breaking laws in the environment.

7. Communication of Research – Accomplished

For management-oriented audiences the artifact and what it can and can't do are explained in Chapter 4. For technology-oriented audiences the appendix A and chapter 2, Prior Research, come into play, as they describe more technically what the artifact consists of and how it works. For these reasons, this guideline is accomplished.

6. Discussion and Implications

This chapter describes the findings and their implications as well as answers the research questions stated in Chapter 1.

To start with, it is feasible to use commodity hardware for large-scale virtualization environment and it's not necessary to use special, often expensive, hardware, such as servers or hardware specifically designed for virtualization. What seems to be the limiting factor when using commodity hardware for virtualization, is the amount of RAM there is. CPU usage as well as storage usage was not that limiting. CPU power and RAM are somewhat harder to increase, whereas storage is easy to get more of. With 32 GB of RAM, 12-core CPU and 1 TB of SSD storage, it's possible to create at least 400 containers before RAM starts to fill up. Furthermore, because Linux Containers are system containers, they isolate software from its environment and ensure that it works uniformly ("What is a Container", n.d.). This means that much more can be done with them than only network routing, such as hosting and running different services.

As mentioned earlier, there isn't much research concerning building large-scale virtual networks. This study's research questions aim to shed some light into that subject, so the main implications of this study are the answers to the research questions.

RQ1: What feasible virtualization solutions are available for creating a large-scale virtual environment for network routing?

Hypervisors are a good solution, because they have a direct access to the hardware resources, meaning they are very efficient and enable great scalability, robustness and versatility ("Virtualization Overview", n.d.). However, hypervisors produce overhead due to virtualization (Soltesz et al., 2007). The more VM's there are, the more overhead is produced, even if it is a small amount per VM. Creating hundreds of VM's is not a feasible solution due to the overhead and the amount of virtualized HW. This is why containers are useful, as they offer an environment close to one gotten from a VM but without the overhead that comes from running a separate kernel and simulating all the hardware ("Linux Containers", n.d.).

Using both of these approaches in combination is an option as well, as was done in this study. This enables even greater configurability and adaptability of the environment. The hosted approach is not feasible because it is too heavy as the virtualization layer does not have direct access to the hardware.

RQ2: How to build a flexible virtual environment that can simulate hundreds of nodes in a network, using only commodity hardware?

The detailed instructions for building an environment similar to what was done in this study are in appendix A, but the main steps are:

1. Install and setup KVM
2. Create and setup VMs and nodes
3. Create networks using bridges
4. Create a route outside using bridges and macvtap

KVM installation requires only a few commands to install a few packages. After KVM installation, a virtual router needs to be created to enable communication between Virtualization PC and the VMs. Then VM's can be created, connected to the virtual router, and configured. VM configuration includes installing LXD version 3.0.3 and enabling proper Quagga functionality for when it is used later. After that, bash scripts can be used to create and configure nodes. Node configuration includes installing software packages from which Quagga is the most important.

After nodes are created and configured, network creation using bridges can be performed. Different topologies can be made using scripts. After topologies are created, routing must be enabled on the nodes by configuring Quagga in each node, which consists of configuring zebra daemon and OSPF daemon. Communication between nodes in different VMs can also be done at this stage. All can be done using scripts, for ease of use.

The last stage is creating a route to the real world. This can be done using only IP protocol, but for routing purposes this requires having an OSPF capable router in the real world. The process starts with creating macvtap interfaces that bridge VMs' interfaces to the physical NIC. Then nodes' interfaces are bridged to the VMs' interfaces that are already bridged to the physical NIC. Lastly OSPF is enabled on the nodes' interface.

7. Conclusions

This was a study that used design science research methodology to research computer networks, routing protocols, virtualization and network virtualization in an attempt to understand how they work. That information was used in creating an artifact capable of running a virtualized network of many hundred nodes. The artifact was developed using a hybrid approach consisting of a hypervisor and system containers. Links between nodes and also into the real world were achieved by using bridges. The artifact was conceived for a company and had requirements to fulfil in order be useful. Most of the requirements were fulfilled while some were questionable depending on the definition of the requirement. DSR guidelines were followed and most of them were accomplished as well.

The study contributes to the research community with information concerning the subjects of networking, virtualization and network virtualization, as well as with information regarding how to build an artifact capable of creating hundreds of network nodes using only commodity hardware. Furthermore, this information is useful for the organization this study was done for. This study has shown that each node in a network requires a certain amount of memory storage as well as CPU memory, also often called RAM. Approximately 500 MB of storage is required per node, and with 32 GB of RAM memory, 400 nodes were created and running while having around 90% CPU's RAM usage. CPU processing power was not deemed to be very crucial.

There are some limitations to the study. Major limitations include not meeting all the artifacts requirements or all the DSR guidelines. Further limitations include having old performance tests between hypervisors and containers. The technologies concerning hardware and virtualization have evolved since those tests and it might be beneficial to see the performance differences with current technologies. Furthermore, this study used KVM and Linux Containers which were not included in the tests, as the tests used Xen for hypervisor approach and VServer for container approach. Lastly, due to time limitations, there were not many tests performed with the artifact.

For future research, as somewhat mentioned, it would be interesting to learn the performance differences between hypervisor and container approach with current technologies concerning both hardware and virtualization. Another topic for future research would be to study how to optimize the performance when using hypervisor approach together with container approach. For example, is it performance-wise better to have few well-equipped VM's with plenty of containers or is there some kind of balance to be found? Lastly, future work might include integrating other routing protocols that are not supported by Quagga into the artifact, such as OLSR.

References

Bănică, L., Rosca, D., & Stefan, C. (2007). Virtualization: an old concept in a new approach. University Library of Munich, Germany.

Bhatia, S., Motiwala, M., Muhlbauer, W., Mundada, Y., Valancius, V., Bavier, A., ... & Rexford, J. (2008, December). Trellis: A Platform for Building Flexible, Fast Virtual Networks on Commodity Hardware. In *Proceedings of the 2008 ACM CoNEXT Conference* (p. 72). ACM.

Blank, A. G. (2006). *TCP/IP Jumpstart: Internet protocol basics*. John Wiley & Sons publishing.

Computer Network | Types of area networks – LAN, MAN and WAN. (n.d.) Retrieved February 25, 2019, from https://www.geeksforgeeks.org/computer-network-types-area-networks-lan-man-wan/

Containers at Google – The Google Way. (n.d.) Retrieved February 27, 2019, from https://cloud.google.com/containers/

Dong, Y., Yang, X., Li, J., Liao, G., Tian, K., & Guan, H. (2012). High performance network virtualization with SR-IOV. *Journal of Parallel and Distributed Computing, 72*(11), 1471-1480.

Data Communication and Computer Network. (n.d.) Retrieved January 29, 2019, from https://www.tutorialspoint.com/data_communication_computer_network/data_communication_computer_network_tutorial.pdf

Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Huici, F., & Mathy, L. (2008, December). Towards High Performance Virtual Routers on Commodity Hardware. In *Proceedings of the 2008 ACM CoNEXT Conference* (p. 20). ACM.

Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Huici, F., Mathy, L., & Papadimitriou, P. (2010). A Platform for High Performance and Flexible Virtual Routers on Commodity Hardware. *ACM SIGCOMM Computer Communication Review, 40*(1), 127-128.

Fortz, B., Rexford, J., & Thorup, M. (2002). Traffic engineering with traditional IP routing protocols. *IEEE communications Magazine, 40*(10), 118-124.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS quarterly, 28*(1), 75-105.

Introduction – About Scapy. (n.d.) Retrieved February 18, 2019, from https://scapy.readthedocs.io/en/stable/

Introduction to Computer Networks. (n.d.) Retrieved January 29, 2019, from https://www.studytonight.com/computer-networks/overview-of-computer-networks

Jacquet, P., Muhlethaler, P., Clausen, T., Laouiti, A., Qayyum, A., & Viennot, L. (2001). Optimized link state routing protocol for ad hoc networks. In Multi Topic Conference, 2001. *IEEE INMIC 2001. Technology for the 21st Century. Proceedings. IEEE International* (pp. 62-68). IEEE.

Linux Containers. (n.d.) Retrieved January 28, 2019, from https://linuxcontainers.org/

Main Page (2016, November 7*). KVM*. Retrieved February 1, 2019 from https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792

Maltz, D. A., Xie, G., Zhan, J., Zhang, H., Hjálmtýsson, G., & Greenberg, A. (2004). Routing design in operational networks: A look from the inside. *ACM SIGCOMM Computer Communication Review, 34*(4), 27-40.

Rahore, M. S., Hidell, M., & Sjodin, P. (2012, March). PC-based Router Virtualization with Hardware Support. In *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on* (pp. 573-580). IEEE.

Rouse, M. (2018). Network Engineer. Retrieved February 25, 2019, from https://searchnetworking.techtarget.com/definition/network-engineer

Scapy - About Scapy. (n.d.) Retrieved February 18, 2019, from https://scapy.net/

Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., & Peterson, L. (2007, March). Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review* (Vol. 41, No. 3, pp. 275-287). ACM.

Understanding BGP. (n.d.) Retrieved January 29, 2019, from https://www.juniper.net/documentation/en_US/junos/topics/concept/bgp-routing-overview.html

Virtual Networking. (n.d.) Retrieved January 15, 2019, from http://www.vmware.com/files/pdf/virtual_networking_concepts.pdf

Virtualization Overview. (n.d.) Retrieved January 15, 2019, from http://www.vmware.com/pdf/virtualization.pdf

Wang, A., Iyer, M., Dutta, R., Rouskas, G. N., & Baldine, I. (2013). Network Virtualization: Technologies, Perspectives, and Frontiers. *Journal of Lightwave Technology, 31*(4), 523-537.

What is a Container. (n.d.) Retrieved January 15, 2019, from https://www.docker.com/resources/what-container

Why Xen Project? (2013) *Xen Project*. Retrieved February 1, 2019, from https://www.xenproject.org/users/why-the-xen-project.html

Appendix A. Artifact Instructions

For the purpose of this document, containers will be called nodes to enforce the idea of using containers as nodes in a network.

Virtual Machines Creation and Setup

This part explains virtual machine creation and how to set them ready for node creation. It's expected that the Virtualization PC is already configured with basic functions, such as working apt-get.

Install KVM and Create VMs

Run the following command in the Virtualization PC to install KVM:

```
sudo apt-get install qemu-kvm libvirt-bin virt-manager bridge-utils
ovmf libguestfs-tools sshpass
```

Make sure that the current user is part of libvirt group, and reboot afterwards:

```
sudo adduser 'id –un' libvirtd
reboot
```

Undefine the default pool used by KVM and define different pools for networks, images and VM's (called guests):

```
virsh pool-destroy default
virsh pool-undefine default
mkdir <name for network folder>
mkdir <name for images folder>
mkdir <name for VM's folder>
virsh pool-define-as –name networks --type dir –target <path to
network folder>
virsh pool-define-as –name images –type dir –target <path to images
folder>
virsh pool-define-as –name guests –type dir –target <path to guests
folder>
```

Then set autostart –flag up and start the pools. Do this for all three pools created in previous step. Move installation media to the images-pool's folder before mounting them for VMs.

```
virsh pool-autostart <pool name>
virsh pool-start <pool name>
mv <installation media> <path to images folder>
```

Create control-bridge that the VM's can connect to for communication:

```
nano <xml file>.xml

<network>
        <name>network_name</name>
        <bridge name="interface_name" />
```

```
        <ip address='192.168.1.1' netmask='255.255.255.0'>
                <dhcp>
                        <range start='192.168.1.2'
end='192.168.1.254'/>
                </dhcp>
        </ip>
</network>

virsh net-define <xml file>.xml
virsh net-autostart <network_name in the xml file>
virsh net-start <network_name in the xml file>
```

Also, disable the default network:

```
virsh net-stop default
virsh net-autostart default --disable
```

You can use virt-manager to create VM's using GUI, or use virt:

```
virt-install \
--connect qemu:///system \
--virt-type kvm \
--accelerate \
--boot uefi \
--ram <RAM in MB's, e.g. 1024> \
--vcpus <vCPU's, e.g. 2> \
--disk pool=guests,size=<storage size in
GB's>,bus=virtio,sparse=false,format=raw,cache=none \
--network <network_name> \
--cdrom <path to installation media, e.g. images folder> \
--name <name> \
--os-type linux \
```

Ensure that the created VM's connect to the control-bridge using the --network parameter. Ensure good processing powers and memory. Storage, in gigabytes, should be at least half of the number of nodes planned to create, e.g. 100 planned nodes means 50GB of storage.

Configure VMs

NOTE: From now on, all commands are supposed to be used in the chosen VM or all VMs.

Set root password and allow root SSH login, for ease of access and usability:

```
sudo passwd root
nano /etc/default/sshd
        permitRootLogin = yes
```

Enable proper quagga functionality when it is used later:

```
sysctl fs.inotify.max_user_instances=2048
```

Install LXD version 3.0.3:

```
apt-get install liblxc1=3.0.3-0ubuntu1~16.04.1
apt-get install lxcfs=3.0.3-0ubuntu1~16.04.1
apt-get install lxd-client=3.0.3-0ubuntu1~16.04.1
apt-get install lxd=3.0.3-0ubuntu1~16.04.1
```

In the organization, there is a folder called niko which has all the scripts created for the artifact and all software packages and images needed for the artifact. The artifact can be done without them, it's only takes more time. This appendix is mainly meant for the organization. Copy the niko folder to each VM:

```
scp -r niko/ <user>@<VM's IP>:~/
```

Configure LXD

Set LXD image updating off and add the user to LXD group:

```
lxc config set images.remote_cache_expiry 100
lxc config set images.auto_update_interval 100
lxc config set images.auto_update_cached false
usermod --append --groups lxd testuser
```

If there is no Internet connection available, create new OS image for LXD:

Note! This requires you to have the OS images meta.tar.xz (or meta.yaml) and rootfs.tar.xz. Meta.tar.xz can be created from meta.yaml with `tar -cvzf meta.tar.xz metadata.yaml`. Meta.yaml can be edited normally if needed. In this case the required files are in niko/ubuntu_image/ -folder.

```
lxc image import ~/niko/ubuntu_image/meta.tar.xz \
~/niko/ubuntu_image/rootfs.tar.xz –alias ubuntu_nigel
```

Edit the default profile nodes use when first created and remove the eth0 interface. New interfaces will be created for each link separately when creating links:

```
lxc profile edit default
<remove eth0 interface from the profile>
```

Edit lxd-bridge to disable the default bridge LXD uses:

```
nano /etc/default/lxd-bridge (lxd-bridge.upgraded if LXD 3.0.3)
USE LXD BRIDGE = "false"
LXC BRIDGE = ""
```

Nodes Creation and Setup

All the scripts used throughout this document are in niko/ folder. This part shows how to create and setup nodes.

Create Nodes

Nodes are created from a custom OS image which was imported to LXD earlier. The command requires only the custom OS image's name and the node's name. Nodes can be initiated or launched. The difference between these two is that the launch option also starts the node right away.

```
lxc init <OS image> <node name>
lxc launch <OS image> node name>
```

For example, initiating node1 and launching node2:

```
lxc init ubuntu_nigel node1
lxc launch ubuntu_nigel node2
```

There is a bash script create_containers.sh that can initiate a wanted number of nodes:

```
./create_containers.sh <starting node ID> <ending node ID>
```

For example, creating 100 nodes with IDs from 1 to 100, e.g. node1, node2 …, node100:

```
./create_containers.sh 1 100
```

There is also a bash script create_and_setup_containers.sh that launches and installs some needed software packages to them:

Note: This script starts the nodes as well, because nodes need to be running in order to install packages in them. Furthermore, this doesn't install Quagga on the nodes.

```
./create_and_setup_containers.sh <starting node ID> <ending node ID>
```

Install Software

To easily setup nodes it's suggested to use the create_and_setup_containers.sh script. However, software packages, or any other file, can be transferred to nodes in three ways; scp over ssh, cp locally or LXD file pushing:

```
scp <package> root@<node IP>:<directory where to copy>
cp <package> /var/lib/lxd/containers/<node>/rootfs/<directory where to
copy>
lxc file push <package> <node>/<directory where to copy>
```

For example, copying tcpdump-package to node50's (IP e.g. 10.0.0.5) /tmp/ -folder using all variations:

```
scp tcpdump-package root@10.0.0.5:/tmp/
cp tcpdump-package /var/lib/lxd/containers/node50/rootfs/tmp/
lxc file push tcpdump-package node50/tmp/
```

After the software package is in the node, it can be installed using dpkg in two ways; directly giving the executing command to the node, or opening bash inside the node and installing it locally:

```
lxc exec <node> -- dpkg -i <package>

lxc exec <node> bash
dpkg -i <package>
```

For example, installing the previously transferred tcpdump-package in the same node50's /tmp -folder:

```
lxc exec node50 -- dpkg -i /tmp/tcpdump-package

lxc exec node50 bash
dpkg -i /tmp/tcpdump-package
```

Doing this manually to many different software packages and all nodes is time consuming, thus there are two scripts for installing software packages to nodes. The first bash script, called install_all_tools.sh, installs all required tools to the nodes specified. The installed software packages are in niko/ubuntu_packages/ -folder. The second bash script, install_tool.sh, installs a specified tool to all the nodes specified:

```
./install_all_tools.sh <starting node ID> <ending node ID>
./install_tool.sh <starting node ID> <ending node ID> <software
package>
```

For example, installing all software packages and additionally installing tcpdump to nodes from 1 to 50:

```
./install_all_tool.sh 1 50
./install_tool.sh 1 50 ubuntu_packages/tcpdump-package
```

Setup Quagga

Quagga is installed separately from the other tools, because it requires configuration. To install quagga package, seven dependency packages need to be installed first in the following order: (1) libc-bin, (2) libblas-common, (3) libblas3, (4) liblinear3, (5) liblua, (6) libpcap, and (7) lua-lpeg. After they are installed, Quagga can be installed using any of the above-mentioned methods. Quagga consists of Zebra daemon that can run routing protocols, and routing protocol daemons that produce the routing protocol.

In this case, zebra and OSPF daemon must be configured. First, bash must be started in the node that has Quagga installed. Then configuration file samples must be moved to Quagga folder. The text ".sample" must be removed from the configuration files names. Then zebra and OSPF daemon for Quagga must be enabled by modifying "daemons" file. A login password must be written for zebra and OSPF daemons. Furthermore, enable-password can be disabled for them. Lastly Quagga must be restarted:

```
lxc exec <node> bash
cd /etc/quagga/
cp /usr/share/doc/quagga/examples/*.conf.sample .
mv ospfd.conf.sample ospfd.conf
mv zebra.conf.sample zebra.conf
nano daemons
        zebra=yes
        olsrd=yes
nano ospfd.conf
        password zebra
        !enable password
nano zebra.conf
        password zebra
        !enable password
/etc/init.d/quagga restart
```

This can be bothersome to do for all nodes separately, thus a bash script setup_quagga.sh installs and configures Quagga, i.e. zebra, and optional protocol daemon, for all wanted nodes:

```
./setup_quagga.sh <starting node ID> <ending node ID> <optional
protocol to be configured>
```

In this case, zebra and OSPF daemon must be configured, for example, to nodes 1 to 100:

```
./setup_quagga.sh 1 100 ospf
```

Network Creation

This chapter explains how to create networks between nodes within a single VM and then how to connect nodes between different VMs. Furthermore, it explains how to enable OSPF routing in nodes.

Create Networks in Same VM

This requires the creation of bridges, then attaching that bridge to two nodes. First create that bridge, then attach it to both nodes. Both nodes then need to add that new interface in their /etc/network/interfaces file:

```
lxc network create br<from_node>_<to_node>
lxc network attach <bridge> <first_node>
lxc network attach <bridge> <second_node>
lxc config show <first_node> | grep <bridge>.name | cut -d ' ' -f4
printf "\nauto <interface>\niface <interface> inet dhcp"
/var/lib/lxd/containers/<first_node>/rootfs/etc/network/interfaces
lxc config show <second_node> | grep <bridge>.name | cut -d ' ' -f4
printf "\nauto <interface>\niface <interface> inet dhcp"
/var/lib/lxd/containers/<second_node>/rootfs/etc/network/interfaces
lxc restart <first_node>
lxc restart <second_node>
```

For example, creating a bridge between nodes 1 and 2:

```
lxc network create br1_2
lxc network attach br1_2 node1
lxc network attach br1_2 node2
iface=$(lxc config show node1 | grep br1_2.name | cut -d ' ' -f4)
printf "\nauto ${iface}\niface ${iface} inet dhcp"
/var/lib/lxd/containers/node1/etc/network/interfaces
iface=$(lxc config show node2 | grep br1_2.name | cut -d ' ' -f4)
printf "\nauto ${iface}\niface ${iface} inet dhcp"
/var/lib/lxd/containers/node2/etc/network/interfaces
lxc restart node1
lxc restart node2
```

This can be done using bash script create_network_link.sh:

```
./create_network_link.sh <from node> <to node>
```

For example, creating a link between nodes 1 and 2:

```
./create_network_link.sh 1 2
```

Furthermore, chain and mesh topologies between multiple nodes can be created using bash scripts create_network_chain.sh and create_network_mesh.sh:

```
./create_network_chain.sh <starting node ID> <ending node ID>
./create_network_mesh.sh <starting node ID> <ending node ID>
```

For example, creating a chain topology from node 1 to 20 and a mesh topology from node 20 to 25:

```
./create_network_chain.sh 1 20
./create_network_mesh.sh 20 25
```

Enable OSPF in Nodes

This requires that Quagga has been installed and setup already in nodes. By default, the links between nodes are in different network spaces and there is no routing yet enabled. There are two ways to enable OSPF in interfaces; dynamic run-time configuration, or static configuration. To enable OSPF routing, each interface on a node that has a connection to other node must be added configured to OSPF daemon. To do this dynamically, first it must be known what the port is that OSPF daemon uses. Then telnet connection is to be formed on the port and OSPF daemon is to be configured to run on the wanted interface. This is done by using the following commands:

```
nmap localhost
telnet localhost 2604
(password = zebra)
enable
conf t
router ospf
network <first three octets of interfaces IP>.0/<network prefix> area
<area number>
<same for second interface that has a link to another node>
<same for third interface, etc.>
exit
```

For example, there is node77 that has 3 links to other nodes using the following interfaces and IPs: eth0 with IP of 10.211.23.44, eth1 with IP of 10.212.40.19, and eth2 with IP of 10.4.11.5

Enabling OSPF on that node would require:

```
telnet localhost 2604
(password = zebra)
enable
conf t
router ospf
network 10.211.23.0/24
network 10.212.40.0/24
network 10.4.11.0/24
exit
```

This must be done to all nodes and it does not last over rebooting a node. Thus, there is the second, static, way of configuring it. This requires to first write the interfaces that are to be used for OSPF to zebra.conf and ospfd.conf located at nodes /etc/quagga/ -folder. Then Quagga must be restarted. Let's use an example node 2 that has two interfaces, eth0 and eth1, connected to two other nodes with IP's of 10.232.27.188/24 and 10.167.30.149/24, respectively. They will all be in OSPF area 0. Then zebra.conf must consist of:

```
hostname router
password zebra
!enable password zebra
interface eth0
        ip address 10.232.27.188/24
        ipv6 nd suppress-ra
interface eth1
        ip address 10.167.30.149/24
        ipv6 nd suppress-ra
interface lo
ip forwarding
line vty
```

And ospfd.conf must consist of:

```
hostname ospfd
password zebra
!enable password zebra
log stdout
interface eth0
interface eth1
interface lo
router ospf
        network 10.232.27.0/24 area 0.0.0.0
        network 10.167.30.0/24 area 0.0.0.0
line vty
```

To save time, use the bash script conf_quagga.sh. For example, when enabling OSPF on all the nodes between, and including, nodes 1 and 50:

```
./conf_quagga.sh 1 50
```

Create Networks Between Nodes in Different VMs

For a communication medium between VMs, it's possible to use the virtual router created to enable communication between Virtualization PC, or a new one can be created for this specific purpose. If a new virtual router is created, the following steps are needed: (1) create a new network router with virsh, as was done in the beginning, (2) attach it to VMs, (3) shutdown VMs, and (4) start VMs.

Attaching a virtual router or network to a VM creates a new interface in that VM. In the following example, a bridge has been created in the Virtualization PC, using virsh, and it has been attached to all VM's, creating a new interface in them, called ens9. This information is needed in the next step.

In any case, a new profile called betweenroute is made from the default LXD profile. This can be done using:

```
lxd profile cp default betweenroute
```

Ensure it has the following contents by using `lxd profile edit betweenroute`

```
config:
  environment.http_proxy: ""
  user.network_mode: ""
description: Default LXD profile
devices:
  eth0:
    nictype: macvlan
    parent: ens9
    type: nic
  root:
    path: /
    pool: default
    type: disk
name: betweenroute
used_by:
```

The `parent` parameter tells LXD which VM interface this new macvlan-type bridge will be attached to. The other end of that bridge is then attached to the node the profile is attached to.

Then add the profile to the nodes you want, using `lxd profile attach <node> betweenroute`. Attaching the profile to a node creates a new interface in the node.

Modify the new interface in the nodes /etc/network/interfaces – file to get dhcp address from the router or add a static IP. Lastly, enable OSPF on that interface. The steps after creating the betweenroute-profile can be done using a bash script:

```
./conf_ospf_connection.sh <node> <ospf_area> between
```

Route Creation to Outside

NOTE: This is suggested to do last, i.e. when there is network topology between nodes and VMs. This requires to manually set IP address on the interface that a node will use for outside connection. If more links are created on a node that has a route outside, but the interface doesn't have IP address, the newly created links won't get an IP address. This is because the outroute link doesn't have an address, making networking stuck to that point. One workaround is to remove the outroute profile from the node and restart the nodes, then all links created afterwards should work. Just remember to add the outroute profile when the topology creation is done.

This chapter requires a router in the outside world if routing is to be tested. It needs to have an IP address and OSPF protocol configured and working. This will be explained using an example, where the real-world router has OSPF running, an IP address of 10.0.10.10/24, and the Virtualization PC's eth1 interface will be used.

Creating Passthrough Bridge in Virtualization PC

On the physical Virtualization PC, edit the interface you want to be used as a passthrough bridge, using `nano /etc/network/interfaces`. In this example, interface eth1 will be used as a passthrough bridge:

```
allow-hotplug eth1
auto eth1
iface eth1 inet manual
```

Create a passthrough bridge in a Virtualization PC, starting with a new xml file. Then use virsh to define and start that bridge. Start with creating the xml using using `nano macvtap1-net.xml`

```
<network>
        <name>macvtap-net1</name>
        <forward mode="bridge">
                <interface dev="eth1"/>
        </forward>
</network>

virsh net-define macvtap1_router.xml
virsh net-autostart macvtap1-net
virsh net-start macvtap1-net
```

The VM must be made to use that newly created bridge. This can be achieved using either GUI or terminal. If GUI is used, shut down the VM's and use virt-manager to add the network device to the VM using the following settings:

```
Network source: host device eth1: macvtap
source mode: Passthrough
device model: virtio
```

If terminal is used, use the command `virsh edit <VM name>` and add the following lines below the currently used interface:

```
<interface type='direct' trustGuestRxFilters='yes'>
      <mac address='52:54:00:b5:25:6c'/>
      <source dev='eth1' mode='passthrough'/>
      <model type='virtio'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x09'
function='0x0'/>
</interface>
```

It is suggested to use terminal, to ensure that multicast packets are transferred using the `trustGuestRxFilters` -parameter.

Shutdown the VM and boot up, it should have a new interface (e.g. ens10) which is now a straight passthrough to eth1 on the Virtualization PC (set with virsh edit or virt-manager). However, still on the VM you must edit /etc/network/interfaces to set that new interface to manual mode. The new interface must also be brought down and up to enable the new configuration. This can be achieved with:

```
nano /etc/network/interfaces
        auto ens10
        iface ens10 inet manual
ifdown ens10
ifup ens10
```

Connecting Nodes to the Passthrough Bridge

The last step is to also connect nodes to the passthrough bridge. Start with copying the default LXD profile and then editing the new profile. This example creates a profile called `outroute` (used by scripts).

```
lxc profile cp default outroute
lxc profile edit outroute

config
  environment.http_proxy: ""
  user.network_mode: ""
description: Default LXD profile
devices:
  eth1:
    nictype: macvlan
    parent: ens10
    type: nic
  root:
    path: /
    pool: default
    type: disk
name: outroute
used_by:

lxc profile add <node ID> outroute
```

Give the newly created interface on a node a static IP (in this case eth2 was created, even though the profile says eth1) using `nano /etc/network/interfaces`

```
auto eth2
        iface eth2 inet static
        address 10.0.10.1/24
```

Reboot the node and remember to enable OSPF on that new IP address, e.g. 10.0.10.0/24 area 0 in this case.

Enabling OSPF can be done using a script, however the interfaces IP address must still be set manually. Enabling OSPF can be done by using the following script:

```
./conf_ospf_connection.sh <node ID> <ospf_area> out
```

Useful Commands and Scripts

This chapter has useful scripts that might be needed when working with the artifact.

Show Networks and Links

There is a python script that parses all the links into a readable form. This has been further refined into a bash script links.sh:

```
./links.sh <optional, nodeID> <optional, to nodeID>
```

For example:

`./links.sh` → shows all the links in the network and their information
`./links.sh 5` → shows all the links and their information that node5 has
`./links.sh 5 6` → shows the link and its information that is between node5 and node6

LXD's build-in "list" command also be used. For example:

`lxc ls` → shows lxc information from all nodes
`lxc ls node10` → shows lxc information from all nodes whose name starts with "node10"
`lxc ls node10 .` → shows lxc information from node10
`lxc ls node10*` → shows lxc information from nodes 1, 10, 100, 1000, etc.
`lxc ls node1*` → shows lxc information from nodes 1, 11, 111, etc.

Deleting Bridges

Deleting a link between two nodes requires to first bring stop the nodes, then detaching the nodes from the bridge lastly deleting the bridge. All are done through LXD:

```
lxc stop node1
lxc stop node2
lxc network detach br1_2 node1
lxc network detach br1_2 node2
lxc network delete br1_2
```

Or use the script `delete_link.sh <first node> <second node>`. There is also a script called `delete_all_links.sh` that takes starting node ID and ending node ID as a range parameter and deletes all links between, and including, the nodes in that range. For example, deleting all the links starting from node 10 and ending in node 20:

```
./delete_all_links.sh 10 20
```

Note: This doesn't delete the interfaces settings from /etc/network/interfaces. This can be done by pushing a default interfaces –file to each node, using script `push_file.sh`. First the default interfaces-file must be created and then the script can be used for pushing any file to any directory in a range of nodes. For example, if an interfaces -file was created and wanted to be pushed to nodes 10 to 20 after deleting links between them:

```
./push_file.sh 10 20 interfaces /etc/network/
```

Configuring Nodes

Quick info of a node:

```
lxc info <node>
```

Profile handling:

```
lxc profile list
lxc profile show <profile>
lxc profile edit <profile>
lxc profile apply <node> <profile>,<profile2>,...
```

Node configuration:

```
lxc config show <node>
lxc config show –expanded <node>
lxc config edit <node>
lxc config set <node> <key> <value>
```

Node are located at /var/lib/lxd/containers/

Controlling Nodes

For starting, stopping, deleting or restarting created nodes, there scripts for each of them:

```
./start_containers.sh <starting node ID> <ending node ID>
./stop_containers.sh <starting node ID> <ending node ID>
./delete_containers.sh <starting node ID> <ending node ID>
./restart_containers.sh <starting node ID> <ending node ID>
```

Commands inside node:

```
lxc exec <node> bash
lxc exec <node> <command>, may require the use of separators for parsing
lxc exec <node> -- <complicated command>
```

File control:

```
lxc file pull <node>/<path> <destination/path>
lxc file pull <node>/path> -
lxc file push <source> <node>/<path>
lxc file edit <node>/<path/and/file>
```

Static IPs on Nodes

Create a file /etc/default/dns.conf that will have each nodes' static IP, using `nano /etc/default/dns.conf`

```
dhcp-host=<node>,<IP>
dhcp-host=<another node>,<another IP>
```

Add that file into /etc/default/lxd-bridge to section LXD_CONFILE and restart lxd-bridge service. Use `nano /etc/default/lxd-bridge` to add the location and then restart the service:

```
LXD_CONFILE=”/etc/default/dns.conf”
```

```
service lxd-bridge restart
```

Restart nodes.

Each nodes' IP can also be set manually one by one:

```
lxc stop <node>
lxc network attach <bridge or interface> <node> <nodes interface>
lxc config device set <node> <nodes interface> ipv4.address <ip
address>
lxc start <node>
```

For example, adding lxdbr0 interface from VM to eth0 interface in node5 and giving the eth0 a static IP address of 10.0.0.2:

```
lxc stop node5
lxc network attach lxcbr0 node5 eth0
lxc config device set node5 eth0 ipv4.address 10.0.0.2
lxc start node5
```