Security and Network Engineering — UvA

MSc Research Project 1

University of Amsterdam

Containerized deployment of SURFnet8 service layer network

Pim Paardekooper & Iñigo Gonzalez

March 7, 2021

Supervisor(s): Migiel de Vos, Marijke Kaat & Peter Boers

Abstract

This project is about creating a virtual testbed using containers, done in collaboration with the ICT organization SURF. SURF already has a virtual testbed that makes use of virtual machines. The problem is that the virtual machines for this virtual testbed are very resource-intensive. Therefore it is infeasible to create big topologies. This project had a look at containerization to solve these resource constraints. Containers are less resource-intensive and could therefore be used to create bigger virtual networks. This containerized solution makes use of a containerized routing protocol (cRPD) from Juniper. In this project, a testbed has been created using cRPD instances. These instances are deployed with *k8s-topo* on a Kubernetes cluster that uses the Container Network Interface *meshnet*. To test this virtual testbed a use case is built to measure BGP (Border Gateway Protocol) route convergence with different sized topologies and routing tables. Furthermore, the time it takes to deploy and configure the topology has been measured. The tests showed a decreased performance of *meshnet* as the number of links between routers increased. For this reason, it was infeasible to test one of the topologies for even a small number of nodes due to long startup times. The results prove that the combination of *k8s-topo* and Kubernetes is able to provide a testbed that can be started and recreated in a fast and easy manner.

Contents

CHAPTER 1

Introduction

The use of virtual testbeds is fairly widespread in the industry among both companies and research institutions. They provide the means for testing real hardware-based configurations and setups in a lightweight and virtualized manner separate from the production environment. Network engineers and operators depend a great deal on testbeds to analyze and evaluate the introduction of new features. One of the main advantages of virtual testbeds is the potential cost savings compared to the physical alternative where more pieces of hardware need to be purchased. Even though virtual testbeds also have some licensing costs associated with it, often times physical devices end up costing more. A virtual testbed can offer more ease of management when new links or nodes need to be added. In a physical testbed, installing a new node, adding new link between devices or any other maintenance task requires the presence of an operator on site. On the other hand, these same actions can be performed remotely on the virtual testbed by adding the corresponding virtual devices. Overall, a virtualized setup offers an economical and manageable testbed at the expense of losing a certain amount of fidelity and available features compared to physical testbed[1].

In a previous research study[2] such a testbed was developed for SURF and their SURFnet8[3] project making use of vMX, a virtual router running Junos OS developed by Juniper[4]. Even though a working virtual testbed was produced, the high resource usage of the vMX[5] turned out to be a scalability bottleneck.

Fortunately, the vMX is not the only available product to virtually emulate the MX series routers of Juniper. A containerized solution known as containerized RPD (cRPD)[6] exists which is also developed by Juniper. It consists of an image packaged with Juniper's routing protocol process (*rpd*), the same process run by the Junos OS. The main feature to be studied is its potential to scale up the network topology due to its low resource requirements[7]. This supposes a significant improvement compared to the vMX and provides the potential to develop a testbed that overcomes the aforementioned scalability problem. Therefore, this project will focus on exploring the possibility of using cRPD to build a containerized testbed.

1.1 Project Purpose

This research project aims to find out a more lightweight solution using a containerized architecture by answering the following question:

- How can a containerized testbed using cRPD be scalable in terms of the number of instances to help SURF engineers test their network setup?

To help answer this question, we will interview some of the engineers at SURFnet to set the most important requirements for this testbed. The information gathered will help us define a use case and a prototype will be built according to that. Then a test will be carried out and the results are going to be used to evaluate the scalability of the containerized testbed.

CHAPTER 2

Related work and Background

In this chapter we will first look at tools that create virtual testbeds, especially the ones that uses containerization. After that we will look at the research that has been done at SURF at creating a virtual testbed with vMX's. This to create a better understanding of what is required for a testbed for SURF and why a containerized solution might be useful. Also a bit of background about containers is explained.

2.1 Emulation tools

Our project is focused on building a virtual network testbed. Many tools for this have been developed throughout the years [8]. Mininet, an emulation tool with a high rate of citation in scientific papers, and many other tools work with VMs as their virtualization tool [9] [10]. The problem with VMs is that they are resource-intensive because they require significant RAM and CPU [11]. Therefore the possibility of using containerization is explored in emulation tools such as DockSDN and vSDNEmul [12] [13]. These tools were researched in papers by deploying three kinds of topologies: tree, star and mesh [11] [9]. Then those topologies were scaled up to see how the CPU and RAM usage changed with bigger networks. The scalability of these containerized emulations can be improved by integrating it with something like Kubernetes as stated in the future work in the research about vSDNEmul[9].

2.2 Virtual testbed for SURFnet

In a previous research study[2] a virtual testbed was developed for SURF and their SURFnet8[3] project making use of vMX. This vMX is a virtual router running Junos OS developed by Juniper[4]. It provides the operational consistency of its physical counterpart, the MX Series routers which is the main equipment used for the SURFnet8 network.

Even though the previous study produced a working virtual testbed, the heavy resource usage of the vMX[5] turned out to be a scalability bottleneck. One instance of the vMX router running in *lite* mode [14] requires 4 cores and 3 GB of memory to be allocated in order to boot successfully. This puts a hard limit on the created testbed due to its constrained resource availability, resulting in only 8 router nodes being able to be deployed. The study pointed out that this level of resource requirements was only achieved by running the vMX in *lite* mode, which is only meant for lab usage with low throughput. This means that also the performance is constrained and it will require 9 cores and 5 GB of memory to run in *performance* mode [14]. Attempting to boot the vMX with fewer resources allocated than the minimum specified will fail.

2.3 Containerization

Containerization can be explained as a form of operating system virtualization where running processes are isolated making use of cgroups and namespaces [15]. It packages up all the libraries, system tools and dependencies to form a portable standard unit of software also known as a container image. If compared with virtual machines (VM), containers provide an abstraction at the application layer making use of the kernel of the underlying OS, whereas VM's are an abstraction of the physical hardware. One of the practical differences of VM's compared to containers, is the ability to be created using any OS while containers are tied to the OS of the host. Because of this same reason, container images are typically considerably smaller that VM images and most important, container boot times are significantly lower than those of VM's.

Containerization is achieved thanks to Control Groups (or cgroups) and namespaces. The first provides a mechanism for limiting, accounting and isolating resource usage of a group of processes. Namespaces also provide isolation but they do so by separating a set of processes (Process ID, Network or Mount [16]) from each other that there is no interference between their resources and processes.

CHAPTER 3

Description of software used

The testbed produced in this project makes use of certain software. To create routing instances for the network cRPD is used. This is not a full fledged router, therefore to understand what is possible with it a description of what it can do is given. For deployment of the virtual testbed *k8s-topo* that uses Kubernetes with a CNI plugin called *meshnet* is used. These tools are also described in this chapter.

3.1 cRPD

The containerized routing protocol (cRPD) is a Juniper's routing protocol process (*rpd*) decoupled from their Junos OS and packaged in a Docker container to run in Linux-based environments [17] [6]. The *rpd* is a user space application that learns route state through various protocols. It maintains that state in a routing information base (RIB), a routing table. The state can then be downloaded into the forwarding base (FIB) by the *rpd*, using a routing policy to determine which routes to download. In Juniper routers, the Packet Forwarding Engine (PFE) holds the FIB and performs packet forwarding. For cRPD, this role is done by the Linux Kernel. The *rpd* can download the FIB into the Linux Kernel with Netlink which acts as an interface to the kernel components [18]. Netlink messages are used for the following tasks:

- install FIB state into the Linux Kernel
- interact with mgd (management process) and cli for configuration and management [19]
- maintain protocol sessions using PPMD (Prediction by partial matching)
- detect liveness using Bidirectional Forwarding Detection (BFD), which detects link failures
- let *rpd* learn interface attributes such as their name, addresses, MTU settings and link status.

The cRPD is packaged in a Docker container. Network interfaces learned by the underlying OS are exposed to the *rpd* on Docker containers. The *rpd* learns these interfaces and adds route state to all these interfaces. Multiple Docker containers can access the same network interfaces. When multiple cRPD instances are running the containers are connected to the host network stack through bridges, connected in bridge mode. The containers can communicate over these bridges. By default the docker bridge enables NAT. External communication is possible by connecting the OS network interfaces to the bridge.
The cRPD docker container doesn't need lots of resources as can be seen in table 3.1. If more memory is allocated, more RIB routes can be held by a single cRPD instance (see table 3.2).

Description	Minimum value
CPU	1 core
Memory	256 MB
Disk space	256 MB

Table 3.1: cRPD minimum resources [7]

RIB/FIB route scale	Minimum memory
32,000	256 MB
64,000	512 MB
128,000	1024 MB
1,000,000	2048 MB

Table 3.2: cRPD scaling [7]

The features that are supported by cRPD are described by Juniper on their website [6]:

- BGP Route Reflector in the Linux container
- BGP add-path, multipath, graceful restart helper mode
- BGP, OSPF, OSPFv3, IS-IS, and Static
- BMP, BFD, and Linux-FIB
- Equal-Cost Multipath (ECMP)
- JET for Programmable RPD
- Junos OS CLI
- Management using open interfaces NETCONF, and SSH
- IPv4 and IPv6 routing
- MPLS routing

To run these cRPD features a license needs to be installed. This can be done through the Junos OS CLI. The Juniper license that is used by us was provided by Juniper. Configuration of these features can be done in multiple ways, via the Junos OS CLI or loading an ASCII file with configuration statements [20].

3.2 Kubernetes and k8s-topo

Kubernetes (k8s) is a container orchestration tool [21]. In a two part blog post it is described how to create a Container Network Interface (CNI) plugin to let containers communicate with each other and how a tool called *k8s-topo* works to build user-defined topologies [22] [23].
CNI plugins need to have certain requirements to work with k8s and are stated in the blog post as follows:

- All containers can communicate with all other containers without NAT after the CNI is installed
- All nodes can communicate with all containers without NAT
- The IP address that a container sees itself as must be the same as others see it as.

This means that communication between containers happens at layer 3, the containers need to be able to communicate and with only one IP address you cannot simulate layer 2 protocols other then ARP (Address Resolution Protocol). This is because if all pods need to be connected directly, there are only LAN's of two routers connected to each other. Also, one IP address means one interface which makes creating arbitrary network topologies impossible. Both these reasons makes network simulation impossible in k8s. To make network simulation possible a specialized CNI plugin is needed. This specialized CNI plugin is called *meshnet* [24]. It allows k8s to build arbitrary network topologies out of point-to-point links, on top of a master plugin that keeps the above requirements which keeps it compatible with k8s. The master plugin can be any CNI that connects all the pods together. The *meshnet* CNI is build to merge with any of them, such as

Weave Net, Flannel, Calico or Canal [25].

A CNI plugin can be seen at a high level as just a binary and a configuration file installed on k8s worker nodes. When any k8s pod is scheduled a local node agent, *kubelet*, calls a CNI binary and passes all necessary information to it with the help of environment variables and a CNI configuration file. Then the CNI binary connects and configures the network interfaces and returns the result to *kubelet*. The CNI plugin *meshnet* interconnect pods via direct point-to-point links according to a user-defined topology. The plugin uses two types of links: "veth" is used to interconnect pods running on the same node and "vxlan" to interconnect pods running on different nodes. Point-to-point links do not use bridges but direct connections. The blog post describes the *meshnet* plugin as having these three main components:

- *etcd*: private cluster storing topology information and runtime pod metadata
- *meshnet*: a CNI binary called by *kubelet*, responsible for pod's network configuration
- *meshnetd*: a daemon responsible for vxlan link configuration updates.

This CNI plugin is not stateless because it has a daemon to maintain VXLAN links between different nodes with HTTP PUT requests. The *etcd* component needs to have topology data about which pods need to be connected to which other pod uploaded to it. This information needs to be accessed by other pods. Also to connect those pods with *meshnet* certain ports on those pods need to be exposed. Then also configuration files might need to be uploaded to the pods to make it easy to configure network routing software that is used for network simulation. To not have to manually do all those things an orchestration tool is built *k8s-topo* [23]. What it does according to the blog post is:

- upload topology information into *etcd*
- create a pod for each network device mentioned in the topology file
- if present, mount devices startup configuration as volumes inside pods
- expose internal HTTPS port on every device. K8s uses NodePort, a service type, to expose a service on a port on a k8s node's IP address [26].

There are many supported device types that are Docker images with network routing software installed, such as Quaqqa devices and Juniper vMX devices. Quaqqa is a routing software suite that support many routing protocols as OSPF and RIP [27]. Juniper vMX is already described in section 2.2.

CHAPTER 4

Interviews and provided use cases

In this chapter, we define the use case to be studied. For that, we define a set of requirements and limitations of the containerized testbed. To do so, three employees of different positions at SURF were interviewed and asked about the most relevant use cases for their work. This information, together with the advice of our supervisors, was used to establish a use case to evaluate cRPD as a possible alternate architecture for a testbed at SURF.

4.1 Interviews and use cases

As mentioned before, three employees from three different positions at SURF were interviewed: a Network Engineer, a Software Developer and a Software/NetOps Engineer. A few questions were previously selected as a guide to follow through the conversation and to make sure we gathered the information needed:

1. What do you do at SURF? (introduction)
2. What should the use cases be for the virtual testbed? (what are you going to use it for, give us some scenarios. Which one is most relevant for you?)
3. What should in your opinion the new testbed require?
    - How big should the virtual testbed be for you?
    - What are the minimum requirements of the virtual testbed, size, protocol stack?
    - cRPD can scale to use more routes in the RIB, what should the minimum and optimal number be? (table 3.2)
    - cRPD does not have a routing engine, does the virtual testbed require the data plane? (should we focus on finding a solution to this problem?)
4. What would make this testbed manageable for you? Are there specific tools that need to be integrated?
5. What do you think is feasible in terms of what you can do with a testbed?

The first question was meant to provide a small introduction about the interviewee and their main responsibilities at SURF. The second and third questions were used as guidelines to initiate a conversation about their current virtual testbed and the requirements for an improved testbed that meets their needs. The information obtained in these interviews varied significantly from one interviewee to another. The proposed used cases and requirements differed due to the difference between the tasks performed in the different positions at SURF. The use cases mentioned by the interviewees consisted of testing a specific protocol or a set of tools used in the production environment. Next list shows the most important use cases mentioned:

- EVPN virtual private wire and VLAN aware systems
- (Automatic) Test functionality of service models
- Development environment to test orchestrator's production setup
- Route convergence running BGP
- Route convergence with IS-IS

When asked about the minimum requirements of the testbed for size and the protocol stack, the obvious answer was to have a testbed as big as possible with as many protocols as possible. The production network of SURF consists of more than 300 routers. Building a containerized testbed with that amount of router instances is not strictly necessary to have a useful testbed. Many of their tests can be run with just 6 router nodes, however, it would also be useful for them to be able to deploy different instances of the testbed alongside each other. Therefore, the test case to be studied should take into account different network sizes. Regarding the required protocols we noted down a list of the most mentioned protocols when answering question 3:

- MPLS
- IS-IS
- EVPN (over MPLS)
- BGP which will also require configuring route reflectors.
- LLDP, LACP

Question number four is aimed at finding out what are the most common tools used by the engineers. This was done in order to find out the requirements necessary to integrate these tools into the testbed. Even though this falls outside of the scope of the project we considered it useful to be mentioned for future work. The following list shows the main tools mentioned:

- Northstar
- NETCONF
- NSO (Network Services Orchestrator)

Finally, a few more points were gathered through the last two questions in terms manageability of the testbed. These points do not necessarily influence the choice of a particular test case against another. These are general features preferred by the engineers, therefore they can be useful to help answer the main research question:

- Should be easy to delete and recreate.
- Easy to revert to a base configuration.
- Easy to create many instances of the network running in parallel.

After evaluating the information obtained, we came up with three possible use cases to implement:

1. Implement BGP protocol and test route convergence for different scales (small, medium and large topology). This can be done by:
    - Put a link down and measure the time until all nodes have the same routing info.
    - Add a new node and measure time until all nodes have the same routing info.

This use case could be repeated for different amounts of route reflectors. For example, find out how many route reflectors would be necessary for a given topology scale to be able to deliver a reasonable time for route convergence.
The case would provide a method for testing the performance of the containerized testbed by implementing one protocol on different network topologies. It is a simple setup with a defined methodology for testing the performance of the protocol. This way, the performance of cRPD can be evaluated by looking at the BGP convergence time results.

2. Test the performance of EVPN protocol to different network topologies.
   Similar to the case described above, this case would provide the means for testing the containerized testbed by implementing a single protocol and testing it for different network sizes and configurations. A simple setup for evaluating how well cRPD performs on different network topologies by measuring the performance of a single protocol.

3. Implement as many protocols as possible (to emulate production setup) and measure the spin-up time of the testbed to different scales of the topology.
   This use case can provide a way for measuring how fast cRPD can be started when many protocols are implemented and how stable it can be. This could be done by measuring the time it takes for the cRPD container to be scheduled in Kubernetes until the moment that the configuration is loaded and all the protocols in the stack are operational.

With the help of our supervisors, we decided to implement the first use case. This will be explained in detail in the next section.

4.2 Use case: BGP route convergence for different network scales

The Border Gateway Protocol (BGP) is a path vector routing protocol that allows Autonomous systems (AS)[1] to exchange routing information. This data is kept in the Routing Information Base (RIB) tables and gets maintained and updated dynamically by the exchange of *keepalive* and *update* messages. In practice, BGP maintains three different types RIB tables [2] but for the purpose of this study we are only going to focus on the local RIB table.

Since cRPD does not have a forwarding plane, the route convergence of the control plane is going to be measured. RFC 4098[28], provides the following definition for BGP control plane route convergence:

"A routing device is said to have converged at the point in time when the DUT [3] has performed all actions in the control plane needed to react to changes in topology in the context of the test condition."

Following this definition, we are going to manually add or remove a route from the RIB of one router and measure the time it takes for this update to propagate to the rest of the routers in the network. We will perform this test for 6 different network topologies: three different amounts of router nodes in a full mesh configuration and in a ring shape where each node is directly connected to two other nodes. These connections are made using IPv4. A case could be made to use IPv6 instead. We are going to use IPv4, as IPv4 addresses are shorter and therefore in our opinion easier to work with.

The aim of this use case is to study the response of cRPD to different network topology scales. The results obtained for the convergence time will be compared with results from similar studies [29] [30]. This will help to evaluate cRPD as a suitable tool for testing scalable network

[1]Autonomous system: a network or a collection of networks that are all managed and supervised by a single entity or organization
[2]Adjacent RIB Out (Adj-RIB-Out), Adjacent RIB In (Adj-RIB-In) and Local RIB (Loc-RIB)
[3]Device under test

configurations. In chapter 5 a detailed explanation of the test setup and measuring method is provided.

CHAPTER 5

Building the use case

In this chapter, we define in detail the specifics of the setup. First the topologies we are going to use in our experiment in regards to size and the router configuration. Then we also describe how we are going to conduct the experiment, how we generate the set of BGP routes to use in each topology and how we keep all other factors that influence BGP stable. Finally, we explain how the convergence time is measured.

5.1 Network topology Configuration

For the purpose of testing route convergence of BGP, we are going to define a network setup where each routing instance represents a full AS. This allows for a simple configuration making use of only external BGP without the necessity of implementing other protocols for internal routing. A simplified layout like this provides a good setup for measuring route convergence of the BGP protocol in isolation from other networking protocols.

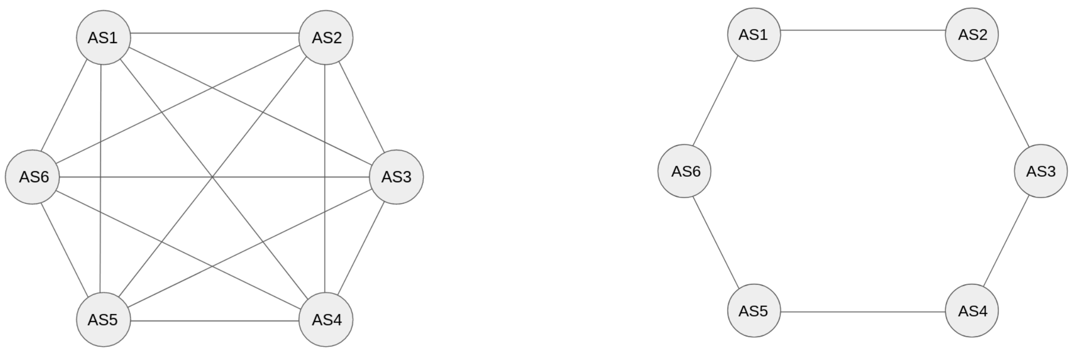

Figure 5.1: Full Mesh (left) and Ring (right) topologies

The network topology used in the experiment is going to be composed of n router nodes in two different configurations: a full mesh and ring-shaped setup. In the full mesh, each router node will have a direct link to each of the other nodes. In the ring setup, each router has a direct connection to only two other router instances. For both configurations, the test is going to be performed on three different sizes: small, medium and big. We use 6 nodes for the small setup since it is also the number of nodes used in the current vMX setup of SURF and it is also the baseline configuration used in the main reference study [30]. For the medium size, an amount of 30 nodes is going to be set up. This number is in the next order of magnitude and just inside the maximum amount that could be scheduled in the Openstack cluster at SURF[2]. Lastly, for the big size we set up a topology of a 100 nodes since this is the smallest number in the next order of magnitude from the medium size.

5.2 Building routing table

To fill up the routing table we will be using a program called *ExaBGP* [31], which is a tool to interact with BGP. It has the nickname "The BGP swiss army knife" because it has many use cases concerning BGP. Such as network protection against DDOS attacks, network monitoring and route announcements. It can do all this because it can make a BGP peer connection to a BGP speaker and can translate any BGP message in readable text or JSON and vice versa. This allows for a text file to be created full of route announcements that can be injected into any BGP peer. By deploying the tool inside a Docker container it can be deployed alongside the topology. Then it can create a BGP peering, by creating a configuration file in which the ExaBGP container's router-id, local-address/IP-address and local-as is specified and the router it wants to peer with its AS and IP address is specified as can be seen in figure 5.2. The IP address used in this configuration file are chosen as to not collide with the prefixes it wants to announce.

```
process announce-routes {
   run /usr/local/bin/python /announce_routes.py;
   encoder json;
}

neighbor 172.16.2.1 {                  # Remote neighbor to peer with
    router-id 172.16.2.2;              # Our local router-id
    local-address 172.16.2.2;          # Our local update-source
    local-as 65000;                    # Our local AS
    peer-as 1;                         # Peer's AS

    api {
        processes [announce-routes];
    }
}
```

Figure 5.2: ExaBGP configuration file

It can then also inject routes that are specified in the text file, which are called from a python script called in the "api" block in figure 5.2. These routes in the text file have a certain format you need to specify which route you want to advertise and the only mandatory option next-hop address which we set to "self" [32]. This will change the next-hop advertised to the IP address of the ExaBGP container specified in its configuration file. Normally eBGP will advertise the next-hop as its own IP address without needing to specify, but in ExaBGP you have to mention explicitly that it needs to put in its own IP address [33]. This by putting the IP address of the ExaBGP container after "next-hop" or putting the option "self" there. An example of a route is the following:

```
announce route 210.162.198.0/24 next-hop self
```

5.3 Factors that influence performance

How to benchmark BGP route convergence in the control plane is specified in RFC 4098[28]. For this study, the most relevant factors that influence the time to reach route convergence are [1]:

- Number of Peers
- Number of Routes per Peer

[1]Flap damping and churn have been left out since are not affecting the results in the test

- Policy Processing/Reconfiguration
- Interactions with other protocols

Only the first two factors are going to be taken as independent variables for the test. Therefore other factors should not influence our experiment and their effects should be eliminated or kept to a minimum.

The influence of "Policy Processing/Reconfiguration" can be eliminated by simply not implementing any policy and thus allowing all sent and received route advertisements. This prevents spending processing time on policy-related decision making. The most obvious way for reducing the impact of the "Interaction with other protocols" is by implementing the least amount possible. Other protocols like IP or TCP are required for BGP to work and they cannot be left out for this test. Therefore the interaction between protocols is kept to a minimum by implementing one routing protocol (BGP). A network configuration where each router instance is also its own AS is a scenario where this single protocol setup is feasible. However, a setup with a multi-node AS where also IGPs are configured results in a scenario where proper protocol isolation cannot be provided.

There exist also implementation specific factors impacting the route convergence. These are caused by conditions internal to the DUT[2] and in this test case do not influence the results. However, we point out why some of these factors have no effect:

- Forward Traffic: the presence of data traffic can put stress on the control plane. In this test, however, no data plane is used therefore this will not have an influence on the measurements.
- Timers: this accounts for the (transmission, propagation, queuing or processing) delay and hold-down settings at the link layer. RFC4098[28] states that these should only be reported if non-default values are used. Since we don't make use of custom values for those settings, the effects of them are left out.
- TCP parameters underlying BGP transport: all BGP communication is done over TCP therefore all relevant parameters must be provided: slow start, max window size, maximum segment size, or timers [28].
- Authentication: since this is achieved by using the TCP MD5 signature option, it can take up significant processing time when a device has a large number of BGP peers. Authentication is disabled by default in Juniper's products [34] and since we are not enabling it, this factor will not have any effect.

5.4 Measuring route convergence

Route convergence has as a definition that the DUT[3] has performed all actions in the control plane needed to react to changes in the network. Measuring route convergence will be done by looking at the timestamps of the BGP update messages in the BGP speakers log files. To trigger the exchange of these messages, we are going to make use of an experiment taken from previous studies [29] [30]:

- UP experiment: from an origin node advertise a single prefix and let all the routers converge.

Then the route convergence is measured as the total amount of simulation time it takes from the origin node to send out the update message to the earliest time after all routers have received this update.

[2]Device under test (in this case is cRPD)
[3]Again, in this case cRPD

CHAPTER 6

Test Description

In this section, we explain in detail the test configuration. How cRPD is deployed for the different topologies and the configuration used to enable BGP communication between peers. Then a description is given about how the routing tables are built for the peers and finally how measurements are extracted from the logs.

6.1 Deployment using Kubernetes and k8s-topo

Due to the high resource requirement of the hundred node test, the engineers from SURF provided us with an Azure[35] account to be able to provision the necessary VMs to setup a Kubernetes cluster. Given that each cRPD instance requires one core and one gigabyte of memory, this test requires a total amount of a hundred cores to be able to schedule every instance. Using a platform like Azure allowed us to add as many worker nodes as we needed to meet those resource requirements. We made use of a total amount of 6 VMs, dedicating one VM for the control plane and a total of 5 worker nodes. All of the VMs had Ubuntu 18.04 installed in them. The control plane was assigned to the smallest VM with 2 vcpu and 4 gigabytes of memory. Then from the worker nodes, 2 of them had 8 vcpu and were used for the small size topology tests. Then for the big size topologies, 3 more nodes were added with 32 vcpu each. All of the worker nodes had more than double the amount of memory in gigabytes compared to the vcpu, which was more than what the cRPD containers require.

```
VERSION: 2
driver: veth
PREFIX: crpd
conf_dir: ./conf_bgp_ring6
custom_image:
    crpd: crpd
links:
  - endpoints: ["crpd-1:eth1:10.0.0.1/24", "crpd-2:eth1:10.0.0.2/24"]
  - endpoints: ["crpd-2:eth2:10.0.1.1/24", "crpd-3:eth1:10.0.1.2/24"]
  - endpoints: ["crpd-3:eth2:10.0.2.1/24", "crpd-4:eth1:10.0.2.2/24"]
  - endpoints: ["crpd-4:eth2:10.0.3.1/24", "crpd-5:eth1:10.0.3.2/24"]
  - endpoints: ["crpd-5:eth2:10.0.4.1/24", "crpd-6:eth1:10.0.4.2/24"]
  - endpoints: ["crpd-6:eth2:10.0.5.1/24", "crpd-1:eth2:10.0.5.2/24"]
```

Figure 6.1: K8s-topo configuration example

On these VMs we installed Kubernetes using *kubeadm*[36]. Then we installed Weave Net[37] as our network add-on of choice and the *meshnet* CNI plugin, as it was explained in section 3.2. Then, we make use of *k8s-topo*[38] to upload the different topology information presented in section 5 and deploy the cRPD instances. These cRPD instances are deployed in the cluster in the form of Kubernetes Pods [39] which simply run one container. The deployments are configured using *YAML* formatted files.

Figure 6.1 shows the configuration file used to deploy the 6 router ring topology. This file is all that is necessary to set the router instances, links, NIC names and IP addresses of the NICs. Each item inside the *links* list in figure 6.1 defines a point-to-point connection. Both elements inside the list in the *endpoints* field contain the pod name, NIC name and IP number separated by a colon in that respective order. Taking the first element in the list as an example, *k8s-topo* will create pods *crpd-1* and *crpd-1* and a *Topology* manifest with the given NIC and IP address (see figure 6.2). This in turn will make *meshnet* create the extra network interfaces in both pods with that information. The pods which are created run also an *init* container that will check and wait until all the NICs needed to be configure for the pod are created. This way the main container running cRPD in the pod will not start until all the NICs are configured. Every topology configuration will make use of the corresponding amount of cRPD pods, plus an extra pod running the *ExaBGP* container that builds the routing tables (as explained in section 5.2).

```
apiVersion: networkop.co.uk/v1beta1
kind: Topology
metadata:
  name: crpd-1
spec:
  links:
  - local_intf: eth1
    local_ip: 10.0.0.1/24
    peer_intf: eth1
    peer_ip: 10.0.0.2/24
    peer_pod: crpd-2
    uid: 1
...
```

Figure 6.2: Topology manifest example

6.2 Building routing tables

Besides the 6 pods running cRPD, an extra node is used running a custom image with *ExaBGP* installed in it, that will connect to one of the pods to setup a BGP peering connection to the cRPD instance running in that pod. Figure 6.4 below shows the *Dockerfile* that is used to create the image. This *DockerFile* is created from a standard python3.7 image, then the *ExaBGP* Python package is installed and some mandatory files are created for ExaBGP to log messages to. As explained previously in section 5.2, *ExaBGP* is used to interact with BGP and it allows routes defined inside a text file to be injected into a cRPD instance. When the pod with the ExaBGP image installed is running, a file with the routes to be announced is copied to the ExaBGP container so it can be accessed by ExaBGP commands. These routes all have a prefix length of 24. Within this ExaBGP pod the function: "exabgp ./conf.ini" can be called. This function will execute a file "conf.ini" which specifies the BGP peer connection it needs to setup, so the IP address and peer-as of the pod it is connected to. It then loops through the file with routes and announces them over the BGP peer connection. After that the cRPD instance will advertise these routes to the other BGP peers it has. Eventually every cRPD instance will have all prefixes that were announced inside its routing table.

```
FROM python:3.7

EXPOSE 179

RUN pip install exabgp
RUN apt-get update && apt-get -y install vim

CMD mkfifo //run/exabgp.{in,out}
CMD chmod 666 //run/exabgp.{in,out}
CMD /bin/bash

ENTRYPOINT /bin/bash
```

Figure 6.3: Custom Dockerfile with ExaBGP installed

6.3 Configuration of cRPD

The configuration of cRPD is done through ASCII files containing Juniper configuration statements [20]. In this case, we are configuring each cRPD instance as its own AS and implementing only BGP as routing protocol. Next, an example of part of one of these files is shown:

```
routing-options {
    autonomous-system 2;
}

protocols {
    bgp {
        traceoptions {
            file bgp-traces-crpd-2 size 4294967295;
            flag update receive;
        }

        group external-peers {
            type external;
            neighbor 10.0.0.1 {
                peer-as 1;
            }
            neighbor 10.0.1.2 {
                peer-as 3;
            }
        }
        ...
    }
}
```

Figure 6.4: cRPD configuration file

The first block, *routing-options*, shows the AS number of the router. In the *protocols* sections the configuration of the BGP protocol is defined. Each *group external-peers* object specifies a peer BGP connection to another cRPD instance directly connected with a link. The *traceoptions* block enables the logging of BGP update messages to be written to a file.

6.4 Test procedure

In this section, the procedure of the experiment is given. First is stated how the experiment is setup and then how the measurements are taking. The results of the project can be obtained by following this procedure.

6.4.1 Experiment setup

The experiment varies two variables: the number of routes in the routing table and the number of routing instances. The number of routes that are used is 0, 10, 100, 1000 and 10000. The number of nodes is 6, 30 and 100. For each combination of the two variables a topology is created. The topology *yaml* files and cRPD configuration files are all created before the experiment, this at the hand of a Jinja template [40].

6.4.2 Topology setup measurement

The first measurements are taken when all Kubernetes pods in the topology are in a ready state. The second measurement is when all cRPD instances have their configuration file and license key installed. The configuration files are mounted onto the k8s pods with the help of *k8s-topo* and the license key is copied to each k8s pod with the command "kubectl cp". Then the configuration files are loaded with the Junos OS CLI command "configure load merge /mnt/flash/startup-config" where "/mnt/flash/startup-config" is the config file that is mounted onto the pod. The license key is installed with the Junos OS CLI command "cli request system license add license". Both these commands are executed for each k8s pod through the "kubectl exec" command.
Executing the "exec" commands for each pod is done sequentially. Once all commands have finished running all cRPD instances are configured.

6.4.3 Filling the routing tables

After that, all routing tables need to be filled. First the *ExaBGP* pod needs to have the full file of routes we want to announce copied with "kubectl cp". The file full of routes is generated by a python script that gets a random IP address and adds it to a file in the format shown in section 5.2. Second, the routing table for each cRPD instance will be filled by calling "exabgp ./conf.ini" inside the *ExaBGP* pod. We will then wait till the routes reach the whole network. We know when we can stop waiting, by first getting the number of routes in each cRPD instances routing table. You get this number by first running the command "show route summary" in the Junos OS CLI for each cRPD instance and then only printing the first lines, which gives something in the following format:

```
inet.0: 32 destinations, 34 routes (31 active, 0 holddown, 1 hidden)
```

Then by using "grep" to get the number before routes you have the number of routes in the routing table. Keep this number for each cRPD instance and keep getting this number until all these numbers increase to the number of routes it started with plus the number of routes that are injected by *ExaBGP*.

6.4.4 BGP route convergence measurement

Afterwards the experiment can start by injecting one other route, which is not yet in any of the routing tables. This route is 10.0.254.0/24, which gets injected with the *ExaBGP* pod. The start time of injecting this route is written in a log file in the *ExaBGP* pod, it is the first message with the announced route in it and the line starts with the timestamp. The moment the network has converged is when all cRPD instances have received the BGP update for this route announcement. In the cRPD configuration files *traceoptions* is activated so that BGP update messages are written down in each cRPD instance in a file called: "bgp-traces-crpd- " + indicator number. In this file we get the line where the route in the route announcement first shows up with the help of "grep". At the beginning of this line there also is a timestamp written. The end time is the most recent timestamp, of the BGP update, from all cRPD instances.

6.4.5 Gathering results

After that the convergence time, number of routes, number of nodes, time to create topology, time to configure all cRPD instances and iteration number is appended to the results file. The iteration number is added, because each experiment is executed 5 times. Then lastly, the topology is destroyed. After that the next combination of number of routes and number of nodes is used to perform all measurements again.

CHAPTER 7

Results and Discussion

First, we look at the creation and configuration time of the topologies in the results in figure 7.1. The average deployment time for creating the topology and configuring it increases with the number of pods. Each pod that gets added needs time to deploy and configure itself. This time decreases a bit when the number of pods increases as can be seen in table 7.1. An explanation for this could be that the overhead of scheduling the pods is diminished by the number of pods deployed. The important point is that the increase in creation time per added pod is slightly less than linear. The figure shows that the creation and configuration time of the topology with 100 nodes is less than 3 minutes and that it does not vary significantly looking at the standard deviation. Adding more nodes increases it by 0.8s per pod in the worst case. Creating a topology with 100 nodes in less than 3 minutes is reasonable because the topology is for network engineers to test services. To test these services these topologies do not need to be taken down and started up every minute but are most likely running for longer periods of time. Therefore only looking at the deployment time, cRPD instances are a valid choice to create network topologies.

Second, if we look at figure 7.2 it shows the results of the BGP convergence time for different topology sizes. Also looking at table 7.2 it can be seen that the convergence time increases per pod by 0.1s. In a ring topology update messages only need to travel one pod further to put the route in all the cRPD instances. Therefore the test setup stays stable with a bigger topology as each pod always takes 0.1s to forward the prefix. The performance of each individual router instance does not change when more routers are added to the network.

Lastly, we look at how the convergence time changes when the cRPD instances hold more routes which is shown in figure 7.3. In section 5.3 it is mentioned that the number of active routes in the routing table has an influence on the route convergence time. However, the routes in the routing tables of the cRPD instances are not active as they don't send update messages at the time of the experiment (it is not in the update interval). Therefore increasing the routes in the routing table has no influence on the route convergence. What this does show is that the test setup remains stable even when the cRPD instances use more of their allocated memory. It is more intensive on the resources to hold more routes in the routing table. The amount of memory usage does not affect the performance of cRPD instances since our experiment shows the same results with different amounts of routes in the routing tables.

Now we discuss an experiment we tried to do that did not work, as it shows some limitations that our setup has. This is useful as it points out what the setup still needs to improve on to make it suitable for SURF engineers. During the test run for the full mesh topology, a significant increase in the creation time was observed in comparison with ring topology setup. As explained in section 6.1, the *init* container configured in each pod will wait until all NICs are configured preventing the main container (cRPD) from starting. For the full mesh topology, this *init* container will remain running for a long period of time (in the order of hundreds of seconds), meaning therefore that the NICs were taking a long time to be configured by *meshnet*. This was

caused by the high amount of links needed to be created in total which increases by $n(n-1)/2$, with n being the number of router instances being deployed.
The increase in links puts bigger stress on the workload needed to be handled by *meshnet*, hence delivering a very slow response. Running a full mesh topology with ten nodes already was taking more than ten minutes to start. This made it unfeasible to run the medium and big size test therefore this topology setup was dropped from the experiment cases. Our setup can therefore not create networks where routers need a lot of interfaces which makes some experiments impossible to do. This is a shortcoming that needs to be overcome. It can maybe be done by using a different CNI than *meshnet* or another underlying infrastructure architecture. To be certain this should be researched further.

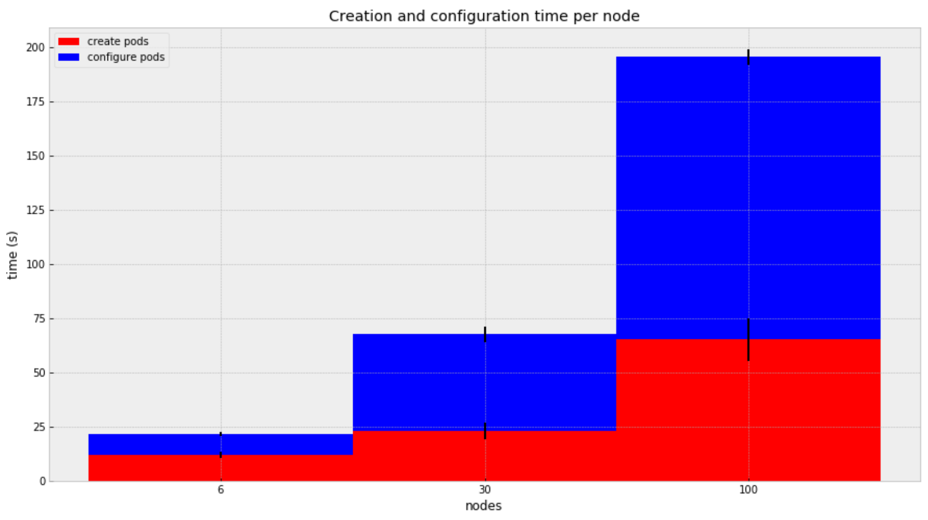

Figure 7.1: Ring Topology average deployment time (including loading router configuration) for 5 iterations.

nodes	creation time (s)	pod creation time (s)	configuration time (s)	pod configuration time (s)
6	12	2	10	1.66
30	23	0.76	44	1.47
100	65	0.65	130	1.3

Table 7.1: Shows pods that get created and configured per second for different sized topologies. This is done by dividing the creation time and configuration time through the number of node.

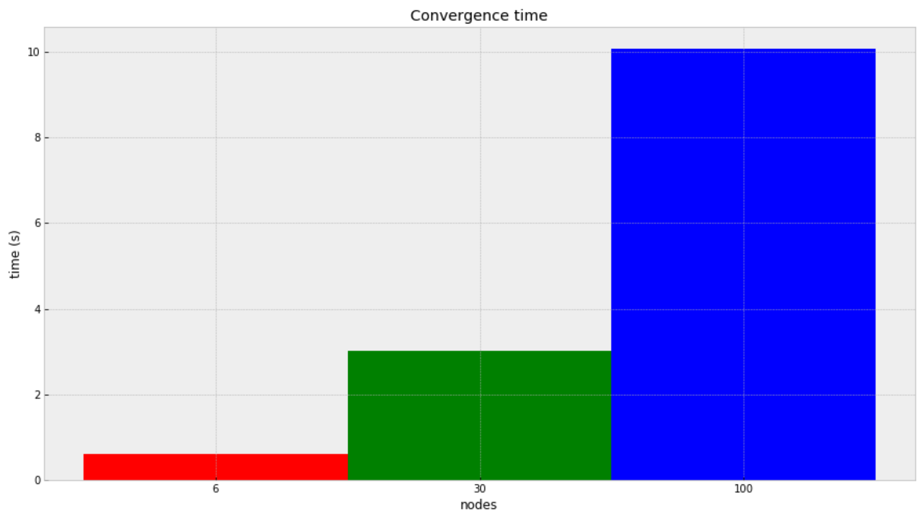

Figure 7.2: Average Route Convergence time in seconds for Ring Topology for 5 iterations.

nodes	convergence time (s)	increase convergence time per pod (s)
6	0.6	0.1
30	3.0	0.1
100	10	0.1

Table 7.2: Shows convergence time increase per pod with different sized topologies. This is done by dividing the convergence time through the number of node.

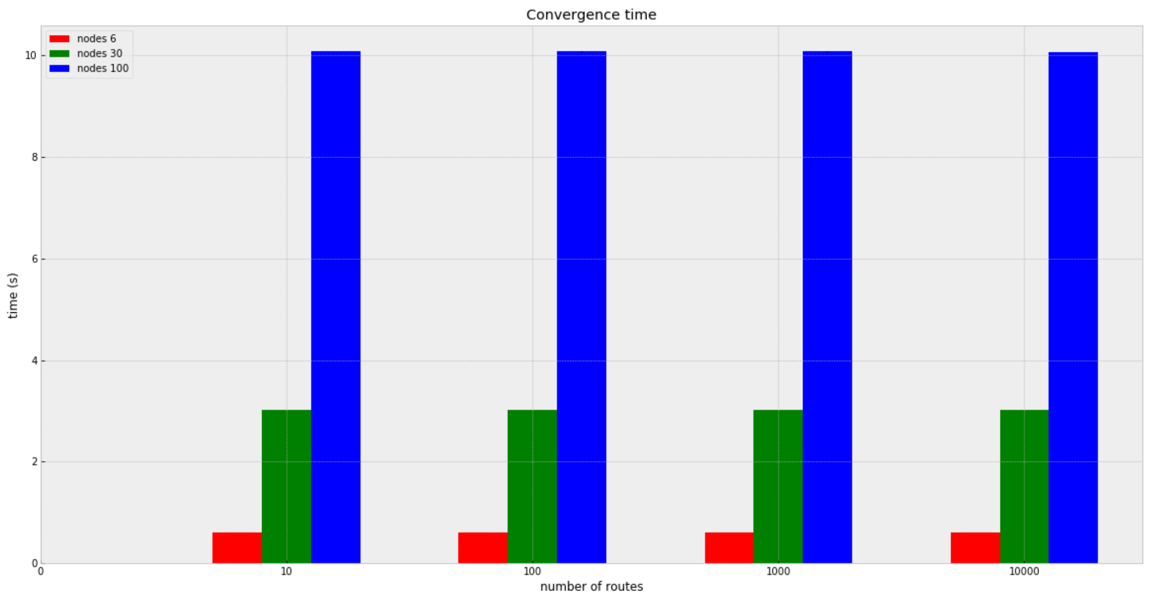

Figure 7.3: Average Route Convergence time in seconds for Ring Topology and different number of routes in the cRPD instances routing table.

CHAPTER 8

Conclusions and Future Work

In this paper, we presented and tested a use case for a containerized testbed using cRPD. The study was focused on evaluating the scalability of the model to find out how it can help engineers at SURF test their network setup.

First, we conducted interviews that point to the requirements that this testbed needs. These point out the necessity of building a testbed where a deployed configuration can easily be reverted, deleted and recreated. Out of the interviews came a use case that we used to test the created testbed. The use case is to test BGP route convergence on the testbed and how it scaled with different topology sizes. The test was designed using a simplified setup with one routing protocol and a small number of topology configurations. Although these configurations do not represent a realistic setup, they are useful for making the first evaluation of a containerized testbed using cRPD. In the experiments, a ring topology was successfully configured and BGP convergence was measured for the three topology sizes and four different numbers of routes in the routing tables. This shows how a containerized testbed using cRPD can be used for testing network configurations. The test setup described in section 6.1 shows how the use of *k8s-topo* and Kubernetes provides a solution for quickly deploying different network and topology configurations.

In the results, it is stated that the convergence time is reasonable if used to create a topology that does not need to be taken down shortly after creation. Which is the case for using it to test network services. Also, the convergence time increases linearly with the number of pods. Therefore the test setup is stable even when the number of pods increases. Furthermore, the test setup remains stable when using more routes in the routing table. The route convergence remained the same even when cRPD uses more of its allocated resources. A problem with the test setup is that *Meshnet* turned out to be a scalability bottleneck, delivering an increasingly slow performance as the number of links in the topology increased. This shows that the testbed designed for this study is not suitable for testing every topology configuration and that the startup time is heavily influenced by the number of links between routers.

Our research question was:

- How can a containerized testbed using cRPD be scalable in terms of the number of instances to help SURF engineers test their network setup?

The containerized testbed we used scales well with the number of router nodes and number of routes in the routing table. This is an argument for using a containerized testbed for cRPD instances and tells that it is scalable in the number of router nodes and routes in the routing tables. However, because of the bottleneck the CNI *meshnet* has it cannot be used yet by SURF network engineers to test their setup. Therefore more research needs to be done to know if this bottleneck can be overcome, before answering the question if a containerized testbed using cRPD can be used by SURF engineers and if it is also scalable with the number of interfaces per pods.

8.1 Future Work

Due to the lack of similar studies, there is plenty of research studies that could be carried out around cRPD and containerized testbeds for network configurations. During the test case definition phase of this project, a few different use cases were proposed with the information gathered from the interviews (section 4.1). However, we want to point out the work that could be developed on the problems encountered during the testing phase of this study.

One of the main remarks of the conclusions above is the scalability bottleneck that *meshnet* introduced for topologies with a high number of links between routers. A possible research study could focus on solving this issue by testing a different network plugin for Kubernetes or testing a different underlying infrastructure setup. To extend on the latter, our Kubernetes cluster was made up of one *master* node and five *worker* nodes (see section 6.1). Three of those nodes were big host VMs which accounted for 85% of the total amount of available resources in the cluster. This meant that most of the router instances would be deployed on those nodes and would therefore put high stress on the *meshnet* daemon running in each of those nodes. The use of a different architecture using several smaller host VMs (as *worker* nodes) could help redistribute the load between several daemons.

Other studies could be directed towards investigating the factors that influence the startup time of the containerized testbed. In this study, the configuration loading time increased significantly with the amount of router instances. The development of a better-suited method for loading this configuration could reduce this time further. Also, another study could be carried out to see the effect on the start up time of more extensive router configurations with more protocols.

8.2 Acknowledgments

This project has been done in collaboration with people of SURF. We want to thank our supervisors for all the help they have given us. We also thank the people interviewed in this project for the time they took to answer our questions. Lastly, we want to thank Juniper for giving us access to the cRPD software and the license. Without it, we could not have done this research project.

Bibliography

[1] J. Crussell et al. “Virtually the Same: Comparing Physical and Virtual Testbeds”. In: *2019 International Conference on Computing, Networking and Communications (ICNC)*. 2019, pp. 847–853. DOI: 10.1109/ICCNC.2019.8685630.

[2] Cees Portegies. “Virtual testbed for SURFnet, Tool evaluation and prototype”. In: *University of Amsterdam* 1 (June 2018). DOI: https://esc.fnwi.uva.nl/thesis/centraal/files/f304980442.pdf.

[3] SURF Nederland. *SURFnet8 Project.* URL: https://www.surf.nl/project-surfnet8.

[4] Juniper Networks. *vMX Overview.* URL: https://www.juniper.net/us/en/products-services/routing/mx-series/vmx/.

[5] Juniper Networks. *vMX Minimum Hardware and Software Requirements.* URL: https://www.juniper.net/documentation/us/en/software/vmx/vmx-getting-started/topics/concept/vmx-hw-sw-minimums.html.

[6] Juniper Networks. *Understanding Containerized RPD.* URL: https://www.juniper.net/documentation/us/en/software/crpd/crpd-deployment/topics/concept/understanding-crpd.html.

[7] Juniper Networks. *cRPD Resource Requirements.* URL: https://www.juniper.net/documentation/us/en/software/crpd/crpd-deployment/topics/concept/crpd-hardware-requirements.html.

[8] *List of network simulators and virtualization tools - NIL - Network Information Library.* June 2020. URL: https://nil.uniza.sk/network-simulation-virtualization-software-list/.

[9] Fernando N. N. Farias et al. “vSDNEmul: A Software-Defined Network Emulator Based on Container Virtualization”. In: *CoRR* abs/1908.10980 (2019). arXiv: 1908.10980. URL: http://arxiv.org/abs/1908.10980.

[10] Mininet Team. URL: http://mininet.org/.

[11] E. Petersen and M. A. To. “DockSDN: A Hybrid Container-Based SDN Emulation Tool”. In: *2020 IEEE Latin-American Conference on Communications (LATINCOM)*. 2020, pp. 1–6. DOI: 10.1109/LATINCOM50620.2020.9282297.

[12] Jvrmaia. *DockerSDN.* URL: https://github.com/jvrmaia/docker-sdn-base.

[13] Fernnf. *vSDNemul.* Nov. 2018. URL: https://github.com/fernnf/vsdnemul.

[14] Juniper. *Juniper Performance and lite modes.* URL: https://www.juniper.net/documentation/us/en/software/vmx/vmx-getting-started/topics/task/vmx-chassis-flow-caching-enabling.html.

[15] Docker. *Docker Containers.* URL: https://www.docker.com/resources/what-container.

[16] Wikipedia. *Linux Namespaces.* URL: https://en.wikipedia.org/wiki/Linux_namespaces.

[17] *Docker.* URL: https://www.docker.com/.

[18] *Netlink.* URL: https://man7.org/linux/man-pages/man7/netlink.7.html.

[19] *Junos OS Routing Engine Components and Processes.* Jan. 2020. URL: https://www.juniper.net/documentation/en_US/junos/topics/concept/junos-software-components-and-processes.html.

[20] *Methods for Configuring Junos OS.* URL: https://www.juniper.net/documentation/en_US/junos/topics/concept/junos-software-configuration-methods-overview.html#jd0e132.

[21] *Production-Grade Container Orchestration.* URL: https://kubernetes.io/.

[22] Michael Kashin. *Large-scale network simulations in Kubernetes, Part 1 - Building a CNI plugin.* Nov. 2018. URL: https://networkop.co.uk/post/2018-11-k8s-topo-p1/.

[23] Michael Kashin. *Large-scale network simulations in Kubernetes, Part 2 - Network topology orchestration.* Nov. 2018. URL: https://networkop.co.uk/post/2018-11-k8s-topo-p2/.

[24] Michael Kashin. *Meshnet CNI.* URL: https://github.com/networkop/meshnet-cni.

[25] Justin Ellingwood. *Comparing Kubernetes CNI Providers: Flannel, Calico, Canal, and Weave.* Mar. 2019. URL: https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/.

[26] *Service.* Jan. 2021. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[27] *Quagga.* URL: https://www.quagga.net/.

[28] Marianne Lepp et al. *Terminology for Benchmarking BGP Device Convergence in the Control Plane.* RFC 4098. June 2005. DOI: 10.17487/RFC4098. URL: https://rfc-editor.org/rfc/rfc4098.txt.

[29] Timothy Griffin and Brian Premore. "An Experimental Analysis of BGP Convergence Time". In: Jan. 2001, pp. 53–61. DOI: 10.1109/ICNP.2001.992760.

[30] Nenad Laskovic and Ljiljana Trajkovic. "BGP with an adaptive Minimal Route Advertisement Interval". In: vol. 2006. May 2006, 8 pp. –142. ISBN: 1-4244-0198-4. DOI: 10.1109/.2006.1629400.

[31] Exa-Networks. *Exa-Networks/exabgp.* URL: https://github.com/Exa-Networks/exabgp.

[32] *exabgp.conf.* URL: https://manpages.debian.org/testing/exabgp/exabgp.conf.5.en.html.

[33] *A fresh look at BGP's NEXT_HOP.* Oct. 2015. URL: https://www.noction.com/blog/bgp-next-hop.

[34] Juniper. *BGP Route Authentication.* URL: https://www.juniper.net/documentation/en_US/junos/topics/topic-map/bgp_security.html.

[35] Microsoft. *Microsoft Azure.* URL: https://azure.microsoft.com/nl-nl/.

[36] Kubernetes. *Creating a cluster with Kubeadm.* URL: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/.

[37] Weaveworks. *Weave Net network add-on.* URL: https://www.weave.works/.

[38] Michael Kashin. *K8s-topo.* URL: https://github.com/networkop/k8s-topo.

[39] Kubernetes. *Kubernetes Pods.* URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[40] URL: https://jinja.palletsprojects.com/en/2.11.x/.