Bachelor Informatica

University of Amsterdam

Performance of Docker Overlay Networks

Arne Zismer (10495800)

June 8, 2016

supervisor: Dr. Paola Grosso

signed: Dr. Paola Grosso

Abstract

The emergence of the virtualization technology Docker increased the need for connecting its containers across multiple hosts. Overlay networks are solving the problem of static network configurations by abstracting the physical networks. My research examines the performance of Docker overlay networks `libnetwork`, Flannel and Weave on the GÉANT Testbed Service (GTS). Previous research has shown that the UDP throughput of the overlay networks is significantly lower than the throughput of a VM whereas their TCP throughput even partly outperforms that of the VM. I will demonstrate that this unexpected behaviour is caused by the combined effects of VXLAN encapsulation overhead and hardware offloading which has a greater impact on TCP throughput than on UDP throughput.

Furthermore, I examine the performance of Weave's multi-hop-routing feature and find that this feature does not introduce any latency penalty. The UDP throughput only slightly decreases with each additional hop whereas the TCP throughput dramatically drops after the first hop.

CONTENTS

1 INTRODUCTION

Docker is an open-source software that allows to deploy applications inside of software containers. Even though the technology of software containers already existed before the advent of Docker, it was too complex to apply this technology on a large scale. The emergence of Docker greatly facilitated the use of software containers, which makes them a more efficient, and therefore more popular, alternative to traditional virtualization. As a result of the wide application of Docker containers, the need to connect them to a network occurred. Connections across different hosts are achieved by network overlays which can be implemented in different ways. The recent introduction of Docker's native networking library `libnetwork` allows third-party solutions to integrate with Docker containers. Currently, the most commercial and community backed third-party solutions are Weave and Flannel. [3]

Whereas plenty of researches about performance differences between Docker containers and traditional virtual machines were conducted, little research exists about the performance of Docker overlay networks. In February 2016 Hermans & de Niet examined the performance of different Docker overlay solutions in their research project "Docker Overlay Networks - Performance analysis in high-latency networks"[3] . My research continues and expands the research of Hermans & de Niet and further investigates the performance of the overlay solutions `libnetwork`, Weave and Flannel. The main question for this research is:

> **What is the performance of the Docker overlay network solutions `libnetwork`, Weave and Flannel?**

This question can be divided into more precise sub-questions that are answered in this project. Previous research revealed that the UDP throughput of all overlay network solutions is significantly lower than the throughput of a VM. This discrepancy cannot be observed for TCP throughput. This unexpected behavior leads to the first sub-question:

(1) **Why is the UDP throughput of the Docker overlay solutions Flannel, Weave and Libnet so much lower than their TCP throughput?**

Furthermore, more complex application communication scenarios need to be researched. Hermans & de Niet thoroughly examined the performance of different overlay solutions for point-to-point connections as well as for networks in a star-topology in which a central node is stressed with an increasing amount of concurrent sessions. The results of these two topologies need to be supplemented with more complex scenarios. One of Weave's features which is not supported by other overlay networks is hop-routing. This leads to the second research question:

(2) **What is the performance of Weave's hop-routing feature with respect to throughput and latency?**

The remainder of the this paper is structured as follows: Chapter 3 on page 5 explains the Docker project, the examined overlay networks and their common encapsulation protocol VXLAN in more detail. Furthermore, it provides an overview of the GÉANT Testbed Service which is used for all experiments. The results and conclusions of related researches are presented in chapter 2 on the next page. Chapter 4 on page 12 explains the methodology that is used to conduct the experiments. The results of these experiments are presented in chapter 5 on page 19 and discussed in chapter 6 on page 25. Chapters 7 on page 27 concludes the project and presents ideas for future research.

2 RELATED WORK

In the past, different aspects of the performance of Docker containers have been researched. These researches mainly focus on the differences between traditional virtual machines and Docker containers.

Scheepers [13] compared hypervisor-based and container-based virtualization. He concluded that hypervisor-based virtualization, such as XEN, is more suitable for applications that require equally distributed resources that should not be affected by other tasks executed on the same system whereas container-based virtualization, such as *Linux Container* (LXC), is more applicable for making efficient use of resources. Morabito [11] also compared those two virtualization technologies and came to the same result.

Claassen [1] examined the performance of the kernel modules `veth`, `macvlan` and `ipvlan` that interconnect containers on the same host and also link them to the network. He came to the conclusion that `macvlan` in bridge mode is the most efficient module on a single host deployment. On a switched environment, `macvlan` and `veth` are performing equally good. `ipvlan` turned out to be not production-ready due to stability issues. Claassen also experimented with various overlay network solutions but noticed that they were not mature enough to be tested.

Even though Docker containers are a heavily researched topic, little research has been conducted on Docker overlay networks. The main reason for this lack of research is the fact these solutions are still quite new. Furthermore, due to the fast paced development of the existing overlay solutions and the recent introduction of Docker's native networking library `libnetwork`, most of the performance analysis is already outdated.

Recently, Hermans & de Niet [3] examined the usability and performance of `libnetwork` and the third-party overlay networks Weave and Flannel. According to their experience, Weave was the easiest to deploy, because it does not require a key-value store, like `libnetwork` and Flannel, which needs to be installed separately and makes it more complex to set up the overlay network. This property makes Weave especially suitable for agile development and rapid prototyping.

Even though the three solutions use very different approaches to realize the routing of packets, point-to-point measurements revealed no significant differences in latency and jitter. Compared to a VM that does not run Docker, the overlay networks merely introduced a negligible increase in latency and jitter. However, the examination of TCP and UDP throughput yielded unexpected results. `iperf` measurements showed that the TCP throughput of some overlay networks is even higher than that of a virtual machine. Furthermore, the overlay's UDP throughputs turned out to be merely 50% of the virtual machine's throughput. The results can be seen in figure 1. The first research question seeks to investigate this unexpected behaviour.

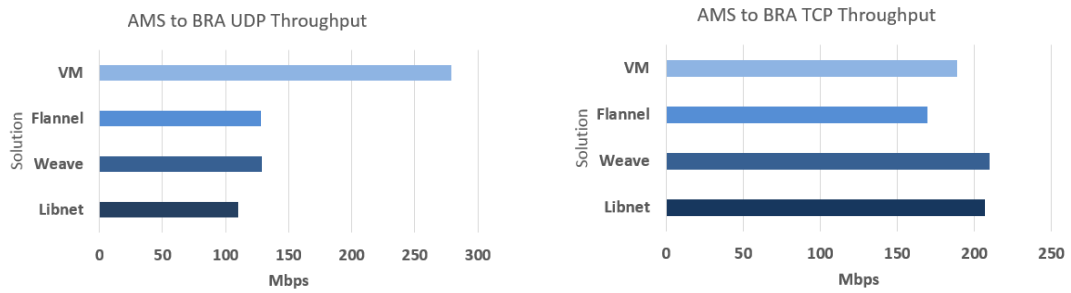


Figure 1: Discrepancy between UDP and TCP throughput of a VM and overlay networks from Amsterdam to Bratislava [3]

Additionally, Hermans & de Niet examined the performance of the overlay networks in a star-topology by stressing a central node with an increasing amount of clients streaming media files. The results of this streaming scenario showed, that latency and jitter of the overlay networks are very similar and are not significantly different from those of a virtual machine. Furthermore, with only 2 milliseconds of jitter for nine concurrent sessions, the jitter is far below the critical limit of 30 milliseconds that Cisco defined for video streams.

3 THEORETICAL BACKGROUND

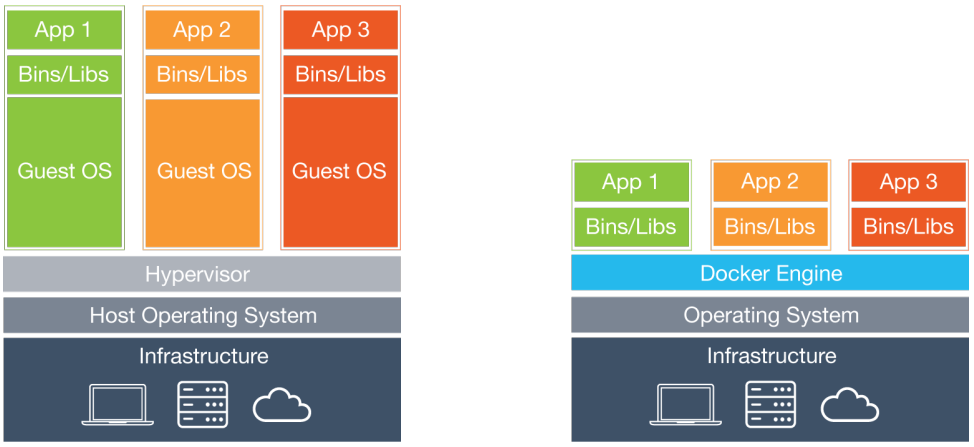
This chapter briefly explains the technologies that are used in this research. After discussing the Docker platform in general, the different overlay network solutions and the network virtualization technology VXLAN, which all of the overlay networks rely on, are covered in more detail.

3.1 THE DOCKER PLATFORM

Docker is an open-source platform which aims to improve the application deployment process by means of operating-system-level virtualization with so called *software containers*. This technology already existed before the advent of Docker, but Docker managed to implement it in a more usable way which is the key to its success.

All dependencies that are necessary for an application to run are included in the Docker container which makes it independent from the environment it is running in. This greatly increases the portability of an application since it can easily be deployed on different kinds of machines, such as local hosts, physical and virtual machines and cloud platforms.

Docker containers are managed by the *Docker Engine* which makes use of the kernel's containerization features `cgroups` and `namespaces` to isolate applications from each other. It runs directly on the host's kernel. Consequently, neither a hypervisor nor a guest operating system is needed for running Docker containers. The differences between the two architectural approaches can be seen in figure 2. As this figure implies, Docker containers are running with a significantly lower overhead. Furthermore, Docker containers are very lightweight, due to the fact that they merely contain the files that are essential for the application to run. These characteristics make dockerized applications very flexible and scalable because they can easily be deployed and updated. [7]



(a) Architecture of a virtual machine

(b) Architecture of Docker

Figure 2: The two different architectural approaches of virtual machines and Docker [7]

Docker containers are run from read-only templates called *images*. These images consist of different *layers* which describe the changes that are made to the previous layer. These layers

can easily be added, modified or removed without having to rebuild the whole image. Internally, layers are implemented by the *advanced multi-layer filesystem* (aufs) which allows files and directories of separate file systems to be transparently overlaid. [6]

Images can either be pulled from the Docker Hub, a centralized repository which holds a large collection of official images, or created by writing a `Dockerfile`. Such a file lists all the instructions that are used expand a base image. Each instruction adds another layer to the image. These instructions include running commands, adding files and directories and defining which processes should be started when launching a container of that image. Once a container is launched from that image, a thin writable layer is created, which allows to write changes to the container while it is running. These changes can then be committed to create an updated image which can then be used to create exact replicates of the current container [6].

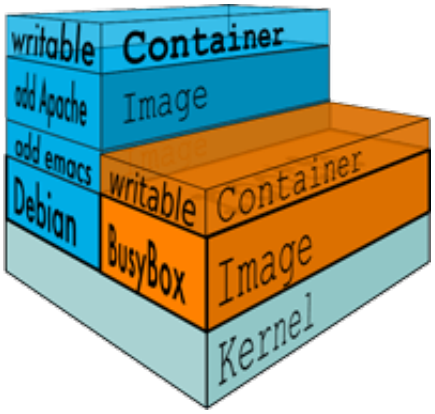


Figure 3: Illustration of the layers of two Docker containers running on the same kernel. [7]

As seen in figure 3, Docker containers are running on the same kernel, which allows developers to build an entire multi-tier application on one single workstation, without the overhead of multiple operating system images that traditional virtualization would introduce. A virtual bridge named `docker0` handles the communication between the containers and the host machine. It is connected to a subnet from a private IP range and every running container connects to this subnet via a virtual Ethernet (`veth`) interface. Inside a container, this interface usually appears as a regular `eth0` interface. Since all containers that are running on the same host are connected by the same subnet through the `docker0` bridge, they can communicate with each other. [5]

3.2 DOCKER OVERLAY NETWORKS

Communication between containers and external machines is handled by the local host which uses *network address translation* (NAT) to forward traffic to the external machine. Therefore, directing traffic from a container to an external server can easily be achieved by default Docker networking. Before Docker 1.9 was published, initiating a connection from an external machine to a container required some extra configuration which mapped ports to specific containers statically. Setting up connections between containers on different hosts was a complex task and the resulting networks were very inflexible. [15]

Overlay networks are solving the problem of static network configurations by abstracting the physical network. This abstraction allows for dynamically creating logical links between endpoints which are not coupled to the hardware infrastructure of the underlying network. By establishing an overlay network between containers that are running on different hosts an IP address is assigned to every container. Thus, containers are not statically linked by a port number anymore and can connect to each other independently of their hosts IP address and

location. The next subsections present different overlay network solutions which can be used to easily connect Docker containers across different hosts.

3.2.1 Libnetwork

In March 2015 SocketPlane, a small start-up that strived to implement multi-host networking between Docker containers through *system-defined networking* (SDN), joined the Docker team [12]. Their approach is to connect containers to an *Open vSwitch* (OVS) port instead of a virtual bridge. As of Docker version 1.9, SocketPlane's solution is included in Docker's main release in the native networking library `libnetwork`, which holds all of Docker's networking code. This modular approach, which follows the Docker philosophy of developing small, highly modular and composable tools, allows local and remote drivers to provide networking to containers [15].

`Libnetwork` implements *Container Network Models* (CNM) which allow containers to be addressed like standalone hosts. [4] As seen in figure 4, the CNM consists of three main components: *Sandboxes*, *endpoints* and *networks*. A sandbox contains the configuration of a container's network stack, such as the container's interfaces, its routing table and DNS settings. An endpoint joins a sandbox to a network. Sandboxes can have multiple endpoints that are connecting them to multiple networks (see the middle container in figure 4). Such an endpoint could for example be implemented as a `veth` interface or an OVS internal port. A group of endpoints that are able to communicate directly are forming a network. Implementations of networks could be a Linux bridge or a `VXLAN` segment. [4]

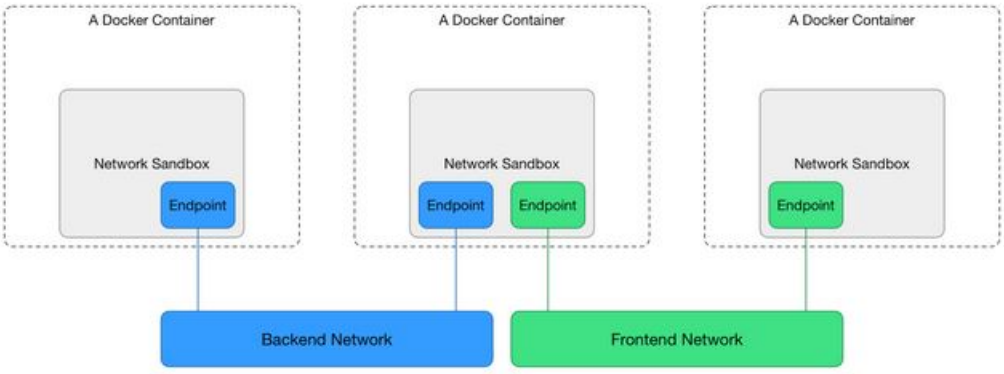


Figure 4: `Libnetwork`'s Container Network Model. [4]

The `Driver` objects of the CNM abstracts the actual implementation of a network and is responsible for managing it. There are two different kinds of `Driver` objects: Built-in drivers and remote drivers. The built-in bridge driver connects containers on the same host. When installing Docker, the `docker0` bridge is created by default and Docker automatically creates a subnet and gateway to this network. When running a container, Docker automatically connects them to this network. The default `host` and `none` networks are also built-in networks which are required by the Docker installation for historical reasons. However, they are not meant for interaction and are therefore not explained in further detail.

According to Docker themselves, the default bridge network `docker0` is error prone and is better to be replaced by a user-defined network. It is possible to create multiple user-defined networks on the same host. Containers can only communicate within networks but not across them. After creating a user-defined network, a container needs to be connected to it explicitly when it is launched. User-defined bridge networks are isolated from external networks and cannot be linked. However, by exposing container ports, a portion of the bridge network can be made available to outside networks.

Networks that connect containers across different hosts are managed by the overlay network driver, which supports multi-host networking out-of-the-box. This driver is based on the network virtualization technology *Virtual Extensible Local Area Network* (VXLAN). The overlay network driver requires a key-value (KV) store, which stores information about all participating hosts. At the time of writing `Libnetwork` supports the KV-stores Consul, Etcd and ZooKeeper. After setting up a connection to the KV-store on each host, a Docker overlay network can be created. Just like with bridge networks, launched containers need to be connected to the overlay network explicitly. Every container on that overlay network has its own virtual interface and can directly communicate with containers on other hosts that are part of that overlay network. [5]

3.2.2 Weave

Docker also supports remote network drivers that allow third-party providers to register with `libnetwork` via the plugin mechanism. Weave is such a plug-in.

A Weave overlay network consists of several software routers which are placed in Docker containers. Every machine that participates in the overlay network requires such a Weave router container. Weave routers establish a TCP connection over which they exchange information about the network topology. Additionally, they also establish UDP connections to transfer encapsulated network packets. In order to connect containers to the overlay network, Weave creates its own network bridge on each host. This bridge connects the Weave router with other containers that connect via their `veth` interface.

The Weave router only handles traffic between local and external containers. Communication between different local containers and communication between the host and local containers is routed straight over the bridge by the kernel. Packets destined to an external container are captured by the Weave router and encapsulated in a UDP packet which is then sent to a Weave router on another host. The the Weave router captures packets that are heading to the same external host within a short time, it tries to encapsulate as many as packets as possible within a single UDP packet.

Since the routers are aware of the complete network topology and also propagate topology changes via their TCP connection, they are able to decide to which Weave router a packets should be forwarded to, instead of having to broadcast the packet to every router. Furthermore, Weave supports hop-routing which allows to route packets through partially connected networks.

Weave also supports encryption of its TCP and UDP connections which is implemented with the *Networking and Cryptography library* (NaCl), which was chosen for its good reputation and clear documentation. [17]

Weave 1.2 introduced *Fast Data Path*, an optimization which greatly increases network performance. Instead of sending arriving and departing packets to the router to process, the router tells the kernel how to process them. As a consequence, packets can travel between containers and the external network directly, without the detour over the router, which is running in user space. Figure 5 on the next page illustrates this optimization. Prior to version 1.2 Weave used a custom format for encapsulating the packets. However, in order to enable the kernel to route the packets, VXLAN (explained in more detail in section 3.3 on the following page) is used for encapsulation. At the time of writing, Fast Data Path does not support encryption yet. [18]

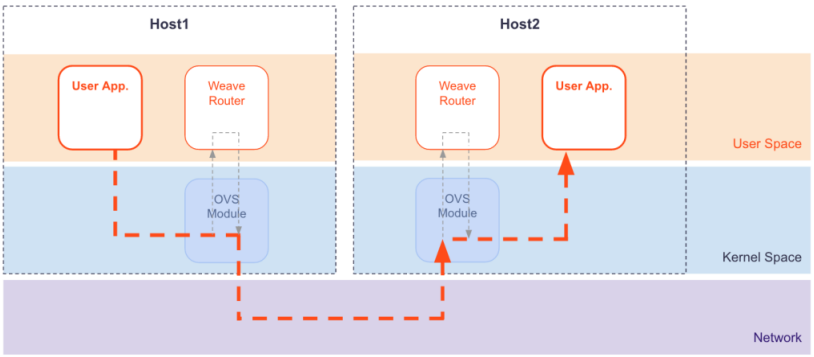


Figure 5: Weave's *Fast Data Path* optimization [18]

3.2.3 Flannel

Flannel was developed as a component of CoreOS and was originally designed for Kubernetes, an open-source container cluster manager designed by Google, for large scale environments with a multitude of containers. In contrast to the Weave solution, Flannel does not integrate with `libnetwork` and does not deploy separate container instances. Instead it creates a daemon (`flanneld`) on each container host. This background process controls the overlay network. [19]

In order to synchronize all services, Flannel requires the `etcd` KV-store, which is also created as a component of CoreOS and stores essential information such as the mappings between virtual container IP addresses and host addresses and the user-defined subnet. Another important configuration that has to be made in the KV-store is the VXLAN forwarding via Open vSwitch. Otherwise Flannel uses UDP tunneling which dramatically reduces network performance [2]. The configuration used during this project can be seen in listing 3.1.

```
$ etcdctl set /coreos.com/network/config '{"Network":"10.100.0.0/16", "Backend": {"Type":"vxlan"}}'
```

Listing 3.1: Flannel configuration defining the overlay subnet and encapsulation type

Once a host joins the overlay network Flannel automatically assigns a randomly picked subnet (`/24` by default) to it from which Docker can allocate IP addresses to the individual containers. In order to assign the picked subnet to a host, Flannel attempts to update the overlay network configuration in the `etcd` store. If the subnet is already assigned to another host, the update will fail and flannel will pick another subnet. If the subnet is still available, Flannel acquires a 24-hour lease which will be extended one hour before expiration. In order to route packets correctly, Flannel keeps track of all entries in the `/coreos.com/network/subnets` location of the `etcd` store and uses this information to maintain its own routing table. In the future, Flannel is planning to support authentication and encryption of IP packets with `IPsec`. [19]

3.3 VXLAN

Even though all the discussed Docker overlay networks are using different approaches to connect Docker containers across multiple hosts, they all make use of the overlay encapsulation protocol *Virtual Extensible LAN* (VXLAN). This protocol is of great importance for the performance of the discussed overlay networks. VXLAN extends the data link layer (OSI Layer 2) by encapsulating MAC-based layer 2 Ethernet frames with network layer (OSI Layer 4) UDP packets. The resulting Layer 2 overlay network is called `VXLAN segment`. Each of these segments is uniquely identified

by a 24-bit *VXLAN Network Identifier* (VNI). Therefore, VXLAN can support up to 16 million VXLAN segment within the same administrative domain. These segments are isolated, meaning that only VMs joined by the same segment can communicate with each other. [10]

VXLAN connects to VMs via *VXLAN Tunnel Endpoints* (VTEP) that are located within the hypervisor that houses the VM. Thus, the VM is unaware of the VXLAN tunneling. When VXLAN is used to connect Docker containers instead of VMs, the VTEPs are located within the Docker Engine. VTEPs can be uniquely identified by the combination of their MAC address and VNI. [10]

When a VM wants to communicate with another VM on a different host, it sends a MAC frame addressed to the target VM. The VTEP on the source host intercepts this frame and verifies that the target VM is on the same VXLAN segment as the source VM by checking the VMI. VXLAN uses multicast to discover unknown endpoints by flooding `unknown destination` frames. If the frame is allowed to travel to its destination, it is encapsulated into a UDP packet. The additional headers of such an encapsulated packet can be seen in figure 6. The target end point learns the mapping of the inner MAC source address to the outer IP source address. This mapping is stored in a table to avoid `unknown destination` flooding, when sending a response packet back to the source VM. [10]

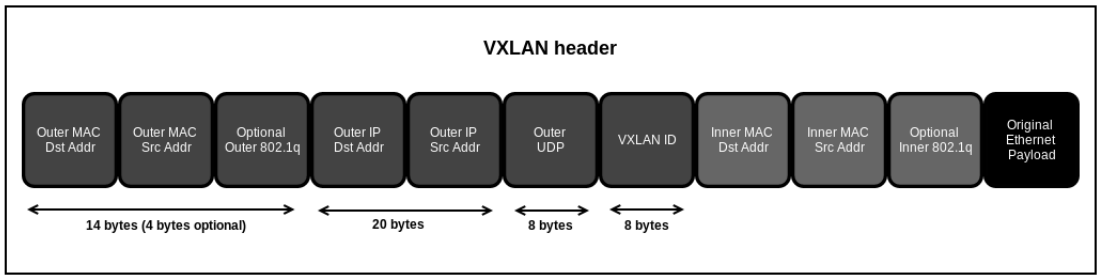


Figure 6: Illustration of the VXLAN header. Encapsulating a regular Ethernet packet adds 50 bytes. [9]

The UDP encapsulation adds 50 extra bytes to the frame. Therefore, it is advisable to configure the MTU of the VMs interface in such a way, that the size of the sent frame is 50 bytes less than the MTU of the underlying network, in order to avoid fragmentation. Furthermore, the encapsulation increases the packet processing workload significantly. It adds 21% of CPU cycles to the processing of a MTU-sizes packet, resulting in a throughput penalty of 31.8% and a latency increase of 32.5%. Some network interface controllers (NIC) can take over some of that workload which relieves the CPU and therefore improves the overlay network's performance. [16]

3.4 GÉANT TESTBED SERVICES

GÉANT is the European research and education network which spans across whole Europe. The GÉANT Testbed Service (GTS) connects five sites of this network and offers the ability to create geographically dispersed experimental networks at scale in an environment that is more controlled than the internet. This service allows experimenters to define their own realistic and flexible network topologies. A high-level overview of the architecture can be seen in figure 7 on the next page. The sites are connected by dedicated point-to-point circuits 10Gbps optical waves. Each of this sites consists of several physical machines which allocate VMs when they are needed for a testbed and deallocates them as soon as the testbed is released. The testbed communication travels via the *data plane infrastructure*. The *control & management network* is used for maintenance

as installing software or various kinds of configurations. A *Internet Access Gateway* (IAGW) offers experimenters the possibility to load their own software from external servers. [14]

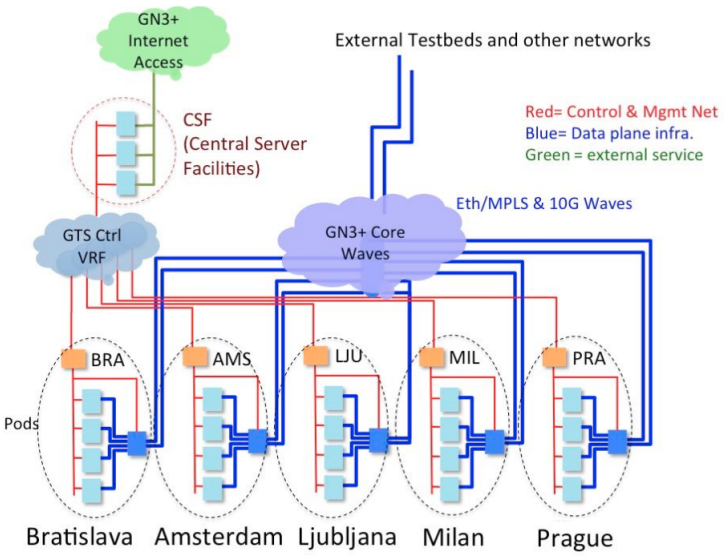


Figure 7: Architecture of the GÉANT Testbed Services. [14]

The GTS' own *Domain-specific language* (DSL) is used to describe how resources, such as VMs, virtual circuits, OpenFlow switches and external domains should be composed to create a user-defined network. Furthermore, the DSL can also be used to specify certain parameters, such as the VM's disk size, the link's capacity or the directionality of ports. However, not all of the features described in the GTS manual seem to be fully functional.

Once a topology is specified, the resources are allocated dynamically. The virtual hosts are running on Kernel-based Virtual Machine (KVM) and are deployed with Ubuntu 14.04 LTS and kernel version 3.13. *PCI passthrough* allows direct and full access to the physical host's network interface controller (NIC).

The physical host on which a VM is allocated cannot be specified. Thus, two VMs on the same location are not guaranteed to be running on the same physical host. Yet, according to the GTS team, the VM's network performance is guaranteed to be the same for all physical machines. Even though Testbeds are isolated by design, the bandwidth of the data plane infrastructure is shared, and might interfere, with other users. [14]

The research of Hermans & de Niet [3] showed that there were no day-night cycles within the GTS. This finding, combined with the small amount of jitter they detected, led them to the conclusion that there is little to no concurrent use of the GTS taking place.

During this project, the GTS was updated to version 3.1. The new version mainly introduces improvements of security and usability. HTTP communication to the GTS services is now protected through TLS, passwords are encrypted with `bcrypt` and a maintenance flag in the GUI warns the user about ongoing maintenance and restricted user access. Furthermore, the performance for the reservation and activation of resources and the testbed view in the GUI were sped up. [8]

4 METHODOLOGY

The experiments that are necessary to investigate the two research questions, are conducted in the GTS. I used different topologies for each research question. These topologies are defined in a JSON-like DSL format. An example of such a DSL specification that defines a simple point-to-point topology between Amsterdam and Bratislava can be seen in listing 4.1. The entire DSLs are listed in appendix A on page 33.

```
AMS_BRA {
  description="AMS-BRA"
  id="AMS_BRA"

  host {
    id="vmAMS"
    location="AMS"
    port {id="port"}
  }
  host {
    id="vmBRA"
    location="BRA"
    port {id="port"}
  }

  link {
    id="vmLink"
    port {id="src"}
    port {id="dst"}
  }

  adjacency vmAMS.port, vmLink.src
  adjacency vmBRA.port, vmLink.dst
}
```

Listing 4.1: DSL definition of a point-to-point topology

GTS topologies are dynamically provisioned but remain static throughout their reservation which means that they cannot be modified. Instead, a new topology has to be created and the VMs need to be configured anew.

All VMs are provided with an `eth0` interface by default. This interface is connected to the *control & maintenance network* through which the VMs can be managed. It should not be used for testing as it is not related to the topology that was created by the DSL. The interface to the *data plane infrastructure*, which was defined by the `port` property of the `host` object in the DSL (see listing 4.1) needs to be configured manually. At the time of writing, this configuration cannot be done by the GTS itself because it does not support port mapping. Future versions will allow this configuration from within the GTS GUI [14]. Listing 4.2 on the next page shows how a new `eth1` interface with IP address `10.10.10.10` is configured. Other VMs within the topology need to be configured with an IP address in the same range. Thus, their address needs to be `10.10.10.x` where `x` can be any number between 1 and 254 inclusive. Writing the configuration to `/etc/network/interfaces` makes the changes permanent and guarantees that they are still active after rebooting the VM.

```
$ ifconfig eth1 up 10.10.10.10 netmask 255.255.255.0

# add the following lines to /etc/network/interfaces
auto eth1
iface eth1 inet static
address 10.10.10.10
netmask 255.255.255.0

$ ifconfig eth1 down
$ ifconfig eth1 up
```

Listing 4.2: Configuration of an `eth1` interface with IP address `10.10.10.10`

4.1 DEPLOYMENT CONSIDERATIONS

During their research, Hermans & de Niet created a GitHub repository [1] which includes scripts that install Docker and an optional third-party overlay network. Furthermore it configures the Flannel overlay to use VXLAN, which is not used by default. As Docker's native overlay driver requires a kernel version of 3.16 or higher, the script also updates the VMs kernel from version 3.13 to 3.19. Since my research is based on their work, I used these scripts to setup the test environment.

The configuration of a `libnetwork` overlay network is not done by the bootstrap script. This can easily be set up manually. Just like Flannel, `libnetwork` also requires a KV-store to store information about the network state. Since Flannel specifically requires an `etcd` KV-store, this KV-store is also chosen for `libnetwork` in order to maintain a consistent environment. A simple script that starts an `etcd` KV-store for `libnetwork` can be seen in listing 4.3. It configures the KV-store to manage a cluster of two hosts. In production mode, this configuration should be placed in a separate file to facilitate maintenance and increase flexibility.

```
#!/bin/bash
./etcd --name infra1 \
  --initial-advertise-peer-urls http://10.10.10.20:2380 \
  --listen-peer-urls http://10.10.10.20:2380 \
  --listen-client-urls http://10.10.10.20:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.10.10.20:2379 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=http://10.10.10.10:2380,infra1=http://10.10.10.20:2380 \
  --initial-cluster-state new
```

Listing 4.3: Starting an `etcd` KV-store that manages a cluster of two hosts

Weave does not require a separate KV-store because it implements its own mechanism of propagating the addresses of participating nodes. Launched Weave routers only need to be pointed to the address of at least one other Weave router to join the overlay network and will automatically learn about the other nodes on the network.

[1] The repository is available at https://github.com/siemhermans/gtsperf

4.2 MEASUREMENT TOOLS

To examine the performance of the overlay networks, I createad a custom `Dockerfile` which can be seen in appendix B on page 36. This file specifies a Docker container which is so generic that it can be used for all conducted experiments. It is based on the Ubuntu 14.04 image and has the necessary benchmarking tools `iperf3` and `netperf` installed. Furthermore it contains the Python programming language as well as its libraries `numpy` and `matplotlib` which are used for processing and plotting the results of the experiments. Moreover, the `ethtool` utility used for modifying the settings of the NIC is installed, the Python scripts for the different experiments are copied into the container and the necessary ports are exposed.

In addition, I used the `stress-ng` tool, which offers a variety of different ways to stress the CPU, to generate additional CPU workload. The Docker container is built and deployed as shown in listing 4.4.

```
$ cd <path to Dockerfile>
$ docker build -t gtsperf .
$ docker run -id --name <name> -v /home/gts/result:/root/result --privileged --net=<network> gtsperf
```

Listing 4.4: Creating a Docker container for performance measurements

The `docker build` command downloads the Ubuntu image and adds additional layers to create a Docker image named `gtsperf`. An instance of this image is then launched as a Docker container by the `docker run` command. In order to move data between the VM and the container, a `result` directory on the VM is shared with the container. All data within this directory can be accessed by the VM and the container. The `--privileged` option grants special permissions to the container. This is essential for the `ethtool` utility to run correctly. If the container should be connected to any other network than the `docker0` bridge, this has to be specified by the `--net` option. An exception is the Flannel overlay network to which the container connects automatically, when launched. To gain a better insight into the VM's resource usage, `cAdvisor` [2] is used. This tool launches a Docker container which collects and visualizes information about running containers. It reports the resource usage per container as well as the overall resource usage on the host.

4.3 EXPERIMENT DESIGN

Since the two research questions require different topologies, a dedicated testbed was used for each question. Previous experiments by Hermans & de Niet showed that there is no day-night-cycle on the GTS [3]. Therefore, no attention is given to the time of the day, during which experiments are run. However, in order to obtain authentic results, only one test was run at a time to avoid falsified results that could be caused by simultaneous test runs. All measurements of `iperf3` and `netperf` are run for two minutes. These measurements are repeated 20 times to balance outliers. All experiments consist of a set of measurements with different values for a certain parameter, to examine how this parameter affects performance.

For each experiment I wrote a dedicated Python script which performs the measurements and writes the results of each of the 20 iterations to disk. At the end of the experiments, this data is plotted as bar charts visualizing the mean values and standard deviation of the results. The fact that the data is written to disk before being plotted makes it easier to adjust the resulting plots. Furthermore, this approach also allows to examine the obtained results in more detail. Since these scripts are running up to 10 hours, they are writing their progress to a log file which helps to monitor their execution.

[2] `cAdvisor` is available at https://github.com/google/cadvisor

Due to the similar structure of the experiments, the dedicated scripts are quite similar. They are all repeating the experiment 20 times, are calling `iperf` or `netperf` with different parameters and are reading the obtained data to plot the results. For the sake of usability, these similar scripts are still kept self contained and are not merged to a single generic script which could perform *all* experiments. Such a script would be a lot more difficult to maintain, harder to use and be more error prone. Since the exact implementation of these scripts is not relevant to this research, the code is not added to the appendix but is made available in a GitHub repository [3] forked from Hermans & de Niet.

4.3.1 Experiment 1: UDP and TCP throughput

In order to examine the reasons for the discrepancy between the low UDP throughput and high TCP throughput, a topology of four separate point-to-point connections was created (see figure 8). The VMs on the first connection do not host any Docker containers. The VMs on the remaining three connections are equipped with Docker containers that are connected via `libnetwork`, Flannel and Weave, respectively. Since the performance of the GTS connections between different locations varies, all four point-to-point connections are between the same locations. In order to make the results comparable to the UDP and TCP throughput results that Hermans & de Niet presented in figure 1 on page 3, the topology connects the locations Amsterdam and Bratislava as well.

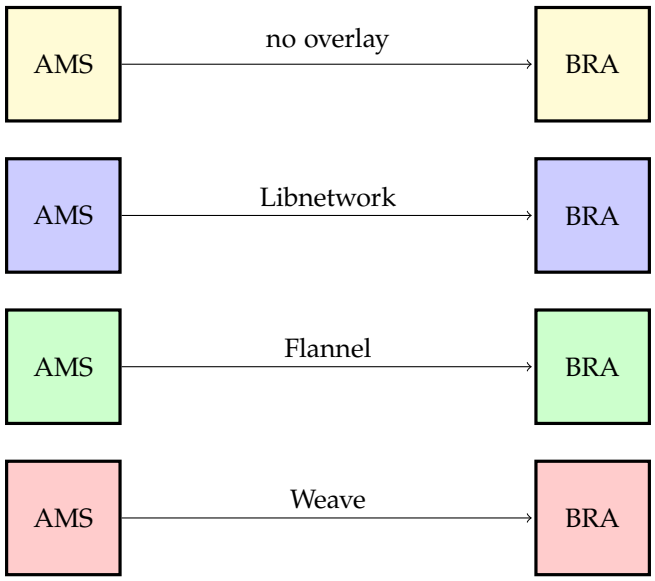


Figure 8: Point-to-point GTS topology for comparing overlay networks and native VM communication. Each overlay network connects a container from Amsterdam to a container in Bratislava.

Correlation between throughput and CPU usage

Claassen [1] and Hermans & de Niet [3] hypothesized that the low UDP throughput observed in the overlay networks is caused by the CPU intensive jitter measurement performed by `iperf`. This presumption is investigated by a series of experiments that examine the correlation between throughput and CPU usage. First of all, UDP and TCP throughput measurements were run on all connections, while monitoring the CPU usage with `cadvisor`. This experiment clarifies in which case the throughput is limited by CPU capacity. The `iperf3` commands used for these measurements can be seen in listing 4.5 on the next page. These are the basic measurements without any special flags.

[3] The forked repository is available at `https://github.com/hopfenzapfen/gtsperf`

```
# measuring UDP throughput
$ iperf3 -u -c <iperf3-server address> -b 1000M -t 120

# measuring TCP throughput
$ iperf3 -c <iperf3-server address> -t 120
```

Listing 4.5: Basic `iperf3` measurements without any special parameters specified

The next experiment examines how UDP and TCP throughput is affected by limited CPU capacity. The first idea was to create a second version of the existing topology in which a lower CPU capacity is assigned to the hosts. However, even though the option of specifying a hosts CPU capacity in the DSL is mentioned in the GTS manual [14], it did not have any effect on the allocated VMs. Instead, `stress-ng` was chosen to generate additional CPU workloads of different intensities. This tool offers a wide range of options to control which resources should be stressed in what manner. One of these options controls the percentage of the CPU capacity that should be occupied. This option turned out to be of little use, because it did not create any additional workload when `iperf3` was occupying 100% of the CPU's capacity. Apparently, `stress-ng` found that the CPU is already busy enough and does not need to be stressed more. Since there is no other way to control the intensity of the CPU stressing, the final solution to this problem was to run `stress-ng` simultaneously with `iperf3` for different portions of the total measuring time. This will reveal the effect that different amounts of CPU cycles available to `iperf3` will have on throughput. The `iperf3` commands used during this experiment are the same as those of the previous experiment, shown in listing 4.5.

Impact of fragmentation on throughput

A CPU intensive task of UDP transmission is the fragmentation of packets which occurs when the total datagram size is higher than the interface's *maximum transmission unit* (MTU). In such a case the packet needs to be split up into smaller packets which can be passed to the interface. The MTU of a standard Ethernet interface is 1500. The MTUs for the Docker overlay networks are smaller due to the VXLAN encapsulation which adds 50 byte to the packet header. Therefore the MTU of a `libnetwork` and Flannel interface is 1450 bytes and the MTU of a Weave interfaces is even smaller with just 1410 bytes. This experiment runs the `iperf3` command shown in listing 4.6 with different datagram sizes in the range from 1000 bytes to 2000 bytes with a step size of 100 bytes. This range was chosen to examine the throughput of traffic with datagram sizes around the MTU. Since the fragmentation is relatively CPU intensive and the CPU capacity seems to be the limiting factor of the throughput, it is expected to drop as soon as a datagram is exceeding the interface's MTU.

```
$ iperf3 -u -c <iperf3-server address> -b 1000M -t 120 -l <datagram size>
```

Listing 4.6: `iperf3` measurements with different datagram sizes specified by the `-l` flag

Impact of UDP and TCP offloading

Similar to UDP, TCP also splits data into chunks that are encapsulated in packets. This process is called TCP segmentation. As mentioned in section 3.3 on page 9, some NICs have the ability to take over CPU intensive tasks such as UDP fragmentation and TCP segmentation. The `ethtool` utility is able to view the NIC's offloading capabilities and can enable or disable certain kinds of offloading, if they are supported by the NIC. Since UDP and TCP offloading relieves the CPU, this optimization could have a positive impact on the throughput. This experiment examines this hypothesis by running `iperf3` with and without offloading.

For UDP offloading, the impact is expected to become more significant as the degree of UDP datagram fragmentation increases. Therefore, UDP traffic with different datagram sizes between $\approx$ 1400 and $\approx$ 10000 bytes is generated. Due to the different MTUs of their interfaces, the VM and the different overlay networks have different fragmentation thresholds. In order to allow a fair comparison of the throughputs, the step size by which the datagram size is increased during the experiment depends on the interface's MTU. Measurements are always taken for the next larges datagram size before the fragmentation threshold.

The `ethtool` commands that are used to control TCP segmentation offloading and UDP fragmentation offloading is shown in listing 4.7. Modifying the offloading settings of the NIC within Docker demands special permission which requires Docker to run in `privileged` mode.

```
# TCP segmentation offload
$ ethtool -K <interface> gso <on/off> tso <on/off>

# UDP fragmentation offload
$ ethtool -K <interface> ufo <on/off>
```

Listing 4.7: `ethtool` commands used to control UDP and TCP offloading

4.3.2 Experiment 2: Weave's multi-hop routing

The second research question is about the performance of Weave's hop-routing feature with respect to throughput and latency. The topology used for this test is shown in figure 9 on the following page. It consists of six geographically dispersed hosts which are connected by five network segments. Each host is merely connected to its immediate neighbors. On the GTS level, each network segment is a separate subnet. It is important to note that the VMs in this topology can only contact their neighboring VMs. To illustrate: The VM in Bratislava cannot transmit data to the VM in Milan because they are not directly connected. However, when creating a Weave overlay network by launching Weave routers on each VM, the attached Docker containers can contact *all* containers connected to the Weave overlay network. Weave routers automatically propagate the network topologies and learn about the addresses of other participating nodes. That is why Weave is capable of routing a packet across the overlay network to its destination.

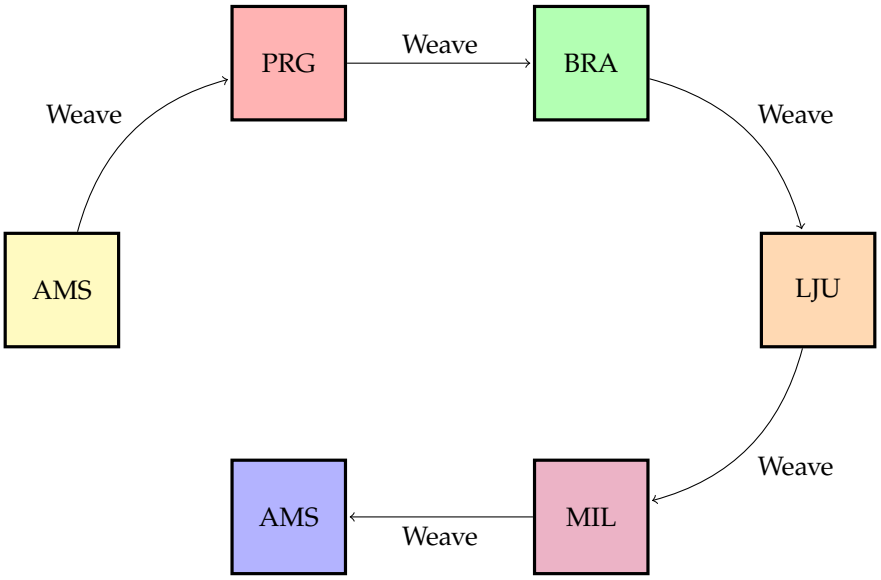


Figure 9: Linear GTS topology for examining Weave's hop-routing. Containers on each location are connected to their immediate neighbors by a Weave overlay network.

The performance tests examines two aspects of the overlay network: Latency and throughput. The latency experiment makes use of `netperf`, which is used to measure the latency between the Docker container on the first host and the Docker containers on the other hosts. Thus, an instance of the `netperf` client is running on the first Docker container whereas all other Docker containers are running an instance of the `netperf` server. This results in the latencies for 1 - 5 hops. In order to bring these results into perspective, they are compared to the cumulative sum of the single latencies per network segment. For example, the overall latency between the first Docker container located in Amsterdam and the third Docker container located in Bratislava is compared to the sum of the latencies of the connection between Amsterdam and Prague and the connection between Prague and Bratislava. The difference of the two resulting latencies represents the overhead that was caused by the intermediate router. The `netperf` command that was used to measure the latency is shown in listing 4.8.

```
1 $ netperf -l 120 -H <netperf-server address> -t UDP_RR -- -O mean_latency
```

Listing 4.8: `netperf` command used to measure latency

The UDP and TCP throughput measurements are performed with `iperf3`. The experiment is set up in the same way as the latency experiment. It measures the throughput from the container on the first host to the other five containers.

5 RESULTS

5.1 UDP-TCP TEST RESULTS

The results related to the first research question are presented in this section. The first results examine how the throughput of the VM and the overlay networks is influenced by the hosts CPU capacity. The following results show how CPU-intensive fragmentation affects UDP throughput. Subsequently, the impact of hardware offloading is presented. All experiments were repeated 20 times to balance outliers.

5.1.1 Correlation between throughput and CPU usage

The first experiments examined the correlation between the throughput and CPU usage. It turned out that starting the `iperf3` client immediately raised the CPU usage to 100% on the VM and all Docker containers. This behaviour could be observed for TCP measurements as well as for UDP measurements. A screenshot of the CPU utilization visualized by `cadvisor` can be seen in figure 10. It clearly illustrates how the CPU usage bursts to 100% the moment the `iperf3` client is started. This suggests, that the bottleneck of the throughput is indeed the CPU capacity.

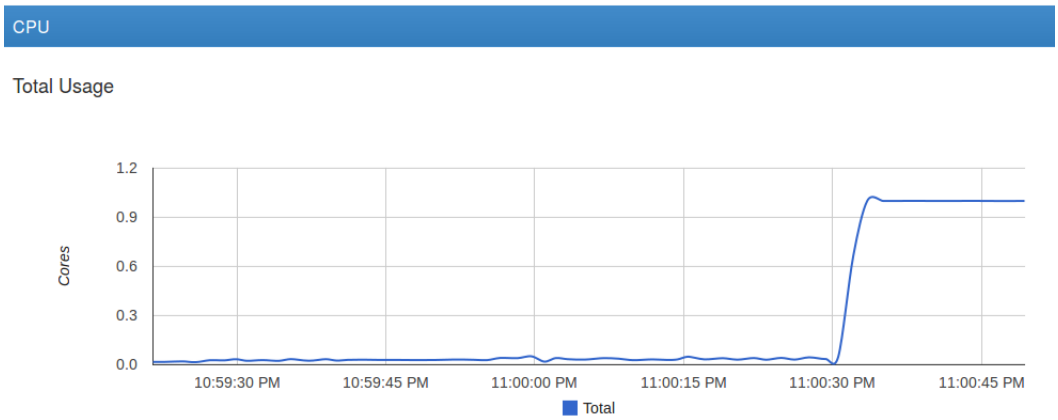


Figure 10: CPU usage shown by `cadvisor` after starting `iperf3`.

The plots in figure 11 on the next page are visualizing the results of the experiments that examined the impact of additional CPU workload on TCP and UDP throughput. As expected, increasing the additional workload on the CPU and thus decreasing the amount of CPU cycles available to `iperf3` reduces the TCP throughput. Furthermore, the obtained throughputs without additional workload are matching the results of Hermans & de Niet in which the TCP throughput of the overlay network outperforms that of the VM.

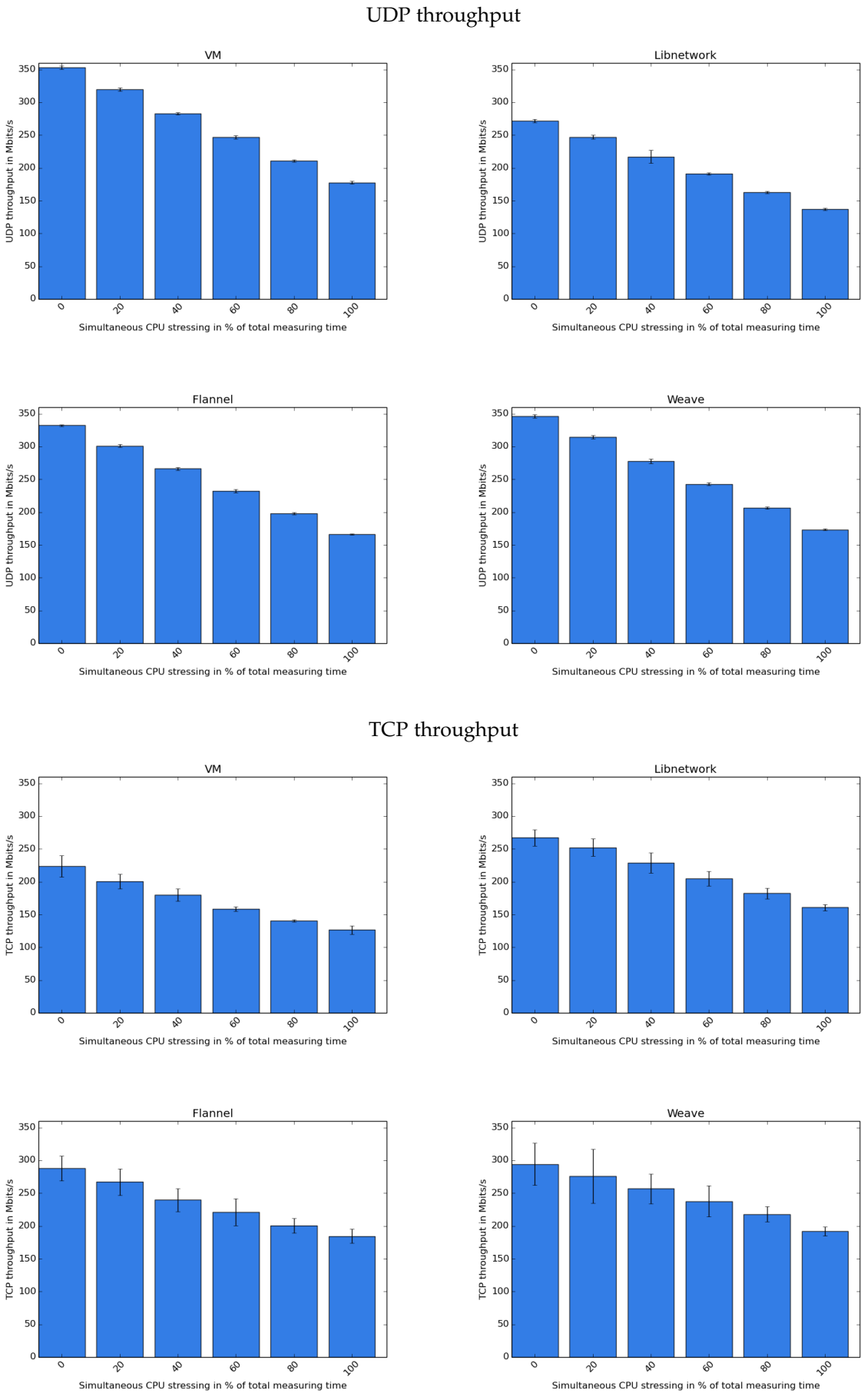


Figure 11: UDP and TCP throughput of measurements with different additional CPU workloads

The UDP throughput shows the same behaviour. Additional CPU workload decreases its throughput. The results for throughputs without additional workload are higher than those that Hermans & de Niet obtained because they used `iperf` whereas all throughput measurements of this project were run with `iperf3`. `iperf` uses a default datagram size of 1470 bytes. The default datagram size of `iperf3` is 128 mega bytes. A significantly larger datagram size seems to increase the UDP throughput.

These results strengthen the presumption that the amount of CPU cycles available to `iperf3` directly impact the throughput. The throughput of both protocols on connections between the VM and all three Docker containers show the same behaviour.

5.1.2 Impact of fragmentation

The impact of datagram fragmentation on throughput can be clearly seen in figure 12. As soon as the datagram's size exceeds the network's MTU size, the throughput drops by $\approx 30\%$ due to fragmentation. The process of splitting up the data and encapsulating the resulting fragments into datagrams is so CPU intensive, that it dramatically reduces the throughput. The fact that the threshold for fragmentation depends on the MTU becomes more clear when comparing the result of the Weave overlay to the other three plots. As mentioned earlier, the MTU of the Weave interface is the lowest of all overlays with merely 1410 bytes. Therefore, a datagram of 1400 bytes encapsulated in a VXLAN packet exceeds the MTU and needs to be fragmented. The VM, `libnetwork` and Flannel can transmit 1400 byte datagrams without fragmentation since their MTU is larger.

Furthermore, the results indicate that as long as no fragmentation occurs, a larger datagram size provide higher throughput. Since the CPU overhead per transmitted datagram is approximately the same, independently of its size, sending a full datagram is inherently more efficient than sending a datagram whose capacity is not fully exploited. The results of the next experiment show how a wider range of datagram sizes affects the throughput.

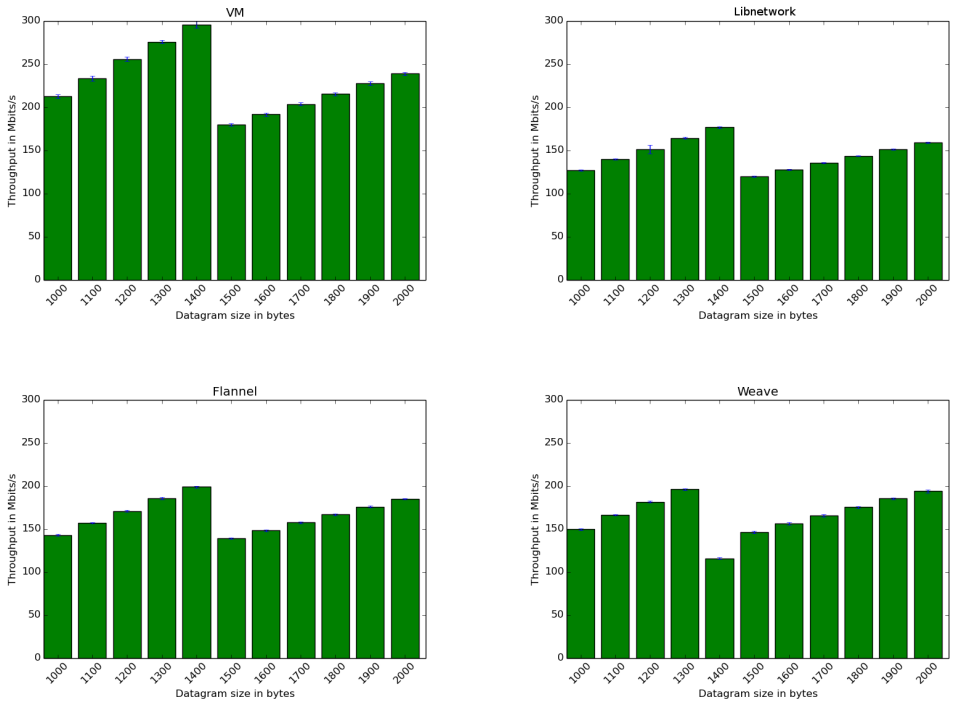


Figure 12: UDP throughput of measurements with datagram sizes between 1000 and 2000 bytes.

5.1.3 Impact of offloading

During the experiment it turned out that the VMs on the GTS support neither TCP segmentation offloading nor UDP fragmentation offloading. `ethtool` is capable of listing which kinds of offloading are enabled on the VM but is not able to modify the NIC settings. UDP and TCP offloading are both disabled. Surprisingly, offloading is enabled by default on the Docker containers that are running on those VMs. Thus, eventhough the VMs themself do not support offloading, the Docker containers do support it. Furthermore, `ethtool` is capable of modifying these NIC settings from within a Docker container.

The results of the UDP throughput can be seen in figure 13. A comparison of the throughput of the VM and overlays without offloading reveals the overhead caused by the VXLAN encapsulation. These results coincide with the findings of previous research that concluded that VXLAN encapsulation decreases the throughput by up to 31.8% [16]. A portion of this penalty can be absorbed by UDP offloading which has a positive impact on throughput. As expected, the impact of offloading increases as the degree of fragmentation grows. However, even for large datagram sizes, the VM throughput still outperforms the throughput of the overlay networks.

Another interesting result is the fact that an increase of datagram size on a larger scale increases throughput, even though the degree of fragmentation increases as well. This also explains the reason why UDP measurements with `iperf3` measured higher throughputs than the `iperf` measurements performed by Hermans & de Niet.

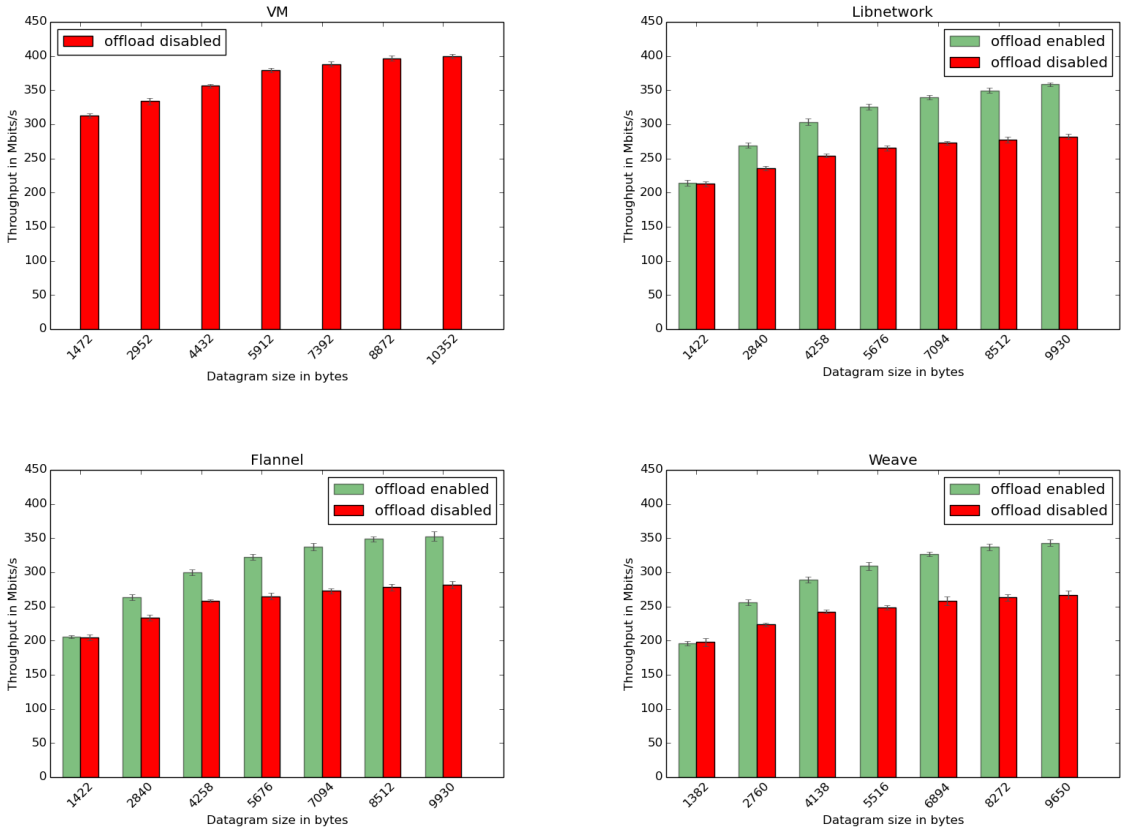


Figure 13: UDP throughput of measurements with increasing datagram sizes with and without UDP fragmentation offloading.

As seen in figure 14, the TCP throughput also suffers from the VXLAN encapsulation overhead. The throughput of the overlay networks without segmentation offloading is significantly lower than the throughput of the VM. These results are similar to those of the UDP results as the throughput also decreases by ≈ 30%. However, the positive effect of offloading appears to have a much greater impact on TCP throughput. The results show that this optimization almost doubles TCP throughput and therefore outperforms the VM's throughput by far.

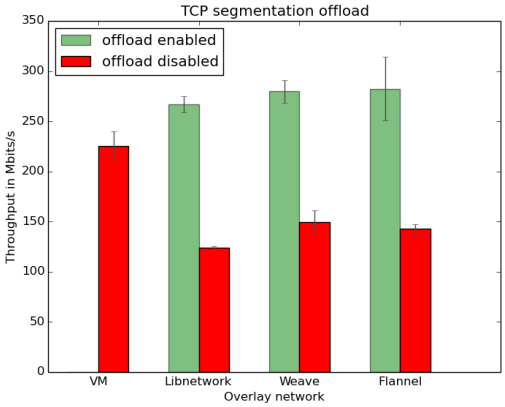


Figure 14: TCP throughput of measurements with and without TCP segmentation offloading.

5.2 WEAVE'S MULTI-HOP ROUTING TEST RESULTS

The configuration of a linear Weave overlay network in which Docker containers communicate via multiple hops proved to be fairly simple. Weave routers merely had to be provided with the addresses of their immediate neighbours and were directly able to communicate with Docker containers that were multiple hops away.

Furthermore, as seen in figure 15, no additional latencies are introduced when routing data across multiple hops. For each number of hops the overall latency is equal to the cumulative sum of the single latencies measured per network segment. The figure also indicates that there is no variance of latency.

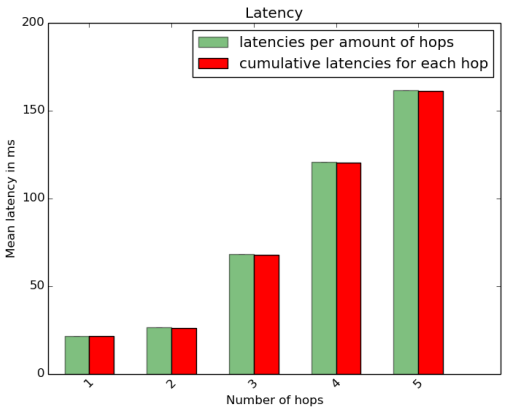


Figure 15: Latency across multiple hops

During the throughput measurements it turned out that some of the Weave routers occasionally stopped using the Fast Datapath optimization, illustrated in figure 5 on page 9, and fell back to `sleeve` mode which greatly reduced the throughput. This problem has never occurred during earlier experiments. According to the Weave developers, Fast Datapath is always used by default and can only be disabled when a Weave router is launched with the according flag. Since the only way to reenable Fast Datapath was to relaunch the Weave router, it was not able to measure the throughput with the dedicated Python script. Instead, the measurements were conducted by manually executing the `iperf3` commands five times inside the Docker containers and restarting Weave routers if necessary. Due to time limitations, these measurements were not repeated 20 times, like all the other experiments of this project.

The obtained results are shown in figure 16. TCP throughput unexpectedly drops significantly after the first hop and then slightly decreases with each additional hop. This result is especially surprising when comparing it to the results of UDP throughput which barely suffers from being routed across multiple hops. The reasons for this unanticipated behaviour would be an interesting topic for further research.

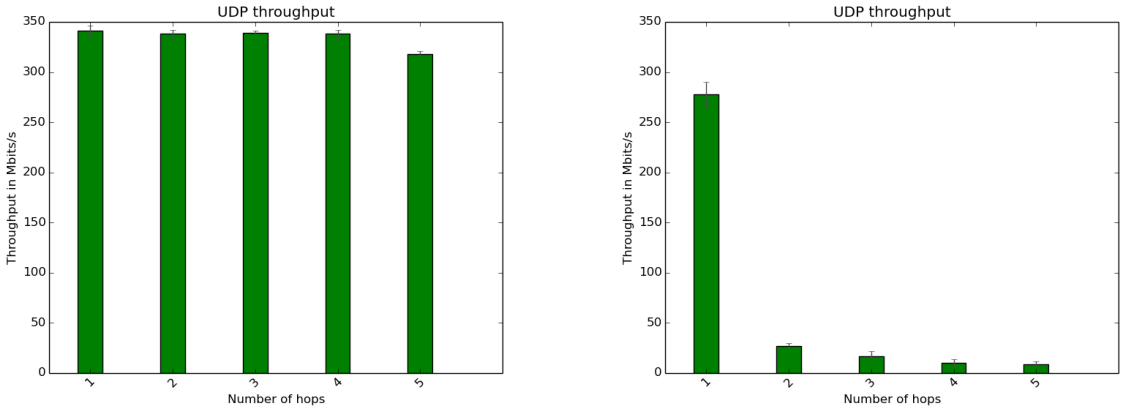


Figure 16: UDP and TCP throughput across multiple hops.

6 DISCUSSION

6.1 OVERLAY USABILITY

During the setup and execution of the experiments with the different overlay solutions I had the chance to evaluate the usability of the different overlay solutions. I found that Weave is the easiest to install and maintain. The fact that Weave does not require the installation of a separate KV-store to share the network's state makes the deployment very straightforward. Since Weave independently propagates changes of the network topology to participating nodes, the overlay network is very flexible and easy to manage since new routers just need to be connected to *one* other router to learn about the whole network topology. The multi-hop-routing feature allows communication between Weave routers that are not directly connected. In addition, out of the three examined overlay solutions, Weave is the only one that supports encryption.

Since `libnetwork` is built-in to Docker, it does not need to be installed separately. In contrast to the other two overlay networks, it is able to accommodate multiple overlay networks and local networks that coexist on a single hosts.

Flannel requires the installation of the Flannel daemon `and` a KV-store and is therefore more complex to set up than `libnetwork` and Weave. Another disadvantage is the fact that VXLAN needs to be activated explicitly which is necessary to reach the performance of the other two network overlays. Furthermore, Docker containers automatically connect to the Flannel overlay network when started. This proved to be useful when setting up test environment but could cause problems when isolation of certain containers is important.

In essence, `libnetwork` seems to be useful for smaller and rather static topologies in which the network state rarely needs to be reconfigured. This solution is also the right choice for topologies that require multiple isolated overlay networks. Weave is practical for more complex topologies which are changing regularly, since Weave can propagate the current network state automatically, without needing to be reconfigured. Since Flannel is more complex to setup and does not offer any features that are not covered by `libnetwork` or Weave, I cannot think of any application for which the Flannel overlay network would be more suited than the other solutions.

6.2 UDP-TCP TEST

The results of the UDP-TCP experiments showed that, in all cases, the limiting factor of throughput is CPU capacity. Datagram size turned out to have a significant impact on throughput. Datagram sizes slightly beyond the fragmentation threshold achieve a lower throughput than datagram sizes that are slightly smaller. However, increasing datagram size on a larger scale turned out to *increase* throughput rates. This is especially true if fragmentation offloading is possible since it relieves the CPU.

Additionally to the fragmentation overhead, previous research showed that VXLAN encapsulation introduces a dramatic throughput penalty of more than 30%. This affects both UDP and TCP transmissions and can be clearly seen in the results when comparing the different throughput measurements that were taken without offloading. This phenomenon explains why the UDP throughput measured by Hermans & de Niet is so much lower for the overlay networks than for the VM. Their measurements also showed that the overlay networks outperform the VM with respect to TCP throughput, even though the results of this project clearly indicate that TCP suffers from VXLAN encapsulation as well. As the offload experiments revealed, the high TCP throughput is achieved by offloading workload to the NIC. The results show that offloading has

a greater impact on TCP throughput than on UDP thoughput. TCP throughput benefits from this optimization to such a great extent that it outweighs the penalty of VXLAN encapsulation and even outperforms the VMs throughput.

However, the experiments also showed that the NIC's offloading settings can merely be controlled from within the Docker container but not from the VM the container is running on. This issue was also presented to the GTS support but no explanation for this behaviour was found. It would be interesting to investigate this issue in more depth. Conducting the offloading experiments on a different infrastructure could reveal if the issue is GTS specific.

6.3 WEAVE MULTI-HOP ROUTING

Setting up a linear topology of Weave routers proved to be very straight forward. Hop-routing turned out to introduce no additional latency overhead. UDP throughput decreases only slightly when being routed across multiple hops. Yet, TCP throughput drops dramatically after the first hop. This might be due to the fact that the TCP protocol is more complex and requires data to be sent in both directions. Another explanation could be that TCP fails to negotiate the window size across multiple hops which would prevent the window size from being increased in order to scale up throughput. The reson for this result should be investigated in more detail.

Another issue that occurred during the experiment was the occasional fall-back of Weave routers from Fast Datapath to `sleeve` mode. This problem never appeared during the experiments of the first research question. Furthermore, the latency experiments using `netperf` did not cause any routers to fail either. Only the throughput measurements using `iperf3` across multiple hops triggered this problem. Since Fast Datapath is definitely possible on the kernel being used, and proved to be reliable on point-to-point connections, this issue seems to be a bug of the hop-routing feature and would be another topic of future research.

7 CONCLUSION

The advent of Docker increased the popularity of deploying microservices in software containers. The growing utilization of Docker containers increased the need for connecting these containers across multiple hosts which is achieved by overlay networks that abstract the physical networks. In my research, I have examined the performance of the native Docker overlay network and the third-party overlay solutions Weave and Flannel.

First of all, I investigated the reasons for the UDP throughput of all overlay networks being significantly lower than the throughput of a VM. Furthermore, I explored why the TCP throughput of the overlay networks does not show the same behaviour but remains as high as the TCP throughput of a VM. My results indicate that the bottleneck of UDP and TCP throughput on the VM and the overlay networks is the hosts CPU capacity. The VXLAN encapsulation introduces remarkable CPU overhead which reduces both UDP and TCP throughput. On the other hand, the positive effect of hardware offloading, which sources CPU intensive tasks out to the NIC, increasaes throughput but proved to have a greater impact on TCP than on UDP. The combination of these two factors results in a lower throughput for UDP than for TCP of the examined Docker overlay networks. Since these conclusions are based on experiments that were run on the GTS, they are specific to this environment. Especially the capabilities of NIC offloading might vary on different infrastructures and could therefore yield different results.

Secondly, I examined the performance of Weave's multi-hop-routing feature with respect to latency and throughput. The results revealed that this feature does not introduce additional latencies. Also, UDP throughput does not suffer significantly from being routed across multiple Weave routers. However, TCP throughput decreases drastically after the first hop.

7.1 FUTURE WORK

During this research I encountered several unexpected situations and results which require further investigation. I discovered that NIC offloading is only possible from within a Docker container but cannot be activated on the VM it is running on. Thus, even though the physical NIC obviously supports offloading, the VM cannot exploit this optimization. It would be very interesting to find out if this situation is GTS specific. This could be done by repeating the same experiments in different environments such as Amazon's *Elastic Compute Cloud* (EC2).

The results of the second research questions indicate that only TCP throughput suffers from being routed across multiple routers. The reasons for this dramatic decrease after the first hop should also be investigated in more detail. Furthermore, the occasional failing of the Fast Data Path feature could be explained by further research.

Other topics that could expand my research would be a deeper examination of Weave's routing capabilities. It would be particularly interesting to investigate how packets are routed in a topology that provides different paths between two routers. Lastly, once Flannel implements its encryption mechanism, it would be interesting to compare it to Weave's encryption feature with respect to performance and security.

ACKNOWLEDGEMENT

I would like to thank Susanne Naegele-Jackson and Fabio Farina from the GÉANT GTS support group for their rapid and elaborate answers regarding technical questions about the GTS infrastructure. Furthermore, I would like to thank Siem Hermans from the Master of System and Network Engineering at the University of Amsterdam for his helpful advice about the interface configuration of the GTS virtual machines.

BIBLIOGRAPHY

[1] Joris Claassen. Container network solutions. http://rp.delaat.net/2014-2015/p45/report.pdf, 2015. [Online; accessed April-13-2016].

[2] CoreOS. Configuring flannel for container networking. https://coreos.com/flannel/docs/latest/flannel-config.html, 2015. [Online; accessed May-2-2016].

[3] Siem Hermans & Patrick de Niet. Docker overlay networks: Performance analysis in high-latency environments. http://rp.delaat.net/2015-2016/p50/report.pdf, 2016. [Online; accessed April-4-2016].

[4] Docker. Libnetwork design readme. https://github.com/docker/libnetwork/blob/master/docs/design.md, 2015. [Online; accessed April-29-2016].

[5] Docker. Understand docker container networks. https://docs.docker.com/engine/userguide/networking/dockernetworks/, 2015. [Online; accessed April-28-2016].

[6] Docker. Understand the architecture. https://docs.docker.com/engine/understanding-docker/, 2015. [Online; accessed April-27-2016].

[7] Docker. What is docker? https://www.docker.com/what-docker, 2015. [Online; accessed April-27-2016].

[8] GÉANT. Changes in gts by version. https://gts.geant.net/changelog, 2016. [Online; accessed June-8-2016].

[9] The Random Security Guy. Vxlan. http://www.therandomsecurityguy.com/vxlan/, 2014. [Online; accessed April-29-2016].

[10] Dutt et al. Mahalingam. Vxlan: A framework for overlaying virtualized layer 2 networks over layer 3 networks. https://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-09#section-5, 2014. [Online; accessed May-2-2016].

[11] Roberto Morabito, Jimmy Kjallman, and Miika Komu. Hypervisors vs. lightweight virtualization: a performance comparison. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 386–393. IEEE, 2015.

[12] Brent Salisbury. Socketplane excited to be joining docker to collaborate with networking ecosystem. https://blog.docker.com/2015/03/socketplane-excited-to-be-joining-docker-to-collaborate-with-networking-ecosystem/, 2015. [Online; accessed April-29-2016].

[13] Mathijs Jeroen Scheepers. Virtualization and containerization of application infrastructure: A comparison. In *21st Twente Student Conference on IT*, pages 1–7, 2014.

[14] Fabio Farina Susanne Naegele-Jackson, Michal Hazlinsky and Nicolai Iliuha. GÉant testbed service (gts) user and resource guide. http://services.geant.net/GTS/Resources/PublishingImages/Pages/Home/User_Guide_v2.0.pdf, 2015. [Online; accessed May-1-2016].

[15] Keith Townsend. Socketplane strives to alleviate docker networking challenges. http://www.techrepublic.com/article/socketplane-strives-to-alleviate-docker-networking-challenges/, 2015. [Online; accessed April-28-2016].

[16] Jagath Weerasinghe and Francois Abel. On the cost of tunnel endpoint processing in overlay virtual networks. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 756–761. IEEE Computer Society, 2014.

[17] Weave works. How weave works. http://docs.weave.works/weave/latest_release/how-it-works.html, 2015. [Online; accessed May-1-2016].

[18] Weave works. Weave networking performance with the new fast data path. https://www.weave.works/weave-docker-networking-performance-fast-data-path/, 2015. [Online; accessed May-1-2016].

[19] Eugene Yakubovich. Introducing flannel: An etcd backed overlay network for containers. https://coreos.com/blog/introducing-rudder/, 2015. [Online; accessed May-1-2016].

LIST OF FIGURES

APPENDICES

A DSL SPECIFICATIONS

```
AMS_BRA {
  description="Four point-to-point connections"
  id="AMS_BRA"

  host {
    id="vmAMS"
    location="AMS"
    port {id="port"}
  }
  host {
    id="vmBRA"
    location="BRA"
    port {id="port"}
  }
  host {
    id="libAMS"
    location="AMS"
    port {id="port"}
  }
  host {
    id="libBRA"
    location="BRA"
    port {id="port"}
  }
  host {
    id="weaveAMS"
    location="AMS"
    port {id="port"}
  }
  host {
    id="weaveBRA"
    location="BRA"
    port {id="port"}
  }
  host {
    id="flanAMS"
    location="AMS"
    port {id="port"}
  }
  host {
    id="flanBRA"
    location="BRA"
    port {id="port"}
  }

  link {
    id="vmLink"
    port {id="src"}
    port {id="dst"}
  }
  link {
    id="libLink"
    port {id="src"}
    port {id="dst"}
  }
  link {
```

```
    id="weaveLink"
    port {id="src"}
    port {id="dst"}
  }
  link {
    id="flanLink"
    port {id="src"}
    port {id="dst"}
  }

  adjacency vmAMS.port, vmLink.src
  adjacency vmBRA.port, vmLink.dst

  adjacency libAMS.port, libLink.src
  adjacency libBRA.port, libLink.dst

  adjacency weaveAMS.port, weaveLink.src
  adjacency weaveBRA.port, weaveLink.dst

  adjacency flanAMS.port, flanLink.src
  adjacency flanBRA.port, flanLink.dst
}
```

```
WeaveHop {
     description = "Test Weave's hop-routing feature"
     id="weave-hop"

     host {
         id="hostA"
         location="AMS"
         port {id="port10"}
     }
     host {
         id="hostB"
         location="PRG"
         port {id="port11"}
         port {id="port12"}
     }
     host {
         id="hostC"
         location="BRA"
         port {id="port13"}
         port {id="port14"}
     }
     host {
         id="hostD"
         location="LJU"
         port {id="port15"}
         port {id="port16"}
     }
     host {
         id="hostE"
         location="MIL"
         port {id="port17"}
         port {id="port18"}
     }
     host {
         id="hostF"
         location="AMS"
         port {id="port19"}
     }

```

```
    link {
        id="AtoB"
        port {id="src"}
        port {id="dst"}
    }
    link {
        id="BtoC"
        port {id="src"}
        port {id="dst"}
    }
    link {
        id="CtoD"
        port {id="src"}
        port {id="dst"}
    }
    link {
        id="DtoE"
        port {id="src"}
        port {id="dst"}
    }
    link {
        id="EtoF"
        port {id="src"}
        port {id="dst"}
    }


    adjacency hostA.port10, AtoB.src
    adjacency hostB.port11, AtoB.dst

    adjacency hostB.port12, BtoC.src
    adjacency hostC.port13, BtoC.dst

    adjacency hostC.port14, CtoD.src
    adjacency hostD.port15, CtoD.dst

    adjacency hostD.port16, DtoE.src
    adjacency hostE.port17, DtoE.dst

    adjacency hostE.port18, EtoF.src
    adjacency hostF.port19, EtoF.dst
}
```

B DOCKERFILE

```
# Docker image to measure effect of UDP datagram size on throughput
# Build command: docker build -t gtsperf .
# docker run -id --name <name> -v /home/gts/Docker/result:/root/result --net=<network> gtsperf
# Install run this container on both hosts.
# Within the container run iperf3  and netperf as server on one host:       iperf3 -sD; netserver
# Within the container on the other run one of the Python scripts:        python perf.py <overlayNetwork > <iperf3 server address>

FROM ubuntu:14.04
MAINTAINER Arne Zismer, <arne.zismer@student.uva.nl>
LABEL role="Docker overlay network performance measurement"

# Set timezone
ENV TZ=CET

# Set correct directory
ENV dir /root
WORKDIR ${dir}

# Update sources & install essential tools and iperf 3
RUN apt-get -qq update && apt-get install -yq   wget build-essential git software-properties-common make nano ethtool
RUN add-apt-repository -y "ppa:patrickdk/general-lucid" && \
  apt-get -qq update && \
  apt-get install -yq iperf3

# install iperf
RUN apt-get install -yq iperf

# install netperf
RUN wget --no-check-certificate ftp://ftp.netperf.org/netperf/netperf-2.7.0.tar.gz && tar -xzvf netperf-2.7.0.tar.gz && \
  cd netperf-2.7.0 && ./configure --enable-demo=yes && make && make install

# install stress-ng and create link to search path
RUN wget http://kernel.ubuntu.com/~cking/tarballs/stress-ng/stress-ng-0.03.11.tar.gz && \
  tar zxf stress-ng-0.03.11.tar.gz && \
  rm stress-ng-0.03.11.tar.gz && \
  cd stress-ng-0.03.11 && \
  make && \
  cd $dir && \
  ln -s  ${dir}/stress-ng-0.03.11/stress-ng /usr/bin/stress-ng

# install python libraries (for testing performance and plotting results)
RUN apt-get install -yq python python-matplotlib python-numpy

# copy test scripts
ADD measureDatagramEffect.py ${dir}/DatagramPerf.py
ADD measureCPUeffect.py ${dir}/CPUperf.py
ADD measureWindowEffect.py ${dir}/WindowPerf.py
ADD measureOffloadEffect.py ${dir}/OffloadPerf.py

# Expose the default ports, statically linked (iperf TCP/UDP, iperf3, netperf)
EXPOSE 5001:5001 5002:5002 5201:5201 12865:12865
```