University of Amsterdam

Master's Programme in System and Network Engineering

MSc Final Research Project

Measuring The Impact of Docker on Network I/O Performance

Author:
Rohprimardho

Supervisor:
Prof. dr. ir. Cees T. A. M. de Laat

21 August 2015

Abstract

This paper investigates whether running applications in Docker would have a significant network I/O performance degradation. The underlying technologies behind Docker are explained and analyzed to look for any possible sources of performance degradation. Multiple measurements were performed in a certain setup while also took into account some options to optimize the performance i.e. CPU spinning and affinity. In the end, results are explained and conclusions are drawn.

Contents

1. Introduction

Docker is an open source platform that simplifies the process of developing, shipping, and running applications. These applications are packaged with all their dependencies into a standardized unit called a container.

These containers run in an isolated way on top of the operating system's kernel. Having this additional layer of abstraction may lead to a performance degradation.

This paper investigates whether running an application in Docker has any impact on the network I/O performance.

1.1. Motivation

High-frequency trading (HFT) is an umbrella term for different automated trading strategies that utilize computers and ultra-low-latency networks.

An HFT firm deploys latency sensitive applications that communicate messages with the system of stock and derivative exchanges. These applications read a stream of UDP multicast datagrams sent by the exchanges and react on some of those datagrams by sending TCP response messages back (so it is: UDP-in, TCP-out). This response messages are TCP messages because the exchanges only accept TCP connection as they prefer reliability for the incoming messages. The reaction time, i.e. the latency between the incoming UDP datagram and the outgoing TCP response message is important to be as low as possible.

To simplify the deployment of these applications, this HFT firm is considering to use Docker. It is important for them to know whether this simplicity has a trade off in terms of network I/O performance degradation.

The attempt to find out whether performance degradation actually exists is the main motivation of this research.

1.2. Research Questions

As mentioned in Section 1.1, the initiative to find out possible network I/O performance degradation is the foundation of this research. It leads to the main research question: how big is the impact of using Docker on the network I/O performance?

A several sub questions are posed to support the main question:

- How to convert an application to a Docker container?
- What kind of test scenarios can be created to accurately measure the performance?

- What are the factors that contribute to the performance loss that can be avoided or minimized?

The aspect of the network I/O performance that becomes the focus in this paper is the network latency.

1.3. Related Work

There are papers that compare the network performance between applications running with and without Docker. Eder [1] concludes that the network performance of applications running in Docker is equivalent to the one without Docker. Yet Kratzke [2] and Felter et al. [3] suggest the opposite, that using Docker containers affect the general network performance. These three papers focus on different aspect of network performance by using various measurement methods and tools.

Eder uses industry standard *netperf*[1] benchmark tool to measure the round-trip latency between two Docker containers, each running on different physical machine. Eder also uses kernel-bypass technology that lets applications and network drivers run together in the user space and therefore bypass the kernel space [4]. The result shows that there are no significant difference in performance between running applications with or without Docker. The average round-trip latency is about 4.5 µs as shown in Figure 1.1

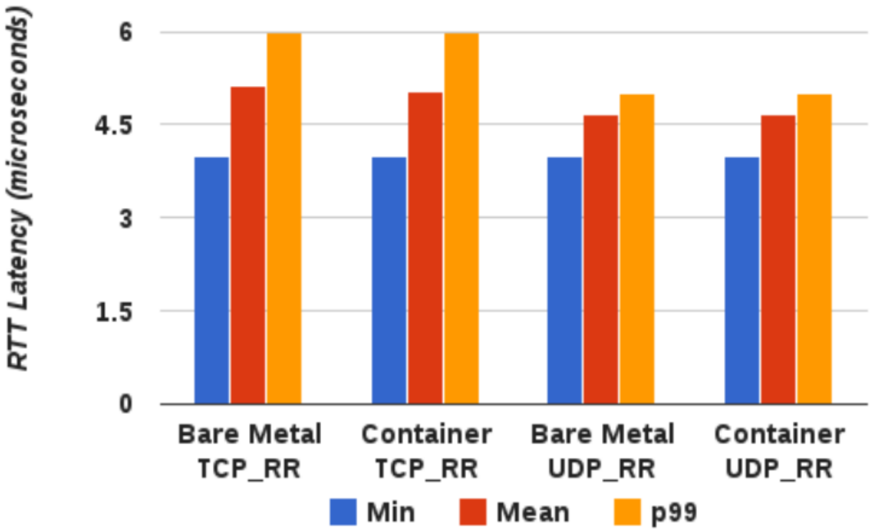

Figure 1.1.: Round-trip latency measured by Eder [1]. Bare Metal indicates the measurement without Docker and Container indicates the measurement with Docker

[1] http://netperf.org

The work of Felter et al has a broader focus. They measured several aspect of network performance and one of them is the network latency (the rest are measuring network bandwidth, memory bandwidth, block I/O, and some other tests), which was also measured by using *netperf*. The result shows that the latency is doubled when Docker is used as shown in Figure 1.2.

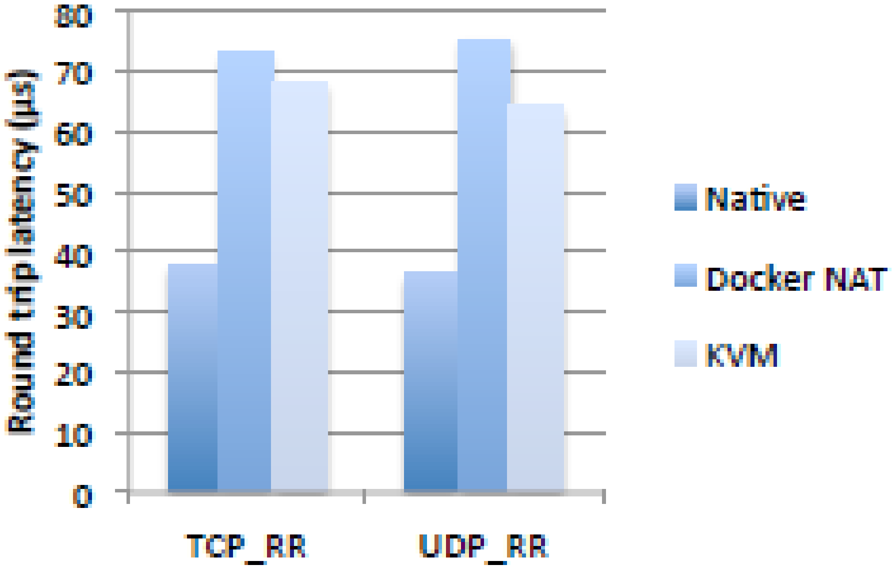

Figure 1.2.: Round-trip latency measurement result by Felter et al [3] that shows that native latency is half of the Docker NAT

Kratzke focuses on the data transfer rate instead of the network latency by using *apachebench*[2] as the measurement tool. The result shows that for messages smaller than 100 KB there is an 80% performance reduction which goes to 90% for messages bigger than 100 KB. Even though it is not measuring network latency, it shows in general that using Docker has impact on the performance.

Since Kratzke focuses on different aspect, a direct comparison can only be drawn from the result of Felter et al. and Eder. Both measure the network latency with the same measurement tool. Kernel-bypass is the major difference between these two papers, which explains the difference on their results.

From these researches, we might conclude that kernel-bypass technology could be the factor that made the difference between Eder's result and the others. In general the performance with container is reduced to a certain extent unless an additional (optimization) technique is used, which is in this case the kernel-bypass.

In this paper, the research will be performed by using a tool called *sfnt-pingpong*[3] and with a measurement setup that will use hardware timestamp instead of software timestamp (which is used by *netperf*). Because of time constraint, there are no measurements

[2] `http://httpd.apache.org/docs/2.2/programs/ab.html`

[3] It will be further explained in Section 2

could be performed to test the effectiveness of kernel-bypass.

To sum up, these previous works, despite using different methods and having different focuses, have one thing in common, i.e. comparing network performance of applications running with and without Docker. They give insight into the approach of measuring the network performance.

1.4. Scope

The focus of this paper is the implementation of Docker in an environment and setting up the measurement topology and tools to get a proper result. The underlying technology of Docker will also be explained.

1.5. Contribution

The result of this paper, which shows there is no scientifically significant performance degradation while running in Docker contributes greatly to the field of high performance computing and performance in containers or virtualization in general.

2. Background Information

This section will describe the Linux network stack to understand the network I/O performance that is actually measured in this paper. An introduction about Docker and its underlying technology will also be explained. Moreover, a brief information about hardware timestamps will be explained as well.

2.1. Linux Network Stack

In a high-level view of the Linux network stack, there are seven layers in which each layer has a different job description to allow Linux machines to communicate on the network. These layers are divided into three major sets: user space, kernel space, and physical space (as seen in Figure 2.1) [5].

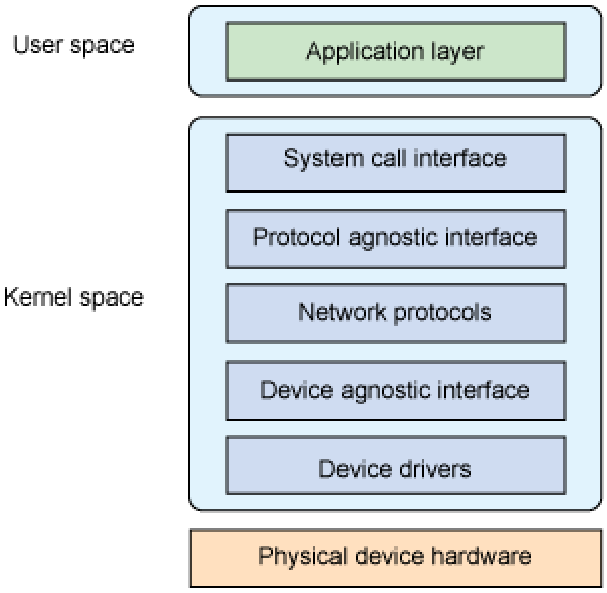

Figure 2.1.: Linux high-level network stack architecture [6]

The focus of this research is on the network I/O performance in terms of the time it takes for a network packet to travel from the physical space (where it enters), to go to the user space through the kernel space and go back to the physical layer. This is an important aspect in a low-latency network environment.

There are various reasons why unnecessary latency (jitter) can occur when network packets travel within a network stack. Some of them are context switching, waiting time for arriving packets to be polled, interrupt, and cache miss. The following subsections will explain these reasons along with the solution on how to minimize the negative impact to achieve latency as low as possible.

2.1.1. Kernel-bypass

As mentioned briefly in Section 1.3, kernel-bypass is a way to let applications and network drivers run together in user space and therefore bypass the kernel space [4]. This would increase the network performance because it is avoiding altogether context switching between user space and kernel space.

Context switching itself describes a condition where the operating system suspends the execution of the process running on the CPU, store the CPU's state in the memory, and resume the execution of some other process previously suspended by retrieving its state from memory and put it in the CPU's register [7][8]. Related to Linux network stack, context switching occurs between user space and kernel space when applications running on user space transmit data to the kernel in the network stack (e.g. to check for new network packets). Context switching also occurs when network adapter notifies CPU for the arrival of incoming network packets. Therefore context switching occurs often and it can affect the performance [9].

The disadvantage of using this technique is to lose the possibility of using network tools that heavily rely on kernel features, such as [10]: `netstat`[1], `ethtool`[2], and `tcpdump`[3].

There are several solutions available that apply kernel-bypass technologies, such as solution offered by *ntop*[4], *netmap*[5], *Intel*[6], *Napatech*[7], and *OpenOnload*[8].

Eder's work [1] uses *OpenOnload*. It uses a combination of a specially designed hardware (network card) and optimized drivers to achieve extreme low-latency while maintaining application compatibility and support for TCP/IP protocol. One of the claim is that it can achieve below 1.7 µs for application-to-application [9]. *OpenOnload* implementation enables transfer data from the user space directly to the NIC because it is linked into an application's address space and granted direct access to network hardware [11].

2.1.2. CPU Affinity

Processes running on Linux are handled by a scheduler. A scheduling policy determines when and how to select a new process to run. Other than the default scheduling policy, it is possible to create a custom scheduling policy. The default scheduling policy is based on the time sharing technique and priority of processes [7][12].

[1] `http://linux.die.net/man/8/netstat`

[2] `http://www.linuxcommand.org/man_pages/ethtool8.html`

[3] `http://linux.die.net/man/8/tcpdump`

[4] `http://www.ntop.org/products/pf_ring/dna/`

[5] `http://info.iet.unipi.it/~luigi/netmap/`

[6] `http://www.intel.com/p/en_US/embedded/hwsw/technology/packet-processing`

[7] `http://www.napatech.com/products/network_adapters.html`

[8] `http://www.openonload.org`

[9] `http://www.openonload.org`

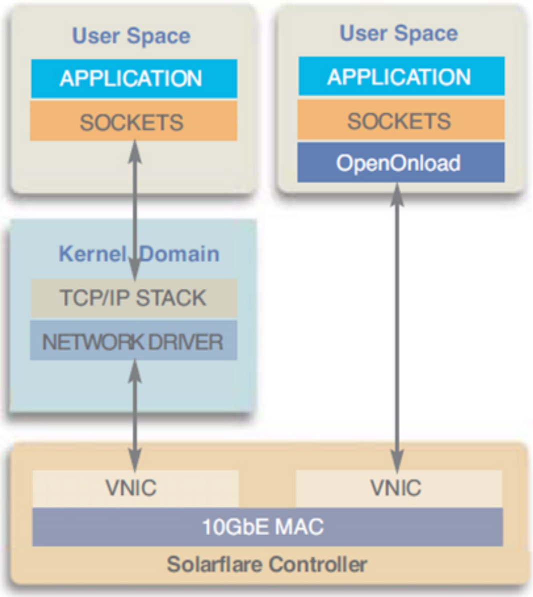

Figure 2.2.: OpenOnload enables applications to transfer data directly to the NIC by using user space library

Time sharing means that the CPU time is divided into time slices, one for each process. At one moment, one processor can run only on one process. If a process is still running beyond the assigned time slices, the scheduler will migrate this process to another CPU.

The scheduler also uses the priority of the processes to determine which processes get to run on CPU. The scheduler keeps track of what processes are doing and adjust their priority periodically. Processes that have not used CPU for long time will have their priority increased and on the opposite, processes that have been running for a long time will have their priority decreased. The biggest chance of having CPU time goes to the process with highest priority.

CPU affinity is the ability in Linux to bind processes to a certain processors [13]. There are two types of CPU affinity: soft and hard affinity. Soft affinity, or also known as natural affinity, is the default way of scheduler to try to keep processes on the same CPU as long as possible. The process will be moved to another processor if it becomes impossible to keep it on the same CPU based on the scheduling policy. Hard affinity is a way to force a process to run on a certain CPU without any possibility of being moved to another processor. The implementation of hard affinity is provided by Linux system call `sched_affinity` (since Linux kernel v2.6) [7].

One of the benefit of configuring CPU (hard) affinity is cache optimization. When a process runs on a processor, a local cache contains data from a processor is created. If a process keeps bouncing between processors, this local cache will likely to be invalid because each process also likely to have different data thus invalidating the old cache and creating new one. It means cache miss will grow large.

Another benefit that related to a time sensitive application, which is very common in an HFT environment, is performance. By configuring a single process to bound to one processor and let other processes run on other processors, it directs all attention and resource of a processor to a single process [13]. Therefore the process can run without any significant interruption that can be caused by CPU.

2.1.3. Spinning

An interrupt is a signal to the kernel that an event has occurred and therefore changes the sequence of instructions that is executed by processors [14]. Interrupt can be caused by software (software interrupt) or by hardware (hardware interrupt). Software interrupt is caused by an application running on user mode to show an exceptional events. Hardware interrupt is used to let processors know that an event created by the hardware needs their attention.

This also applies to network cards. When a network packet arrives on a network card, it will send an interrupt notification to CPU [15]. It means CPU will need to handle interrupt for each incoming packet. Every time CPU handles an interrupt it will create an overhead because CPU will need to do context switching. On top of that, there are thousands of packet coming in short period of time, therefore most NIC drivers (and supported by Linux kernel) [16] are using polling (called device polling) to regularly handle arriving packets [17] [18]. In a network stack that experiences high load of incoming network packets, device polling shows better performance than per-packet interrupt [16] [19].

In a low-latency sensitive environment, both options are not optimal. Per-packet interrupt will create too many context switches and device polling will create unnecessary latency by letting incoming packets to wait before being handled by processor.

An option to achieve better performance in terms of latency is to constantly ask network device for new packets, which is known as spinning. To realize this in Linux, application can be configured to either repeatedly invoke on of the polling system calls (`poll()`, `epoll()`, `select()`) on the application level or to use Busy Polling socket option (`SO_BUSY_POLL`), which is included in the kernel since v3.11 [7]. The implementation of busy polling as the socket option will only work in combination with network driver that implements `ndo_busy_poll()` callback and Linux kernel that compiled with `CONFIG_NET_RX_BUSY_POLL` option [20]

By using Busy Polling, the networking stack actively ask the device driver for new packets for a given amount of time. If there are newly arriving packets, the network driver sends them directly through the network layer to the socket. When the poll call is back to the networking stack, it checks directly for any pending data in the socket queue. By doing this, there will be no time wasted by letting packets waiting in the queue.

The major advantage of spinning is that it minimizes the number of context switch that occurs when packets arrive [21]. It then will reduce latency and jitter [22]. However, it causes greater CPU utilization on the core that is doing the poll, which will eventually have an impact on the general performance. An additional side effect is that because of busy polling, CPU will have no time to sleep and hence using more power. Therefore, a careful tuning must be considered to achieve the desired performance [23].

2.2. Docker

Docker is an open source platform that automates the process of developing, shipping, and running applications. Docker packages an application (with all of its libraries and dependencies) into a standardized unit called a (Docker) container [24]. Docker combines the principle of operating system-level virtualization [25] with tools to simplify the managing and the deploying process of these containers.

2.2.1. Comparison with Virtual Machine

As seen in Figure 2.4, Docker containers include the application and all of its dependencies but share the same operating system's kernel with other containers. On the other hand, traditional virtual machines include the entire guest operating system (as seen in Figure 2.3).

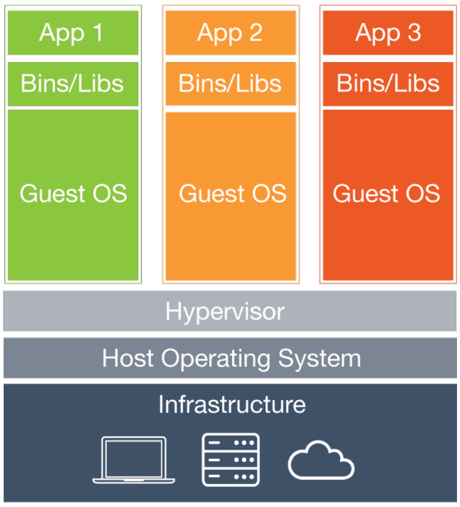

Figure 2.3.: Virtual machine architecture [24]

2.2.2. Inside Docker

There are three components in Docker internally:

- **Docker image** - it is a read-only template used to create a container. Every image starts from a base image. Base images are mainly operating system images, e.g. Fedora 20 image[10] or Ubuntu 14.04 image[11]. Operating system images are

[10] https://registry.hub.docker.com/_/fedora/

[11] https://registry.hub.docker.com/_/ubuntu/

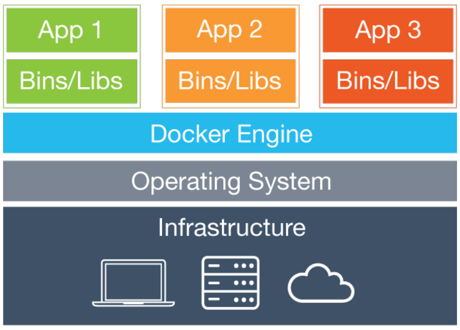

Figure 2.4.: Docker container architecture [24]

examples of images that would create a container with a fully working operating system. It is also possible to create a base image from scratch [26]. The base image can be modified by adding necessary applications, which then can be converted to be a new image. This process is called "committing a change". This is one of the two ways to build an image. The other way is by using a Dockerfile. A Dockerfile is a script that consists of instructions to build an image in an automated way [27][28].

- **Docker container** - as explained previously in Section 2.2, it is a standardized unit that has everything necessary for an application to run in an isolated way. A container is created from a Docker image. For instance, an image of Ubuntu with Apache will create a container that has Apache running on Ubuntu [29][30].

- **Docker registry** - it holds the Docker images. It operates similar to source code repositories in which images can be downloaded and uploaded ("push" and "pull" are the proper terms) from a single source. A registry can be private or public. A public Docker registry is called Docker Hub[12] where everybody can push their images and also pull publicly available images without any need to create an image from scratch. This feature allows images to be distributed (either publicly or privately) to a specific location [31][32].

2.2.3. Underlying technology

To allow containers to run in an isolated way, Docker makes use of several Linux kernel features [33].

- **namespaces** - the main idea of Linux namespaces is to separate different resources of a group of processes to have different view of the system than another group of processes [34]. The implementation of namespaces allow processes to be put into different namespaces, with all processes in each namespace has no clue about the existence of other processes in other namespaces. It then provides a form of lightweight virtualization and resource isolation. This capability is the reason why

[12] `https://hub.docker.com`

Docker uses this kernel feature to create containers. It is to provide them with an isolated workspace and their own environment without having access outside it. There are several types of namespaces used by Dockers to accomplish their goals of building isolated containers [34].

  - `pid` - used for process isolation. The `pid` namespaces allow multiple processes in different `pid` namespaces to have the same `pid`. This is possible because processes in different namespaces cannot see other processes in the other namespaces. This is the foundation of Docker container because it means that all processes in a container will not be able to see other processes outside of the container and therefore will not be able to influence or affect other processes in other container.
  - `net` - used for managing network interface. It provides isolation of the system resources related to networking. Each network namespace therefore can be configured to have different network configuration (network device, IP address, routing tables, etc) than other network namespace.
  - `ipc` - used to isolate certain interprocess communication (IPC) resources, namely System V IPC objects and POSIX message queues. It means that each IPC namespace has its own set of System V IPC identifiers and its own POSIX message queue filesystem.
  - `mnt` - used for managing mount points. A filesystem that is mounted in a mount namespace will not be seen by other mount namespaces. Therefore it allows processes to have its own view of a filesystem and of their mount points.
  - `uts` - used for isolating kernel and version identifiers. It can isolate two system identifiers (nodename and domainname) in which allows each container to have its own hostname and domain name.

- **control groups (cgroups)** - it is Linux kernel layer that provides resource management and resource accounting for groups of processes. Docker implement cgroups so that available hardware resources can be shared fairly between containers and also to limit it if necessary.

- **union file system (UnionFS)** - it is a file system that operates by creating layers which is used to provide the building blocks for containers.

- **container format** - it is a wrapper that combines all the previously mentioned technologies. Despite the fact that *libcontainer* is the default container format, Docker also supports LXC (Linux Container)[13].

The implementation of namespaces and cgroups by Docker to create containers are fairly lightweight since it is only separating some Linux kernel administration and therefore have insignificant impact on the performance [35].

[13] `http://linuxcontainers.org`

2.2.4. Networking Mode

Since the focus of this research is the network I/O performance, it is crucial to take a look at the network setup of containers in Docker.

There are four possible network setups in Docker [36]:

- **bridge** - this is the default networking setup when a Docker container is created. Each container has its own network namespace so that it has its own isolated network stack. Moreover, an Ethernet bridge is created in the host machine which is connected to the Ethernet interface of the container. Then depending on the preference, it can be configured to use bridging or NAT (Network Address Translation).

- **host** - this mode does not create a separated network stack for a container. The container shares the same network stack with the host which means the container also has full access to the network interfaces of the host.

- **container** - it lets a container to share the same isolated network stack with another container such that running processes on these two containers can communicate to each other via the loopback interface.

- **none** - a container will have its own isolated network stack but it is left unconfigured and it is all for the user to setup the network stack.

Only host and bridge networking mode will be the focus in this research because these two modes can be used to connect a container to other networks outside the host. Whereas container mode are used only to communicate between containers.

These various networking modes are created by the network namespace mentioned in Section 2.2.3. How each of these networking modes affect the performance is investigated in Section 4.

2.3. Hardware timestamps

Hardware timestamps are timestamps added to packets in the ingress port. This is done by a dedicated hardware component in certain hardware (mostly switches). By having a packet timestamped by dedicated hardware in the ingress port, it gives better accuracy then software timestamps [37]. In short: hardware timestamping is more accurate and precise than software timestamping.

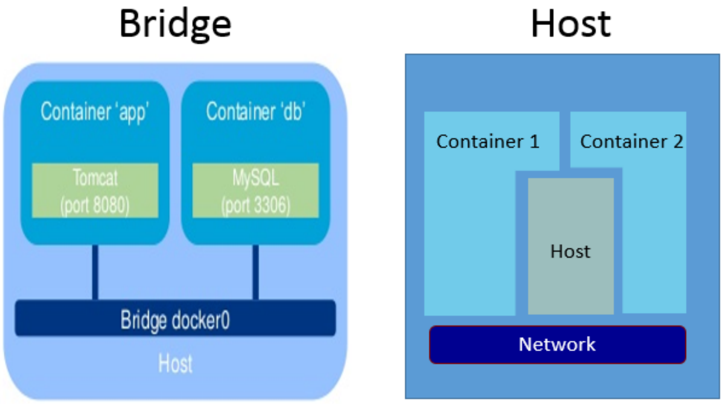

Figure 2.5.: The difference in architecture between bridge networking mode [38] and host networking mode in Docker.

3. Methodology

3.1. Approach

This research uses a bottom-up method. First of all, a topology will be set up. This topology is designed to do the measurement accurately. Then the baseline will be determined by measuring the network I/O performance of applications running with and without Docker. Other measurements will be performed with some network tuning to optimize and improve the baseline result. All measurements will be done multiple times to get repeatable and deterministic results. In the end, these results will be compared to each other. By comparing them, a conclusion can be taken whether the implementation of Docker actually impacts the network I/O performance.

3.2. Topology

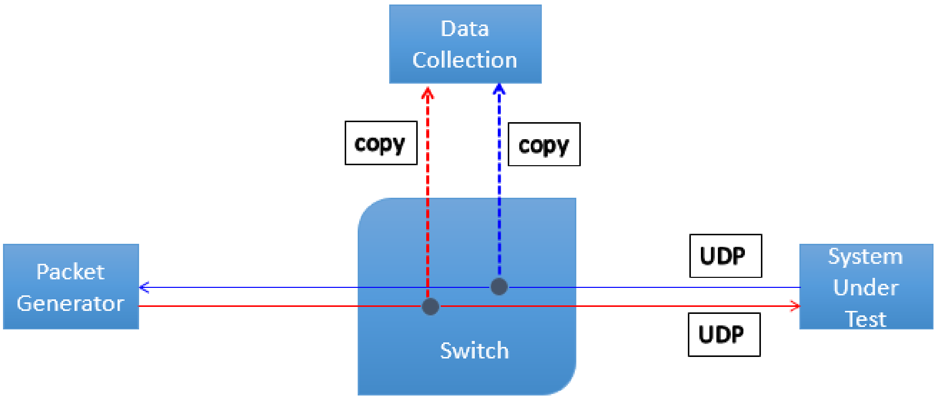

Figure 3.1.: The topology for the measurement

The topology is set up as shown in Figure 3.1 which consists of four components: *packet generator*, *data collection*, *switch*, and *system under test*. It is assumed that there is zero latency caused by the cables that connects each hardware.

The process is as the following:

1. The *packet generator* sends a UDP packet to the *system under test* through the *switch*.

2. In the *switch*, the packet is copied to the *data collection*.

3. The original packet is received by the *system under test* and it is sent back to the *packet generator*.

4. This returning packet is then also copied in the *switch* to be sent to the *data collection*.

5. Meanwhile the original returning packet continues its way to the *packet generator*.

6. In the *data collection* the copy of the original and the returning packet are timestamped and the difference is calculated by subtracting the timestamps of the original packet from the timestamps of the returning packet.

Packets coming from the *packet generator* are copied as late as possible on the egress port. It is to avoid any delays that can occur inside the *switch*. For the same reason, packets returning from the *system under test* to the *packet generator* are copied as early as possible on the ingress port. Therefore, both original and returning packets are always copied exactly on the same points.

Furthermore, the copied packets are sent to the *data collection* with dedicated lines to avoid collision between copied packets that would add unnecessary delays.

The *system under test* is a server with Intel®Core$^{\text{TM}}$i7-4790K CPU @ 4.00GHz. It has 8 MB of L3 cache. The size of cache is important because the larger the cache the less cache miss can happen which will eventually lead to less time needed to process network packets.

The application that is used to send the UDP packets is *sfnt-pingpong*. It is an open-source application that is used to measure network I/O performance. More detail about *sfnt-pingpong* is explained in Section 3.3. This application is installed in the *packet generator* and the *system under test*. In the *system under test*, it will be installed natively and also inside a Docker container.

UDP-in and UDP-out is used in this research. It is different than the application used in the production environment which is using UDP-in and TCP-out (as mentioned in Section 1.1). Since the focus is to measure the network performance, using UDP-in and UDP-out is sufficient and even preferred because UDP protocol is faster than TCP, the measured results shows the time it takes for a packet to travel through a network stack, without any delay. Other than that, there was not enough time to perform the experiments on the production application.

Fedora 20 is used as the operating system for the *packet generator* and the *system under test*. This is chosen because Fedora is already common in the infrastructure of the company where this research was carried out.

3.3. Sfnt-pingpong

Sfnt-pingpong is one of a set of tools developed by Solarflare[1] to measure network performance on Linux, Solaris, FreeBSD, and MacOSX. This application can be obtained from its website[2].

Sfnt-pingpong has a client and server architecture. The client sends the packet and the server acts as a mirror that bounces the incoming packet back to the client. In this research, the client is the *packet generator* and the server is the *system under test*.

Sfnt-pingpong provides built-in function to optimize the measured network performance, i.e CPU affinity and spinning. As explained in more details in Section 2.1.2 and Section 2.1.3, CPU affinity is dedicating a single CPU core to run the application while spinning is a way for an application to keep checking for new incoming network packets instead of waiting for an interrupt from the network card. For the spinning, *sfnt-pingpong* is using user space `poll()` and `epoll()` system calls instead of using kernel socket option `SO_BUSY_POLL` [39].

It is also possible to set the size of a packet and the number of packets sent. Moreover, it is also possible to test the performance by sending either TCP or UDP packet.

3.4. Dockerizing

Dockerizing is a common term to describe the process of converting an application to run in a Docker container. As *sfnt-pingpong* will run without and also with Docker, it must be *dockerized* as well.

As mentioned in Section 2.2.2, there are two ways of dockerizing an application, by creating a Docker container and updating it internally or by using a Dockerfile. In this research, a Dockerfile will be used to create an image.

The Dockerfile written for this research is shown in Appendix B.1. The instructions in this Dockerfile do the following: use Fedora 20 as the base image, install necessary applications, unpackage and compile *sfnt-pingpong*, and put it into `/usr/local/src/`. A symlink to this directory is then created in `/usr/local/bin`.

This created image has been uploaded and therefore is accessible in Docker Hub[3].

To create an image based on this Dockerfile, the command as shown in Appendix B.2 can be invoked. The option *-t* is to give a name to the created image.

The created image then can be run as a container by using the command shown in Listing B.3. It tells Docker to create a container based on the *fedora-pingpong* image with a terminal (*-t*) and to make an interactive connection (*-i*) so users can get a command prompt inside the created container. It also launches a Bash shell inside the container.

[1] `http://www.solarflare.com/`

[2] `http://www.openonload.org/download/sfnettest/sfnettest-1.5.0.tgz`

[3] `https://registry.hub.docker.com/u/ardho/fedora-pingpong/`

3.5. Test Cases

As mentioned in Section 3.3, the measurement is performed by using *sfnt-pingpong*. The test cases are separated into two scenarios: running the application with and without the optimization options. The optimization options are CPU affinity and spinning. Each scenario will be performed natively (without Docker) and in Docker (with both host networking mode and bridge networking mode).

The following options are used by *sfnt-pingpong* for all measurements:

- **64 bytes data payload size** - it is resulted in an Ethernet packet of 110 bytes (18 bytes Ethernet header, 20 bytes IP header, 8 bytes UDP header, and 64 bytes UDP payload). This is a typical trading-traffic size.
- **1 million packets sent** - it is to get statistically correct measurements.
- **UDP packet** - it is unreliable and best effort protocol which is faster than TCP.

The options are set as parameters on the client side but based on the purpose of the parameters, the related action can be carried out by the server side. Setting the optimization options are the example where the parameters are set on the client side but the action itself is executed on the server side. So in this case, when the client sends the network packets, the server receives it and send them back while performs CPU affinity and spinning as configured on the client side [39].

Furthermore, no kernel-bypass used on this research because of the time constraint.

```
sfnt-pingpong --sizes=64 --miniter=1000000 --maxiter=1000000 udp
    system_under_test
```

Listing 3.1: sfnt-pingpong command to run measurements without optimization options

```
sfnt-pingpong --affinity="2;2" --spinning --sizes=64 --miniter=1000000
    --maxiter=1000000 udp system_under_test
```

Listing 3.2: sfnt-pingpong command to run measurements with optimization options

4. Results

As explained in Section 3.5, there are in general two scenarios: the baseline (without any optimization) and the optimized configuration measurement. Multiple measurements have been performed for each scenarios and setups mentioned in Section 3.5. The following terms are created for each setup to show the results in a simple way:

- **Docker bridge** - Docker with bridge networking mode
- **Docker host** - Docker with host networking mode
- **No Docker** - no Docker is used

For each setup, the results of ten measurements will be shown. The results will be shown in tables and also in CDF (Cumulative Distribution Function) graphs. A single table will show min, median, 95 percentile, 99 percentile, and standard deviation value. They are used to show the percentage of the packets below a certain value.

And to show that these results are always within certain boundaries, all ten measurements results are combined into one. Then the statistical results were created again together with CDF graphs.

In this section, the summary of baseline results and optimized results will be explained. For each complete measurement, please refer to Appendix C and Appendix D.

4.1. Measuring the baseline

Figure 4.1 and Table 4.1 show the comparison of each setup. Without Docker, there is a 50% probability that it would take 6.17 µs or less for network packets to go back and forth the network stack. Meanwhile, with Docker, this number is slightly higher (for both network host and network bridge mode). Having said that, these results are not scientifically significant to prove or to confirm whether there is indeed a performance degradation. The results are too close to each other.

The wide spread of resulted values (as shown in Appendix C) shows inefficiencies and delays during transmission. These inefficiencies could be caused by multiple reasons as mentioned in Section 2, i.e. context switching, polling time, interrupt, and cache misses.

This result was created by *sfnt-pingpong* without any optimization options. The network driver either sends interrupt to kernel for each incoming packets or uses device polling which would hurt the performance since context switching is expensive and polling time wasted unnecessary waiting time for the packets to be picked up by the

	min (in µs)	median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
Docker host	4.98	6.33	9.60	14.69	1.93
Docker bridge	5.43	6.63	8.63	10.7	6.72
No Docker	4.92	6.17	10.08	15.98	2.30

Table 4.1.: Comparison of combined results of ten measurements of each setup without optimization

kernel. It is also using natural CPU affinity during measurement that will make processors to bound to different processes during a single measurement. This will creates cache misses and therefore adds more latency.

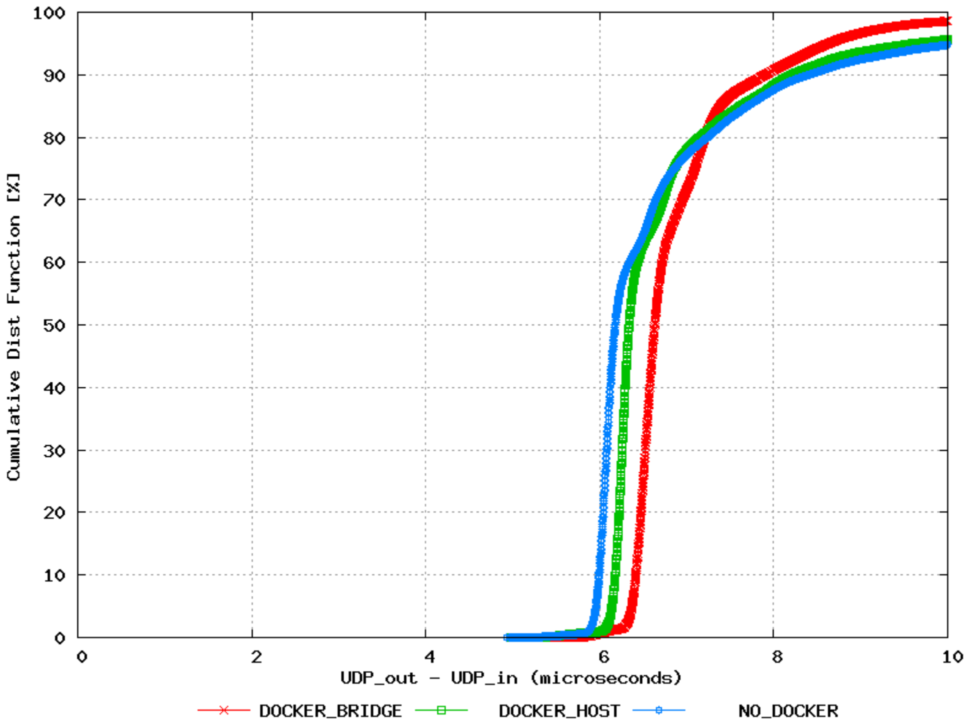

Figure 4.1.: Comparison of combined results of ten measurements of each setup without optimization

4.2. Measuring with optimization

Figure 4.2 and Table 4.2 show the comparison between results of each setup.

It shows that the network I/O performance without Docker and with Docker host networking mode are identical. There is a 50% probability that it would take 4.13 µs or less for network packets to go back and forth the network stack in a Docker container with networking host mode. Without Docker, for the same observation, the result is

	min (in µs)	median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
Docker host	3.47	4.13	4.60	4.85	0.22
Docker bridge	4.38	4.94	5.43	5.67	0.26
No Docker	3.65	4.15	4.64	4.88	0.22

Table 4.2.: Comparison of combined results of ten measurements of each setup with optimization

4.15 µs. Moreover, the standard deviation value indicates that the results are more stable and therefore deterministic, as shown in Appendix D.

The result also indicates that tuning the application with network tunings like CPU affinity and spinning greatly reduce the delays since it minimizes context switch, interrupt, polling time, and cache misses.

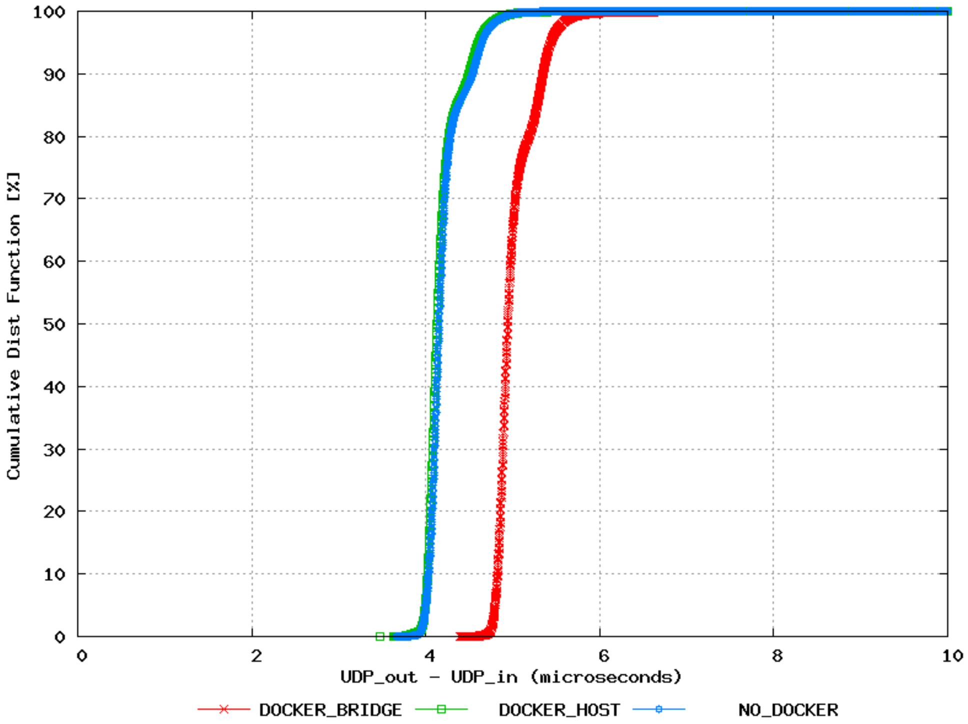

Figure 4.2.: Comparison of combined results of ten measurements of each setup with optimization

5. Conclusions

Docker simplifies the deploying process of applications by packaging applications with all of their dependencies and libraries into a standardized unit.

To find out whether a performance degradation exists while running applications in Docker, a series of measurements was carried out. After setting up a topology and converting a test application to run in Docker, multiple measurements have been performed successfully.

The results indicate that there is a slight performance degradation of the network I/O performance when Docker is used. However, the results were having too high value of standard deviation with really small difference between them. Therefore, the results are not scientifically significant to prove that there is indeed a performance degradation.

Furthermore, if the application running in Docker is properly configured with some network tuning (CPU affinity and spinning) and the network host mode is used, it gives identical performance as without Docker. The reason is that by using CPU affinity and spinning, a few source of latency are minimized, i.e. context switching and cache miss. This proves that having a (close to) native performance is possible when using Docker, even though the network setting of Docker and the running applications must be configured properly.

This result then also shows that the baremetal-like performance can also be achieved without using additional technique like kernel-bypass as has been found by Eder [1]. This result is an encouragement to use Docker in an environment that expect high performance, without implementing kernel-bypass technology. Therefore, it is also feasible for low latency sensitive applications to be deployed by using Docker since there is no significant impact on the network performance which is the requirement for low-latency sensitive applications.

5.1. Future Work

As the topology and supported tools are ready, it is good to expand the measurement to have scenarios with a larger number of packets and also with different sizes of data payload. The result would give an indication whether the native performance with Docker only occurs on a certain size of payload or not.

It is also interesting to run the measurements with having an extra load on the machine either by running multiple Docker containers at the same time or by using a special tool to create loads on the host itself. Another measurement setup would be to test the optimization options separately. The implementation of CPU affinity and spinning could have different effect to the performance when only either one of them implemented on the measurements.

Knowing the results of these measurements are going to be important because then they can be taken into account when building applications that depend on network I/O performance.

Acknowledgments

I would like to thank my supervisor Prof. dr. ir. Cees T. A. M. de Laat who gave me the golden opportunity to do this wonderful project on the topic.

Special thanks to Arno Bakker, as I appreciate his guide and time for helping me to write the report in good way.

I would also like to thank my wife, Indri, and friends who helped me a lot in finalizing this project within the limited time frame.

References

[1] Jeremy Eder. Accelerating Red Hat Enterprise Linux 7-based Linux Containers with Solarflare OpenOnload. Technical report, Red Hat Enterprise, April 2015. http://public.brighttalk.com/resource/core/67389/201504-onload_containers_brief_v10_99139.pdf.

[2] Nane Kratzke. About microservices, contars and their underestimated impact on network performance. In *Proceedings of CLOUD COMPUTING 2015 (6th. International Conference on Cloud Computing, GRIDS and Virtualization), p165-169*, 2015. https://www.researchgate.net/publication/273456042te.

[3] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. Technical report, IBM Research Division, IBM, July 2014. http://public.brighttalk.com/resource/core/67389/201504-onload_containers_brief_v10_99139.pdf.

[4] Larry Neumann. Kernel bypass revving up linux networking. http://www.solacesystems.com/blog/kernel-bypass-revving-up-linux-networking, March 2010. Retrieved: 16 August 2015.

[5] Jarret W. Buse. Linux Network Stack. http://www.linux.org/threads/linux-network-stack.4620/, September 2013. Retrieved: 4 July 2015.

[6] M. Tim Jones. Anatomy of the Linux networking stack. http://140.120.7.21/LinuxRef/Network/LinuxNetworkStack.html, June 2007. Retrieved: 4 July 2015.

[7] Daniel P. Bovet and Marco Cesati. *Understanding The Linux Kernel.* Springer Science+Business Media New York, New York, USA, 2005.

[8] Context Switch Definition. http://www.linfo.org/context_switch.html, October 2004. Retrieved: 18 August 2015.

[9] Li, Chuanpeng and Ding, Chen and Shen, Kai. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, page 2. ACM, 2007. http://dl.acm.org/citation.cfm?id=1281702.

[10] Jeremy Eder. Thoughts on Open vSwitch, kernel bypass, and 400gbps Ethernet. http://www.breakage.org/2012/10/01/thoughts-on-open-vswitch-kernel-bypass-and-400gbps-ethernet/, October 2012.

[11] Steve Pope and David Riddoch. Introduction to OpenOnload Building Application Transparency and Protocol Conformance into Application Acceleration Middleware. Technical report, Solarflare Communication, April 2011. http://www.solarflare.com/content/userfiles/documents/solarflare_openonload_intropaper.pdf.

[12] Chapter 14. Tuning the Task Scheduler. https://doc.opensuse.org/documentation/html/openSUSE_121/opensuse-tuning/cha.tuning.taskscheduler.html. Retrieved: 21 August 2015.

[13] Robert Love. Kernel korner: Cpu affinity. Linux Journal, July 2003. http://dl.acm.org/citation.cfm?id=860375.860383.

[14] Software Interrupt Definition. http://www.linfo.org/interrupt.html, May 2006. Retrieved: 21 August 2015.

[15] Christian Benvenuti. *Understanding Linux Network Internals.* O'Reilly Media Inc., Sebastopol, CA, USA, December 2005.

[16] Luca Deri. Improving Passive Packet Capture: Beyond Device Polling. Technical report, NETikos S.p.A., Pisa, Italy. http://luca.ntop.org/Ring.pdf.

[17] Jonathan Corbet. Low-latency Ethernet device polling. https://lwn.net/Articles/551284/, May 2013. Retrieved: 16 August 2015.

[18] Luigi Rizzo. Device Polling support for FreeBSD. http://info.iet.unipi.it/~luigi/polling/. Retrieved: 20 August 2015.

[19] Vivek Gite. FreeBSD Set Network Polling To Boost Performance. http://www.cyberciti.biz/faq/freebsd-device-polling-network-polling-tutorial/, June 2009. Retrieved: 20 August 2015.

[20] The Linux Kernel Archives, Documentation for /proc/sys/net/*. https://www.kernel.org/doc/Documentation/sysctl/net.txt.

[21] Marek Majkowski. How to achieve low latency with 10Gbps Ethernet. https://blog.cloudflare.com/how-to-achieve-low-latency/, June 2015. Retrieved: 20 August 2015.

[22] Jesse Brandeburg. A way towards Lower Latency and Jitter. Technical report, Intel, 2012. Linux Plumbers Conference, San Diego, California, 29-31 August 2012.

[23] Open Source Kernel Enhancements. Technical report, Intel, 2013.

[24] What is Docker? https://www.docker.com/whatisdocker/. Retrieved: 3 June 2015.

[25] Yang Yu. *OS-level Virtualization and Its Applications.* PhD thesis, Stony Brook University, December 2007. http://www.ecsl.cs.sunysb.edu/tr/TR223.pdf.

[26] Create a base image. `https://docs.docker.com/articles/baseimages/`. Retrieved: 9 July 2015.

[27] Docker images. `https://docs.docker.com/introduction/understanding-docker/#docker-images`. Retrieved: 7 July 2015.

[28] How does a Docker image work? `https://docs.docker.com/introduction/understanding-docker/#how-does-a-docker-image-work`. Retrieved: 8 July 2015.

[29] Docker containers. `https://docs.docker.com/introduction/understanding-docker/#docker-container`. Retrieved: 7 July 2015.

[30] How does a container work? `https://docs.docker.com/introduction/understanding-docker/#how-does-a-container-work`. Retrieved: 8 July 2015.

[31] Docker registries. `https://docs.docker.com/introduction/understanding-docker/#docker-registries`. Retrieved: 7 July 2015.

[32] How does a Docker registry work? `https://docs.docker.com/introduction/understanding-docker/#how-does-a-docker-registry-work`. Retrieved: 8 July 2015.

[33] The Underlying Technology. `https://docs.docker.com/introduction/understanding-docker/#the-underlying-technology`. Retrieved: 8 July 2015.

[34] Rami Rosen. *Linux Kernel Networking. Implementation and Theory.* Springer Science+Business Media New York, New York, USA, 2013.

[35] Fabio Diniz Rossi Tiago C. Ferreto Timoteo Lange Cesar A. F. De Rose Miguel Gomes Xavier, Marcelo Veiga Neves. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments, 2013.

[36] Network Configuration. `https://docs.docker.com/articles/networking/`. Retrieved: 8 July 2015.

[37] Douglas Arnold. Why is IEEE 1588 so accurate? `http://blog.meinbergglobal.com/2013/09/14/ieee-1588-accurate/`, September 2013. Retrieved: 8 July 2015.

[38] Adrien Blind. Docker Networking Basics and Coupling with Software Defined Networks. `http://www.slideshare.net/adrienblind/docker-networking-basics-using-software-defined-networks`, 2013. Octo Technology.

[39] Solarflare Communication. *Onload User Guide*, 2015. Appendix F: Solarflare sfnettest.

A. Automation Script

```bash
#!/bin/bash


trap ctrl_c INT

# Of course, set the functions first..

killall() {
    ssh labnet4 "pgrep -f ’sfnt-pingpong’ | xargs kill -9 &> /dev/null"
    ssh labnet4 "docker ps -a | awk ’{print $1}’ | xargs docker rm -f &> /dev/null"
    ssh labnet4 ’brctl delif docker0 eth1 &> /dev/null’
  pgrep -f ’solar_capture’ | xargs kill -9 &> /dev/null
}

help_message() {
  echo "-d to use with docker"
  echo "-o to use in an optimized way"
  echo "-w to write the filename (default: pingpong.txt)"
  echo "-b to use bridge mode. Need to be used a bit different. The node in the labnet4 must be entered first and run the server with do while loop. Here put the IP address of the docker container."
  echo "-s the byte size"
  echo "-i the number of iteration in one measurement (and the result will be created and inserted into a directory with the filename as the directory name)"
  echo "-p packet numbers"
  echo "-t time it takes to wait between exiting solarcapture and processing the result in seconds. needed when the packet numbers are big"
}

ctrl_c() {
  echo "You wanted to stop. I quit. But cleanup first.."
    killall
    ssh labnet4 ’brctl delif docker0 eth1 &> /dev/null’
    echo "Done. Bye!"
    exit 0
}


# Set the default value first...
ITERATIONS=1
```

```
PACKET_NUMBERS=1000
SIZES=64
OPTIMIZED=""
RESULTFILENAME="pingpong"
PINGPONG_SERVER="ssh labnet4 ’sfnt-pingpong’"
SERVER_IP="9.21.1.61"
CLIENT_IP="9.21.1.60"
BRIDGE_SERVER_IP=""
TIME_TO_WAIT=""

# Checking the parameters
if [ $# -eq 0 ];
then
    help_message
    exit 0
else
  while getopts ":dow:s:i:p:bt:h" opt; do
    case $opt in
    o)
      OPTIMIZED="--affinity=\"2;2\" --spin"
      ;;
    d)
      PINGPONG_SERVER="ssh labnet4 ’docker run  --name=${docker_pingpong} --net=host ardho/fedora-pingpong-cmd’"
      ;;
    w)
      RESULTFILENAME=$OPTARG
      ;;
    s)
      SIZES=$OPTARG
      ;;
    i)
      ITERATIONS=$OPTARG
      ;;
    p)
      PACKET_NUMBERS=$OPTARG
      ;;
    b)
      BRIDGE_SERVER_IP="9.21.1.1"
      PINGPONG_SERVER="ssh labnet4 ’docker run --name=${docker_pingpong} ardho/fedora-pingpong-cmd’"
      ;;
    t)
      TIME_TO_WAIT=$OPTARG
      ;;
    h)
      help_message
    ;;
    \?)
      echo "Invalid option: -$OPTARG" >&2
      exit 1
      ;;
    :)
      echo "Option -$OPTARG requires an argument." >&2
      exit 1
      ;;
    esac
  done
fi

if [ -z $TIME_TO_WAIT ]; then
  TIME_TO_WAIT=5
fi

if [ ! -z $BRIDGE_SERVER_IP ]; then
  SERVER_IP="${BRIDGE_SERVER_IP}"
fi

# This is the sfnt-pingpong command that will be used later.
THECOMMAND="sfnt-pingpong ${OPTIMIZED} --sizes=${SIZES} --miniter=${PACKET_NUMBERS} --maxiter=${PACKET_NUMBERS} udp ${SERVER_IP}"

# Kill everything now..
killall

# Set up directory name
DIRECTORY_NAME=${RESULTFILENAME}_dir
if [ "${ITERATIONS}" -gt 1 ]; then
  echo "Creating directory because has more than 1 result"

  mkdir ${DIRECTORY_NAME} &> /dev/null
fi

for i in $(eval echo "{1..$ITERATIONS}")
do
  if [ ! -z $BRIDGE_SERVER_IP ]; then
    ssh labnet4 'systemctl restart docker'
    ssh labnet4 'brctl addif docker0 eth1 &> /dev/null'
  fi

  while :; do
    CHECK_SOLARCAPTURE_ACTIVE_ETH2=$(pgrep -f "solar_capture_interactive.sh -n -i eth2" | wc -l)
    if [ "${CHECK_SOLARCAPTURE_ACTIVE_ETH2}" -gt 0 ]; then
      break
    else
      echo "Run solarcapture on eth2 of labnet3"
      solarcapture_eth2_cmd="solar_capture_interactive.sh -n -i eth2 -w eth2_sc.pcap \"udp and ip src ${CLIENT_IP} and ip dst ${SERVER_IP}\""
      tmux new-session -d -s solarcapture_eth2 -n sc2 "$solarcapture_eth2_cmd"

      sleep 5 # to make sure it the tmux is created successfully
    fi
  done

  while :; do
    CHECK_SOLARCAPTURE_ACTIVE_ETH3=$(pgrep -f "solar_capture_interactive.sh -n -i eth3" | wc -l)
    if [ "${CHECK_SOLARCAPTURE_ACTIVE_ETH3}" -gt 0 ]; then
      break
    else
      echo "Run solarcapture on eth3 of labnet3"
      solarcapture_eth3_cmd="solar_capture_interactive.sh -n -i eth3 -w eth3_sc.pcap \"udp and ip src ${SERVER_IP} and ip dst ${CLIENT_IP}\""
      tmux new-session -d -s solarcapture_eth3 -n sc3 "$solarcapture_eth3_cmd"

      sleep 5 # to make sure it the tmux is created successfully

    fi
   done

   tmux new-session -d -s session_pingpong_server -n pp "${PINGPONG_SERVER}"

   # Run client pingpong
   echo "Run pingpong client on labnet3"

   ${THECOMMAND}

   echo "Pingpong is done. Kill solarcapture and sfnt-pingpong server"
   killall

   sleep ${TIME_TO_WAIT}

   echo "Convert the pcap using tshark and join them"

   tshark -r eth2_sc.pcap -tad | columnx.sh 1 3 > eth2_sc.txt
   tshark -r eth3_sc.pcap -tad | columnx.sh 1 3 > eth3_sc.txt

   join eth2_sc.txt eth3_sc.txt | col_sub.sh -s 3 2 | columnx.sh 1 3 > ${RESULTFILENAME}

  mv ${RESULTFILENAME} ${DIRECTORY_NAME}/${RESULTFILENAME}-${i}
done



echo "Done!"
```

Listing A.1: A script to create the raw result

B. Docker

```
FROM fedora:20
MAINTAINER rohprimardho

#Preparing the software
RUN yum install -y wget
RUN yum install -y make
RUN yum install -y gcc
RUN wget http://www.openonload.org/download/sfnettest/sfnettest-1.5.0.tgz
RUN tar xzf sfnettest-1.5.0.tgz
RUN cd sfnettest-1.5.0/src
RUN make -C /sfnettest-1.5.0/src all
RUN cp /sfnettest-1.5.0/src/sfnt-pingpong /usr/local/src
RUN ln -s /usr/local/src/sfnt-pingpong /usr/local/bin/
```

Listing B.1: The content of the Dockerfile to create an image of Fedora 20 with sfnt-pingpong installed

```
docker build -t fedora-pingpong /path/to/Dockerfile
```

Listing B.2: The command to create an image from the Dockerfile

```
docker run -i -t fedora-pingpong /bin/bash
```

Listing B.3: The command to run a container from the created image

C. Baseline Measurements

C.1. Docker host

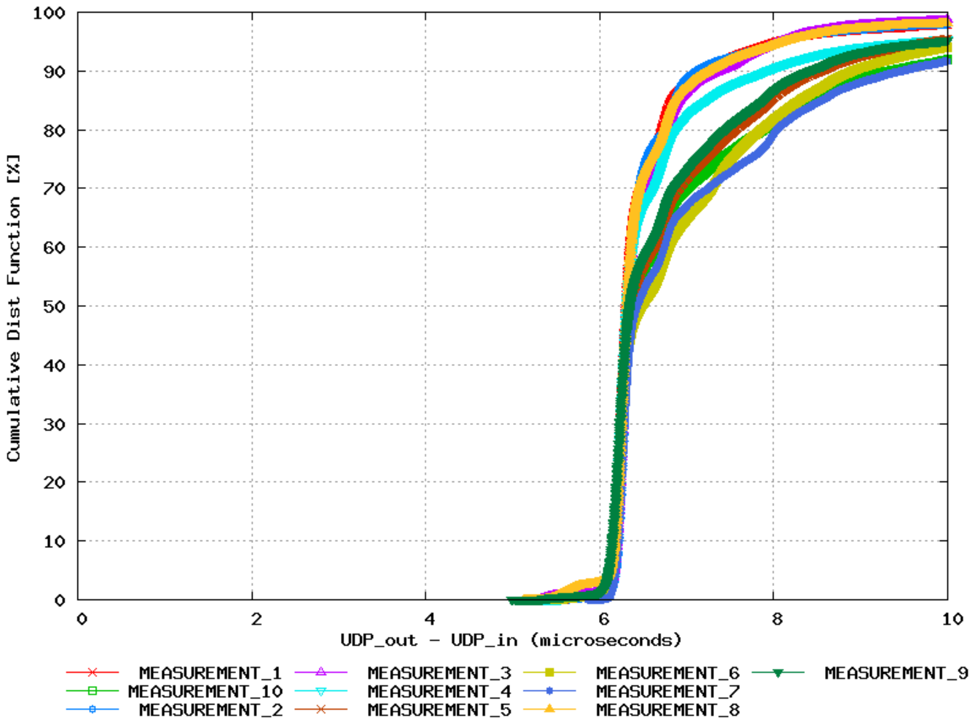

Figure C.1.: Ten measurements using Docker host

	median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
1	6.36	11.30	16.83	2.35
2	6.34	8.08	12.34	1.44
3	6.32	8.08	10.53	1.36
4	6.31	9.58	17.62	2.23
5	6.36	9.83	14.45	1.97
6	6.48	10.42	14.58	2.03
7	6.43	11.36	15.69	2.16
8	6.31	8.12	10.91	1.14
9	6.34	9.86	16.43	2.57
10	6.29	8.05	11.66	1.2

Table C.1.: Results of ten measurements using Docker host

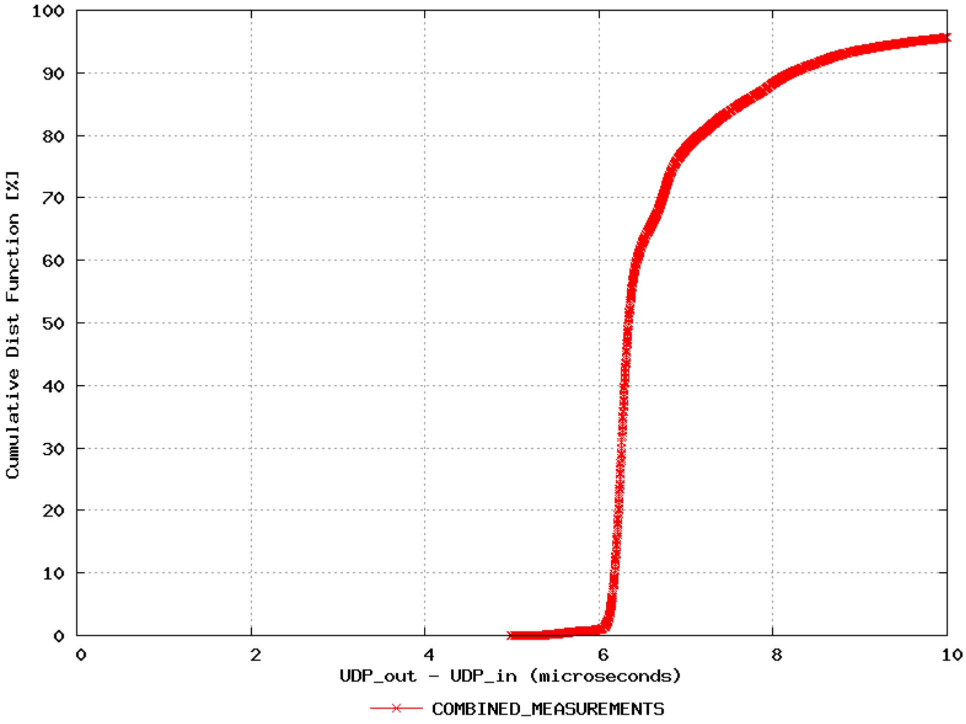

Figure C.2.: Combined results of all ten measurements using Docker host

median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
6.33	9.60	14.69	1.93

Table C.2.: Results of combined ten measurements using Docker host

C.2. Docker bridge

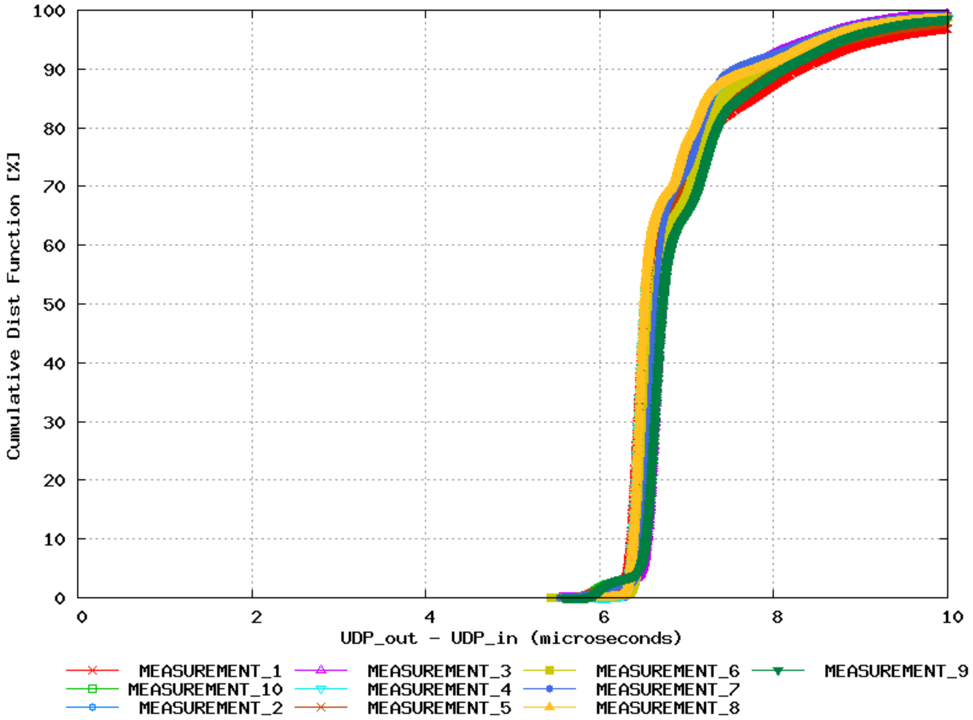

Figure C.3.: Ten measurements using Docker bridge

	median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
1	6.66	8.58	10.09	10.35
2	7.02	8.46	10.11	1.19
3	6.711	8.42	9.66	10.84
4	6.52	8.47	10.44	1.29
5	6.54	8.85	11.50	9.74
6	6.62	8.58	10.22	1.18
7	6.63	8.44	9.97	2.90
8	6.51	8.56	10.65	1.26
9	6.72	8.80	10.62	10.71
10	6.52	9.22	13.20	1.56

Table C.3.: Results of ten measurements using Docker bridge

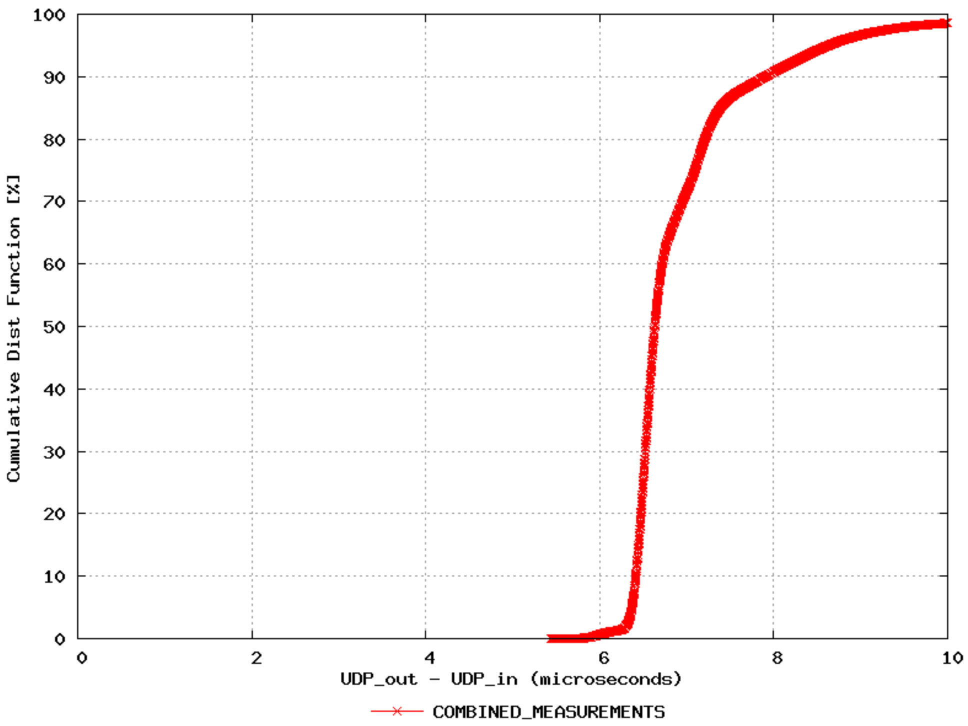

Figure C.4.: Combined results of all ten measurements using Docker bridge

median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
6.63	8.63	10.7	6.72

Table C.4.: Results of combined ten measurements using Docker bridge

C.3. No Docker

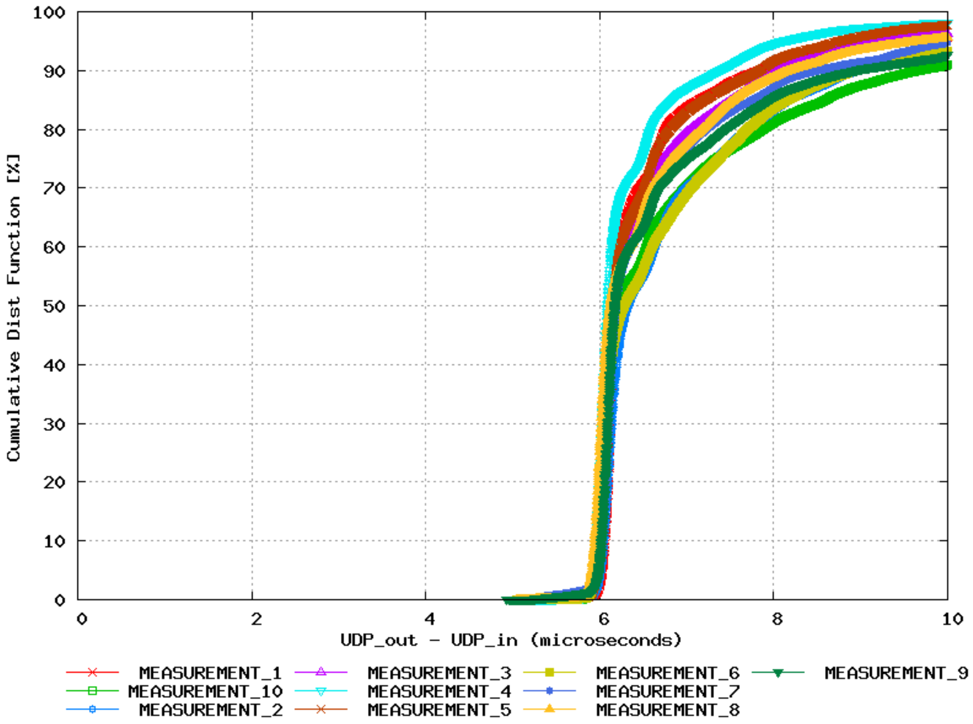

Figure C.5.: Ten measurements without Docker

	median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
1	6.18	11.64	17.10	2.51
2	6.33	11.21	17.33	2.22
3	6.15	9.21	14.06	1.76
4	6.09	8.10	12.38	1.60
5	6.14	8.76	12.66	2.26
6	6.28	10.70	15.43	2.20
7	6.19	10.04	16.14	2.20
8	6.12	9.65	14.89	2.06
9	6.17	10.86	19.64	2.61
10	6.19	9.46	16.54	3.12

Table C.5.: Results of ten measurements without Docker

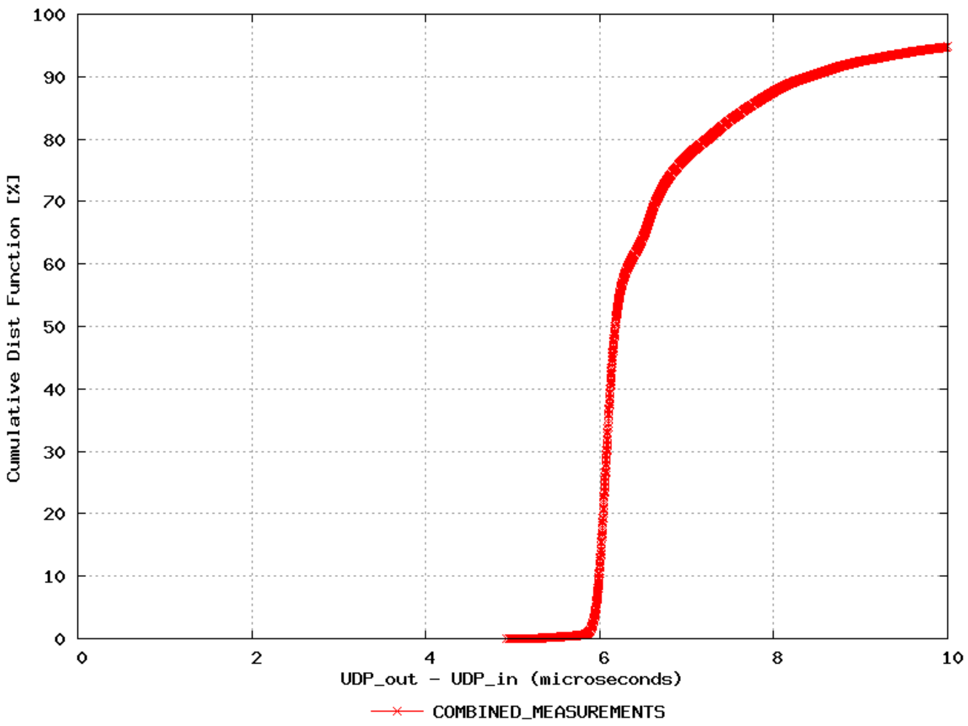

Figure C.6.: Combined results of all ten measurements without Docker

median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
6.17	10.08	15.98	2.30

Table C.6.: Results of combined ten measurements without Docker

D. Optimized Measurements

D.1. Docker host

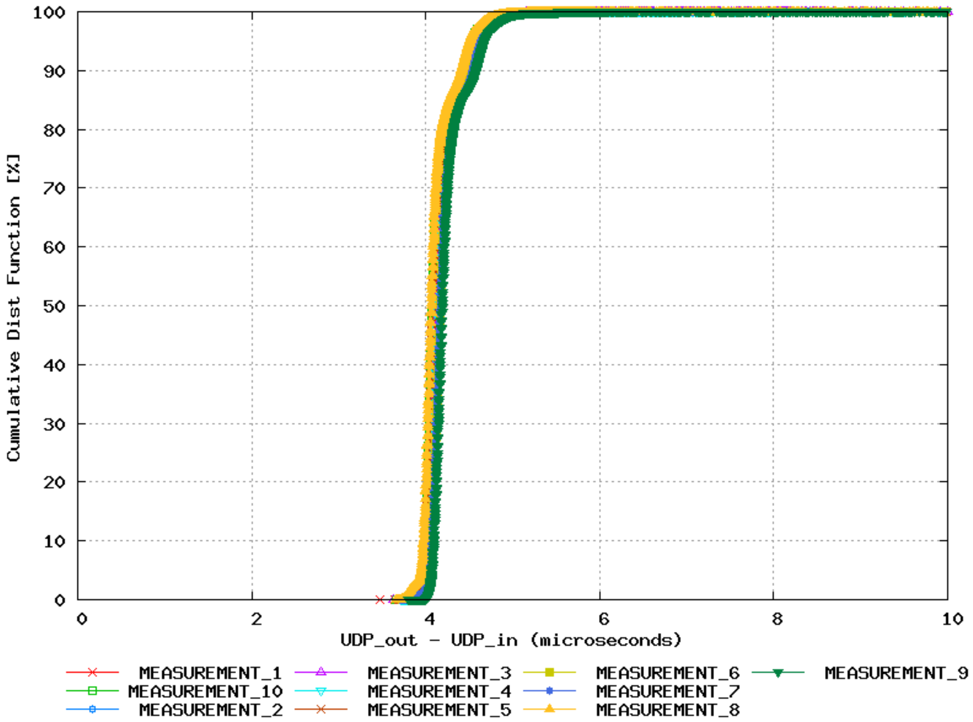

Figure D.1.: Ten measurements using Docker host with optimization

	median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
H	4.13	4.60	4.85	0.22

Table D.2.: Results of combined ten measurements using Docker host with optimization

	median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
1	4.07	4.52	4.76	0.21
2	4.11	4.60	4.84	0.20
3	4.13	4.58	4.81	0.20
4	4.12	4.61	4.87	0.20
5	4.08	4.55	4.86	0.23
6	4.17	4.63	4.88	0.21
7	4.16	4.62	4.88	0.21
8	4.06	4.50	4.74	0.20
9	4.19	4.67	4.93	0.25
10	4.16	4.62	4.86	0.21

Table D.1.: Results of ten measurements using Docker host with optimization

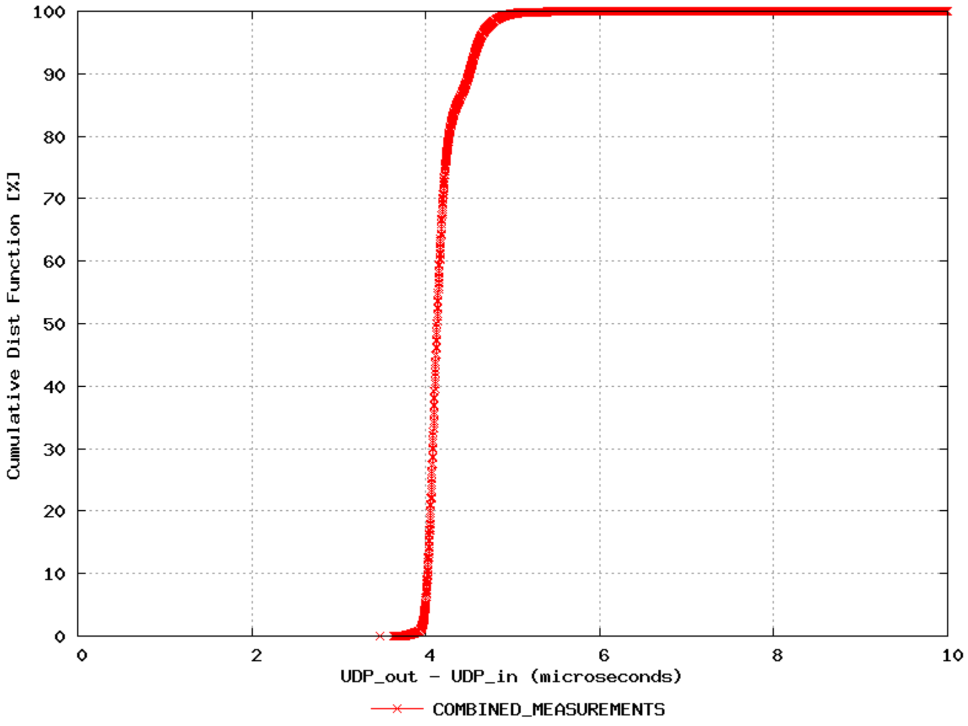

Figure D.2.: Combined results of all ten measurements using Docker host with optimization

D.2. Docker bridge

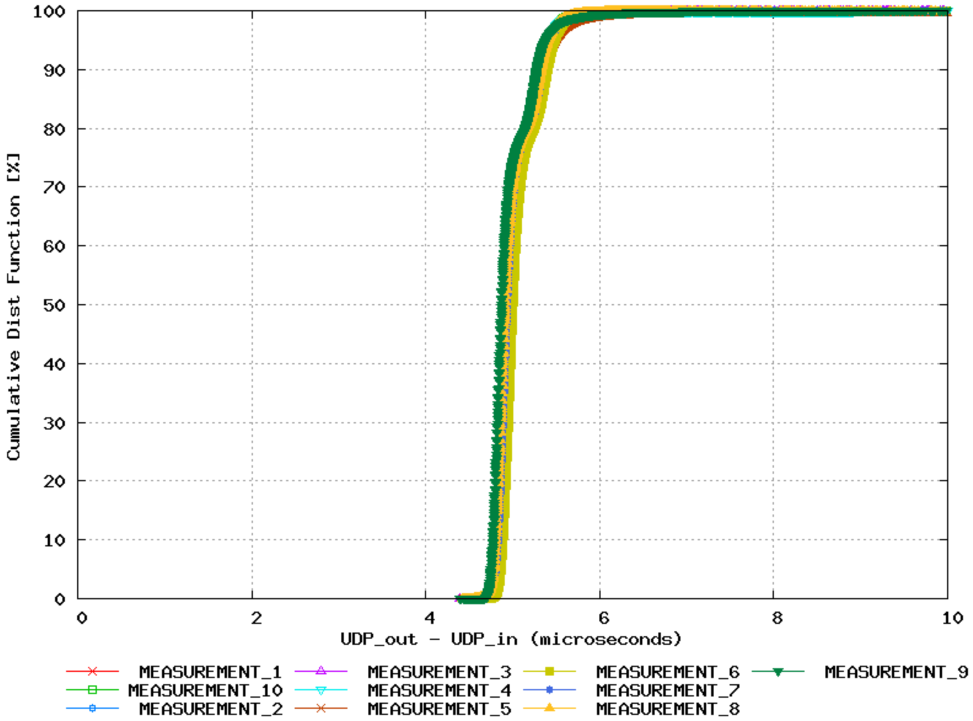

Figure D.3.: Ten measurements using Docker bridge with optimization

	median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
1	4.93	5.41	5.62	0.24
2	4.94	5.42	5.62	0.21
3	4.96	5.44	5.66	0.28
4	4.92	5.41	5.64	0.29
5	4.94	5.49	5.96	0.40
6	5.00	5.48	5.68	0.21
7	4.94	5.43	5.64	0.23
8	4.91	5.40	5.61	0.23
9	4.87	5.38	5.79	0.26
10	4.95	5.42	5.62	0.21

Table D.3.: Results of ten measurements using Docker bridge with optimization

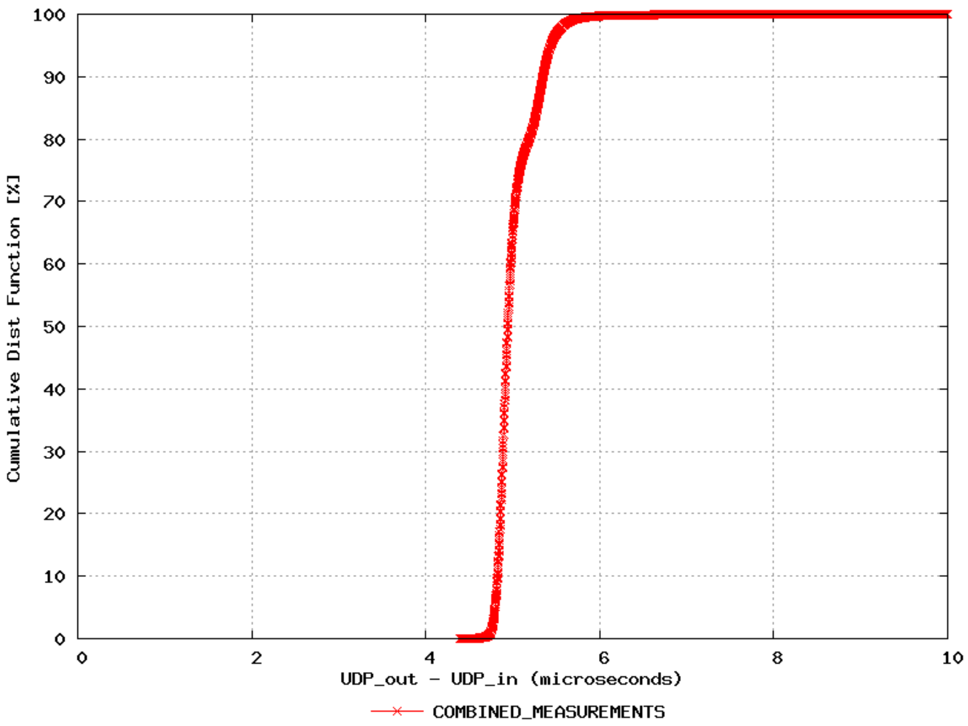

Figure D.4.: Combined results of all ten measurements using Docker bridge with optimization

median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
4.94	5.43	5.67	0.26

Table D.4.: Results of combined ten measurements using Docker bridge with optimization

D.3. No Docker

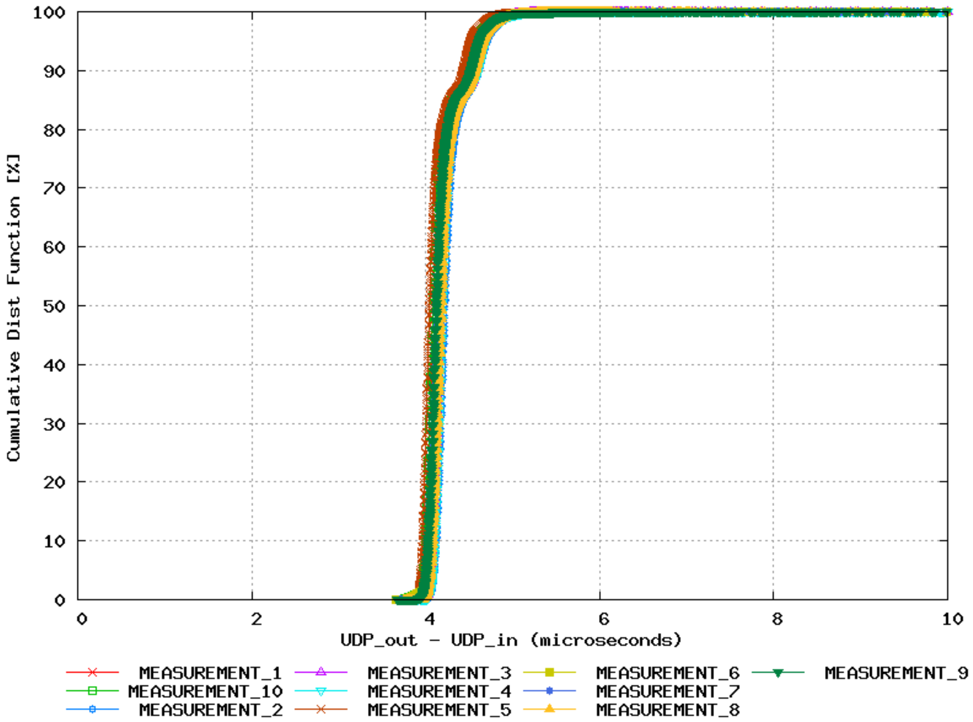

Figure D.5.: Ten measurements without Docker with optimization

	median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
1	4.08	4.58	4.81	0.21
2	4.21	4.70	4.94	0.24
3	4.16	4.66	4.89	0.20
4	4.18	4.68	4.92	0.23
5	4.04	4.51	4.74	0.21
6	4.16	4.65	4.88	0.21
7	4.13	4.62	4.85	0.20
8	4.18	4.66	4.90	0.20
9	4.12	4.61	4.85	0.22
10	4.16	4.63	4.86	0.20

Table D.5.: Results of ten measurements without Docker with optimization

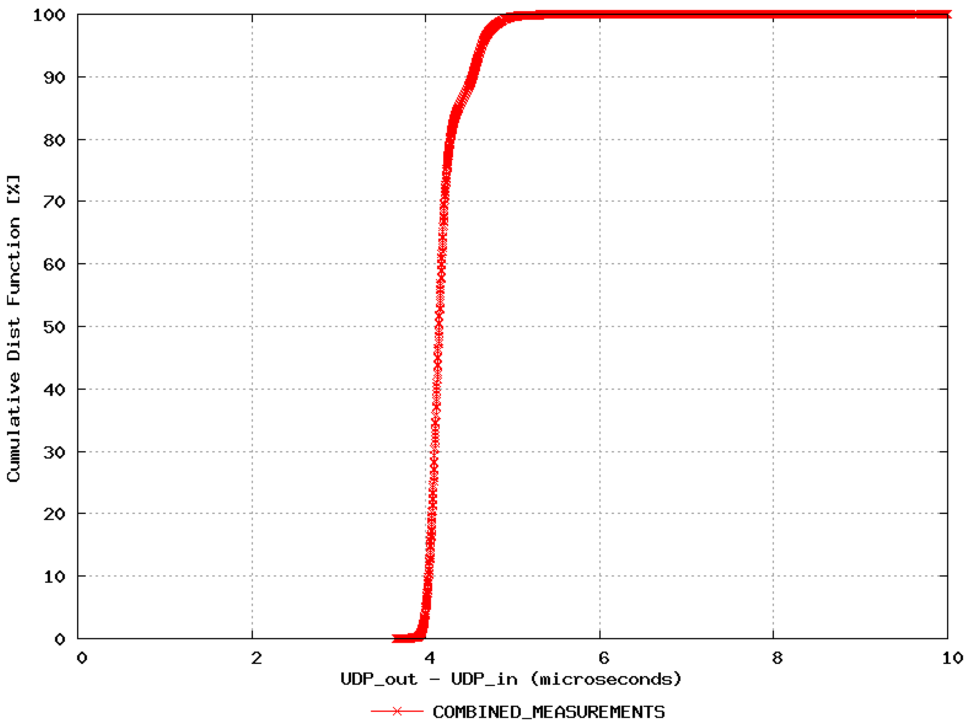

Figure D.6.: Combined results of all ten measurements without Docker with optimization

median (in µs)	95% (in µs)	99% (in µs)	std (in µs)
4.15	4.64	4.88	0.22

Table D.6.: Results of combined ten measurements without Docker with optimization