



# Comparing Model- and Delay-based Congestion Controls in a Kubernetes Cluster

---

Robin Allansson  
Vittcki Yang

---

Faculty of Health, Science and Technology

---

Computer Science

---

C-uppsats 15hp

---

Supervisor: Karl-Johan Grinnemo

---

Examiner: Leonardo Iwaya

---

Date: 2024-01-27

---



# Abstract

Today's society is heavily influenced by the use of cloud-based services and the internet as a whole. This is causing higher demand for stable networks and maximized efficiency inside the cloud industry. The direction the cloud-based industry went was to maximize their hardware using virtualization. To make this happen they had two options one being virtual machines and the other which grew more common were containers. Kubernetes and Docker are top candidates for virtualized environments which also raises the question of whether the current network protocol is up to speed, TCP Cubic has been the standard network protocol in Linux kernels since 2006 but there are newer protocols that challenge the old standard mainly TCP BBR which was announced by Google 2016. The current situation between the explosive growth of network-based applications and the older network-based protocol that was not designed for this type of traffic which begs the question, is the current standard keeping up to the task? Is the virtualization causing an impact on the traffic flow and is the older protocol worth it compared to the newer protocol produced by Google? In this thesis we are going to investigate the impact on performance, particularly delay and throughput, of the mentioned network protocols. This will be done by running a Docker container in a Kubernetes cluster to have a virtualized environment while being able to compare it to a standard test instance outside of the virtualized container. When the instances are set up there will be traffic generated by the use of iperf, netem and socket statistics to get the data needed to compare and evaluate. The experiment that was performed showed an array of different

results. When it came to the main questions that were investigated it was concluded that the TCP protocols BBR and Cubic were not affected by virtualization to the point that it was harmful to use them within Kubernetes, likewise it was concluded during the experiment that when introducing packet loss as a variable the protocols expect differently as theory dictated, TCP cubic handles package loss more aggressively which leads it to have a lower throughput compared to BBR that quickly tries to resume with the speed it had before a packet was lost. So to summarize the experiment, Kubernetes negatively effects both TCP protocols but it is minuscule enough that it will not cause issues when using virtualized environments. When it comes to the two protocols it is shown that BBR handles package loss better than Cubic but both protocols provide a similar throughput when the connection is stable.

# Contents

<b>Abstract</b>	<b>i</b>
<b>Figurer</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Description . . . . .	2
1.2 Thesis Objective . . . . .	2
1.3 Thesis Goals . . . . .	2
1.4 Ethics and Sustainability . . . . .	3
1.5 Methodology . . . . .	3
1.6 Stakeholders . . . . .	3
1.7 Delimitations . . . . .	4
1.8 Disposition . . . . .	4
<b>2 Background</b>	<b>5</b>
2.1 TCP and its Congestion Control . . . . .	6
2.1.1 Maximum Transmisson Unit . . . . .	8
2.2 Cubic . . . . .	8
2.3 BBR . . . . .	10
2.4 Virtualization . . . . .	11
2.5 Virtual Machines . . . . .	12

2.6	Docker Containers . . . . .	13
2.7	Kubernetes . . . . .	13
<b>3</b>	<b>Experiment</b>	<b>17</b>
3.1	Introduction . . . . .	17
3.1.1	Environment . . . . .	17
3.1.2	SSH . . . . .	18
3.2	Network Tools . . . . .	18
3.2.1	Iperf3 . . . . .	19
3.2.2	Socket statistics tool . . . . .	20
3.2.3	Netem . . . . .	20
3.3	Docker . . . . .	21
3.4	Kubernetes . . . . .	22
3.4.1	Kubectl . . . . .	23
3.4.2	Minikube . . . . .	23
<b>4</b>	<b>Results</b>	<b>25</b>
4.1	Introduction . . . . .	25
4.2	Inside a Cluster . . . . .	25
4.2.1	TCP BBR . . . . .	25
4.2.2	TCP Cubic . . . . .	28
4.3	Outside a Cluster . . . . .	31
4.3.1	TCP BBR . . . . .	31
4.3.2	TCP Cubic . . . . .	34
4.4	Comparison . . . . .	36
<b>5</b>	<b>Conclusion</b>	<b>39</b>
	<b>Appendix</b>	<b>47</b>







# List of Figures

2.1	Visual presentation of congestion window [1]	7
2.2	Illustration of TCP Cubic on its growth and packet loss function [2]	9
2.3	Illustration the difference between virtual machines and containers	12
2.4	A figure of a simple Kubernetes Cluster.	15
3.1	Experiment Setup.	18
3.2	The MTU of our machine outside the Kubernetes cluster	19
3.3	The MTU of our machine inside the Kubernetes cluster	19
3.4	The MSS of our machine outside the Kubernetes cluster	20
3.5	The MSS of our machine inside the Kubernetes cluster	21
4.1	Graph over TCP BBR inside the cluster, showing cwnd and ssthresh, 0.001% packet drops	26
4.2	Graph over TCP BBR inside cluster, showing cwnd and ssthresh, 3% packet drops	27
4.3	Graph over TCP BBR inside cluster, showing RTT with 0.001% packet drops	27
4.4	Graph over TCP BBR inside the cluster, showing RTT with 3% packet drops	28
4.5	Graph over TCP Cubic inside cluster, showing cwnd and ssthresh with 0.001% packet drops	28

4.6	Graph over TCP Cubic inside cluster, showing cwnd and ssthresh with 3% packet drops . . . . .	29
4.7	Graph over TCP Cubic inside cluster, showing RTT with 0.001% and 3% packet drops . . . . .	30
4.8	Graph over TCP BBR outside cluster, showing cwnd and ssthresh with package loss of 0.001% . . . . .	31
4.9	Graph over TCP BBR outside cluster, showing cwnd and ssthresh with package loss of 3% . . . . .	32
4.10	Graph over TCP BBR outside cluster, showing RTT with 0.001% and 3% packet drops. . . . .	33
4.11	Graph over TCP Cubic outside cluster, showing cwnd and ssthresh with package loss of 0.001% and at 3% . . . . .	34
4.12	Graph over TCP Cubic outside cluster, showing RTT with 0.001% and 3% packet drops . . . . .	35
4.13	Graphs over TCP BBR and TCP Cubic at maximum interference . . . .	36
4.14	Graphs over TCP BBR and TCP Cubic when TCP Cubic starts to falter	37
4.15	Compressed information in table form over all the data collected . . . .	38

# Chapter 1

## Introduction

TCP is the prevailing transport protocol on the Internet [3] and for many years TCP Reno and TCP New Reno [4] have been the primary TCP protocols and, as such, have been dictating the congestion control standard on the Internet. The use of TCP Reno and TCP New Reno on the Internet has been increasingly questioned with the growth of the Internet, mainly due to applications that require more than the "best effort" service these transport protocols offer. Moreover, with the inception of virtualization, concerns about these protocols' performance in virtualized environments have been raised, not least the extent these protocols affect applications reliant on high bandwidth or short delays. These concerns have been around but have yet to be answered.

Two categories of congestion control algorithms that have earned particular interest are model- and delay-based congestion controls. As the name suggests, model-based congestion control algorithms are built around a model of the network and how it reacts to increases and decreases in send rate. A well-known model-based congestion control algorithm is Bottleneck Bandwidth and Round-trip propagation time (BBR), which was proposed by Google in 2016 [5] and currently being standardized by the Internet Engineering Task Force (IETF) [6].

## 1.1 Problem Description

The increased demand for more cloud-based services and more effective usage of server hardware has led to more virtualization within servers, e.g., virtual machines and containers. A question that has emerged with the ongoing server virtualization is whether current TCP congestion protocols can efficiently work in virtualized server environments, a question that this thesis aims to address. The current industry standard is to use Kubernetes as virtual environment while the protocol was standardized by TCP Cubic and recently moved over to TCP BBR, that setting is what will be used as the experiments foundation. Kubernetes is quite a powerful tool to use and according to Enlyft [7] it is used in a multitude of different companies hence why it is being used as base for the experiment.

## 1.2 Thesis Objective

This thesis aims to investigate the impact on the performance, particularly delay and throughput, of TCP BBR and TCP Cubic by running these congestion control algorithms in Docker [8] Containers in a Kubernetes [9] cluster.

## 1.3 Thesis Goals

To complete the objectives described above we need to start by constructing a Kubernetes cluster [9] configured with a Docker container [8], this will be our inside node which we can transfer data to so we can measure how much virtualization [10] effects both TCP BBR [5] and TCP Cubic [11]. But to be able to compare this, a secondary node needs to be setup outside of the network which was made on a secondary server that is not configured within either Kubernetes nor containers to not be influenced by virtualization. The data necessary to perform a comparison is collected by the use of the

tools Iperf3 [12], Socket statistics tool [13] and netem [14]. The first two tools are for measuring the data transferred between nodes while netem is to add a simulated delay to get more accurate results that are closer to a real live connection.

## 1.4 Ethics and Sustainability

This thesis only evaluates protocols and the performance of virtualization environments, which raises few, if any, questions about ethics or sustainability. In the long term, it could positively affect sustainability, provided the tests result in changes in TCP BBR that would increase internet traffic speeds, leading to less power usage. Still, the decrease in power usage would be minuscule.

## 1.5 Methodology

This thesis project involved setting up a test environment using Kubernetes to evaluate TCP BBR and Cubic algorithms. The aim was to compare the throughput of data transmission within and outside a virtualized network. The test environment consisted of two machines: a server hosted within a Docker container inside a Kubernetes cluster, and a client set up as a node outside the virtualized environment. The traffic generated by the test provided valuable data on the impact on the server side, by comparing round-trip time, congestion windows, and slow-start threshold to the client side, which served as a benchmark.

## 1.6 Stakeholders

This thesis work is being carried out at Karlstads University, with Red Hat being the primary stakeholder. Red Hat is currently working on Kubernetes clusters, and striving

to deliver the best possible solution for their clients. This requires extensive research on the limits of Kubernetes and the protocols that affect its performance.

## 1.7 Delimitations

We conducted this project on a small scale, so the test results may not accurately reflect external factors that can impact speed, throughput, and latency. It is important to note that there are multiple variables within a network that could affect these metrics. Our test environment is built with only the minimum components necessary to test our thesis. However, due to the limited setup, it is impossible to predict how other nodes sending traffic at the same time would affect a real-scale model.

## 1.8 Disposition

This thesis begins by introducing the problem and the reason behind this project in Chapter 1. Chapter 2 presents the congestion control algorithms used in the experiment, along with Docker [8], Kubernetes [9], and other essential tools. Chapter 3 covers the implementation of the experiment and explains how the relevant data was extracted and processed. Moving on to Chapter 4, we dive deep into the experiment's results, analyze the data, and discuss its relevance. Finally, the thesis is concluded in Chapter 5.

# Chapter 2

## Background

This thesis investigates the impact of TCP BBR and TCP Cubic on performance metrics, i.e., delay and throughput. The study involves running these congestion control algorithms in Docker containers configured with the Kubernetes cluster. To evaluate the algorithms, we will use three networking tools, namely `iperf3` [12], the `ss` (socket statistics) tool [13] and `netem` [14]. We create a virtualized environment to generate networking traffic and compare it with a non-virtualized environment. This study interests Karlstad University [15] and Red Hat [16]. Karlstad University is involved in researching the development of transport and application layer protocols, while Red Hat is active in cloud computing and enterprise areas.

The following sections describe Docker containers, Kubernetes, TCP, and the different congestion control algorithms used in this project. Section 2.1 introduces the basics around TCP and its congestion control. Section 2.2 and Section 2.3 explain the two TCP congestion controls used in this project, Cubic and BBR. Section 2.6 and Section 2.7 explain Docker containers and Kubernetes and how they are used together, both used to build our experiment.

## 2.1 TCP and its Congestion Control

The Transmission Control Protocol (TCP) is a reliable transport-layer protocol used in communication networks [17]. It achieves reliability by initializing a connection between hosts and maintaining certain status information to detect damaged, lost, or duplicated data [3]. TCP maintains information such as sequence numbers and window sizes. Sequence numbers detect if the transmitted data has arrived intact without any faults. If a positive acknowledgment (ACK) is received from the receiver, no data is re-transmitted. However, if the ACK is not received within a timeout interval, the data is re-transmitted [3].

Congestion occurs when a network becomes overloaded, which leads to packets being discarded or delayed in TCP. When packets are not acknowledged, the data is re-transmitted, leading to more congestion. Therefore, reasonable congestion control is necessary, and the information maintained about window sizes plays a crucial role in managing congestion. There are two types of window sizes used in TCP: sender and receiver windows. The amount of data that can be sent is determined by how much data the receiver can receive and how much data the sender can send [18]. The receiver window size is advertised together with the acknowledgments. However, the sender-receive window is more complex, as it has to consider the receiver window size. The congestion window (cwnd) determines the sender's flow control. It is based on the window sizes and network conditions. A small cwnd leads to little data being transmitted, meaning underutilized bandwidth. If the cwnd is big, large amounts of data are being transmitted, which can lead to congestion. Therefore, a good TCP congestion control algorithm is needed to ensure good throughput without any congestion occurring.



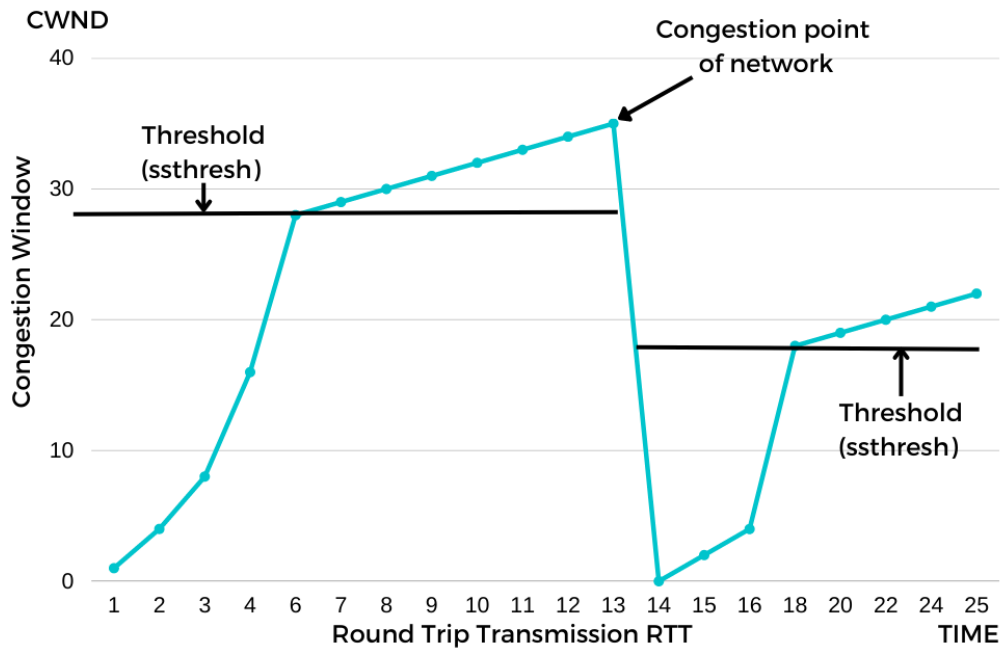


Figure 2.1: Visual presentation of congestion window [1]

As Figure 2.1 illustrates, the congestion window increases with a slow start-up to the threshold value; before hitting the threshold value, it increases faster since it knows the network can handle that transmission speed. After the threshold value, it slows in and increases in smaller increments to get maximal flow without causing packet loss. When the congestion point of the network has been hit, the process restarts and builds up to a new threshold value to prod the network for its maximum flow rate [1].

### 2.1.1 Maximum Transmisson Unit

The Maximum Transmission Unit (MTU) is a term used in TCP that has a significant impact on congestion control and data transmission over networks. This parameter can be modified on different network devices [19], and it controls the maximum size of data that can be sent over a network without getting divided into smaller packets. If a packet is larger than the MTU, it gets divided into smaller packets, transmitted over the network, and then reassembled at the receiver's end.

Fragmentation can cause network inefficiencies and potential delays as it requires reassembly at the receiver's end. This can have a significant impact on congestion control, as it leads to the transmission of more packets that may get lost in flight, which in turn leads to re-transmissions and congestion.

The `cwnd` determines the amount of data that can be sent before waiting for an acknowledgment. Therefore, the MTU size also has an effect on the `cwnd`. A larger MTU size may result in faster growth in the `cwnd` as more data can be sent until congestion occurs.

## 2.2 Cubic

Cubic is a widely-used TCP congestion control method that limits the data transfer rate over a network to prevent congestion. This algorithm has been the default congestion control algorithm in many operating systems, including Linux, since its introduction in 2006 [20].

Cubic regulates data transmission using a window-based technique, gradually increasing the sending rate until it detects packet loss in the network. The algorithm calculates the transmitting rate using a cubic function based on the number of packets in flight and the predicted round-trip time. When packet loss is detected, the algorithm reduces the transmission rate to prevent future loss. After a reset of transmission, it

quickly accelerates to reach the previously measured congestion point and then slows down gradually. Figure 2.2 illustrates the window, with the arrow indicating the point where the algorithm slows down as it approaches the previously measured maximum point [2].

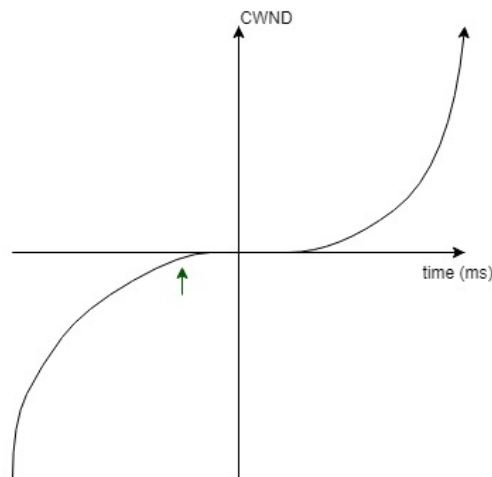


Figure 2.2: Illustration of TCP Cubic on its growth and packet loss function [2]

TCP Cubic has been demonstrated to function well in various networks, including wired and wireless networks, and it is especially effective in high-bandwidth, low-delay networks. Many studies have been conducted, and the algorithm has been shown to outperform other TCP congestion control algorithms, such as TCP Reno and TCP Vegas, in many scenarios [21].

TCP Cubic is a widely used congestion control algorithm for TCP in various network environments. However, in a virtualized setup, it may not perform as well due to the additional layers of abstraction and overhead introduced by virtualization. These factors can affect the accuracy of network measurements and the responsiveness of congestion control algorithms, such as TCP Cubic. Additionally, predicting and responding to network congestion can be more challenging due to workloads moving between virtual machines and hosts. Therefore, TCP Cubic may not always be the best choice for virtualized environments where minimizing virtualization overhead is

critical. TCP BBR, which stands for Bottleneck Bandwidth and RTT, is an alternative TCP congestion control algorithm that may be more appropriate for virtualized settings.

## 2.3 BBR

The early days of the Internet were characterized by loss-based congestion control algorithms. In 2016, Google introduced a new TCP congestion control algorithm called BBR, ten years after the introduction of Cubic. BBR is designed to decrease the sending of packets when lost packets are detected. However, networks have evolved, and we now have much higher bandwidth than before, and it is still increasing [22].

BBR estimates the available network bandwidth and round-trip time (RTT) using a model-based approach. This makes it more accurate and effective than other congestion control algorithms. BBR monitors the packet delivery rate and RTT, and a cubic function is used to manage the congestion window size. The function uses estimated delivery rate and RTT from the monitored traffic. BBR also uses pacing to reduce packet loss and increase efficiency. By correctly calculating and monitoring the available bandwidth and RTT, BBR achieves better transmission rates and lower latency than other algorithms. Its model-based approach makes it adaptive to fluctuations and changes in the network, making it optimal in networks with high variety [6].

TCP BBR is supported by a wide range of operating systems and network applications. However, its implementation and deployment can be complex. Unlike Cubic, BBR is not a standard for most operating systems. Furthermore, BBR might not be appropriate for many networking scenarios and may require tuning to achieve more optimal performance than other algorithms. It also requires modification and coordination across the network stack [23].

BBR is now being used in various network applications, including file transfers and real-time multimedia streaming such as YouTube [24]. Its congestion control is

ideal for applications that require high throughput and low latency. Therefore, BBR might apply to cloud computing and virtualization, where effective data processing and transmission are essential. TCP BBR is also projected to be crucial in developing networking technologies such as the Internet of Things (IoT) and 5G networks, which demand efficient and dependable transfer of data [25].

## 2.4 Virtualization

Virtualization is a common method of utilizing physical computer hardware for maximum efficiency. It is frequently used in cloud computing where servers can be optimized to use their entire capacity for multiple tasks instead of running idle. The way virtualization works is by creating an abstract layer over the actual hardware to simulate multiple containers or virtual machines. Both of these are similar in that they isolate a specific task and run it as a self-contained unit [26].

The benefits of using virtualization are many. It saves resources, is easier to manage, has minimal downtime, and is faster to set up newly acquired hardware into a virtual environment. Prior to virtualization, companies set up servers by having one computer or server running one operating system used for one task. Nowadays, with virtualization, one server can run multiple instances of virtual machines or containers, which utilizes more of the hardware and saves resources.

Downtime is reduced since the instances of both virtual machines and containers can easily be reinstalled, and most companies have spares, so work can continue with minimal downtime by just swapping out an inactive instance for the malfunctioning instance. Furthermore, the instances are normally minimalistic, containing only the necessary functions to perform their intended tasks, leading to easier re-installation or replacement if the need arises for replacement of an instance. This also leads to faster replacement if new hardware is acquired since the instances are designed to be almost

plug-and-play types, which makes them faster to get going. This also leads to easier management since all the machines are running the same type of instances, which makes them easier to control and manipulate if necessary. [10]

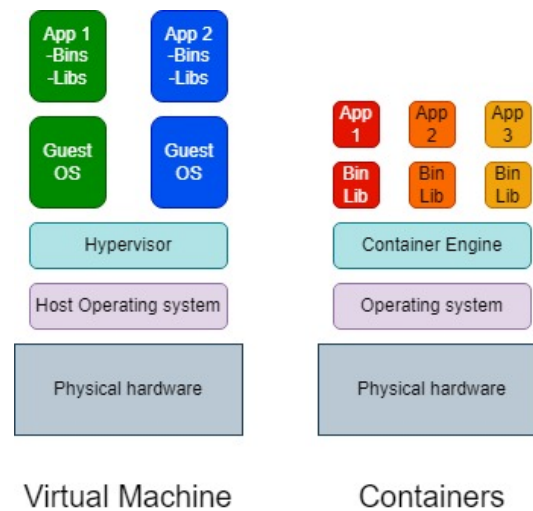


Figure 2.3: Illustration the difference between virtual machines and containers

## 2.5 Virtual Machines

Virtual machines are virtualized environments that operate as a single instance of the underlying hardware. This separates them from both the underlying and other virtualized environments, providing more security between instances. This means that if one virtual machine malfunctions, it will not affect other virtual machines. Additionally, running riskier tasks on virtual machines is safer than running them on the host environment. Virtual machines can be installed on any platform, without limitations like those of containers. For example, you can run a Linux distribution on a Windows platform or run a Windows machine inside a macOS [27].

## 2.6 Docker Containers

Docker container technology was first introduced in 2013, and it is a virtualization technology that allows software applications to be created and managed in isolated environments [8]. This technology uses containerization, which enables multiple separate environments to run on a single host operating system, making it faster and more efficient than virtual machines. Therefore, Docker containers are more suitable for our experiment since the overhead is lower than that of virtual machines [28].

Docker containers consist of several components, including a Docker file, a Docker image, a Docker registry, and a Docker engine. These components are used to create a Docker container, but the Docker image is where all the application code, runtime, and libraries are packaged [29]. In our experiment, networking tools such as `iperf3` are packaged into a Docker image.

Despite the numerous benefits of Docker containers, they also have some drawbacks, such as security risks, complexity, and network issues [30]. Since Docker containers often come in significant numbers, managing them can be challenging. Also, resource constraints can be an issue when running multiple containers on a single host machine. In many scenarios, containers need to communicate with each other, and network issues and difficulties may occur. Therefore, it is essential to follow best practices like image size optimization and container orchestration [31]. Image size optimization involves reducing the size of the Docker images to reduce storage and transport costs, and the image should only include the necessary libraries and code. Container orchestration tools like Kubernetes become necessary when running multiple containers.

## 2.7 Kubernetes

Kubernetes has become the de facto standard for container orchestration in recent years. It is widely used by enterprises worldwide, and companies like Red Hat contributed

to the project. Kubernetes, originally developed and designed by engineers at Google, is an open-source container orchestration platform that automates many processes in scaling containerized applications [16]. The main benefits of using Kubernetes are the features enabling automatic scaling, cluster maintenance, self-reparation, and management. The scalability also makes it a popular choice in the cloud computing and data center industries.

The Kubernetes architecture is made up of many components that work together. The main components are:

1. **Pod:** A Pod is a group of one or more containers. They share network resources and storage. Pods are the smallest and most common unit in a Kubernetes Cluster. A Pod only runs one container. However, it can run multiple containers that need to work together [32]. As seen in Figure 2.4, Pods are deployed inside the Nodes.
2. **Service:** The Service component in Kubernetes enables access and communication with the containers running in a Pod. It makes Pods available on the network so other components and clients can interact with it. It provides an IP address and a DNS name to the set of pods in a Node, creating an abstraction [33] over the Worker Machines as seen in Figure 2.4.
3. **Nodes (Worker Machines):** Nodes, also called Worker machines, are managed by the Control Plane; the Nodes contain services necessary to run the Pods, such as the Kubelet, which reads container manifests and ensures the containers are running [16].
4. **Control Plane:** The Control Plane is the head of the cluster; it manages the Nodes and Pods in a cluster. It comprises components that provide scheduling, self-reparation, and security to the Kubernetes cluster [34]. In Figure 2.4, we can see the components inside the Control Plane. Control Planes typically manage many Worker Machines spread across many physical or virtual machines. To ensure



high availability, Kubernetes clusters in enterprise environments often consist of multiple control planes running on multiple machines.

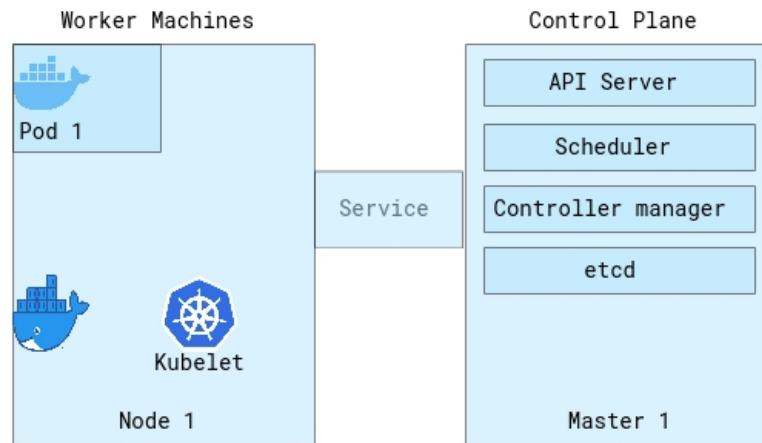


Figure 2.4: A figure of a simple Kubernetes Cluster.

One of Kubernetes' essential characteristics is its networking capabilities, which allow communication between containers running on various nodes and access to services running within the cluster. However, the complexity becomes much higher when looking into how networking works in such an environment. Each component adds overhead to the networking, which can result in increased latency and slow performance. The complexity leads to difficulty troubleshooting network issues, and network optimization requires tuning [35].



# Chapter 3

## Experiment

### 3.1 Introduction

In this chapter, we will explain the experiment in greater detail. We will start with the environment and then explain how the tools were used to provide the results that will be discussed in the next chapter.

#### 3.1.1 Environment

At Karlstad University, the environment is set up on one of their local machines with two virtual machines running for the experiment. These virtual machines are called "server" and "client". The server is responsible for the Kubernetes cluster and running a pod to simulate the necessary parts required for running the tests. The server has two main functions in our tests: first, to serve as a platform for running traffic to an endpoint inside the pods, and second, to act as an endpoint outside the cluster, referred to as "Outside-Cluster". When the traffic passes through the Kubernetes cluster and into the pod, it is called "Inside-Cluster". The illustration in Figure 3.1 shows how the machines communicate inside the experiment.

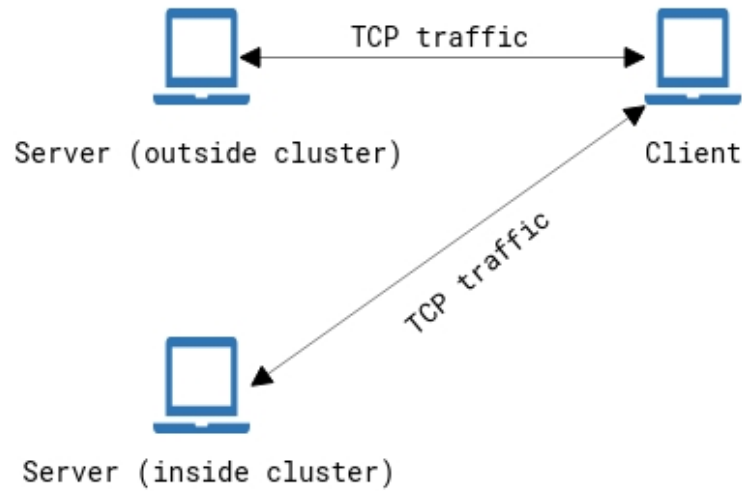


Figure 3.1: Experiment Setup.

### 3.1.2 SSH

For the experiment, we set up a secure environment using Secure Shell (SSH) [36]. This tool helped us to remotely connect to a computer or server through the network, allowing us to conduct the experiment remotely. Additionally, it provided more security by ensuring that the experiment could not be tampered with, which could potentially have led to inaccurate results.

## 3.2 Network Tools

In this experiment, we utilized Iperf3 [12] and socket statistics (ss) [13] to measure and monitor the network traffic. Additionally, we employed netem [14] as a networking tool to introduce delay, packet loss, and other modifications to the network.

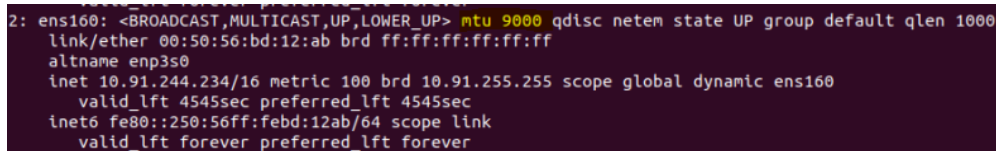
### 3.2.1 Iperf3

In this experiment, we are using iperf3 to generate TCP traffic between the Client and Server. Although iperf3 can be used to monitor the traffic, it only does so once a second and does not provide detailed network traffic monitoring. Therefore, in this experiment, we will mainly use iperf3 to generate traffic between the Client and Server. We will also use the outputs, but to a much lesser extent than the ss tool.

#### Iperf3 parameters

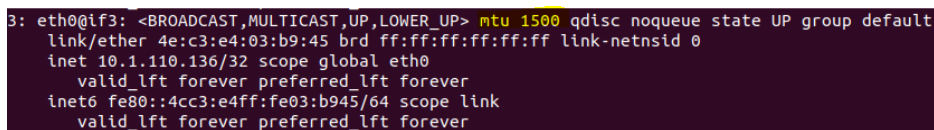
In the context of network performance testing with iperf3, several configurable parameters exist. The Maximum Segment Size (MSS), an important parameter, was modified for this experiment. MSS is calculated based on the Maximum Transmission Unit (MTU), and it represents the maximum amount of data that a segment can hold, excluding the TCP/IP headers [19].

The reason behind the change in the MSS parameter was that, outside the Kubernetes cluster, the MTU value is 9000 bytes, while inside the cluster is only 1500 bytes.



```
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc netem state UP group default qlen 1000
    link/ether 00:50:56:bd:12:ab brd ff:ff:ff:ff:ff:ff
    altname enp3s0
    inet 10.91.244.234/16 metric 100 brd 10.91.255.255 scope global dynamic ens160
        valid_lft 4545sec preferred_lft 4545sec
    inet6 fe80::250:56ff:febd:12ab/64 scope link
        valid_lft forever preferred_lft forever
```

Figure 3.2: The MTU of our machine outside the Kubernetes cluster



```
3: eth0@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 4e:c3:e4:03:b9:45 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.1.110.136/32 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::4cc3:e4ff:fe03:b945/64 scope link
        valid_lft forever preferred_lft forever
```

Figure 3.3: The MTU of our machine inside the Kubernetes cluster

In Figure 3.4, the MSS is 8948 bytes, which results in an average of 471 Mbits/sec. On the other hand, in Figure 3.5, the MSS is only set at 1448 bytes, resulting in an

```

Control connection MSS 8948
Time: Wed, 30 Aug 2023 21:03:58 GMT
Connecting to host 10.91.244.234, port 5201
Cookie: np75qqndwyxrirlcwb6h3viuocolnd5n7kx
TCP MSS: 8948 (default)
[ 5] local 10.91.244.201 port 50002 connected to 10.91.244.234 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 5 seconds, 60 second test, tos 0
[ ID] Interval      Transfer    Bitrate    Retr    Cwnd
[ 5]  0.00-5.00    sec  192 MBytes  322 Mbits/sec  0    6.71 MBytes
[ 5]  0.00-5.00    sec  285 MBytes  478 Mbits/sec  0    6.49 MBytes
[ 5]  5.00-10.00   sec  269 MBytes  451 Mbits/sec  0    6.21 MBytes
[ 5] 10.00-15.00   sec  284 MBytes  476 Mbits/sec  0    6.52 MBytes
[ 5] 15.00-20.00   sec  285 MBytes  478 Mbits/sec  0    6.52 MBytes
[ 5] 20.00-25.00   sec  285 MBytes  478 Mbits/sec  0    6.50 MBytes
[ 5] 25.00-30.00   sec  270 MBytes  453 Mbits/sec  0    6.69 MBytes
[ 5] 30.00-35.00   sec  285 MBytes  478 Mbits/sec  0    6.50 MBytes
[ 5] 35.00-40.00   sec  285 MBytes  478 Mbits/sec  0    6.52 MBytes
[ 5] 40.00-45.00   sec  269 MBytes  451 Mbits/sec  0    6.50 MBytes
[ 5] 45.00-50.00   sec  285 MBytes  478 Mbits/sec  0    6.57 MBytes
[ 5] 50.00-55.00   sec  285 MBytes  478 Mbits/sec  0    6.54 MBytes
[ 5] 55.00-60.00   sec  285 MBytes  478 Mbits/sec  0    6.50 MBytes
-----
Test Complete. Summary Results:
[ ID] Interval      Transfer    Bitrate    Retr
[ 5]  0.00-60.00    sec  3.29 GBytes  471 Mbits/sec  0
[ 5]  0.00-60.05    sec  3.29 GBytes  471 Mbits/sec
sender
receiver

```

Figure 3.4: The MSS of our machine outside the Kubernetes cluster

average of 243 Mbits/sec. Unfortunately, we were unable to modify the MTU inside the cluster to 9000 bytes. Therefore, we agreed with our supervisor to conduct the tests using the lower MSS by adjusting the parameters when running `iperf3`.

### 3.2.2 Socket statistics tool

We can use the `ss` tool to gather detailed information about the networking traffic generated through `iperf3`. This tool provides data about the round-trip time (RTT), congestion window size (`cwnd`), and the slow start threshold for the congestion window (`ssthresh`).

### 3.2.3 Netem

The tool called `netem` is utilized in the experiment to introduce both delay and packet loss. The client and server are present within the same network, which results in a relatively small RTT and no packet loss. Due to this, the `cwnd` tends to increase at a rapid pace, and subsequently, congestion can occur instantaneously. To replicate a

```
Control connection MSS 1448
Time: Wed, 30 Aug 2023 21:02:22 GMT
Connecting to host 10.91.244.234, port 30445
Cookie: gzcelav5mfr2ox4lng5y63cbwgyfkvfi7kql
TCP MSS: 1448 (default)
[ 5] local 10.91.244.201 port 50002 connected to 10.91.244.234 port 30445
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 5 seconds, 60 second test, tos 0
[ ID] Interval      Transfer    Bitrate    Retr  Cwnd
[ 5]  0.00-5.00    sec      140 MBytes  235 Mbits/sec    0   3.43 MBytes
[ 5]  0.00-5.00    sec      149 MBytes  250 Mbits/sec    0   3.43 MBytes
[ 5]  5.00-10.00   sec      140 MBytes  235 Mbits/sec    0   3.41 MBytes
[ 5] 10.00-15.00   sec      149 MBytes  250 Mbits/sec    0   3.44 MBytes
[ 5] 15.00-20.00   sec      139 MBytes  233 Mbits/sec    0   3.22 MBytes
[ 5] 20.00-25.00   sec      149 MBytes  250 Mbits/sec    0   3.37 MBytes
[ 5] 25.00-30.00   sec      139 MBytes  233 Mbits/sec    0   3.23 MBytes
[ 5] 30.00-35.00   sec      148 MBytes  247 Mbits/sec    0   3.40 MBytes
[ 5] 35.00-40.00   sec      140 MBytes  235 Mbits/sec    0   3.17 MBytes
[ 5] 40.00-45.00   sec      148 MBytes  247 Mbits/sec    0   3.30 MBytes
[ 5] 45.00-50.00   sec      149 MBytes  250 Mbits/sec    0   3.42 MBytes
[ 5] 50.00-55.00   sec      139 MBytes  233 Mbits/sec    0   3.18 MBytes
[ 5] 55.00-60.00   sec      149 MBytes  250 Mbits/sec    0   3.31 MBytes
-----
Test Complete. Summary Results:
[ ID] Interval      Transfer    Bitrate    Retr
[ 5]  0.00-60.00   sec    1.69 GBytes  243 Mbits/sec    0
[ 5]  0.00-60.05   sec    1.69 GBytes  242 Mbits/sec
sender
receiver
```

Figure 3.5: The MSS of our machine inside the Kubernetes cluster

normal connection, we included a delay of 10 ms - 50 ms to the experiment setup 3.1.1. We also added packet loss ranging from 0.0001% to 3% to compare how a model-based congestion control handles packet loss compared to that of a delay-based congestion control.

### 3.3 Docker

As mentioned earlier in the background, Docker was chosen as the primary option for the experiment. One of the reasons behind this was that Docker is easily scalable and can be used with minimum resources. This feature was perfect for the experiment as it caused minimal interruption during the actual experiment. Therefore, the container created for the purpose of the experiment contained only the bare minimum, along with the necessary tools that were used.

We needed to create a Docker image that has `iperf3` installed. To achieve this, we used Alpine as our base operating system image, which we pulled from Docker Hub. This was done because Alpine has a smaller image size compared to Ubuntu, which

means fewer resources are used. After adding the base image, we installed `iperf3` using a command that makes up our Docker image. The complete code for the Docker image file is available in Appendix A.

To test the image, we first executed it manually on a local machine. Once the image was up and running, we tested if we could host a server using `iperf3`. To do this, we accessed the terminal of the container and set up a server using a simple `iperf3` command [12].

Finally, after all the testing was done, we uploaded the Docker image to the Docker Hub repository. This allows anyone to easily pull the image and use it in their Kubernetes Cluster.

### 3.4 Kubernetes

There are several versions of Kubernetes available since it is an open-source software. We used Microk8s [37] on our testing machines, but before deploying a Kubernetes Cluster on our testing machines, we first used Minikube [38] and Kind [39] on our local machines. We could test configurations and verify whether our Docker image could be deployed inside a Pod by deploying a local Kubernetes cluster inside our local machines. The main difference between the Kubernetes implementations we tried on our local machine is that Minikube creates a Virtual Machine, essentially a single-node Kubernetes Cluster. In contrast, Kind makes a Docker Container for the cluster [40]. We chose Minikube because we encountered issues with Kind and because Minikube was the more popular option, which meant more documentation was available for its use.



### 3.4.1 Kubectl

Kubectl [41] is a command-line tool used to manage Kubernetes clusters, which allows logging, deployment, and management through the terminal.

### 3.4.2 Minikube

We manually followed the documentation for Minikube and configured a single-node Kubernetes Cluster on our local machine [38]. To build our Docker image file with `iperf3` installed, we used the Minikube Docker ENV, allowing us to use Docker CLI to the Docker Engine inside the Virtual Machine created by Minikube. Docker CLI provides many useful commands and is essential in checking the Docker container status. It also enabled us to build the Docker container locally using the Docker file in Appendix A. However, we faced issues. We pushed the Docker file to the Docker Hub repository, enabling us to pull the image every time we installed a Docker container with `iperf3`.

We then created a single Pod with the Docker container inside, containing `iperf3`. To generate the `iperf3` server, we used a combination of Kubectl and Docker CLI to issue commands inside the container, which is inside a Kubernetes Pod. With the server up and running inside the cluster in Minikube, we had to find a way to reach it outside the cluster.

#### Minikube Tunnel and Service

To get outside traffic into a Kubernetes cluster, a service [33] is needed, it is the component that allows access and communication to the application inside the pod. There are two types of services that are most commonly used in Kubernetes [42]:

1. **LoadBalancer:** A LoadBalancer exposes the service to the Internet, it is done by assigning the Service with an external IP address. The IP address is then used

by the outside world to talk to the application inside the container. When the IP address gets requests, it goes through the service which then forwards it to the ports connected to the Pods, in our case we have a single Pod containing the server.

2. **NodePort:** A NodePort does not expose the service to the Internet, it opens a port and whenever traffic is sent to this specific port, it gets forwarded to the Service. The Service then forwards the traffic to our Pods just like a LoadBalancer.

The service we used in our Minikube testing environment was the LoadBalancer while NodePort was used in our actual testing machines. However, when using a LoadBalancer in Minikube, it needs to be exposed to the Internet by using a tunnel. Minikube tunnel is a command that creates a network route on the host to the Service of the cluster [33]. By creating a LoadBalancer Service connected to our Pod while having the tunnel enabled automatically gave it an external IP-address. This is what our Service looked like in Minikube:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
iperf3-service	LoadBalancer	10.109.198.159	10.109.198.159	8080:31260/TCP	4s

In the PORT(S) column, you can find the ports for the service and the Pod. Port 31260 is where the service accepts traffic from, and then it sends the traffic to port 8080 where the Pod is located. By having an iperf3 server inside the Pod that listens to port 8080, we can send iperf3 traffic from outside the cluster by sending it to the service's external IP-address on port 31260. The traffic then gets sent to port 8080 where our iperf3 server is located. Each component has some effect on the network traffic as it passes through. Therefore, we will run multiple tests in our testing machines, both in a normal connection using TCP BBR and TCP Cubic, and in a connection where the server is hosted inside a cluster.

# Chapter 4

## Results

### 4.1 Introduction

In this chapter the results of our experiment and evaluation will be presented. TCP BBR and TCP Cubic will be matched against each other by discussing the measurements extracted from the experiments, considering the usage inside and outside a Kubernetes cluster.

### 4.2 Inside a Cluster

This section aims to compare the usage of TCP BBR and TCP Cubic inside a Kubernetes cluster. It is done by using the networking tools in Section 3.2 inside a deployment of the Kubernetes Cluster in Section 3.4.

#### 4.2.1 TCP BBR

The following text contains the results of TCP BBR experiments, which are displayed in the figures below. Figure 4.1 displays the recorded value of the `cwnd` with a packet loss of 0.001%. The graph shows that the `cwnd` quickly peaks to an approximately

6200 MMS before hitting the first congestion point, which reduces the value to 4300 MMS. The `cwnd` value remains stable around this number for the rest of the test, with occasional variations. The slow-start threshold remains relatively stable throughout the test after the initial congestion point, remaining at 180 MMS for the entirety of the test.

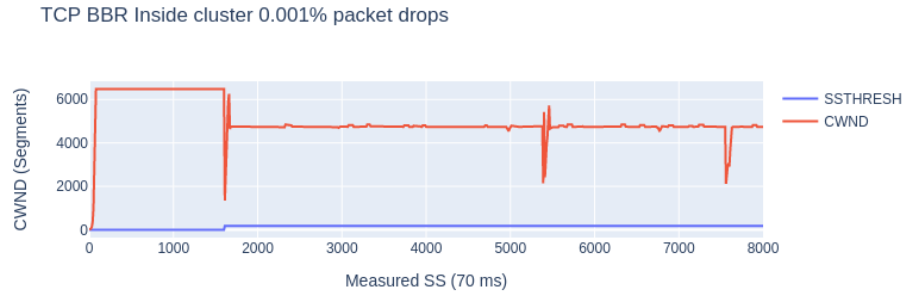


Figure 4.1: Graph over TCP BBR inside the cluster, showing `cwnd` and `ssthresh`, 0.001% packet drops

When we increased the simulated package drop rate to 3% in `netem`, we observed interesting changes in the system's behavior. Figure 4.2 indicates that the `cwnd` started fluctuating more rapidly due to the effect of dropped packages. Additionally, the slow start threshold quickly increased to the value of 2222 MMS and then remained constant for the remainder of the test, indicating that the system was adapting to handle the increased package drop rate.

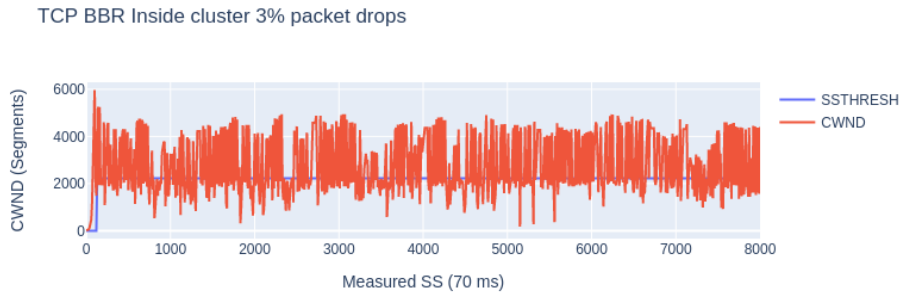


Figure 4.2: Graph over TCP BBR inside cluster, showing cwnd and ssthresh, 3% packet drops

In Figure 4.3, the RTT is depicted. The graph shows the RTT with a package drop rate of 0.001%. The results are stable around 50.2 ms with a deviation of 0.1 ms. The larger spikes on the RTT graph correspond to those on the cwnd, indicating the impact of a packet drop.

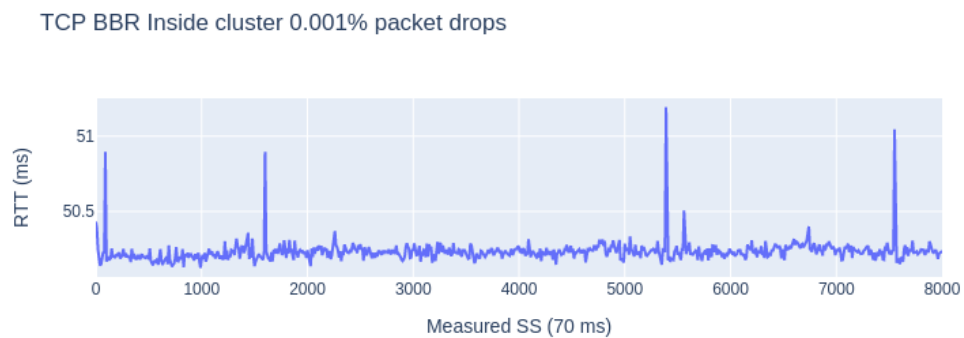


Figure 4.3: Graph over TCP BBR inside cluster, showing RTT with 0.001% packet drops

When increasing the package drop rate to 3% in Figure 4.4 we get the following results, RTT goes from a steady 50.2 ms on 0.001% to an unstable value that ranges from 50.5 ms to 51.2 ms with a few spikes that exceed that range.

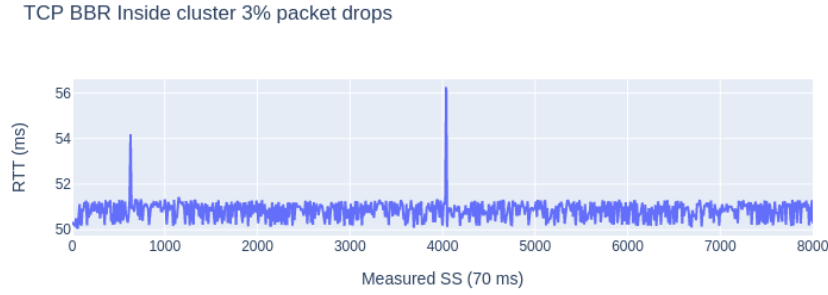


Figure 4.4: Graph over TCP BBR inside the cluster, showing RTT with 3% packet drops

### 4.2.2 TCP Cubic

Below can the graphs of TCP cubic be seen, first graph 4.5 shows the cwnd and slow start threshold with a simulated package drop rate of 0.001%. As shown on the graph, the cwnd follows the normal movement of the protocol, rapidly increasing to 4400 MMS before the first drop, then following up to 3074 MMS for a longer duration before the second drop. The slow start threshold follows the graph down to the 3rd drop, the part from the start where it stays at zero, and after the third drop, they both show the standard curvature of TCP cubic graphs.

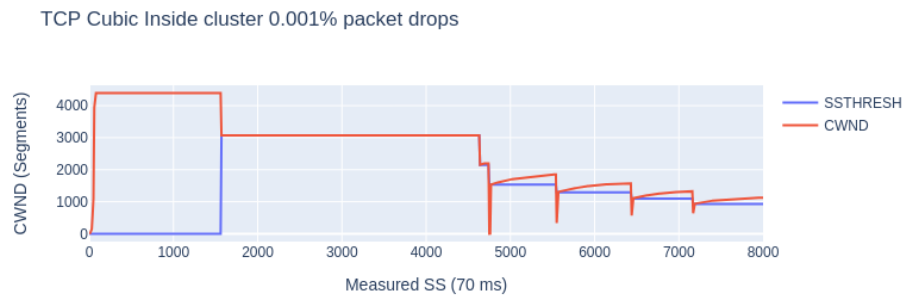


Figure 4.5: Graph over TCP Cubic inside cluster, showing cwnd and ssthresh with 0.001% packet drops

When analyzing the graph depicted in Figure 4.6, which displays the cwnd and slow start threshold with a higher packet drop rate, we observe a more frequent and erratic occurrence of drops in the graph. As the package drops increase, the MSS become much lower, ranging from about 4000 MSS as previously seen to a range of 2 MSS up to a maximum of 23 MSS, with a few jumps exceeding this range. The slow start threshold continues to follow its predicted curvature in this scenario, landing at the beginning of the drop in order to maintain high bandwidth.

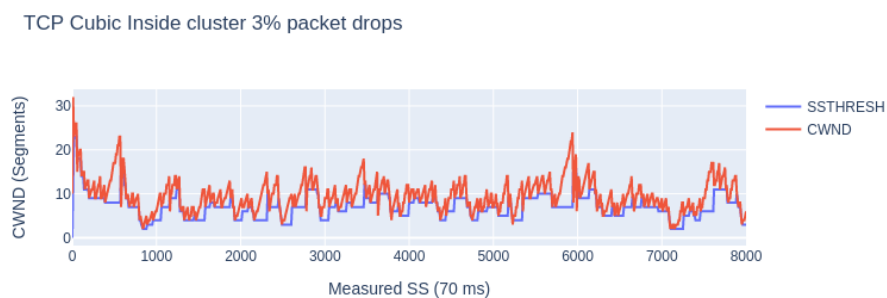


Figure 4.6: Graph over TCP Cubic inside cluster, showing cwnd and ssthresh with 3% packet drops

When examining the round trip time that TCP Cubic experienced, as shown in Figure 4.7, the measurement is relatively consistent, ranging from 50.2 ms to 50.4 ms, with only a few minor deviations from this range. The graph also displays smooth curves, indicating that the frequency of changes is low, which is typical for low package drops.

When comparing these results to those of the 3% package drops shown in Figure 4.7, it is noticeable that the measurements are higher and more frequent, indicating a greater impact from package loss. The measurements range from 50.3 ms to 58.8 ms, with one measurement spiking quite high at 84.2 ms. This spike is an outlier and does not represent the general trend observed in the data. Overall, the results clearly illustrate the impact of package loss on round-trip time and highlight the importance of minimizing

package drops to ensure optimal network performance.

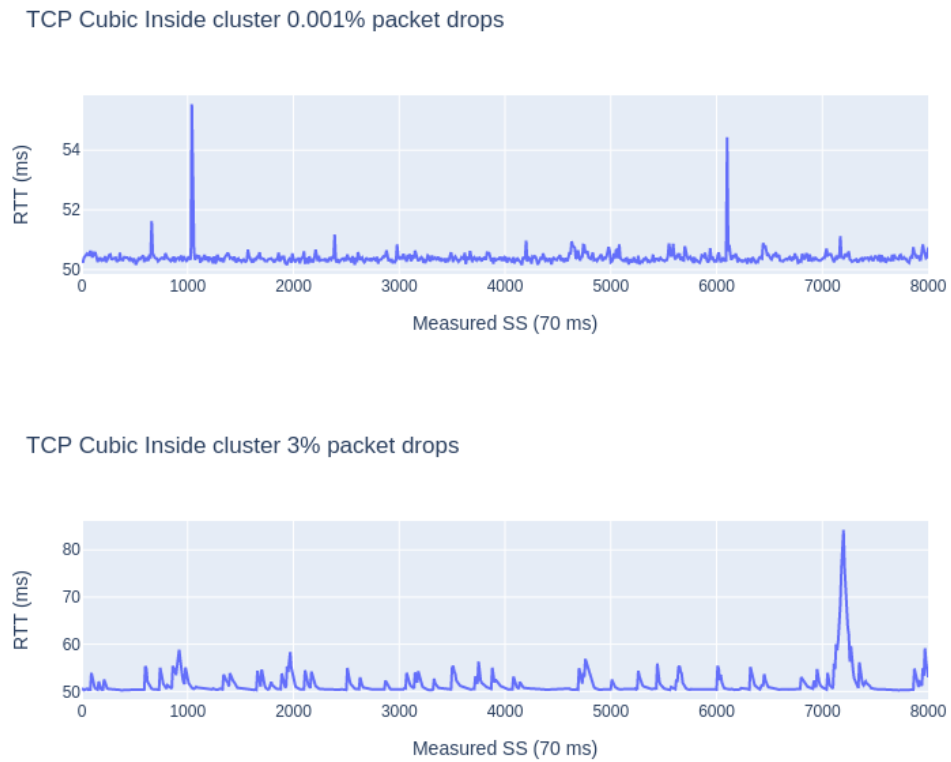


Figure 4.7: Graph over TCP Cubic inside cluster, showing RTT with 0.001% and 3% packet drops



## 4.3 Outside a Cluster

### 4.3.1 TCP BBR

TCP BBR appears to have a minimal effect when tested outside the cluster, as indicated in Figure 4.8. In comparison to the graph inside the cluster, as shown in Figure 4.3, both graphs have a starting point of 6444 MMS that remains constant for the first part of the test. However, two drops can be seen before the drop that affected the slow start threshold. Even after that drop, the curves are similar to the ones inside the cluster.

In contrast, the slow start threshold is much higher outside the cluster. The value jumps from 180 MMS inside to 2248 MMS outside.

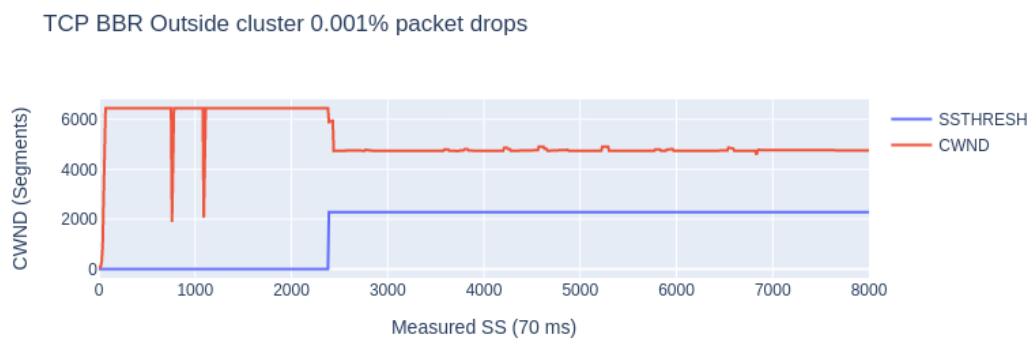


Figure 4.8: Graph over TCP BBR outside cluster, showing cwnd and ssthresh with package loss of 0.001%

When looking at the graphs in Figures 4.2 and 4.9, showing the higher drop rates its noticeable that TCP BBR is equally affected inside the cluster as outside the cluster. The graph below shows TCP BBR outside the cluster with a package loss of 3%, equal to the inside graph with the same conditions.

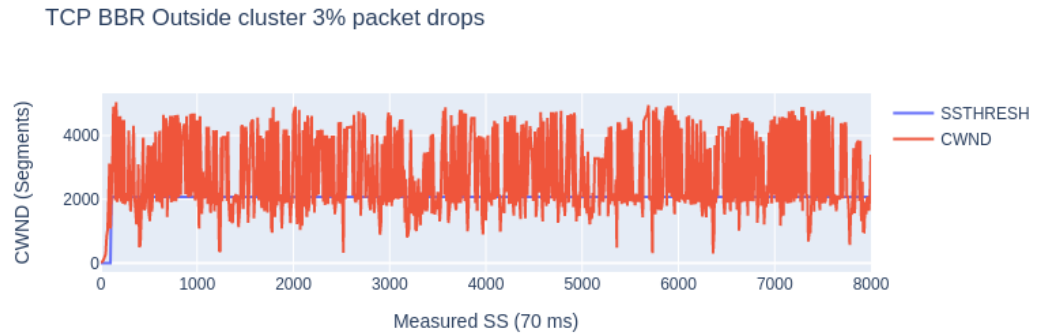


Figure 4.9: Graph over TCP BBR outside cluster, showing cwnd and ssthresh with package loss of 3%

In the analysis of TCP BBR's RTT, it is observed that the RTT outside the cluster is almost the same as the RTT measured inside the cluster. The graphs in Figures 4.10 and 4.3 show that the RTT fluctuates between 50.0 ms to 50.2 ms, which is similar to the RTT measured inside the cluster. Furthermore, the results are consistent for higher packet loss, both inside the cluster (as seen in Figure 4.4) and outside the cluster (as seen in Figure 4.10).

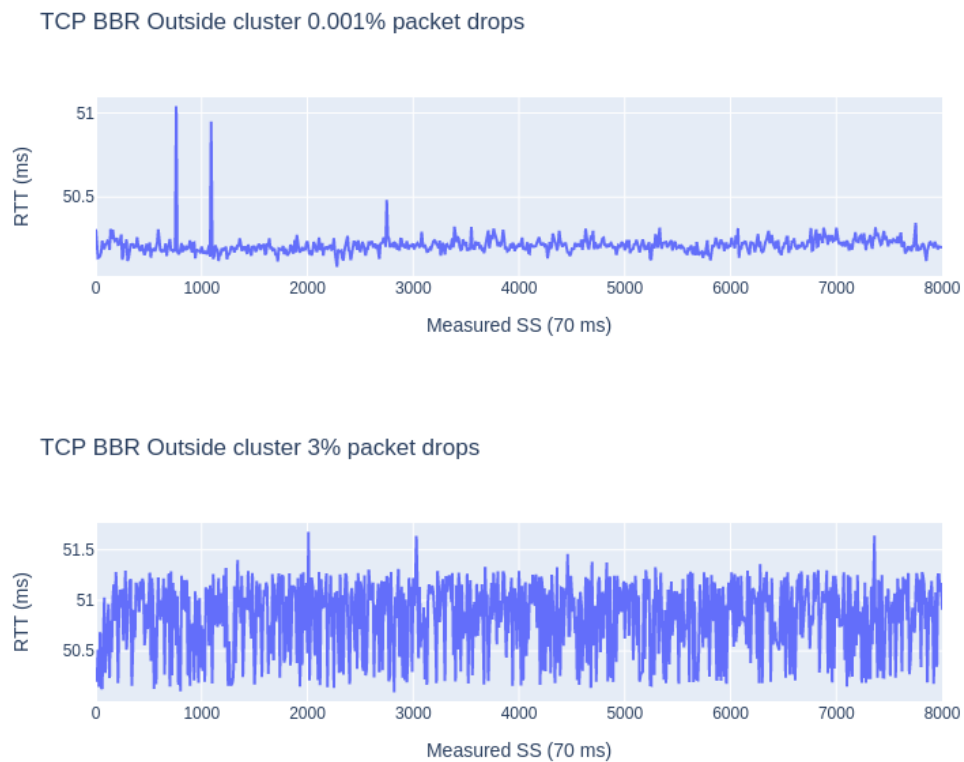


Figure 4.10: Graph over TCP BBR outside cluster, showing RTT with 0.001% and 3% packet drops.

### 4.3.2 TCP Cubic

TCP Cubic exhibits similar behavior in both internal and external clusters, as seen in the graphs in Figure 4.11 and Figure 4.5. The curves follow the same trend with little variation in the measured data. However, the graphs with 0.001% packet drops show some differences between the internal and external clusters. In the internal cluster, the first drop occurs earlier and follows the normal curvature for TCP Cubic. In contrast, the external cluster stays high for a considerably longer time before experiencing the first packet drop. The slow start threshold is slightly higher in the external cluster, but the subsequent drops fall further than in the internal cluster, resulting in less stable throughput outside the cluster.



Figure 4.11: Graph over TCP Cubic outside cluster, showing cwnd and ssthresh with package loss of 0.001% and at 3%

The round trip time for TCP Cubic follows the same trend, as observed from the plot in Figure 4.12., which shows that the value remains roughly the same even with 0.001% package drops. However, there are some differences in the form of a few higher spikes on the outside as compared to the inside as seen in Figure 4.7, while the value still remains around the same, possibly 0.1 higher on the outside. On the other hand, when looking at the 3% package loss graphs on the outside and comparing them to the inside, there is not much difference when considering the overall data.

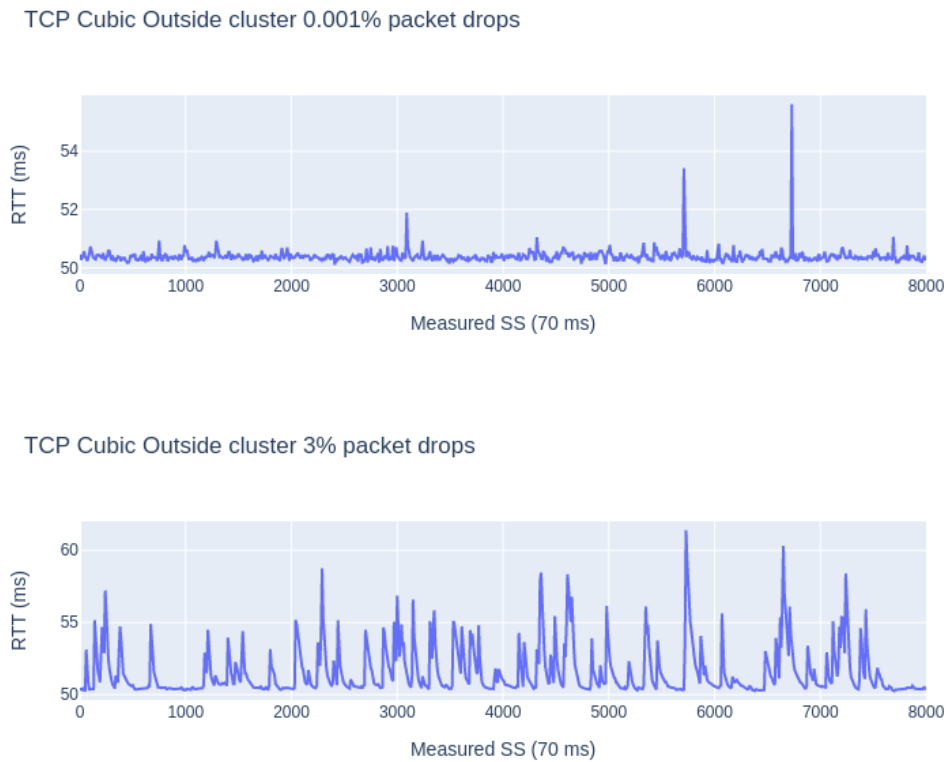


Figure 4.12: Graph over TCP Cubic outside cluster, showing RTT with 0.001% and 3% packet drops

## 4.4 Comparison

Based on the comparison of the graphs presented in Figure 4.13, it is evident that TCP BBR is more efficient in handling interference than TCP Cubic, particularly when the interference level is high. The experiment results demonstrate that while BBR maintains a reasonably high throughput, TCP Cubic slows down significantly to an extremely slow speed. These findings are consistent both inside and outside the Kubernetes cluster, confirming the theoretical understanding that TCP Cubic experiences difficulties in handling a large amount of package loss, while TCP BBR can sustain a good throughput of packages despite the same level of interference.



Figure 4.13: Graphs over TCP BBR and TCP Cubic at maximum interference

As follows from Figure 4.14, we can see that TCP Cubic starts to experience a decline in its throughput when there is a mere 0.1% package drop. Consequently, the throughput is significantly reduced. On the other hand, TCP BBR also experiences an impact due to package drops but maintains a good throughput.



Figure 4.14: Graphs over TCP BBR and TCP Cubic when TCP Cubic starts to falter

The last Figure 4.15 added in this chapter is an overview of the results got per test, it is added as a table to show a compressed version to glance the results and give the ability to compare them.

BBR Inside	0.001%	0.01%	0.1%	1%	2%	3%
CWND	6200 - 4400	5300 - 4900	6200 - 4300	4800 - 2200	4200 - 1800	4100 - 1800
SSThresh	0 - 200	200 - 4000	2300 - 2300	2300 - 4200	2100 - 2100	2100 - 2100
RTT	50.1 - 50.3	50.2 - 50.3	50.2 - 50.3	50.3 - 51.0	50.5 - 51.2	50.6 - 51.3
BBR Outside	0.001%	0.01%	0.1%	1%	2%	3%
CWND	6200 - 4400	6200 - 4400	4300 - 2100	4200 - 1900	4200 - 1900	4100 - 1800
SSThresh	0 - 2100	0 - 200	0 - 300	0 - 2100	0 - 2200	0 - 2000
RTT	50.1 - 50.2	50.1 - 50.2	50.2 - 50.3	50.2 - 50.3	50.2 - 50.8	50.6 - 51.1
Cubic Inside	0.001%	0.01%	0.1%	1%	2%	3%
CWND	4200 - 3100	4100 - 2200	2200 - 60	30 - 17	18 - 8	18 - 5
SSThresh	4200 - 3100	4100 - 2200	2200 - 60	30 - 17	18 - 8	18 - 5
RTT	50.2 - 50.4	50.3 - 50.5	50.4 - 50.9	50.5 - 51.1	50.6 - 52.0	50.8 - 52.6
Cubic Outside	0.001%	0.01%	0.1%	1%	2%	3%
CWND	4300 - 2200	2300 - 1500	2200 - 60	320 - 20	16 - 8	16 - 5
SSThresh	2200 - 1800	1500 - 1200	2200 - 60	26 - 14	15 - 6	11 - 3
RTT	50.2 - 50.4	50.3 - 50.4	50.4 - 50.7	50.5 - 50.9	50.9 - 52.2	51.2 - 53.9

Figure 4.15: Compressed information in table form over all the data collected



# Chapter 5

## Conclusion

This thesis evaluates the performance of two TCP protocols, TCP BBR and TCP Cubic, in a virtual environment of a Kubernetes cluster. The results show that TCP BBR performs better than TCP Cubic in a Kubernetes cluster. The experiment also indicates that running the protocols within a virtual environment has a marginal effect. However, the interference is small enough that it is still worth running servers in a virtual environment.

Based on the results, it can be concluded that TCP BBR performs better than TCP Cubic, especially with an influence of packet loss. Without packet loss, they are not too far from each other, making both viable as transport protocols to use. The experiment also proved that the virtual environment of Kubernetes does not affect the transfer speed enough to be an issue. This is great news considering how many larger servers are currently running Kubernetes and virtual environments in general.

To conclude, the virtual environment within servers does not affect the throughput of protocols TCP BBR and TCP Cubic to any degree that would affect day-to-day usage of the services on the servers. In light of the limited scope of this thesis work, a future experiment involving multiple pods communicating on a larger scale could provide valuable insights into the influence of mixed traffic on the results.



# Bibliography

- [1] Ghassan A. Abed \*, Mahamod Ismail, and Kasmiran Jumari. Exploration and evaluation of traditional TCP congestion control techniques. *Journal of King Saud University - Computer and Information sciences*, 24(2):145–155, 2012. <https://www.sciencedirect.com/science/article/pii/S1319157812000146#f0010>[Accesed: 2023-03-22].
- [2] Josip Lorincz and Zvonimir Klarin. Tcp-based congestion control algorithms, July 2021. <https://encyclopedia.pub/entry/12206>[Accesed: 2023-04-06].
- [3] Jon Postel. TRANSMISSION CONTROL PROTOCOL, September 1981. <https://www.rfc-editor.org/rfc/rfc793.txt> [Accesed: 2023-05-05].
- [4] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP’s Fast Recovery Algorithm, April 2012. <https://www.rfc-editor.org/rfc/rfc6582.txt>[Accesed: 2023-05-05].
- [5] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control Measuring bottleneck bandwidth and round-trip propagation time. *Network Congestion*, 14(5), December 2016. <https://queue.acm.org/detail.cfm?id=3022184>[Accesed: 2023-04-25].
- [6] N. Cardwell, Y. Cheng, S. Hassas Yeganeh, I. Swett, and V. Jacobson. BBR Con-

- gestion Control, March 2022. <https://datatracker.ietf.org/doc/html/draft-cardwell-iccr-g-bbr-congestion-control>[Accesed: 2023-05-06].
- [7] Enlyft. Companies using kubernetes. <https://enlyft.com/tech/products/kubernetes>[Accesed: 2023-05-09].
- [8] Docker. What is a Container? <https://www.docker.com/resources/what-container/>[Accesed: 2023-04-28].
- [9] Kubernetes. Kubernetes, February 2023. <https://kubernetes.io>[Accesed: 2023-04-17].
- [10] IBM. What is virtualization? <https://www.ibm.com/topics/virtualization>[Accesed: 2023-08-15].
- [11] Cubic tcp, September 2022. [https://en.wikipedia.org/wiki/CUBIC\\_TCP](https://en.wikipedia.org/wiki/CUBIC_TCP)[Accesed: 2023-05-09].
- [12] Bruce A. Mah Jeff Poskanzer Kaustubh Prabhu Jon Dugan, Seth Elliott. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. <https://iperf.fr/iperf-doc.php#3doc>[Accesed: 2023-05-05].
- [13] Linux.com Editorial Staff. An Introduction to the ss Command, January 2019. <https://www.linux.com/topic/networking/introduction-ss-command/>[Accesed: 2023-04-09].
- [14] Stephen Hemminger, Fabio Ludovici, and Hagen Paul Pfeifer. NetEm - Network Emulator. <https://manpages.ubuntu.com/manpages/bionic/man8/tc-netem.8.html>[Accesed: 2023-05-10].
- [15] Karin Lundberg. DISCO - DISTRIBUTED SYSTEMS AND COMMUNICATIONS, November 2022. <https://www.kau.se/>

- en/cs/research/research-areas/computer-networking-disco/  
disco-distributed-systems-and-communications[Accesed: 2023-05-07].
- [16] Red Hat. What is Kubernetes?, March 2020. <https://www.redhat.com/en/topics/containers/what-is-kubernetes>[Accesed: 2023-04-02].
- [17] Ibm. TCP/IP TCP, UDP, and IP protocols, March 2021. <https://www.ibm.com/docs/en/zos/2.2.0?topic=internets-tcpip-tcp-udp-ip-protocols>[Accesed: 2023-05-27].
- [18] Kary. Understanding Throughput and TCP Windows, July 2014. <https://packetbomb.com/understanding-throughput-and-tcp-windows/>[Accesed: 2023-04-07].
- [19] How MTU and MSS Affect Networks?, March 2022. <https://www.geeksforgeeks.org/how-mtu-and-mss-affect-networks/>[Accesed: 2023-08-30].
- [20] Matthew Prince. Optimizing Your Linux Stack for Maximum Mobile Web Performance, December 2012. <https://blog.cloudflare.com/optimizing-the-linux-stack-for-mobile-web-per/>[Accesed: 2023-05-02].
- [21] Luis Marrone, Andres Barbieri, and Matt'as Robles. TCP Performance - CUBIC, Vegas Reno, April 2013. <https://www.redalyc.org/articulo.oa?id=638067300005>[Accesed: 2023-05-07].
- [22] Andree Toonk. TCP BBR - Exploring TCP congestion control, February 2020. <https://atoonk.medium.com/tcp-bbr-exploring-tcp-congestion-control-84c9c11dc3a9/>[Accesed: 2024-01-10].

- [23] Patrick Meenan. Optimizing HTTP/2 prioritization with BBR and tcp\_notsent\_lowat, October 2018. <https://blog.cloudflare.com/http-2-prioritization-with-nginx>[Accesed: 2023-05-02].
- [24] Neal Cardwell and Yuchung Cheng. TCP BBR congestion control comes to GCP your Internet just got faster, July 2017. <https://cloud.google.com/blog/products/networking/tcp-bbr-congestion-control-comes-to-gcp-your-internet-just-got-faster>[Accesed: 2023-05-03].
- [25] Josip Lorincz, Zvonimir Klarin, and Julije Oegovic. A Comprehensive Overview of TCP Congestion Control in 5G Networks: Research Challenges and Future Perspectives. <https://www.mdpi.com/1424-8220/21/13/4510>[Accesed: 2023-04-29], Year = 2021, Month = June .
- [26] Preethi Kasireddy. A Beginner-Friendly Introduction to Containers, VMs and Docker, March 2016. <https://www.freecodecamp.org/news/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b/>[Accesed: 2023-08-15].
- [27] "VMWare". What is a virtual machine? . <https://www.vmware.com/topics/glossary/content/virtual-machine.html>[Accesed: 2023-08-22].
- [28] Simran Arora. Docker vs. Virtual Machines: Differences You Should Know, August 2023. <https://cloudacademy.com/blog/docker-vs-virtual-machines-differences-you-should-know/>[Accesed: 2023-05-01].
- [29] Chiradeep BasuMallick. What Is Docker? Meaning, Working, Components, and Uses, July 2022. <https://www.spiceworks.com/tech/big-data/articles/what-is-docker/>[Accesed: 2023-08-23].

- [30] Adam Murray. Docker Container Security: Challenges and Best Practices, February 2023. <https://www.mend.io/resources/blog/docker-container-security/>[Accesed: 2023-05-05].
- [31] Sushant Kapare. How to Reduce Docker Image Size (Best Practices), July 2023. <https://medium.com/@sushantkapare1717/how-to-reduce-docker-image-size-best-practices-c5c3ff71da76>[Accesed: 2023-04-23].
- [32] Kubernetes. What is a Pod?, February 2023. <https://kubernetes.io/docs/concepts/workloads/pods/>[Accesed: 2023-04-22].
- [33] Kubernetes. Services in Kubernetes , March 2023. <https://kubernetes.io/docs/concepts/services-networking/service/>[Accesed: 2023-05-07].
- [34] Kubernetes. Kubernetes Components, May 2023. <https://kubernetes.io/docs/concepts/overview/components/>[Accesed: 2023-04-17].
- [35] Jens Erat. Tuning network sysctls in Docker and Kubernetes, July 2020. <https://medium.com/mercedes-benz-techinnovation-blog/tuning-network-sysctls-in-docker-and-kubernetes-766e05da4ff2>[Accesed: 2023-05-08].
- [36] Marko Aleksic. What is ssh, September 2021. <https://phoenixnap.com/kb/what-is-ssh>[Accesed: 2023-05-07].
- [37] Kubernetes. The lightweight Kubernetes[accessed: 2023-05-07]. <https://microk8s.io/>.
- [38] Minikube. minikube start, October 2023. <https://minikube.sigs.k8s.io/docs/start/>[Accesed: 2023-05-07].

- [39] The Kubernetes Authors. Quick Start, May 2023. <https://kind.sigs.k8s.io/docs/user/quick-start/>[Accesed: 2023-05-07].
- [40] Max Brenner. Minikube vs. kind vs. k3s - what should i use?, December 2019. <https://shipit.dev/posts/minikube-vs-kind-vs-k3s.html>[Accesed: 2023-05-08].
- [41] kubectl, February 2023. <https://kubernetes.io/docs/tasks/tools/>.
- [42] The Kubernetes Authors. Accessing apps, December 2022. <https://minikube.sigs.k8s.io/docs/handbook/accessing/>[Accesed: 2023-05-08].



# Appendix



# Appendix A

## A.1 - DockerFile

```
FROM alpine:3.17.1
RUN apk add --no-cache iperf3 \
    && adduser -S iperf

USER iperf
EXPOSE 5201/tcp 5201/udp

ENTRYPOINT ["iperf3"]

CMD ["-s"]
```