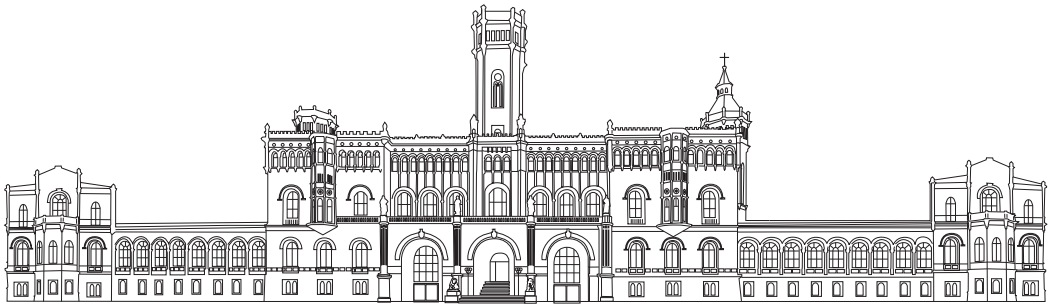

LEIBNIZ UNIVERSITÄT HANNOVER

FAKULTÄT FÜR ELEKTROTECHNIK UND INFORMATIK
INSTITUT FÜR KOMMUNIKATIONSTECHNIK

Evaluation TCP Congestion Control Algorithms using QUIC protocol

Masterthesis

submitted by

AI VIET HOANG

in December 2019

First Examiner	:	Prof. Dr.-Ing. Markus Fidler
Second Examiner	:	Prof. Dr. Jürgen Peissig
Supervisor	:	Anh Vu Vu

SUPERVISORS:
Prof. Dr.-Ing. Markus Fidler
Prof. Dr. Jürgen Peissig
Anh Vu Vu

LOCATION:
Hannover

TIME FRAME:
December 2019

DECLARATION

I hereby declare that I have prepared and written this Masterthesis without the help of any third parties and I used only with the specified sources and tools, which are approved by my supervisor. All passages that have been taken either from any source or from any content have been identified clearly, and this work has not been submitted in the same or similar form to any examination authority.

Hannover, December 2019

Ai Viet Hoang

This work is dedicated to myself and the ones I love.

— Ai Viet Hoang

ABSTRACT

In this work, we will summarize the advantages and disadvantages of the Transmission Control Protocol as well as Quick UDP Internet Connection. The differences between two protocols will be discussed based on the congestion controls and how they interact with each congestion control algorithms. The congestion control algorithms, which will be used in this thesis are loss-based congestion control and delay-based congestion control. The experiment will be emulated using Emulab. After this evaluation, the results can be discussed and compared to the assumption as well as conclusions from other sources or works. The metrics, which we use to compare the protocols include throughput, fairness, link utilization, and responsiveness. With the results we got, we can conclude that QUIC Protocol does not have better performance when we compare the protocol with TCP using the above metrics. In cases with and without loss, TCP shows that it still got the upper hand with better performances when running the same parameters in the experiment as QUIC protocol.

ACKNOWLEDGMENTS

I would like to express my thought and gratitude to Prof. Markus Filder and Prof. Jürgen Peissig for their teachings, experiences, and encouragement during my time studying and working with this thesis.

I also would like to extend my grateful thanks to Msc. Anh Vu Vu, my supervisor, who has spend most of his time to evaluate my work, to give me lots of insight and precision critiques in order to perfect my analysis and measurements.

I would like to thanks the staff and technicians of the Institute of Communications, who have provided me a lab with equipment and resources which I could work with. Without you, this thesis will not be possible.

At last, I would like to give my thanks to my family, my father, Mong Cuong Hoang and my mother Thanh Huyen Nguyen, and my younger brother Tien Bao Hoang. They have given me the opportunity to study here in Germany. I also appreciate the help of Thu Huong Pham, who has given me so many ideas how to do a thesis, and to all my friends, who have always believed in me and supported me in my life and thoroughly this study.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In 2013 QUIC (Quick UDP Internet Connection) has been introduced in a Google Conference as a new protocol for the Internet. With new protocol dedicated for data transportation on the Internet, it brings lots of differences experiments and paper works such as [1](Iyengar and Thomson,2019).

QUIC, which at first is only an experiment, is created mainly to increase the transport performance of HTTPS/2 traffic. By replacing the traditional TCP/TLS and HTTPs stacks, QUIC is developed based on UDP, therefore, allowed QUIC to replace the middleboxes of HTTPS stacks. Meaning QUIC became a cross-layer protocol, which is a combine of 3 parts: Transport, Security and Application layers. According to [2](Adam et al.,2017), QUIC has already wide deployed with more than 30 percent of Google traffic in bytes using this protocol. That makes about 7 percent of total global Internet traffic. QUIC firstly is deployed in many Google applications such as Google Search, YouTube, and Chromium. It reduces approximately 8.0 percent of latency when searching for data using Google Chrome, reduces rebuffed rates of YouTube by approximately 15 percent for users [2](Adam et al.,2017).

Although many studies have been drawn out to measure the performance of QUIC [3](Gratzer,2016) [4](M. Yosofie,2019), however, there is a lack of study about how QUIC performance beside all the statistics of Google. One aspect of the QUIC performances which we are looking for is Congestion Control, which is described by the Internet Engineering Task Force(IETF)[5](Iyengar and Swett,2019). Because QUIC loss recovery mechanisms are built based on existing TCP. There are many algorithms, which we can try to implement into QUIC. Up until now, there are three main types of Congestion Control, loss-based, delay-based and hybrid algorithms [6](B, Kuipers, and Uhlig,2019) with many variations in each class. Because of the shortage of researches in this direction, we need to expand the research to expand our knowledge about this protocol.

Due to the increasing of QUIC usage, in this thesis, QUIC, which is mentioned as a reliable UDP based protocol, will be evaluated based on the congestion control algorithms which has been implemented in old TCP protocol. The strengths and the weaknesses of this new QUIC protocol which are created and maintained by Google will be studied deeply. Both TCP and QUIC will be evaluated, to see the

potential of QUIC protocol when it operated based on TCP algorithms. The Algorithms, which are used in this thesis, are Cubic, represents delay-based congestion control and Vegas, represents Delay-based congestion control.

The experiment will be performed using Emulab, a testbed network emulator, which help researchers to develop, debug as well as evaluate network systems. Using Emulab, which refer to both software system and facility, will reduce a massive amount of time to implement, create simulation, and learn to control a network experiment.

QUIC because of its deployment in user-space facilitated of many applications, there are lots of different implementation using many different programming languages[7](Quicwg,2018). In this thesis, we will use Golang, a powerful open source language, that will reduce a tremendous amount of time to do the implementation thanks to the combination of many older languages. Because of its code structure, which is designed to improve the productivity of programmer in the era of multicore, stacked networked machines with thousand lines codebases.

1.1 MOTIVATION

Nowadays, the rapid development of the mobile Internet and the rising of the IoT (Internet of Things) applications, the network interaction is increased more abundant each day. The content of each webpage, each transmission is also larger and more complex. That is one of the reasons why the users and researchers demand for a new protocol to transmit data in the network, which need efficiency, security and response speed of webpage, which also should be improved.

At first, the developers want to create a protocol, which can increase the stability of the connection, work within the highly variable network and between multiples devices with different operating systems. The transition between local networks, wireless signals, and occasional cellular signals makes mobile Internet connections sometimes unexpectedly very unstable and unreliable. This can cause many problems for the applications, which require constant data streams and low latency with a low ratio of loss. The changing of environment can reduce user experience, which makes the applications worse than they should and that will affect the reputation of the developers or companies.

About this transition point, the TCP protocol has been difficult to improve. Because of this difficulty in improving this old protocol, in order

to avoid it, people have to change the direction of research, instead try to make TCP better protocol, UDP, while it got some drawbacks, this connectionless protocol, which is fast, require fewer resources, seems to be a better choice. What we should do it just give it the advantages of TCP, which including security and reliability. Google is trying to take this new research direction of this problem by developing a new protocol named QUIC over UDP and take advantage of bidirectional bandwidth and TCP congestion control algorithms to avoid network congestion. In other words, we need to integrate the reliability of the TCP, the efficiency and rapidity of the UDP, and combine them into what we have today as QUIC protocol.

The goals of QUIC at first is only an experiment to increase the performance of Google Chrome, but now there are lots of researchers, who want to improve this protocol essentially so that it can be deployed on today's Internet to reduces latency and also solves problems with multiple streams over a single TCP connection. Thanks to that, there are lots of works with the new protocol, which made the foundation for this thesis. They claimed that QUIC will be used widely and the performance will be better than TCP at some performance levels [1](Iyengar and Thomson,2019) [5](Iyengar and Swett,2019), which are parts of the Chromium projects. This project is deployed by Google and most of the researches in this project is used to improve Chrome OS, one of the most famous web browsers today. In this browser, Google claims that they have already used QUIC for almost 80 percent of network traffic. The key features of QUIC include:

- Reduced latency needed for establishing a connection.
- Easier migration of newer connections.
- Improving Congestion control.
- Prevent issue head-of-line blocking problem from TCP/HTTP2.

The Congestion Control of QUIC, which is explained in [8](Molavi et al.,2017), is improved because it is easier to take the information from the program than TCP itself due to the user-space development at the application level. TCP is implemented deeply in the kernel, makes it is almost impossible to gain valuable information to do research or to improve or update the protocol itself. This problem can be neutralize using QUIC. Because QUIC is developed on top of UDP and it is created mainly to increase the performance of modern web browsers and applications, that's why it can be modified as well as show the information quicker than TCP. There are some motivations which we draw out from these works:

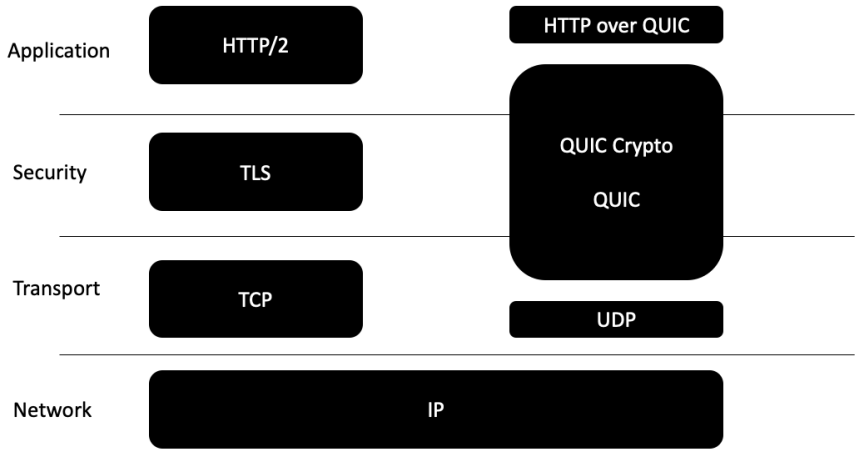

Figure 1.1: Traditional HTTPs stack vs QUIC stack

In [1](Iyengar and Thomson,2019), QUIC has been introduced as a new protocol over UDP, which is also a secured and multiplexed transport. Not only it has the same flow control of HTTP/2, the security power of TLS, but it also supports many of TCP congestion control equivalent algorithms.

In [5](Iyengar and Swett,2019), the author stated that QUIC may be used with other congestion window algorithms, instead of what they choose, which is New Reno. So that to prove that QUIC can be better than TCP, we should take some consideration if QUIC can be also better when we use other congestion control algorithms.

Both conditions above prove that there are many kinds of research that can use QUIC as the main point. In this thesis, we will evaluate the performance of QUIC when compared with TCP at the application level. In this case, is how well both protocols transmit data, which protocol will have more stable transmission, how the congestion control reacts, the amount of data transferred. In the end, we can deduce some assumptions as well as conclusions, if the QUIC can be the next big protocol, which people should look for. Or should people stay will TCP, the one, which has been the foundation for the internet for more than 40 years.

1.2 THE GOAL OF THIS THESIS

After this evaluation, we can get some information about the behavior of QUIC in non-Google-API such as Google Chrome and Chromium. With the implementation of Loss-based and Delay-based algorithms in Golang, a language which is powered by Google, we can check if

the old algorithms from TCP can work well with the newer QUIC protocol or not.

In [9](Brakmo, O'Malley, and Peterson,1994), TCP Vegas has been introduced as well as the new slow start technique, which improves the behavior of the congestion window during the end of the slow start, when the loss packet occurs frequently. TCP Vegas is also fundamental for many later algorithms using delay-based congestion control. TCP Vegas has a better slow start mechanism as well as shows better congestion avoidance due to the using of fine-grained RTT. [6](B, Kuipers, and Uhlig,2019). That is one of the reasons we want to use Vegas for this experiment. And in [10](Hengartner and Bolliger,1999), some scenarios in which to compare the TCP Vegas with TCP Reno have been made, and some conclusions about the performance of both have been drawn out only for TCP.

The Congestion Control evaluation of TCP algorithms is no longer a new subject [6](B, Kuipers, and Uhlig,2019). TCP Cubic has been the main TCP congestion control algorithm for Linux kernel because of its smoothness window. When compared with other traditional loss-based congestion control algorithms such as Tahoe, Reno, Hybla, BIC, etc, Cubic is less aggressive than old ones, which helps it to balance the fairness between flows. The function helps the congestion window to increase faster when it is far away from the congestion window max, it helps the flows to achieve a high rate in a short time. That is a reason why Cubic is good with large bandwidth transmission.[6](B, Kuipers, and Uhlig,2019)

Because of these advantages, we want to measure both protocols using Cubic and Vegas algorithms.

These algorithms, which are deployed widely in TCP, are not introduced as the main algorithm for QUIC. QUIC, on the other hand, has been used New Reno as the main congestion algorithm [5](Iyengar and Swett,2019). So that we should find out why, due to many congestion algorithms TCP have, Google chooses only New Reno is enough, That's why in this thesis, we will examine the behavior of Cubic and Vegas in QUIC Protocol, to conclude that if it is possible to use old algorithms of TCP for the newer protocol.

In this thesis, there are several questions which we try to answer:

- What is QUIC and an overview of QUIC?
- Why QUIC can be better than TCP, which features does QUIC have?

- Can QUIC outperform TCP in Emulab in an experimental environment with the same setup parameters?
- Why Google choose New Reno as the main Congestion algorithm for QUIC?
- And the most important thing, How good Cubic and Vegas algorithms in QUIC protocol?

These questions should be answered at the end of this thesis. There are lots of research testing QUIC in practical applications [11](E. Sy,2019) [12](J. Ruth,2018) but in this thesis, we will run the testing and implementation of QUIC in Emulab, which is a virtual network. Because of the virtual environment, we can easily tune the algorithms, variables, so that we can see the influence of different parameters in the congestion control algorithms.

1.3 STRUCTURE OF THIS THESIS

This thesis will have four main parts:

- Chapter 1 will describe the works, including the introduction, the motivation of the thesis, the target of this experiment and also the Structure of this thesis.
- Chapter 2 will cover the fundamental of TCP and QUIC, also the algorithms which are used in the experiment, Cubic and Vegas will be mentioned, and some related works from the third party such as Golang libs, Emulab, etc.
- Chapter 3 includes the experiment preparation, the values which will be set for the experiment, the topology, the setting which will be the best to evaluate both protocols. The experiment process will be explained so that we can get the best understanding of the experiment, also the measurements will be mentioned so that people can understand the process, why we get these results.
- Chapter 4 concludes the experiments, some explanations as well as the directions for the works in the future will be done. The mentioned papers will be at the end of this chapter.
- Chapter 5 will show the detailed devices and software required for the testing experiment.

1.4 PLANING

In order to have a good evaluation, we need to have a precise plan and balance workflow. The research begins at 5. Juni.2019 and will be ended in 5. December 2019. The time required for the thesis is

approximately 6 months. In this time, the number of works which are required includes:

- Get the foundation understanding of QUIC protocol
- Get used to the testbed Emulab, setting up the experimental environment, such as install all the important software, deciding the optimal topology, planning the process of the experiment.
- Learn how to code using the Go language.
- Have a better understanding of Congestion Control algorithms, such as Cubic and especially Vegas.
- Implement the Vegas algorithms into QUIC. The Cubic algorithms have been already implemented in the default library,
- Testing the code. Write scripts to automate the process. We need at least 25 - 30 samples for each run to calculate the average values.
- Process the raw values into figures and writing the report.

2

FUNDAMENTAL AND RELATED WORKS

This part shows the basics of both protocols TCP and QUIC. The congestion control of QUIC and the two algorithms Cubic and Vegas will be explained here also.

2.1 TRANSMISSION CONTROL PROTOCOL

Since the work of [13](Postel,1981), Transmission Control Protocol or TCP has been a subject in many other types of research and experiments. TCP and Internet Protocol has been the backbone of the Internet until now and the ways they work are the definition of how the data are transferred inside a network.

TCP has always been a reliable type of sending due to its connection-oriented nature. The connection before the data transmission must be established between the sender and receiver. It is the reason for TCP's high reliability as well as security. That's why most of the network traffic such as Email, File Transfer always used TCP.

TCP has also advantages when compared to UDP, which is a fundamental thought of QUIC, due to its Congestion Control. The window in congestion control of TCP defines how much packets can sender transmit to the receiver. The flow control of TCP is why the transmission of data using TCP can be error-free.

TCP, however, has some disadvantages due to the "retransmission ambiguity" [5](Iyengar and Swett,2019). TCP when retransmits the packet, it uses the same sequence number for the retransmitted data, which combines the transmission order of the sender and the delivery order of the receiver. This combination makes some packets, which are retransmitted, have the same sequence number.

The loss epoch of TCP is also not the best when compared to QUIC.[5](Iyengar and Swett,2019) When multiple packets are lost, the loss epoch will increase for some round trips, because TCP has to wait to fill all the empty space of sequence number. This will make the congestion window of TCP to react slower than QUIC.[5](Iyengar and Swett,2019)

2.2 QUICK UDP INTERNET CONNECTION

Quick UDP Internet Connection or QUIC for short, which based on UDP using a multiplexed transport stream has been introduced by Google and designed by Jim Roskind. It supposes to have the advantages of TCP with the connection-oriented transmission, a better congestion control, which UDP doesn't have before, as well as some of the UDP strengths such as the power of establishing multiplexed connections between sender and receiver. The summarize of QUIC can be founded in [14](Projects,2012), which helps us to have a better understanding of QUIC. Besides the base differences of algorithm between TCP and QUIC, which has been explained in [5](Iyengar and Swett,2019), it still has some advantages when using QUIC over TCP. With 3 ways Handshake and with the implementation of TLS into TCP, the time which TCP needs to establish a connection is limited because this time can not be reduced more [1](Iyengar and Thomson,2019). That's why the QUIC has been introduced as a replacement for TCP in the future. With some advantages such as:

- 0-RTT Handshake: when the receiver in QUIC case has already known the crypto keys using Transport Layer Security or TLS 1.3, QUIC does not need to establish to the server, to exchange certificates again.
- Behavior on bad connections: One of the problems with TCP is, the data flow of TCP can not stream forward, in case some packets are missed or discarded. The flow can only continue when the lost packets are retransmitted. QUIC can solve this head-of-line blocking problem of TCP by using multiplexing. Multiple streams are deployed by QUIC and the lost packets only affect one single stream.
- Multiple streams in a single connection create an opportunity for Q to overcome the difficulty of basic TCP. TCP during the transmission can only use a single port for a single connection. While QUIC can transport multiple segments using multiple ports and streams with a 64-bit connection identifier. All streams are checked with stream ID (stream identifier), which created by clients using QUIC.
- Better performance for Google-based programs and applications: According to Google, since the deployment in 2014, most of Google AP such as video buffering, page reload has significantly increased when compared with TCP+TLS.
- "pluggable" mechanisms, which means users can change the setup values, improve the algorithms more flexible and quicker than with TCP.

The main differences between QUIC and TCP are based on the algorithms, which lie at the heart of each respective protocol. For each encryption level (expect for zero-RTT as well as one-RTT), QUIC can use separate packet number spaces. Thanks to this separation process, the sent rate of retransmission of a packet, which are different encrypted, is reduced tremendously.

QUIC has an ability to increase monotonically the sequence number of packets. It divides this process of TCP into two parts: For Transmission order of sender, QUIC uses a sequence number. For the Sending the data, QUIC has Stream frames for each data, and these data are encoded the stream offsets for Delivery order of receiver.

The packet number in QUIC has an important role. It makes the loss detection mechanism of QUIC less complex. The ordering of packet in QUIC is also simpler, the lower packet number data has, the sooner the packet number would be transferred. When a packet is detected as lost, QUIC retransmits the packet but with a new sequence number. Besides, it is also easier to gain information from QUIC than TCP, such as information about RTT measurements.

QUIC is also stated to have better loss epoch algorithms, which reset the loss epoch immediately when a lost packet is acknowledged. According to theory, it makes the calculation of Congestion window more frequent than TCP, which are updated slower due to loss epoch increased for multiple round trips.[5](Iyengar and Swett,2019)

Due to the ability to support multiple ACK ranges, QUIC can run better than TCP in high loss environment and can speed up the recovery process, reduces extra retransmission.

2.3 CONGESTION CONTROL

Congestion is a state occurring in the network layer when the traffic is so heavy that it slows down the network 's response time. In order to have a good network, the congestion algorithms have to work well to calculate this condition. We have to consider if it is really congestion occurred or because the congestion window of the own sender increase so much that makes it think it gets congestion. The sender through the algorithms must determine how to control the flow, how to maintain the fairness between flows, how to utilize the link so that the bandwidth is not wasted because of the fear of congested network. These created lots of congestion control algorithms for TCP as well as QUIC today. These are useful for the calculation of data transmission,

data control, and error detection or correction.

The basic of Congestion Control is to control the transmitted data. When the link is going to be congested, the congestion algorithms will try to change the congestion window, which is related to how many data packets are allowed to transmit. When the congestion is over, the algorithms will give the Client permission to increase the window again. The increasing amount and the smoothness depend on the algorithms.

One of the most important part of flow control in TCP, as well as QUIC, are the congestion control. With TCP there are so many available algorithms, which have been implemented and upgraded over the years, such as Tahoe, Reno and some new congestion algorithms such as Cubic, Vegas, New Reno. Cubic is implemented in Linux Kernel since Version 2.6.19 and Vegas in Version 2.2 / 2.3.

Congestion control of QUIC uses both packet loss as well as the increase of ECN, Explicit Congestion Notification as signals of congestion. Usually, TCP/IP acknowledges congestion using the loss of packets. But by using ECN-aware router, the IP header of the dropped packet would be set and that information would be sent back to the sender by the receiver. The sender will reduce the rate because of this notification and it behaves like that packet was dropped and lost.

Just like TCP, QUIC begins the transmission with the Slow Start phase. The Slow Start Threshold, sshthresh would be set based on the congestion algorithms. When the congestion window decreased below this threshold, QUIC will re-enter this phase immediately. During the transmission, QUIC only increases the congestion window using a number of byteInFlight from packets, which are acknowledged. When the Slow start exits, QUIC changes to the Congestion Avoidance phase. Depending on the algorithms, QUIC should have a different kind of behaviors.

During the transmission, whenever the packet is lost or ECN increased, QUIC has a small amount of time called the Recovery period, which helps the congestion window to not decrease too much when the new lost packets occur continuously. TCP ends the recovery phase when the lost packet is acknowledged while QUIC end the recovery phase only when the first packet, which is sent at the start of the recovery phase, is acknowledged.[5](Iyengar and Swett,2019)

In the case of persistent Congestion, which occurs in the network for a long time. QUIC would change the congestion window into minimum value, in order to behave like the sender of TCP, which responses

to Retransmission Timeout. The congestion control of QUIC is a result of many different works and experiments which are also mentioned in [5](Iyengar and Swett,2019). Because QUIC has acknowledged based detection, so the congestion window would be changed depending on the byteInFlight of the acknowledged packet. That means we need to implement the algorithms based on the values of the acknowledged packet. In [5](Iyengar and Swett,2019) states that QUIC should have these functions:

- OnPacketSent() which controls the behavior of Sender for a non-acknowledged packet. The byteInFlight would increase linearly.
- OnPacketAck() which is called when the sender has received acknowledgment of successful transfer packet.
- OnPacketLost() which is called when the transferred packet is dropped or lost.

2.3.1 *Loss-based Congestion Control*

To detect if the link is overflowed or not, the easiest way is checking for the packet loss. In [15](Geist and Jaeger,2019), this kind of congestion control heavily depends on the binary packet lost input, or feedback. This feedback does not require any information of the network or the state it is in. All the indicators such as retransmission timeout or duplicated ACK signal the sender to reduce the congestion window so fewer packets will be transmitted, therefore, prevent the packet loss in the next segments. One of the advantages of this Congestion control is that the algorithms are set at senders, and the receivers can be using other congestion control, which makes receivers more capable of receiving different data from different senders, which run another kind of congestion control. Loss-based Congestion control is also easy to implement than other congestion control.

However, the problem with this loss-based Congestion control is sometimes, due to having only some indicators, which heavily based on the loss of the packet. It can cause the sender to reduce the congestion window unintentionally, which cause effective throughput of the sender to decrease, even the congestion does not happen. When the congestion window increases to the maximum value, which allowed by the sender. The packet is always indicated as loss or it needs to be dropped because if not the congestion window will increase over the possible bandwidth. This brings lots of problems for the traffic or connection required stability such as real-time communications. That is why almost all loss-based congestion control do not have the highest accuracy when compared to other congestion algorithms. In these loss-based congestion algorithms, Cubic stands out the most due

to its special cubic function. This function when compared to other traditional ones [6](B, Kuipers, and Uhlig,2019).

2.3.2 *Cubic*

Cubic has been researched and became the default TCP congestion control algorithm for Linux kernel since version 2.6.19 due to its stable and independent from RTT, which makes the utilization of the bandwidth better than TCP Reno or BIC [5](Iyengar and Swett,2019). That's why in this thesis, Cubic, which is Delay-based Congestion control, will be used as a method to compare the congestion control of TCP and QUIC.

Cubic uses AIMD (Additive Increase Multiplicative Decrease) as a method to increase and decrease the congestion window. While using the same loss-based approach as other algorithms (Reno, Tahoe), Cubic's actual implementation differs greatly from them. Cubic algorithm has been explained thoroughly[16](Sangtae, Injong, and Lisong,2008) that the performance of this algorithm is so effective due to its cubic function, which is shown in Figure 2.1. The convex and concave functions make the Cubic has better Recovery after loss, which means it can utilize the link better, in case the link has a large bandwidth, which increases the throughput faster after the congestion.

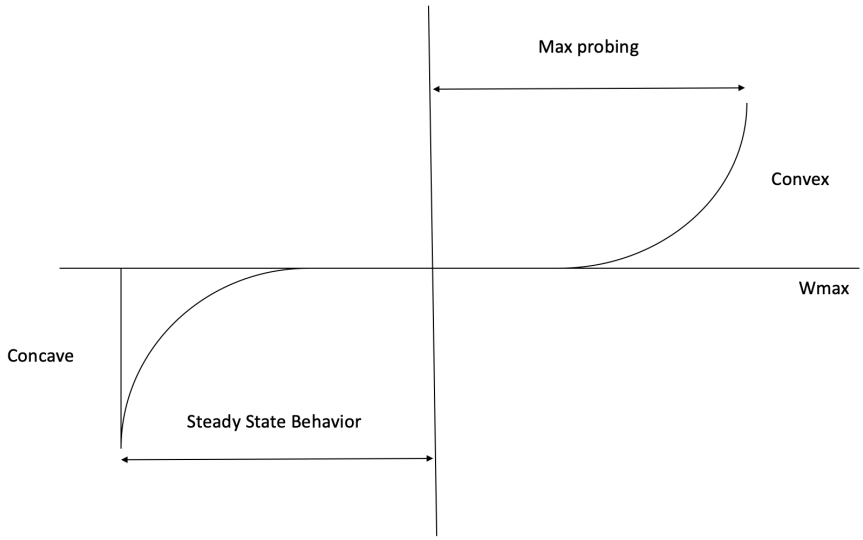

Figure 2.1: TCP Cubic function

Both the Congestion window growth functions of Cubic and the Algorithm, which is implemented in Golang has been introduced using the algorithms in [16](Sangtae, Injong, and Lisong,2008). The Algorithm of Cubic is depending on AIMD, so we must take the additive factor and multiplicative factor into consideration. In [16](Sangtae,

Injong, and Lisong,2008), the α stand for additive factor and β for multiplicative factor. So in the coding process, these two variables must be calculated, in order to gain the average TCP window, to check if the process is in TCP friendly-region in the Congestion window or not. The equations which we use are:

$$\frac{1}{RTT} * \sqrt{\frac{\alpha * (2 - \beta)}{2 * \beta * p}} \tag{2.1}$$

this equation is taken from [16](Sangtae, Injong, and Lisong,2008) but to get a equal TCP window with prefix $\alpha = 1$ and $\beta = 0.5$, which make the congestionwindow equal:

$$\frac{1}{RTT} * \sqrt{\frac{3}{2 * p}} \tag{2.2}$$

So for the coding process, we want to create dynamic factors, which change depending on the number of connections, because QUIC can easily change transmit routes. So with this new dynamic factors α β we may have a better performance for QUIC.:

```
func beta{)
{
        return (number_of_connections * -1 + beta) / number_of_
            connection
}

func alpha()
{
        b = beta()
        return 3 * number_of_connections * (1 - b) / (1 + b)
}
```

with α is an additive factor and β is a multiplicative factor, which are used to find the average window size of AIMD. Function beta() returns dynamic beta values which depend on the number of connection which we used to transmit the data, it also emulates the back off effect on a single loss event of TCP Reno multiple connections. While Function alpha() returns the optimal alpha, which is TCP friendly. Because we want the same setup to do the evaluation, all the optimal values in TCP will be used in QUIC. The mathematical expression of Function alpha is:

$$\alpha = \frac{3 * b}{2 - b} \tag{2.3}$$

while

$$b = 1 - \beta \tag{2.4}$$

both 2.3 and 2.4 are taken from [16](Sangtae, Injong, and Lisong,2008). With these equations, we can achieve the average TCP window which does not depend on the fixed α and β values. The most important thing about QUIC is how the congestion window change when we got the information of the ACKed packet and when the packet is lost. So in the next part, we will describe the functions which do this work.

When the ACK packet is received, first QUIC will check if the transmission is in epoch time and try to change the congestion window with *kCubeScale*, *kCubeCongestionWindowScale*, which are TCP default constant values. These values are determined to calculate the *kCubeFactor*. This factor will be used to calculate the Cuberoot, which is a variant of Cubic function. If the congestion window still not at the max value, the function will try to calculate the new increased window for the next sending. If the current window is already bigger than the max congestion window, which is not allowed, the middle point of the cubic function will be placed at this current window value.

After that, elapse time and congestion window at the origin point between convex and concave profile will be calculated and the target congestion window will be changed using these parameters.

```
if elapsedTime >  timeToOriginPoint
{
        targetCongestionWindow = originPointCongestionWindow +
            deltaCongestionWindow
}
else
{
        targetCongestionWindow = originPointCongestionWindow -
            deltaCongestionWindow
}
```

these functions describe the equation 3.2.1 in [16](Sangtae, Injong, and Lisong,2008), when the time t, which is elapsed time since the last window reduction bigger than time K, which is the time the function cubic need to increase the congestion window to the inflection point or origin point in this case, the new congestion window will increase an amount *deltaCongestionWindow* and Cubic get into convex region. When t is smaller, then the new congestion window must be smaller, since Cubic is still in conclave region. This comparison is used to decide which phase Cubic is in.

When the Packet is lost, QUIC just needs to decrease the current congestion window using a β factor. And after that, the last max con-

gestion window is set based on the congestion window in which the loss occurred. The epoch time will start now, to measure the time till the lost packet is ACKed for QUIC.

```
if currentCongestionWindow < lastMaxCongestionWindow
{
        lastMaxCongestionWindow = betaLastMax *
            currentCongestionWindow
}
else
{
        lastMaxCongestionWindow = currentCongestionWindow
}
Reset time of epoch
return currentCongestionWindow) * beta()
```

2.3.3 *Delay-based Congestion Control*

Delay-based Congestion control seems to have another thought of handling the congestion window. They are called Proactive congestion control by monitoring the RTT of the acknowledgement packet so that they can prevent congestion before it occurred. The steady RTT is one of the indicators for the sender using this congestion control to know that there is no congestion in the link, while the increase in RTT will indicate that the congestion is going to happen, and the sender has to decrease the window. [6](B, Kuipers, and Uhlig,2019) claimed that these algorithms help senders to prevent the packets build up in the queue, and fix some problem in loss-based algorithms such as throughput oscillation.

The main problem of this Congestion Control is, when competing in the link with Loss-based or Hybrid, they always get worse performance, this makes the implementation of this congestion control does not practical when almost all other traffics or other sources use Loss-based algorithms. Because of these loss-based algorithms, the observation RTT which is calculated in Delay-based Congestion control increased a huge amount.

Because these algorithms heavily based on RTT, so any inaccurate samples because of cross traffics, dynamics routing or delays ACKs will tribute to the downgrade performance of the senders. One of the most fundamental representatives of this delay-based Congestion Control is Vegas.

2.3.4 *Vegas*

Aside from the way of detecting congestion based on packet loss like Cubic, there are other TCP congestion controls based on RTT (delay-based). One of them is Vegas. Due to many comparisons of TCP congestion algorithms Vegas might be the second-best algorithm after Cubic because of its smooth congestion window.[6](B, Kuipers, and Uhlig,2019)

TCP Vegas algorithm works based on RTT, which is determined after the packets are sent to the receiver. If this RTT is too big when compared to the based RTT, which is taken from the first successful packet transmission, the sender itself will understand that the traffic is going to be congested. Thus, the Congestion window will be re-calculated while keeping the throughput from degrading too much compared to Tahoe and Reno. The Algorithms are mathematically calculated in [17](Kurata, Hasegawa, and Murata,2016)

In this implementation, we will use only TCP Vegas normal algorithm, not Dynamic TCP Vegas.

So the mathematic equations we used for our implementation are, which drawn out from [18](Li, Zhu, and He,2017):

$$Expectedrate = \frac{cwnd(t)}{BasedRTT} \tag{2.5}$$

The *expected rate* is the throughput, which the sender expects to transmit when the connection is not overflowed.

$$Actualrate = \frac{cwnd(t)}{LatestRTT} \tag{2.6}$$

The *actual rate* is the real throughput, which the sender calculates using the value congestion window of one segment and the RTT of that segment when the sender received the acknowledgment of that segment.

$$Difference = Expectedrate - Actualrate \tag{2.7}$$

The difference is the value that the sender uses to determine the change of the congestion window for the next packets.

$$Tarcwnd = \begin{cases} cwnd(t) - 1, & \text{if } Diff < \alpha \\ cwnd(t) + 1, & \text{if } Diff > \beta \\ cwnd(t), & \text{if } \alpha < Diff < \beta \end{cases} \tag{2.8}$$

For the slow start process, Vegas will behave like new Reno, exponentially increase the Congestion window until the congestion avoidance phase. During the congestion avoidance phase, the two values which are the most important in the Vegas algorithm are Based RTT and Observed RTT. Due to these values, the computer can calculate better and detect sooner when congestion occurs.

With each Acknowledgement(ACK), the computer will take the RTT values and compare it with the old RTT. These values will be stored in Observed RTT. When the Observed RTT gets too high, Vegas will assume that the network is going to be congested and preemptively lower the congestion window. On the other hand, when the Observed RTT becomes low, the congestion window will be increased, so that it can maximize the throughput.

The Based RTT is the RTT, which theoretically calculated before the transmission begins. In [9](Brakmo, O'Malley, and Peterson,1994) it shows the values of the congestion window when there is no congestion. That means it is the best scenario or the best congestion window which can happen in one transmission. In [10](Hengartner and Bolliger,1999), the RTTs are going to be updated when the min RTT is smaller than Based RTT. It means that the throughput can go higher than theoretical values, which are specified in the program.

The function which we will use to calculate the Difference value in code:

$$Difference = \frac{currentCongestionWindow}{BasedRTT} - \frac{currentCongestionWindow}{ObservedRTT} \tag{2.9}$$

where the current Congestion Window is the congestion window in the meantime, which can be obtained during the slow start as well as in the recovery/congestion avoidance part. This value divided with Based RTT will give the expected throughput and the value divided with Observed RTT will give the observed throughput like in [9](Brakmo, O'Malley, and Peterson,1994). The Difference value, together with the measured throughput, would be used for the estimation of available bandwidth. When there are no congestion, expected values, and actual values should be identical.

With the Difference, which is calculated with the function above, comes the congestion avoidance algorithm, which made Vegas better than Reno, because Vegas can predict the congestion before it occurs:

```
if Diff > float64(Vgamma) && currentCongestionWindow <= ssthresh
    {
```

```
                currentCongestionWindow = utils.MinPacketNumber(
                    ssthresh, TarCwnd+1)
}
```

the value gamma here is used to control the transmission rate, so that is it not too quick, which can cause the node to drop the packet unintentionally. If the Diff is too big, while the current congestion window is smaller than the slow-start threshold, we should switch back the congestion window same as the slow start threshold.

Now we get into the function when Vegas gets the ACKed packet. At first, it will check which state the sender is in. If the sender is in limited condition, which should not increase the congestion window, the sender will break from this function. Else it will check if the current congestion window is bigger than the maximum congestion window, which is not allowed, if it is true then the sender also breaks from the function. After that, due to the condition of Vegas, which only update the congestion window when it has enough RTT sample, we make a small variable which increases when new RTT is gained from *rtt_stats.go* when we have enough samples, then the calculation of Vegas algorithm begins:

```
if Diff < alpha / BasedRtt
{
        TarCwnd = TarCwnd + 1
}
else if Diff >= alpha / BasedRtt && Diff <= beta / BasedRtt
{
        TarCwnd = currentCongestionWindow
}
else if Diff > beta / BasedRtt
{
        TarCwnd = TarCwnd - 1
}
```

The algorithm is made using the equations for TCP Vegas in [17](Kurata, Hasegawa, and Murata,2016)
The TarCwnd value here is the targeted congestion window, which will be used for the next packet.
alpha, beta are the values which are from the implementation of tcp_vegas.c, which are the best configuration values for the tcp_vegas module in the Linux kernel. This file is an implementation in the code of [19](Omar and Eitan,2000). Since we want to evaluate both QUIC and TCP, which are using tcp_vegas module of the Linux kernel, so we have to make sure that the parameters used in QUIC would be equals.

The TarCwnd will increase linearly when the Difference value smaller than alpha divided by BasedRTT.

The TarCwnd will decrease by 1 when the Difference value bigger than beta divided by BasedRTT
The TarCwnd will stay equal when the Difference value changes between two of above values.

Another thing we need to consider is that most of the time, Vegas will not always update its congestion window when it does not have enough RTT samples to calculate the new congestion window [20](Richard, Jean, and Venkat,2004). So during the time without a proper RTT sample, it will behave like Reno. So that we need a count variable, to check if the RTT is not updated correctly or enough (most of the time it needs to update 3 latest RTT). This condition has already been done in function *maybeIncreaseCwnd* in Vegas_Sender.

Some disadvantages of TCP Vegas may have occurred in QUIC, such as Re-routing and Unfairness. When RTT increases because the routes are changed by the network, Vegas can make mistake to interpret this Increase as Congestion, thus it would send the packet at a lower rate than expected. For the implementation of behavior when packet is lost, Vegas used the same functionality of Reno but with some upgrades. First, it used new retransmission mechanism, immediately retransmit the packet with only duplicate ACK, while Reno need till third duplicated ACK to initiate the retransmission of the packet.

2.3.5 *Metrics*

To evaluate the performance of each protocol, we have to compare the parameters, which are important in networking traffic. In this thesis, we will have both metrics from network-based and user-based interpretations. Because for some metrics, there are no clear definition of desired goals from the community, who works in the networking field.

The useful parameters included throughput, delay and packet drop rates [21](Floyd,2008). The goal of congestion control usually is to increase the throughput to maximum, so that the sender can transmit the data with the best possible rate. Network throughput represented in bit per second because the packet size of TCP and QUIC are different [3](Gratzer,2016). Because in networking, devices communicate with each other by packets exchange. With high throughput, it indicates how well the packet is delivered from one to other machines. Low throughput creates an unexpected problem for networking applications, such as low-quality Voice over IP, high ping and increased packet drop during online gaming. Because we can easily get the information of throughput through Wireshark, so we will choose throughput as

one a metrics in this experiment.

One of the other metrics we have to consider is fairness [6](B, Kuipers, and Uhlig,2019) [21](Floyd,2008). Fairness can be evaluated between the same protocols or different protocols, which run specific congestion control algorithms. Fairness indicates the ability of the protocol to share the system resources with each other. The more equal data sent from both flows, the better fairness the protocol gets. In this experiment, we use user-based fairness metrics. The fairness will be determined using the results we get from Wireshark. We will calculate the amount of difference data sent between two flows. And then compare the results with other protocols. Smaller difference value we get from the protocol, the better fairness that protocol has.

Next, we can use the utilization of the link as a metric for the comparison [22](Tetcos Engineering,2012). The better the algorithm keeps the link at full utilization, the better the algorithm works [19](Omar and Eitan,2000). In this experiment, the link utilization will be presented as a sum of both data packets, which are sent by both Clients in one period. Then we can check which protocol will have better utilization of the link.

The last metric [21](Floyd,2008) we use in this experiment is convergence. Convergence defined as the time the Clients need to increase the fairness between the existing flow and the joined flow. Convergence is especially important because nowadays most of the network traffic is combined with numerous flows, sometimes thousands of flows at the same time. We will evaluate the convergence of different protocols, which use the same congestion control algorithms. The results we get are also user-based interpretation, which is simpler case of delta-fair convergence time [23](D.Bansal,2001). We check the time when the second flow begin until the time both flows have the same amount of throughput, which is indicated by the time the throughput line of both flows cross each other in Wireshark.

3 EXPERIMENTS

In this section, the information about the experiment including the preparations of the software, the topology of the experiment network in Emulab and the configuration of each measurement. At the end of the section, the results of each measurement will be shown and some conclusions will be drawn out based on these results.

3.1 EXPERIMENT PREPARATION

In this experiment, fairness, link utilization, as well as responsiveness, will be evaluated and they are the based elements to calculate and determine which congestion control algorithms work well with which protocols. The experiment went as follow:

- 4 Nodes: Client1, Client2, Server, Node will be established
- Client1 and Client2 will be connected with Node, Node will be connected to Server
- Client1 and Client2 will transfer the data using the traffic-gen application to generate traffic to the Server
- Client1 will transmit the data first.
- After 10 seconds, Client2 will also transmit the data to the Server and thus causing congestion.
- After 60 seconds, Client2 will stop the transmission. Client1 continues with the transmission.
- After Client2 has stopped for 10 seconds, Client1 closes the transmission.
- The whole process lasts 120 seconds.

After that, the result will be logged into files, which will be processed and evaluated by other programs such as Libre and Matlab.

3.1.1 *Emulab*

The Experiment will be tested on Emulab Testbed [24](University,2019). Emulab is a networking environment for experiments, which creates lots of different Topologies including emulated network nodes. Testers

can visualize the traffics, setup parameters for nodes, links, delays, loss connections using ns scripts. The nodes will be inserted with different images, which basically an operating system for each computer, to help with the data processing, terminal commands. On each node, some programs, which are needed for this experiments are installed such as Go, Wireshark, and Linuxptp.
The setup environment is described as the code in the Appendix section.

All nodes will use a custom Ubuntu image, which is required to install all the necessary programs, we need minimal 4 nodes for this experiment. Both Clients will have access to the middle node between them and the Server with 1000 Mbit link when the link from Node to Server is only 5 Mbit. It is purposely done in order to force the Clients to compete with each other during transmission time for the bandwidth. The link 0 will have a 10-millisecond delay so that the number of nodes, which required for this experiment can be minimized and it will be better to evaluate during RTT-based experiment.

3.1.2 *Golang*

For the Experiment, Golang from Google will be the main programming language. The experiment will be done using an experimental traffic-generator with the QUIC library, which is coded by Lucas Clemente and Marten Seermann. This implementation has been introduced in dotGo conference in 2016, which can be used to do various experiments to draw out the power of QUIC. This library is very useful because many studies about QUIC can be done with little of the resources needed. The cost is low because we do not need Google servers to use this. This library works with traffic-gen in order to create traffic for the emulab nodes. So that we can transfer the packets, which uses TCP protocol and QUIC protocol and by changing the congestion control algorithms, we can get different results. In [17](Kurata, Hasegawa, and Murata,2016) and [4](M. Yosofie,2019), we can get much information of projects which involve QUIC using Golang. The base-drafts which made the specifications of QUIC are also introduced here.

3.1.3 *Tcp_probe*

At first, we will get some idea, how good the program is when compare the results, which produced by TCP-probe with the results, which provided by the traffic-gen program. We want to make sure that the traffic-gen behave equal, or at least similar with the module TCP-

probe, which is the complete program and be maintained by many other programmers and researchers.

The traffic generator we used for testing will be Iperf, the values congestion window will be logged out using TCP_probe and filtered by Libre.
When the traffic-gen, which is written by Go, has the same results as Iperf, we will use this traffic-gen to do more measurements.
After all the processes, the data will be plotted by Matlab.

3.2 EXPERIMENT PROCESS

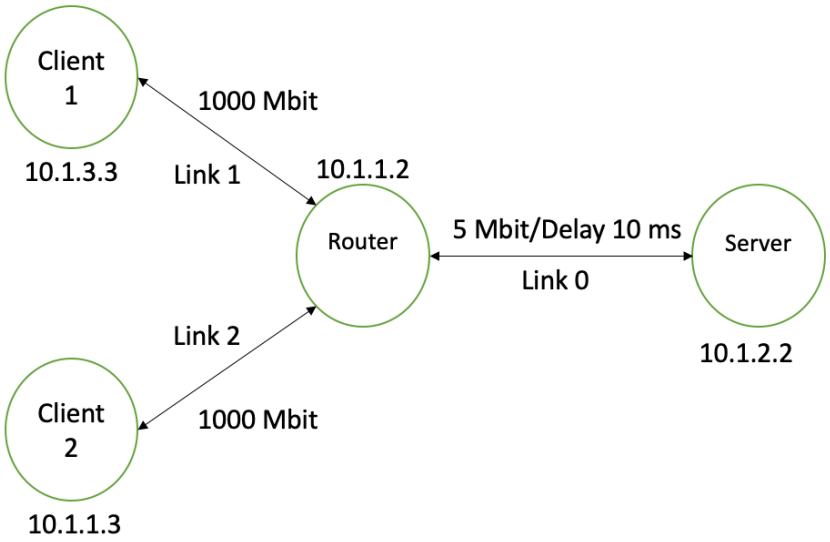

Figure 3.1: Emulab experiment topology

As explained in Section 2, we have two Nodes which represent Client1 and Client2. These two nodes run the sender program to generate packets and send them to Node Server through the middle node. Both Link1 and Link 2 are set with very large bandwidth 1000 Megabit so that both Clients can freely send the data without any obstacles. The Link 0 is set quite low only 5 Megabit so that the Clients have to compete with each other and calculated the rate using its congestion control algorithms. The Server node has to run two different receiver programs so that it can receive the data from two different senders. During the experiments, Wireshark will be run on Node or Server node to check the throughput as well as monitoring the coding in the development process. TCP_probe will be run and generated a log in both Clients so that we can get the values of the congestion window during TCP protocol experiments.

First, the Iperf will be used to determine if the emulated network is working properly. We used the data created by Iperf as a reference for the data created by another traffic generator.
In order to simulate the traffic just like the Iperf, we need to set up the rate so that both Clients will try to take as much bandwidth as possible, without increasing it too much so that it drops the packets. We use nodes in Emulab which are Drop-tail router. So the values are configured in Traffic-gen.go
The data rate is calculated as Equation 3.1

$$data\_rate(bit) = csizeval * arrval * 8 \approx 5,7(Mbit) \tag{3.1}$$

which csizeval is the size of packet data, we set it with 1200 bytes. arrval is the number of packets sent in 1 second, we set it with 600 so theoretically, it should send about 5,7 Mbit, which almost equals the link bandwidth which we set up in Emulab.

We need two folders of mp-QUIC_latency, named M1 and M2. M1 folder will be run on the Client1 node as the first Sender, the M2 folder will act as the Sender for Client2 node.

In the Server node, we have to run both M1 and M2 as two receivers for Client1 and Client2. Because we only care about the data which runs through node and server. So that this setting is acceptable.

At Client1, we need to transmit the data to Server for 120 seconds. At Client2, the time we need to transmit is 60 seconds, after the Client1 run 30 seconds. The syntax we need for Client1 and Client2 is presented in Appendix.

During the running, we used Wireshark as a tool to check the data transmitted.
During the works, all nodes have to sync the time with Linuxptp so that we don't have any un-sync node, which may affect the results we got.

For each algorithm, to have a good conclusion of each protocol, we need to set up the QUIC implementation as equally as TCP implementation in Linux kernel, so that we can have the best comparison for each kind. Because QUIC is a user-space application implementation, we can modify the parameter as much as we want. This can cause a difference between runs.

For Cubic, the most important thing is to create a proper implementation of a cubic function, these variables have been tested out by Google when they implemented these into the Chromium [25](Yolan,2012)

- cubeScale = 40
- cubeCongestionWindowScale = 410
- cubeFactor = 1 « cubeScale / cubeCongestionWindowScale
- defaultNumConnections = 2 number of connections which QUIC can use if it is allowed.
- beta = 0.7 default backoff factor
- betaLastMax = 0.85, this beta is addition backoff factor, it will be used when loss occurs during the transition in concave part of Cubic function. With this factor, the sender can speed up the convergence and give more bandwidth to newer flows.

Both of the below functions are drawn from [16](Sangtae, Injong, and Lisong,2008). These functions are used together with the parameters above to create window growth function in Cubic algorithm of Linux v2.2.

This is a function that is used to determine the growth of Cubic function. During congestion avoidance, when the receiver had an ACK, Cubic will set the next target window equal to W(t+RTT).

$$W(t) = C * (t - K)^3 + W_{max} \tag{3.2}$$

This function is used to calculate the time period the 3.2 needs to increase the value current congestion window to the max congestion window, which is set when the congestion occurs.

$$K = \sqrt[3]{\frac{W_{max} * \beta}{C}} \tag{3.3}$$

CubeScale and cubeCongestionWindowScale parameters are used to calculate the cubeFactor. This Factor is required to calculate value C [16](Sangtae, Injong, and Lisong,2008), which is a part of the window growth function of Cubic 3.2. Using C, we can calculate values K, which is the time period the function uses to increase the current window to max value 3.3. Using both K and C, we can modify the rate of increasing window value, which presents the function line in Figure 2.1. Beta and betaLastMax will be used in case the Packet loss occurs. According to Algorithm 1 [16](Sangtae, Injong, and Lisong,2008), the ssthresh in case of packet loss will be decreased using beta value.

For Vegas, the variables we need, come from the file, which created the module Vegas in Linux kernel for TCP congestion control [26](Arianfar,2012) and [27](Cardwell,2006). These variables are important for the calculation of Vegas algorithms, which are also described in [19](Omar and Eitan,2000).

- $\alpha = 2$
- $\beta = 4$
- $\gamma = 1$

Instead of using average RTT [19](Omar and Eitan,2000), we will use the parameter RTT which is updated by the program. The values which we need for the calculation are:

- latestRTT which is got from rtt_stats.go
- minRTT which is got from rtt_stats.go
- BasedRTT which is set at 20 milliseconds.

3.3 EXPERIMENT RESULTS

The metrics which will be used here include Throughput, Fairness, Responsiveness, Link Utilization [21](Floyd,2008) and [22](Tetcos Engineering,2012). These metrics are useful to measure congestion controls and these have been a guide for others to have a good basis to compare the congestion control mechanisms. In this thesis, the metrics will be measured based on the values which are obtained from the codes or with tcp_probe.

When measuring the performance of a network, throughput, and congestion window are the main metrics which can see directly. First using Wireshark, we can generate a matrix that shows the amount of data which are transmitted during 60 seconds run, when the Client2 begins the transmission. These values will show an overview of how Clients send data during this time. And with that, we can continue to calculate the Fairness and Link Utilization. The Fairness can be calculated using the subtraction of the total data transmitted in 60 seconds of two Clients. And the Link Utilization can be calculated using the average values of total bandwidth usage per second of two Clients. The Responsiveness can be examined using the I/O graph of Wireshark, to see the time the Clients need to change the throughput (or Congestion window) to the average level, which indicated that the Congestion window of two Clients are at the same level.

For the experiment results, first we will evaluate the congestion window behavior of the protocol, after that, we will examine the metrics with the following order: the total data transmitted, the fairness, the link utilization, and lastly the responsiveness.

3.3.1 *Cubic*

With Cubic algorithms, The Congestion Windows of QUIC behaves quite stable like TCP. The ByteInFlight of both flows oscillates in the range between 30 and 50 thousand bytes. However, with the QUIC case, the congestion window is quite stable during the first 10 seconds and the last 10 seconds due to its algorithms. The explanation for this could be that the QUIC has a set fix of Congestion Window which makes QUIC Congestion Window to not increase higher. This could be changed through coding and function in the program. However, this value will affect the result of the Responsiveness. Because of that, we will implement the program with the same parameter as Linux modules of TCP congestion control.

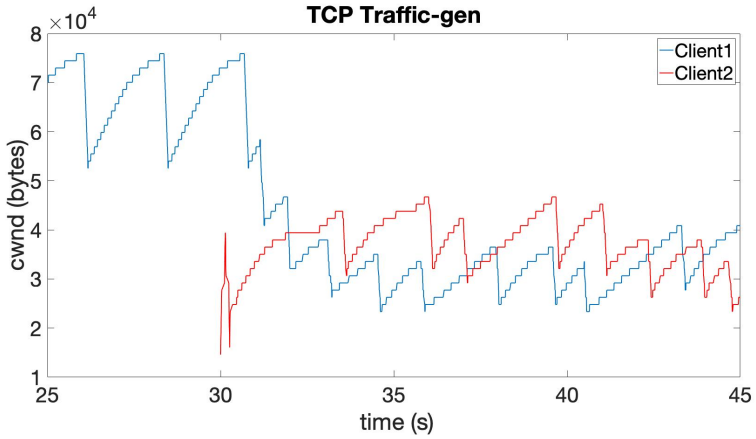

Figure 3.2: TCP run Cubic algorithm

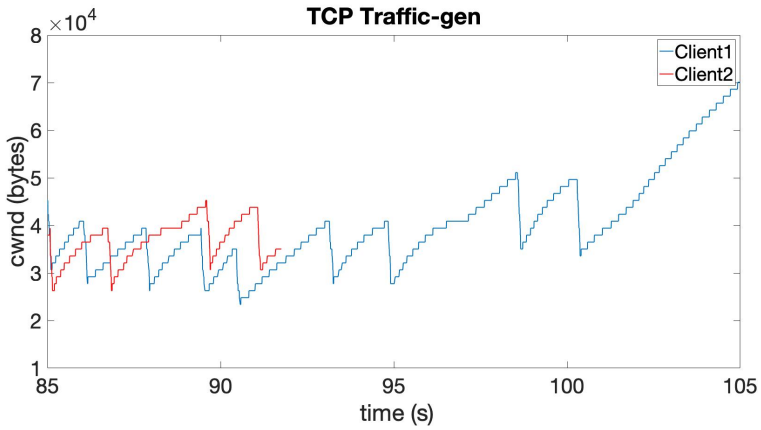

Figure 3.3: TCP Client2 run Cubic algorithm end

In the above figure of TCP Cubic's congestion window using TCP protocol, we can see the behaviors as of the following: We can see that at the first 30 seconds and the last 30 second, the TCP congestion window has a static recovery process, which has a convex and concave

part. However, with the QUIC cubic, the algorithms work not similar to what we expected. The behavior of the congestion window doesn't work like planned. As a result, the comparison using the congestion window is not possible, due to coding issues. Instead, we will compare both protocols base on the throughput, which can be acquired using Wireshark. With Wireshark we can easily check the real data sent through the link and make sure that all the packets are sent in the right type, as well as the data will be received correctly.

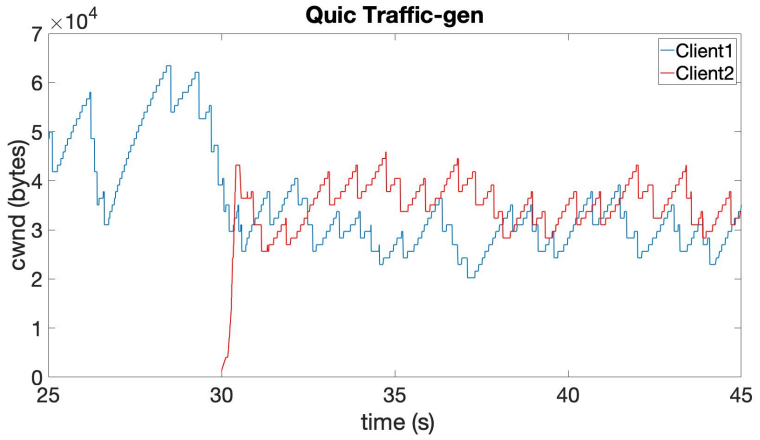

Figure 3.4: QUIC run Cubic algorithm

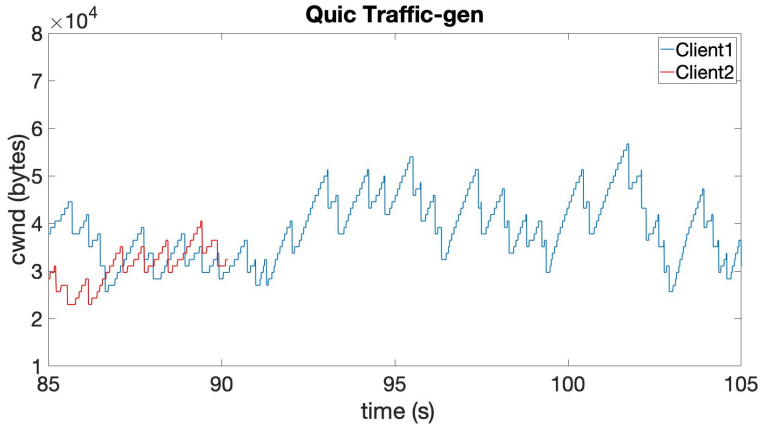

Figure 3.5: QUIC Client2 run Cubic algorithm end

The Responsiveness of QUIC using Cubic heavily depend on the max congestion window, which QUIC handles. The time to respond to the first flow is based on how large is the congestion window, which QUIC calculated.

However due to the link, which we setup, is only 5 Mb. With the formula, we can estimate the biggest congestion window we can has:

$$max\_cwnd(bytes) = \frac{5 * 1000000}{8} \approx 625000(bytes) \tag{3.4}$$

which approximately 625000 bytes, for TCP, the congestion window can not excess 428 segments (1460 bytes pro packet). For QUIC, the congestion window can not excess 462 segments (1350 bytes per packet)

With the information which is filtered from the Wireshark, we can compute the data in Matlab and results are shown in Figure 3.6 as below:

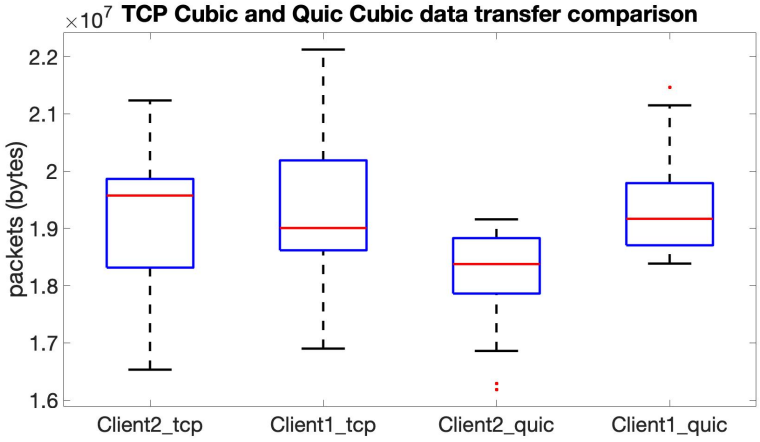

Figure 3.6: TCP and QUIC data transmission comparison

The data transmission of Client1 and Client2 using TCP shows that the data transmission between two Clients is fair when both clients show that they have a similar average data range. The highest and lowest sum of data transferred using TCP is more drastic when compared to QUIC.

Overall, QUIC has a lower mean for both Clients than TCP, which means TCP still can transfer more data than QUIC. However, QUIC still has some advantages when we see the difference values. This value indicated that Client1 and Client2 when using QUIC has quite stable data transmission than TCP. The amount of data transfer of QUIC is more stable for 30 runs.)

On the other hand, TCP still has a better division of data flows between two Sender when both the max and min values of two Clients run TCP are almost identical. When looking at Client1, which runs QUIC, it seems to have the same performance with Client1, which runs TCP for 120 seconds. For the 60 seconds with congestion avoidance,

QUIC performance is not better than TCP.

Next we evaluate the performance of congestion avoidance of Cubic algorithms in the 60 seconds both Clients run:

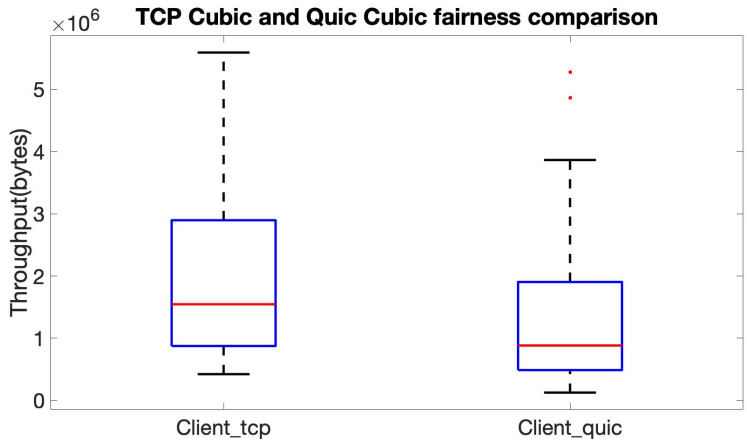

Figure 3.7: TCP and QUIC fairness comparison

In Figure 3.7,the fairness is indicated by y-axis. The fairness is calculated with the subtract of both clients' data transfer in 60 seconds. When throughput equal zero is a perfect balance between two Clients, that's mean both Clients share 50:50 the link bandwidth. The fairness values of the y-axis are presented in absolute value, which means the bigger the values, the worse fairness the protocol has. In the plot, we can conclude that the QUIC has better fairness with the mean value lower than TCP, which means both Clients has a fair share of link bandwidth for 30 runs. However, QUIC still has two outliers values, which are outside of the box Client_QUIC, that's mean there are also possibilities that QUIC has unfair cases.

In Figure 3.8, we can check how good TCP and QUIC can utilize the link when the congestion occurs. With the equation 3.1, we have calculated the maximum throughput of the link, that's mean the closer the data sum of both clients with this link capacity, the better utilization the protocol has.

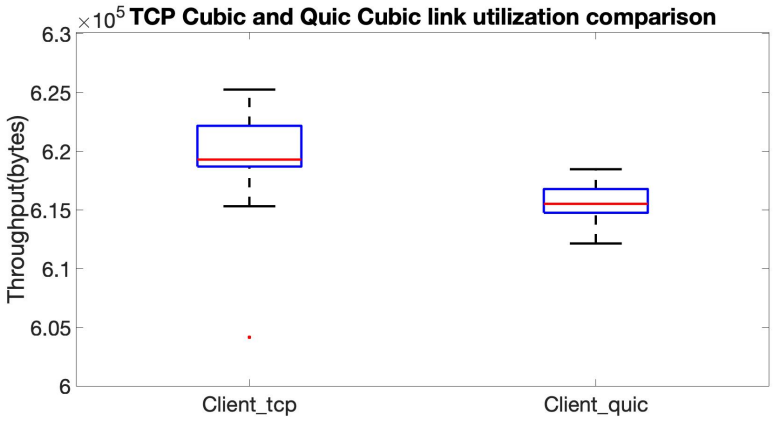

Figure 3.8: TCP and QUIC link utilization comparison

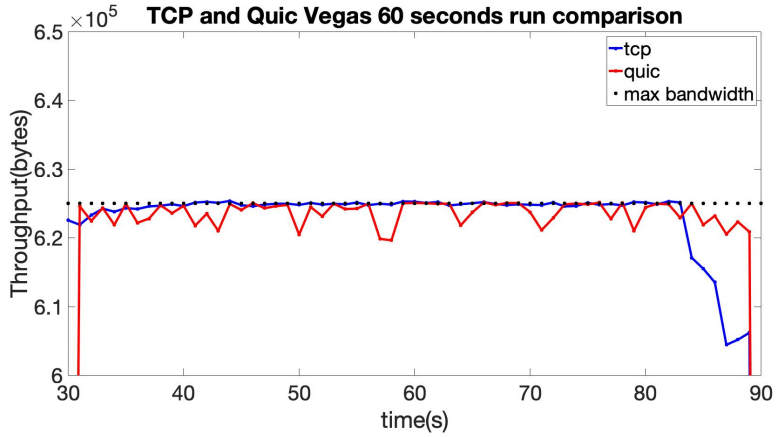

Figure 3.9: TCP and QUIC link utilization comparison in 60 seconds run

From Figure 3.8, we can conclude that the TCP still has better link utilization that QUIC even though it is not too much(the difference is lower than 15000 bytes). Besides the mean value of 30 runs of TCP higher than the mean value of QUIC, the min value of TCP is still higher than the max value of QUIC. So in this metric, TCP still win over QUIC.

The last metric we want to check is the responsiveness of the congestion control. Because the convergence is not static, so we can just only check the responsiveness of the congestion control using the time between the time Client 2 get into the transmission, which is 30th second, and the time both Clients have the same throughput in the transmission, which mean they have almost equal congestion windows. The faster both Clients get into this balance value, the better the responsiveness the protocol has. In hereafter 30 runs. we just show average responsiveness of both protocols.

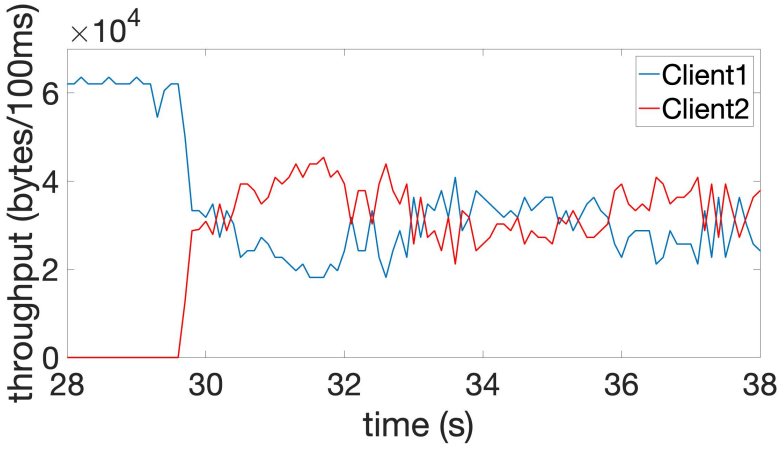

Figure 3.10: TCP Cubic responsiveness

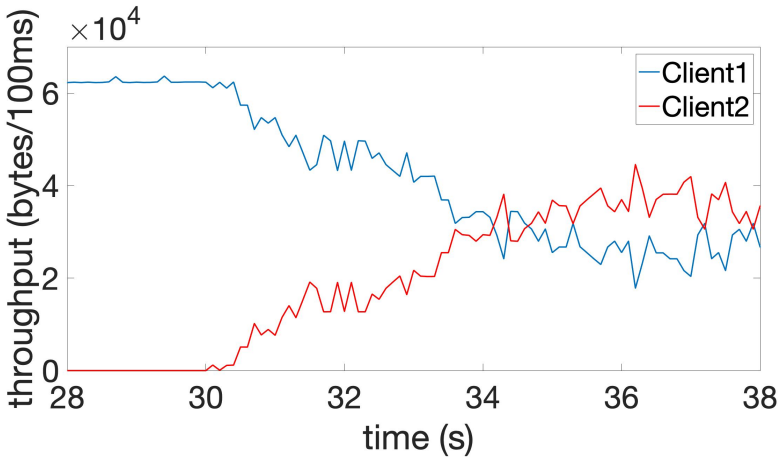

Figure 3.11: QUIC Cubic responsiveness

Figures 3.10 and 3.11 are showing the responsiveness of TCP and QUIC. Most of the time TCP has quick responsiveness when the Client2 joins the transmission. The respond time of TCP is average below 1 second when the respond time of QUIC is average 3 seconds until both Clients remain to converge. In this case, we can conclude that the Cubic behaviors in TCP and QUIC are not similar in responsiveness category.

As can be seen in the first experiment, QUIC still does not outperform TCP using Cubic. QUIC is just slightly better than TCP in fairness criteria, while other metrics are not greater. The performance of QUIC may get better when we setup some loss ratio into the experiment. When the bandwidth is large, TCP still overperform QUIC.

3.3.1.1 *Cubic with 0.005 loss ratio*

In the second measurement, we add some loss ratios for both protocols. In Modify traffic setting of Emulab, we change the value loss ratio into 0.005 to generate some loss into the traffic. Most of the time, we want to check the performance of protocols in a lossy environment.

In Figure 3.12 the Clients of QUIC still have more balanced data transmission than TCP, while the amount of data transfer between both Clients runs TCP is almost identical with the last zero loss Cubic case. When compare to zero loss ratio experiment, QUIC in this experiment has more fairness in transmission, but with bigger values range between the max and min values. That means in the loss case. QUIC using Cubic does not have better stability transmission than TCP. But the distribution of loss, which created by Emulab is not even, so maybe it affects the performance of QUIC.

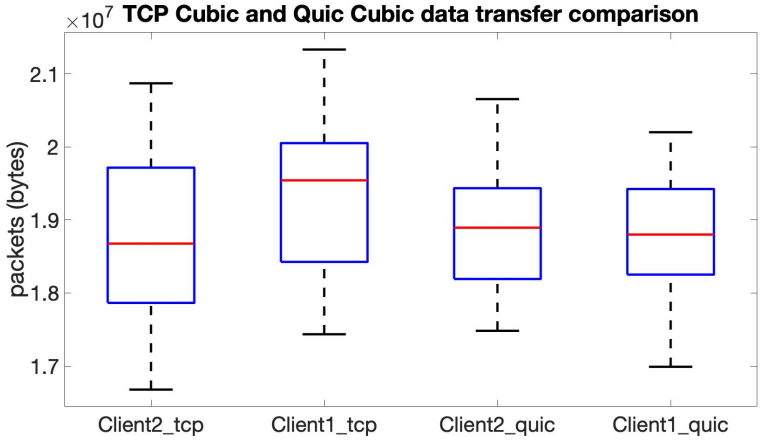

Figure 3.12: TCP Cubic and QUIC Cubic data transfer comparison 0.005 loss

The fairness of QUIC in loss case still win over TCP, while the min values of both protocols are almost equal, the mean values of QUIC are still lower than TCP. In this case, QUIC max values are smaller than TCP and without any outliers, meaning QUIC still has stable fairness even in the loss case.

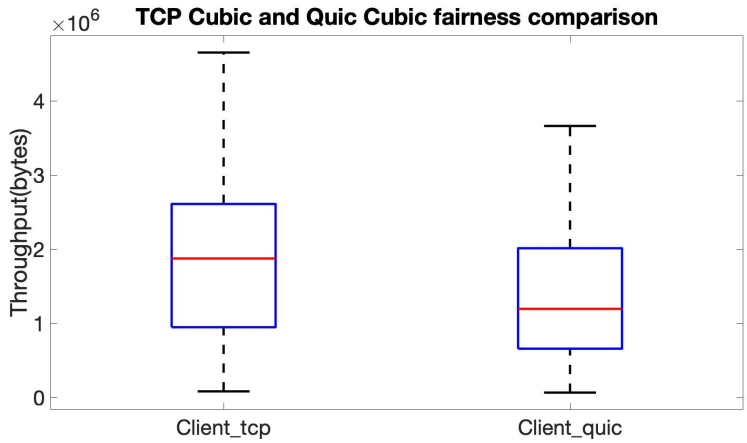

Figure 3.13: TCP Cubic and QUIC Cubic fairness comparison 0.005 loss

There is some unfairness problem between TCP and QUIC when they run together. The problem has been described and proved in [8](Molavi et al.,2017). In most cases, QUIC always gets much more bandwidth than TCP, which makes TCP have worse transmission. This unfairness shows clearly the more TCP transmission are taken place in the link. This unfairness has been proved that it is not a result of a testbed because both protocols have also been tested in Google servers, so we can assure that, this problem of unfairness, as well as fairness, is identical, even with Emulab. In our experiment, QUIC shows that when it is the only protocol in transmission, it has better fairness while in [8](Molavi et al.,2017), QUIC and TCP show the big unfairness with each other. That why we should not use both protocols together.

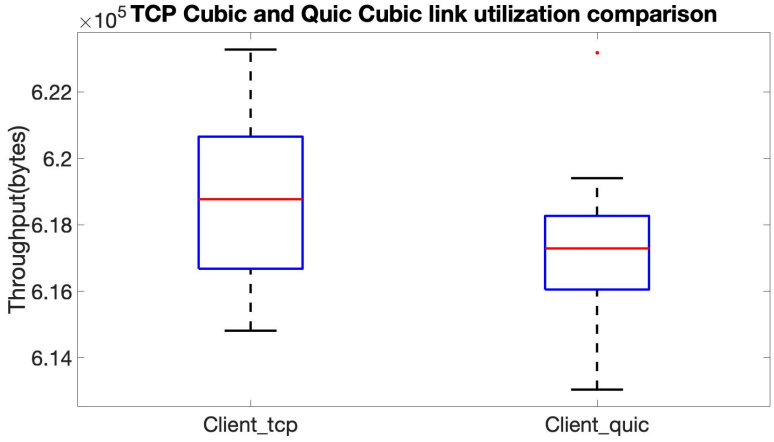

Figure 3.14: TCP Cubic and QUIC Cubic link utilization comparison 0.005 loss

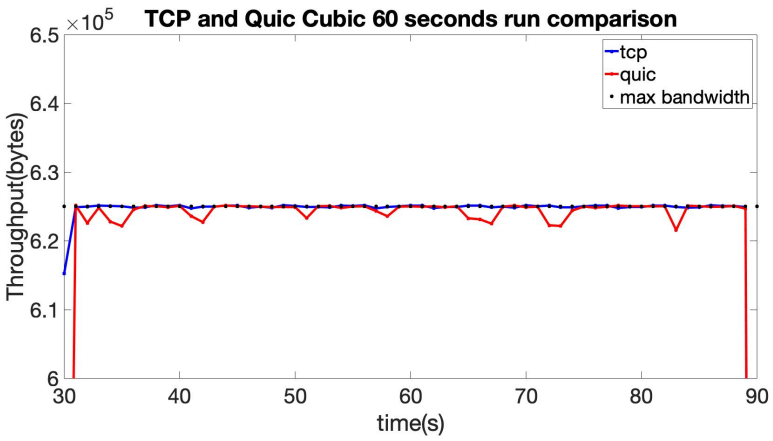

Figure 3.15: TCP Cubic and QUIC Cubic link utilization comparison 0.005 loss in 60 seconds run

By link utilization comparison, QUIC and TCP Clients also have more similar values than the experiment with zero loss. The utilization of TCP is still higher than QUIC however, the different is not much. This difference can be enormous when the latency is high. In [3](Gratzer,2016), it shows that TCP only outperforms QUIC with two conditions fulfils, the bandwidth must big and the latency is low. The bandwidth and the latency we set in Emulab is not favor QUIC so that is reason why QUIC can not have better utilization than TCP. By analyzing the behavior of Cubic for both QUIC and TCP, we can see that TCP still have better congestion control with this algorithm than QUIC.

In Figure 3.15, TCP maintain quite stable performance of keeping the maximum utilization when compared to Quic.

When compare all the metrics: fairness, link utilization, stability, TCP overall have much better results in channel utilization as well throughput than QUIC. The reason for this maybe depend on the condition of the networks, when the network is stable and without much loss, TCP can still be outperforming QUIC in non-website transmission.

3.3.2 *RTT Vegas*

Some problems with TCP Vegas have been described in [20](Richard, Jean, and Venkat,2004). TCP Vegas can not detect the change when rerouting in the network occurred. The problem with finding the optimal constant values α, β, γ is also depended on the systems or testbed, which we run the Vegas on. But most of the time, finding the RTT values of Vegas has been one of the most difficult things to do when researching the behavior of it. These values stay deep below the kernel level of Operating systems. In this experiment, because we want to use nodes, which run the Linux kernel. We have to use the same constants, which runs in the script make Vegas module in the kernel. Because we using QUIC on the application level, so the change of these values is not difficult. Besides we can get the information of QUIC parameters, especially RTT values such as *smooth-RTT, minimal RTT, latestRTT*. With these values, we can get clearer view of the behavior of Vegas algorithm, and the results of the implementation.

On Figure 3.16 we can see the latest RTT update for the 120 seconds transmission. As the assumption about Vegas, the RTT of QUIC using Vegas increase quite stable, with some values increase from the mainline, which is the values around the min RTT. During the run, it shows lots of packets, which has lots of higher RTT, due to the retransmission of drop packets.

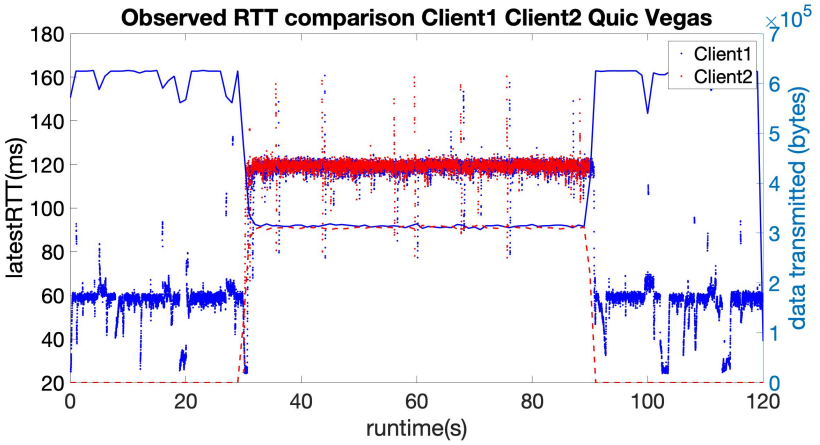

Figure 3.16: QUIC Vegas measurement with latest RTT behavior

The continuous blue line described the throughput of Client1 through 120-second transmission when the dashed line described the throughput of Client2. Because of the loss ratio, during the transmission, we can see the large packets drop at the start period and end period of the Client 1 transmission. When Client 2 intercepts the transmission, Client1 has an increased of RTT, equal to Client 2. Because both have the almost same RTT period, the throughput in this phase of both

Clients are almost the same.

3.3.3 *RTT comparison of Vegas*

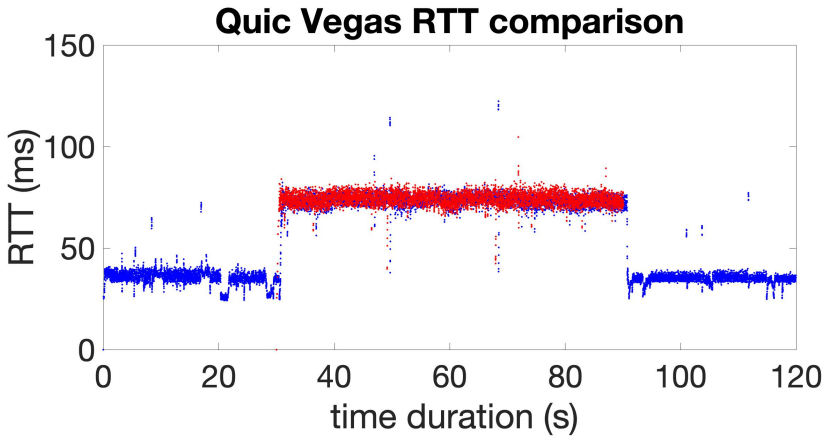

Figure 3.17: QUIC RTT comparison 0 ratio loss

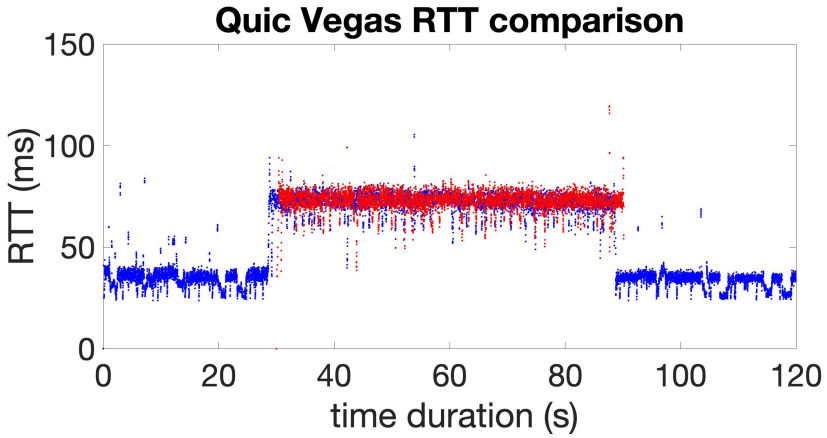

Figure 3.18: QUIC RTT comparison 0.005 ratio loss

In Figure 3.17 and 3.18, we have two RTT comparisons from both zero loss and loss cases. In zero loss case, we can see that the RTT is much lower, the time to sent data and gain ACK back at the sender fluctuate in the range of 20 to 30 milliseconds, which is quite small during the whole transmission. In loss case, the RTT is much higher when it fluctuates most of the time of 100 milliseconds, which is quite high, with this high RTT, it can affect the congestion window during the transmit. That can be a reason why the transmission phase of both Clients using QUIC is more fluctuated than zero loss case. Overall, there are more packets with increased RTT in this loss case, some of them have an about 200-millisecond delay, which are dropped because

of high transmitted rate and the topology, which are all used Drop-tail Routers.

When compare with TCP Vegas RTT usual RTT in Linux [28](A. Vikram Singh,2013), most on the time, we can see that QUIC has stable RTT of each packets without much difference, however the average RTT of QUIC Vegas is higher than TCP Vegas.

3.3.4 *Vegas*

By Vegas, we do the same process as with the Cubic, we will measure the data based on these metrics:

- Total data transferred during 60 seconds process for fairness. The best ideal case that both Clients can transfer an equal amount of data.
- Link utilization, the closer the amount of data the Clients can transfer when compared to the maximum bandwidth, the better the utilization the protocol had.
- Responsiveness of the protocol when the Client 2 join in the transmission.

For the Vegas algorithm, the TCP congestion window would be obtained using Tcp_probe. And the QUIC Congestion Window would be obtained by the values *v.congestionwindow* in vegas_sender.go. Because the running process is 120 seconds, we only care the 60 seconds part, which Client 2 begins to join the transmission.

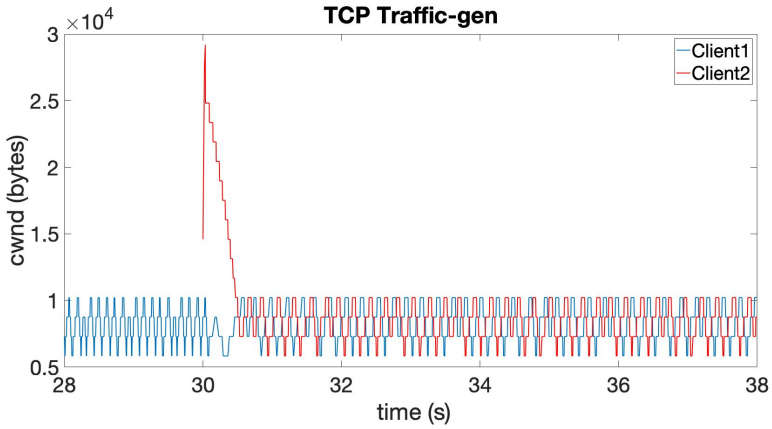

Figure 3.19: TCP run Vegas algorithm with 0 delay

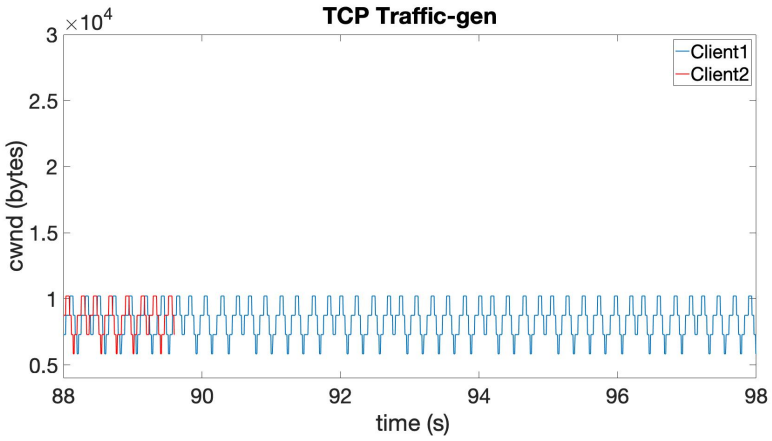

Figure 3.20: TCP Client2 run Vegas algorithm with 0 delay end

TCP works quite well with Vegas in case the delay is so small, almost zero. But in the case of delay, there is a problem with Linux implementation so that there is some large unfairness between two flows. The congestion window in zero-delay cases rises after about 3 seconds of a slow start, the Client2 starts to begin congestion avoidance immediately, and after that, it controls congestion window similar to Client1.

When we set the delay about 10 milliseconds so that minRTT is equal to 20 milliseconds. The unfairness of TCP Vegas increases tremendously. In this case, Client 2 when it joins the transmission, the congestion window of Client 1 reduces a lot and lost its smoothness of the first phase 30 seconds of its transmission.

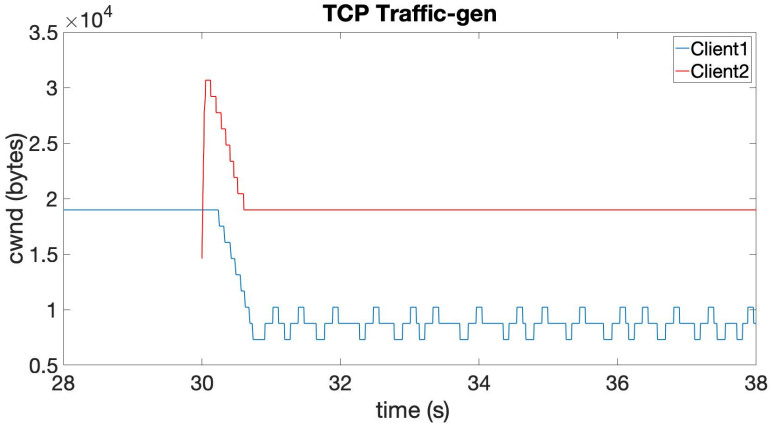

Figure 3.21: TCP run Vegas algorithm with 10 ms delay

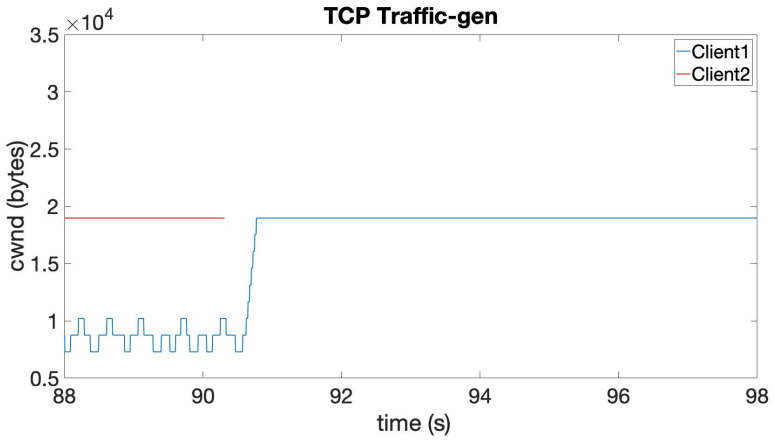

Figure 3.22: TCP Client2 run Vegas algorithm with 10 ms delay end

This unfairness is happened due to the setting problem of Slow-start Threshold [29](X.Wei,2006), which can sometimes cause the unfairness problem between Vegas flows due to the implementation of tcp_vegas.ko in Linux modules. In order to minimize this problem, we need to increase the loss ratio. Otherwise, we have to run more test of Vegas so that the results do not differ with each other because of bad coding.

On the other hand, the congestion window of Client2, which runs QUIC, has more stability than TCP. During the slow start, the QUIC has a slow slow start like TCP, but the decrease to converge value is faster than TCP. During the transmission, the Client2 QUIC seems to not change much and does not try to increase the congestion window in the meantime. This will be changed if the loss ratio is higher (about 0,1 percent) then the congestion window will run almost identical to TCP.

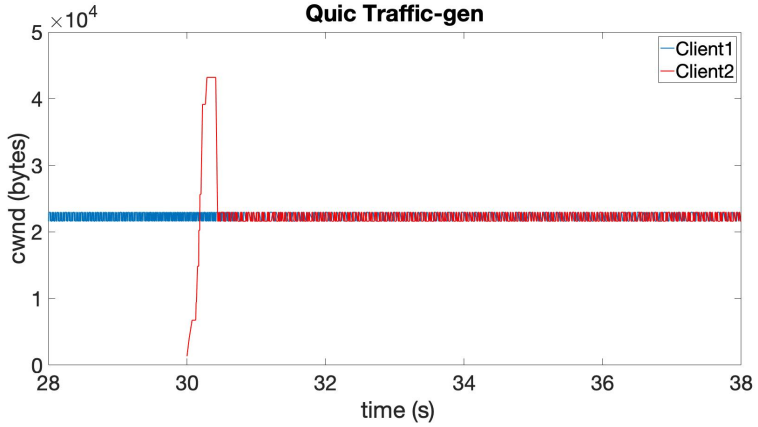

Figure 3.23: QUIC run Vegas algorithm

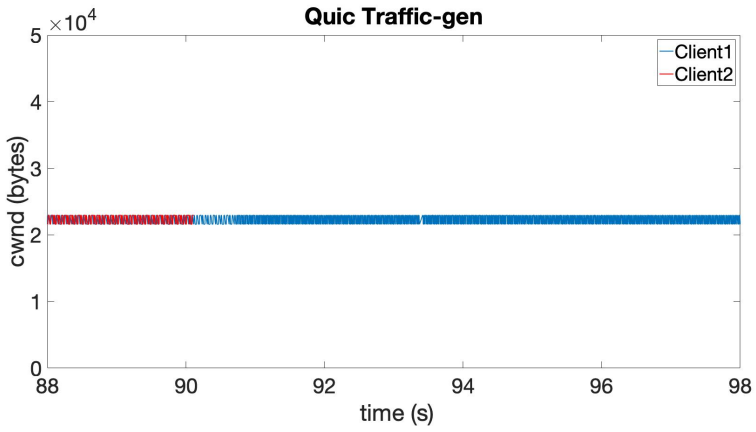

Figure 3.24: QUIC run Vegas algorithm

Because the evaluation based on zero loss with TCP Vegas is trivial, the throughput of both Clients are so smooth when running with TCP, so we need to add some noise so that we can see the better result of the algorithms. We will measure Vegas with both with and without loss. We choose the loss rate for the second part equal to 0.005 ratios. Because the higher ratio will make the results so different with each measurement so that it is difficult to draw a conclusion.

After the second measurement, we can see that. the congestion window of TCP using Vegas, in this case, is no longer smooth as before. The slow start phase of both Clients still increases a lot, until it gets into the congestion avoidance phase. When in congestion avoidance, the congestion window still fluctuates in a small range till the Client 2 join the link. After the same Slow start as Client 1, both Clients used the almost same congestion window range, which below the first phase of Client 1 in 3.25.

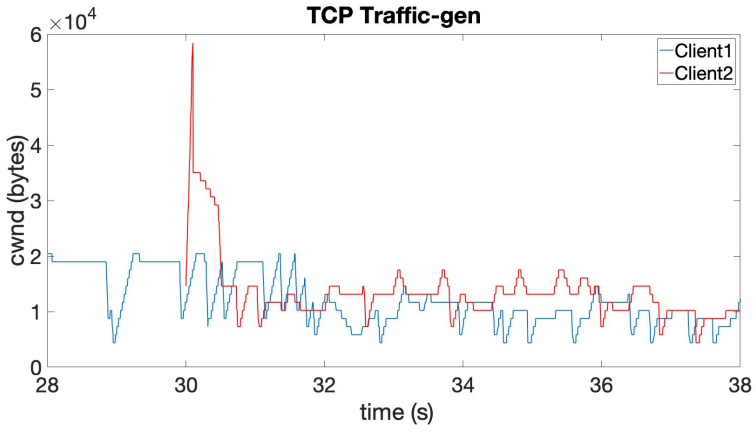

Figure 3.25: TCP run Vegas algorithm 0.005 loss ratio

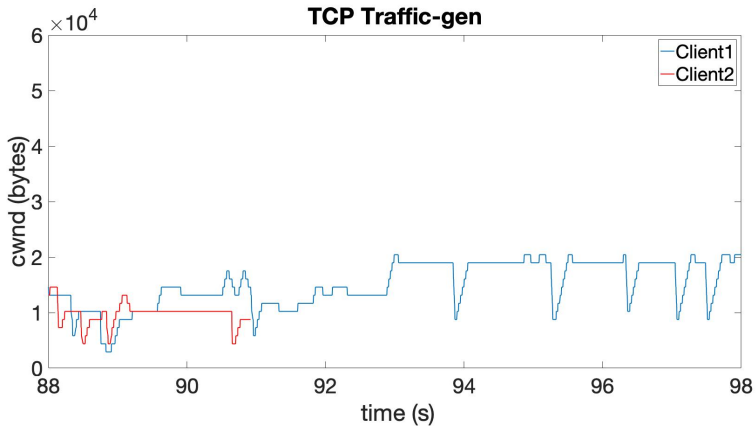

Figure 3.26: TCP Client2 run Vegas algorithm 0.005 loss ratio end

However, In this case with loss, QUIC behaves quite equal with it in the first QUIC case without loss, when compared with TCP. TCP has a huge fluctuation because of the loss link. The RTT differs causes the difference value change, results in the switching of the congestion window. After the Slow start phase, the congestion window of QUIC is maintained much higher than TCP in 3.27. This congestion window shows that the Client1 has quite stable results so that the congestion window does not change drastically like TCP even when the loss occurs. When the Client2 joined in, it gets no chance to increase the congestion window high as Client 1. And during this phase, the fluctuation of both Clients is much lower than the case with TCP Vegas. Because Client 2 get the same congestion window as Client1, we can assume that the fairness comparison in this case with a loss ratio of Vegas would be better than TCP.

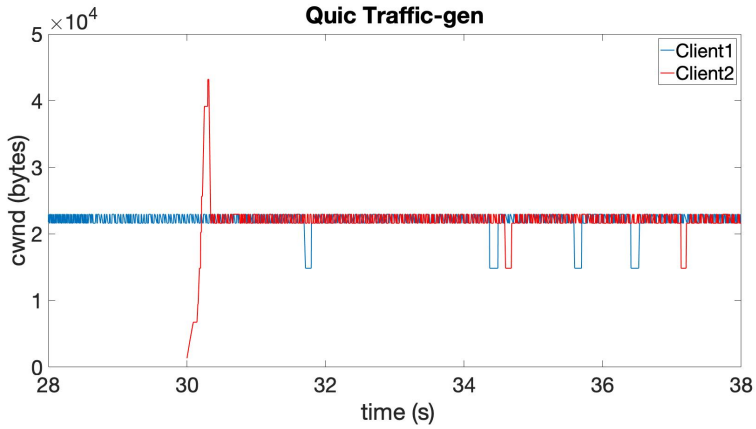

Figure 3.27: QUIC run Vegas algorithm 0.005 loss ratio

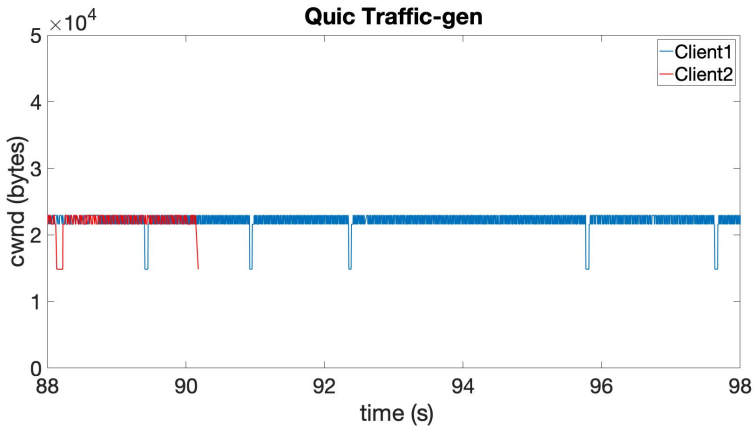

Figure 3.28: QUIC Client2 run Vegas algorithm 0.005 loss ratio end

In Figure 3.27, we can see that this ratio of loss does not affect the result of QUIC Vegas. In the Congestion Avoidance phase, the congestion window still maintains the range of stability, the same as the case without loss. So we have to try to increase the loss ratio to see if the QUIC will have the same congestion window behavior or not.

So in Figure 3.29 is the congestion window behavior when we set the loss ratio in Emulab equal to 0.1. The congestion window of QUIC in this phase becomes less stable and begin to fluctuate just like TCP Vegas 0.005 loss case.

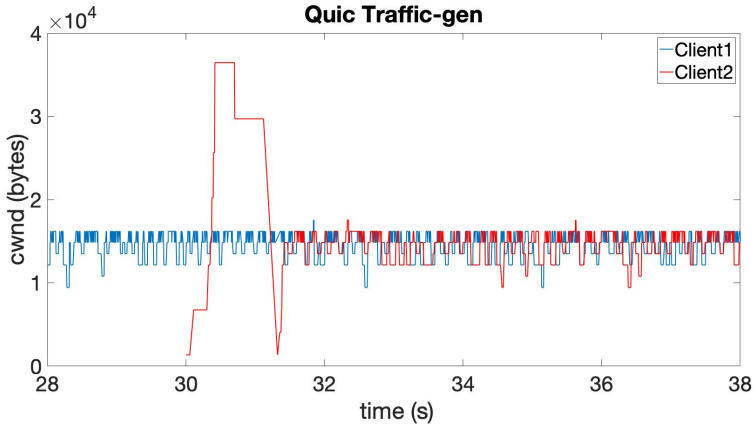

Figure 3.29: QUIC run Vegas algorithm 0.1 loss ratio

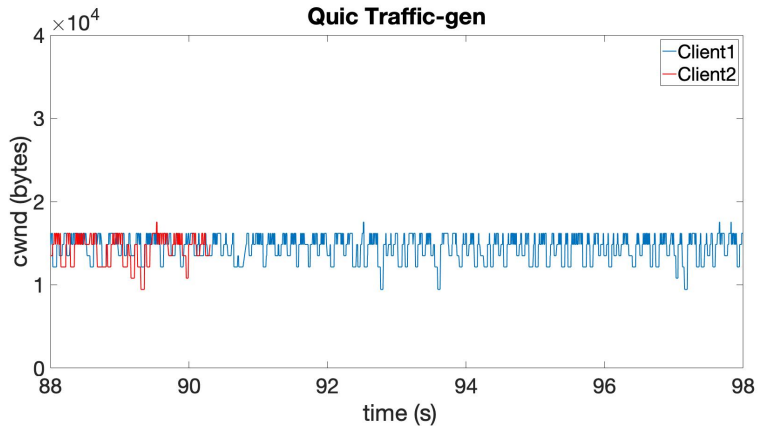

Figure 3.30: QUIC Client2 run Vegas algorithm 0.1 loss ratio end

After comparing the congestion windows, we can see that Vegas seem to work well with QUIC almost equal TCP. The throughput of QUIC seem to better than TCP and because of better RTT calculation, QUIC has better resistance against loss than TCP.

So get a better conclusion, in this evaluation, we still using the throughput comparison, the link utilization as well as the fairness as the main values to compare both protocols.

3.3.4.1 *Vegas with zero loss ratio*

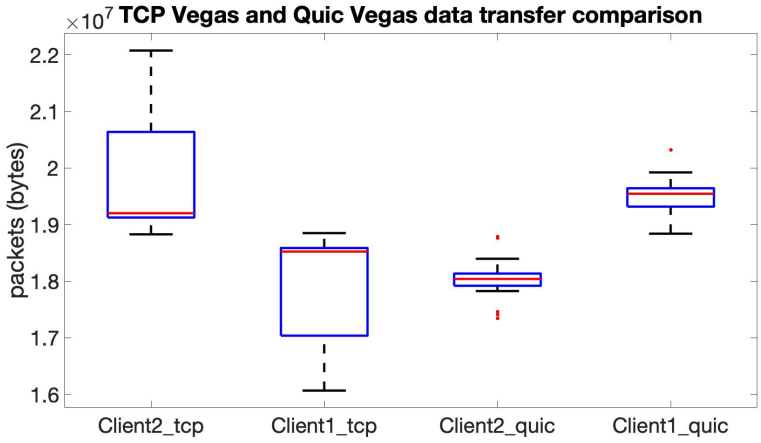

Figure 3.31: TCP Vegas and QUIC Vegas data transfer comparison zero loss

In Figure 3.31 Client2, which runs later, has take lots of link bandwidth when compared with Client1. However, the average data transferred of Client2 and Client1, which runs TCP, are closer to each other than when they run QUIC. However, Clients, which run QUIC, shows that they have better stability than TCP in this case.

The Difference between max and min values of Clients, which run QUIC are lower than Clients, which run TCP. So in this case, we can assume that QUIC run Vegas in condition zero loss are working more stable than expected.

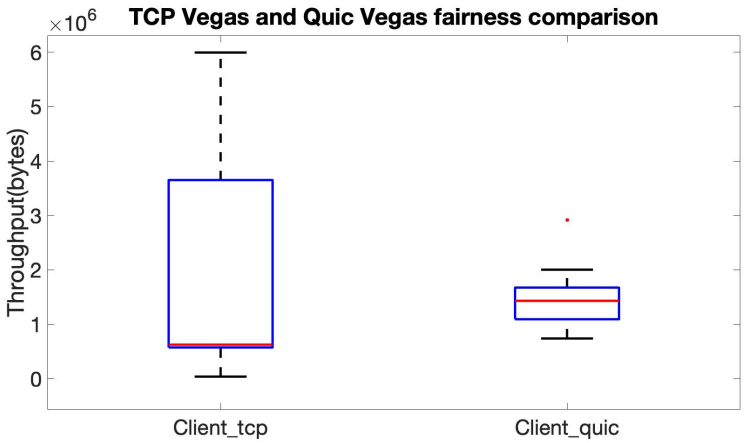

Figure 3.32: TCP Vegas and QUIC Vegas fairness comparison zero loss

Next, we examine the fairness of both Protocols, in Figure 3.32, TCP shows unexpected results with some high degree of unfairness. With Figure 3.31 We can conclude that in some cases, Client2 with TCP had taken more bandwidth, transferred more packets than Client1 with TCP and this behavior creates a massive unfairness in the transmission. However, the mean of TCP is still lower than QUIC so we can say that TCP while does not have the stability of QUIC, it still got the better fairness result.

On the other hand, Client1 and Client2 run QUIC have more stability, and the link is shared fairer, quite different when compared to the last case with Cubic, when the Client2 join the link for the transmission, Client2 is always suffering the loss of bandwidth due to the unfairness.

Overall, when look at the means values in Figure 3.33, TCP still have a better fairness than QUIC when using Vegas with only 0.73 million bytes different than optimal value, which is zero value of y-axis. But for stability, then QUIC may have upper hand in this case over TCP.

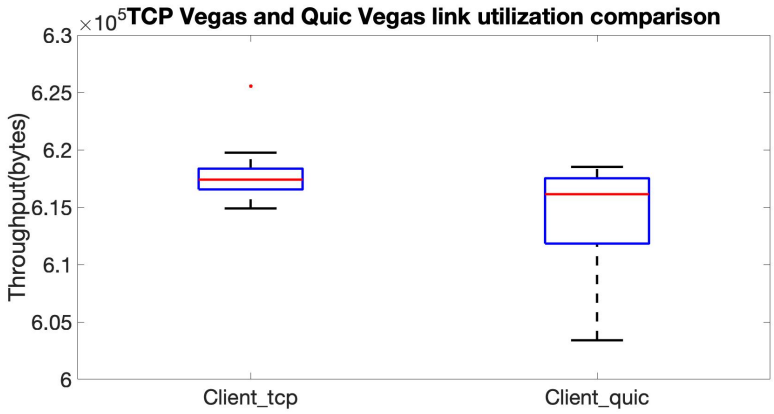

Figure 3.33: TCP Vegas and QUIC Vegas link utilization comparison zero loss

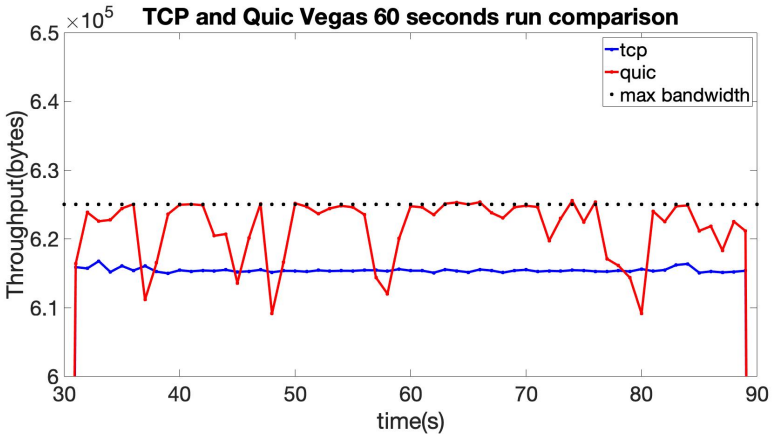

Figure 3.34: TCP Vegas and QUIC Vegas link utilization comparison zero loss in 60 seconds run

Both protocols show quite similar performance in this comparison. The means values of both TCP and QUIC are almost equal. So in this case, QUIC is almost good as the TCP.

However when we compare the 60 seconds run, Quic has higher utilization than TCP but the performance is not stable through the whole transmission.

When combine three comparisons, we can safely assume that, TCP Vegas algorithms work better with QUIC than TCP Cubic. While it still suffers from the fairness problem between two clients, the Link utilization and the data transfers are definitely works better than QUIC Cubic. The link utilization in this QUIC Vegas zero case has also improved a lot so that it safe to say QUIC has slightly better congestion control in this experiment than TCP.

3.3.4.2 *Vegas with 0.005 loss ratio*

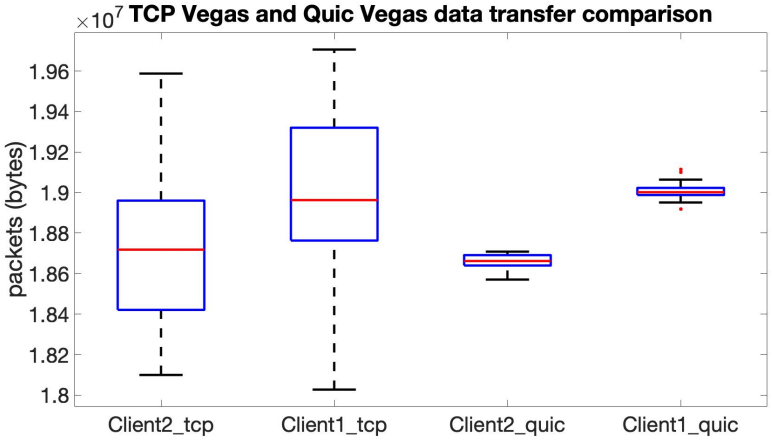

Figure 3.35: TCP Vegas and QUIC Vegas data transfer comparison 0.005 loss ratio

When comparing the amount of data transfer of the TCP and QUIC using Vegas, it shows that the QUIC still has a better balance and stable than QUIC. For the 30 transmissions, QUIC from Client1, which begins first, has a better data transmission during 60 seconds joining of Client2. The max, min values of this amount in Figure 3.35 prove that QUIC has shown the stability for both Clients. Client2, which runs TCP using Vegas suffers quite a lot when comparing the transferred amount with the data amount of Client1. The outliers of both Client1-QUIC and Client2-QUIC show that Client1 in 30 runs has multiple times take more link bandwidth to transfer the data than Client2 and it is so greedy when compared to TCP. So in the case of evaluation of throughput, we can conclude that QUIC Vegas has more stability than TCP Vegas for both Clients, while these have run the same settings.

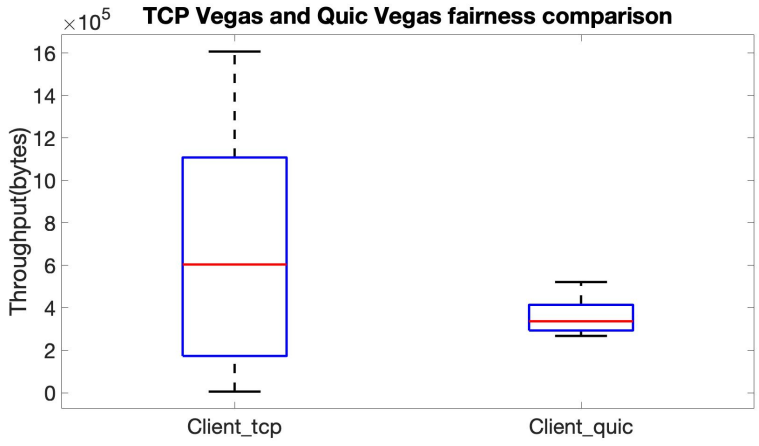

Figure 3.36: TCP Vegas and QUIC Vegas fairness comparison

To evaluate fairness, we have to evaluate the difference between the data transmitted of two Clients. So that the more the results equal zero, the better fairness the protocol has. Although in Figure **??** QUIC Client2 shows that it always sent lower than Client1, which means it should have worse performance in fairness than TCP, the value shows that because of the stable transmission through 30 runs, QUIC has better fairness performance than TCP. In this case, QUIC shows that it has a better average value than TCP, which means the mean value is closer to zero. When compared with Figure 3.35 we can conclude that Client1 run Vegas take more bandwidth to transfer data than Client2 for both protocols. So the fairness of QUIC is better than TCP thanks to the stable transmissions.

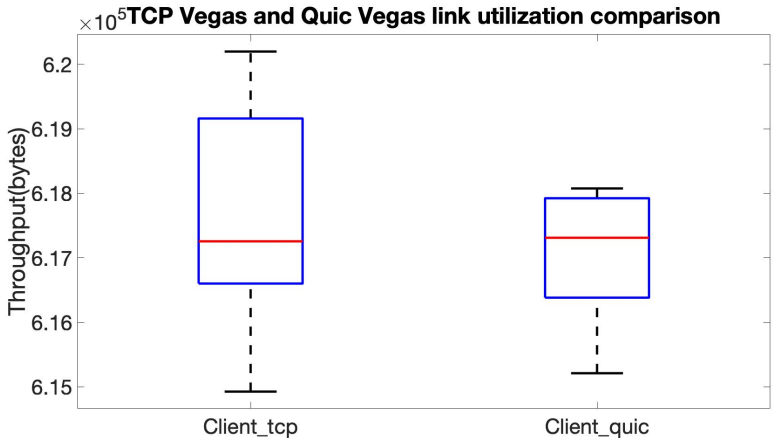

Figure 3.37: TCP Vegas and QUIC Vegas Link utilization comparison

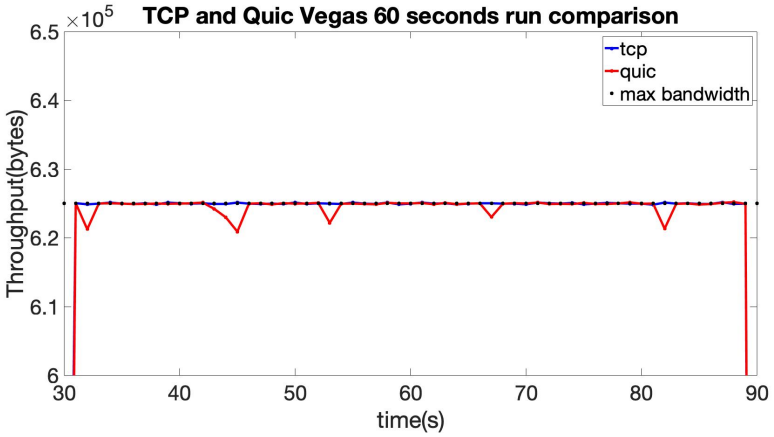

Figure 3.38: TCP Vegas and QUIC Vegas Link utilization comparison in 60 seconds run

In Figure 3.37 we can see the link utilization of both protocols. When the max link bandwidth is about 625000 bytes, TCP shows that the link utilization quite variant than QUIC. The difference of Max and

Min values is greater than Difference of QUIC while the means values of both protocols are almost identical.

In Figure 3.38, both protocols achieve almost perfect link utilization, while in some intervals, Quic still have some downgrade in the performance. But we can safe to say than both protocols keep the utilization quite high and stable in case of noise environment.

Overall, TCP and QUIC has quite similar utilization with the average values are equal with approximately 6.172. However, when compare with Link utilization of Cubic cases, QUIC with Vegas shows that it. in some cases, can achieve the level of link utilization of TCP.

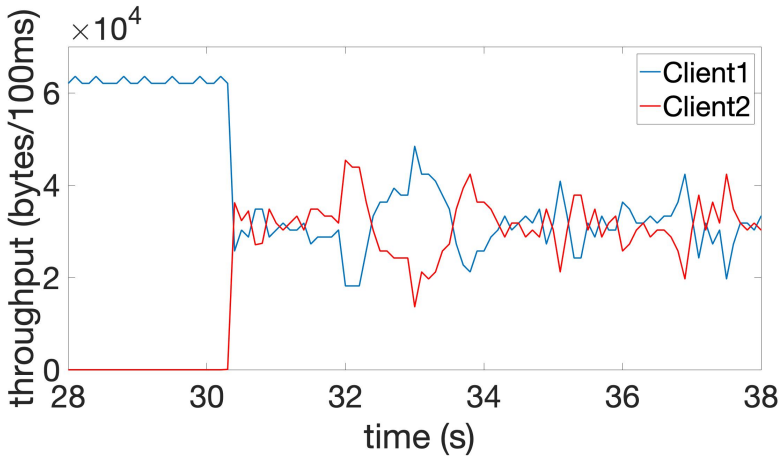

Figure 3.39: TCP Vegas responsiveness

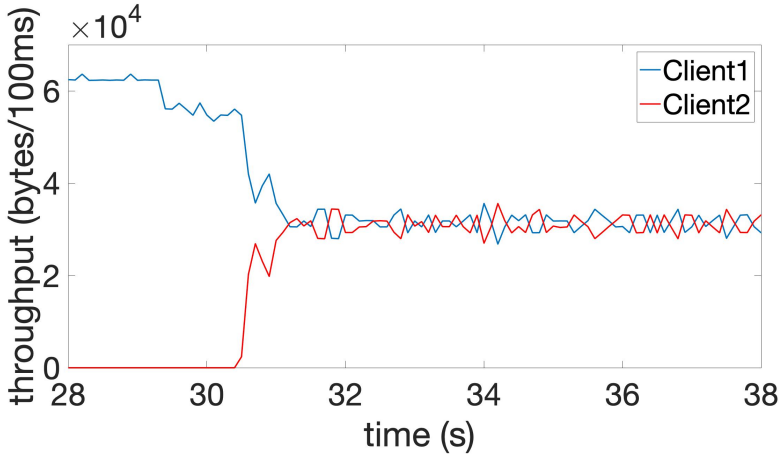

Figure 3.40: QUIC Vegas responsiveness

In 3.10 shows the first 5 seconds when Client2 using Vegas join the transmission flow, we can see that the responsiveness of both protocols are quite different. TCP seem to have more trouble to balance of

two flows. While QUIC using Vegas has some stable degree in the transmission in 3.11

In Conclusion, QUIC with Vegas algorithm has made a better performance than TCP in this case with loss deployed in the link. While QUIC win over TCP using Cubic only in fairness criteria. in loss case, QUIC using Vegas has achieved the link utilization level of TCP with high degree of stability during transmissions. This means the algorithm of Vegas works well with TCP thanks to the easier selections of RTT values and implementation of constant values in Golang.

4

CONCLUSION AND WORKS IN THE FUTURE

4.1 CONCLUSION

After this evaluation, we can conclude that QUIC, when it is not using New Reno as [5](Iyengar and Swett,2019) said. QUIC pales in comparison to TCP in a lot of metrics. While it still works well when with the link utilization, other metrics are behind and not better than TCP.

With Cubic, QUIC has an almost equal implementation like TCP, the slow start, as well as congestion avoidance, work almost perfectly identical, although QUIC still suffer a little worse performance than TCP.

With Vegas, QUIC has a quite high throughput while it has zero loss ratio, however, in this case, its congestion window algorithms does not behave like TCP. When we increased the loss ratio, QUIC Vegas shows that it has a huge resistance against loss, while the congestion window changes differently from TCP. It shows smoothness and a stable congestion window, which we predicted it should have like the Vegas behavior in TCP. The fairness, in this case, is also different, while during the zero loss case, TCP Vegas has better fairness, but when we increase the loss ratio, QUIC Vegas shows that both its Clients have better performance while maintaining a high level of fairness. So in this case with Vegas, QUIC while suffering a little lower performance in throughput against TCP, it shows lots of potentials when working in lossy condition. We may say that QUIC Vegas win over TCP Vegas in this experiment.

Since QUIC is implemented by Google, so in the means time, it will be better only in some of Google program such as Chromium with load page time, YouTube buffering rate, etc.
And the power of QUIC based on UDP, which can increase when working with the Multipath modules, which is why we can expand this work with Multiple path TCP and Multiple path QUIC. Which hopefully, that QUIC can save some reputation in the battle with TCP.

4.2 WORKS IN THE FUTURE

We can expand the work into Multiple path cases which may increase the chance of QUIC beat TCP due to the abilities to expand to flows

and paths of QUIC using its way of Numbering Packets.

We can also fix and update the implementation of Cubic and Vegas algorithms so that maybe it can increase the stability of QUIC especially with Vegas, there are so many ways in which we can increase the potential of this algorithm due to the easy implementation and control over RTT values. Due to some values such as *alpha*,*beta* and *gamma*, which we use for QUIC are the optimal values for TCP, to do the evaluation and measurements, so that maybe they are not the best values for QUIC. To find these optimal values, we need better insight and time.

To achieve better power of QUIC, we need to adjust the parameters not only in the programming code but also in the algorithms itself, there is some variation of Vegas, such as Vegas-A and Vegas-L. Besides, we need more testing scenarios which can make the observations of the congestion control more precise and easier. We should believe that the QUIC protocol will not stop here when more and more research will be carried out in the future to maximal the potential of QUIC. We hope that this work will be the small step to understand better the behavior of QUIC and to support future Congestion Control implementations.

BIBLIOGRAPHY

[1] J. Iyengar and M. Thomson. "Quic: A UDP-Based Multiplexed and Secure Transport." In: *Quic: A UDP-Based Multiplexed and Secure Transport* Internet-draft (Sept. 2019). URL: https://www.chromium.org/quic (cit. on pp. 1, 3, 4, 10).

[2] L. Adam, R. Alistair, W. Alyssa, V. Antonio, K. Charles, Z. Dan, Y. Fan, K. Fedor, S. Ian, I. Janardhan, B. Jeff, D. Jeremy, R. Jim, K. Joanna, W. Patrik, T. Raman, S. Robbie, H. Ryan, V. Victor, C. Wan-Teh, and S. Zhongyi. "The QUIC Transport Protocol: Design and Internet-Scale Deployment." In: SIGCOMM '17 (2017), pp. 183–196. DOI: 10.1145/3098822.3098842. URL: http://doi.acm.org/10.1145/3098822.3098842 (cit. on p. 1).

[3] F. Gratzer. "QUIC - Quick UDP Internet Connections." In: *Seminar Innovative Internet-Technologien und Mobilkommunikation SS2016* (2016). URL: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2016-09-1/NET-2016-09-1_06.pdf (cit. on pp. 1, 21, 37).

[4] B. Jaeger M. Yosofie. "Recent Progress on the Quic Protocol." In: *Seminar IITM WS 18/19, Network Architectures and Services* (May 2019), pp. 77–81. DOI: 10.2313/NET-2019-06-1_16. URL: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2019-06-1/NET-2019-06-1_16.pdf (cit. on pp. 1, 24).

[5] J. Iyengar and I. Swett. "Quic Loss Detection and Congestion Control." In: *QUIC Loss Detection and Congestion Control* 5/27/2019 (May 2019), pp. 7–9,24–28,31–46. URL: https://quicwg.org/base-drafts/draft-ietf-quic-recovery.html (cit. on pp. 1, 3–5, 9–14, 53).

[6] Turkovic B, Fernando A. Kuipers, and S. Uhlig. "Fifty Shades of Congestion Control: A Performance and Interactions Evaluation." In: *CoRR* abs/1903.03852 (2019). arXiv: 1903.03852. URL: http://arxiv.org/abs/1903.03852 (cit. on pp. 1, 5, 14, 17, 18, 22).

[7] Quicwg. "Quic implementation database." In: *Implementations/quicwg - basedraft(Github.com)* 1 (2018), p. 1. URL: https://github.com/quicwg/base-drafts/wiki/Implementations (cit. on p. 2).

[8] K. Arash Molavi, J. Samuel, C. David, N. Cristina, and M. Alan. "Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols." In: IMC '17 (2017), pp. 290–303. DOI: 10.1145/3131365.3131368. URL: http://doi.acm.org/10.1145/3131365.3131368 (cit. on pp. 3, 36).

[9] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. "TCP Vegas: New Techniques for Congestion Detection and Avoidance." In: *SIGCOMM Comput. Commun. Rev.* 24.4 (Oct. 1994), pp. 24–35. ISSN: 0146-4833. DOI: 10.1145/190809.190317. URL: http://doi.acm.org/10.1145/190809.190317 (cit. on pp. 5, 19).

[10] U. Hengartner and J. Bolliger. "TCP Vegas revisited." In: *Proceedings - IEEE INFOCOM* 3 (Dec. 1999), pp. 1–6. DOI: 10.1109/INFCOM.2000.832553. URL: http://www.cs.cmu.edu/~uhengart/infocom00.pdf (cit. on pp. 5, 19).

[11] C. Burkert E. Sy. "A Quic Look at Web Tracking." In: *Proceedings on Privacy Enhancing Technologies 2019* (Jan. 2019), pp. 1–12. URL: https://svs.informatik.uni-hamburg.de/publications/2019/2019-02-26-Sy-PET_Symposium-A_QUIC_Look_at_Web_Tracking.pdf (cit. on p. 6).

[12] I. Poese J. Ruth. "A First Look at QUIC in the Wild." In: *Passive Active Measurements Conference (PAM), 2018* abs/1801.05168 (Jan. 2018). URL: https://datatracker.ietf.org/meeting/101/materials/slides-101-maprg-a-first-look-at-quic-in-the-wild-00 (cit. on p. 6).

[13] J. Postel. "Transmission Control Protocol DARPA Internet Program Protocol Specification." In: *RFC 793* 1.1 (Sept. 1981), pp. 1–7. URL: https://tools.ietf.org/html/rfc793 (cit. on p. 9).

[14] Chromium Projects. "Quic at 10000 feet." In: *Quic Wg Documentation (Last Access: 10/2019)* (Jan. 2012), p. 1. URL: https://docs.google.com/document/d/ (cit. on p. 10).

[15] M. Geist and B. Jaeger. "Overview of TCP Congestion Control Algorithms." In: Seminar IITM WS 18/19 (May 2019). URL: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2019-06-1/NET-2019-06-1_03.pdf (cit. on p. 13).

[16] H. Sangtae, R. Injong, and X. Lisong. "CUBIC: A New TCP-friendly High-speed TCP Variant." In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 1–11. ISSN: 0163-5980. DOI: 10.1145/1400097.1400105. URL: http://doi.acm.org/10.1145/1400097.1400105 (cit. on pp. 14–16, 27).

[17] K. Kurata, G. Hasegawa, and M. Murata. "Fairness Comparisons between TCP Reno and TCP Vegas for Future Deployment of TCP Vegas." In: *web.archieve.org, Internet Archive Wayback Machine (Last Access: 10/2019)* (Jan. 2016), pp. 1–20. URL: https://www.internetsociety.org/inet2000/cdproceedings/2d/2d_2.htm (cit. on pp. 18, 20, 24).

[18] L. Li, J. Zhu, and N. He. "TCP Vegas-L An Adaptive End-to-End Congestion Control Algorithm over Satellite Communications." In: *INNOV 2017 : The Sixth International Conference on Communications, Computation, Networks and Technologies* 978-1-61208-596-8 (2017), pp. 1–5. URL: https://www.thinkmind.org/index.php?view=article&articleid=innov_2017_1_10_70012 (cit. on p. 18).

[19] A. Omar and A. Eitan. "Analysis of TCP Vegas and TCP Reno." In: *Telecommunication systems, IEEE International Conference on Communications* 15 (Dec. 2000), pp. 381–404. URL: https://www-sop.inria.fr/members/Eitan.Altman/PAPERS/telecom-sys.ps (cit. on pp. 20, 22, 27, 28).

[20] L. Richard, W. Jean, and A. Venkat. "Issues in TCP Vegas." In: *Issues in TCP Vegas* (Jan. 2004). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.8588&rep=rep1&type=pdf (cit. on pp. 21, 38).

[21] S. Floyd. "Metrics for the evaluation of congestion control mechanisms." In: *RFC 5166* (Jan. 2008), pp. 3–7. URL: https://www.icir.org/floyd/papers/metrics.ps (cit. on pp. 21, 22, 28).

[22] Tetcos Engineering. "Comparison of TCP congestion control algorithms." In: *Comparison of TCP congestion control algorithms (Last Access: 9/2019)* Internet-draft.1 (Jan. 2012), p. 1. URL: https://www.tetcos.com/downloads/TCP_Congestion_Control_Comparison.pdf (cit. on pp. 22, 28).

[23] D.Bansal. "Dynamic Behavior of Slowly-responsive Congestion Control Algorithms." In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pp. 263–274. ISSN: 0146-4833. DOI: 10.1145/964723.383080. URL: http://doi.acm.org/10.1145/964723.383080 (cit. on p. 22).

[24] Utah University. "Emulab total network testbed." In: *Emulab Documentations(Last Access: 10/2019)* (2019). URL: https://wiki.emulab.net/Emulab/wiki (cit. on p. 23).

[25] Chromium authors Yolan. "Chromium projects." In: *Google Git, Git Repositories on Chromium (Last Access: 10/2019)* (2012), p. 1. URL: https://chromium.googlesource.com/chromium/src/net (cit. on p. 26).

[26] S. Arianfar. "TCP's Congestion Control Implementation in Linux Kernel." In: *Network Protocols in Operating Systems* (2012), pp. 3–5. URL: https://wiki.aalto.fi/download/attachments/69901948/TCP-CongestionControlFinal.pdf (cit. on p. 27).

[27] N. Cardwell. "A TCP Vegas Implementation for Linux." In: *niel.nu/linux-vegas (Last Access:9/2019)* (2006). URL: http://neal.nu/uw/linux-vegas/ (cit. on p. 27).

[28] Dr. S. Agarwal A. Vikram Singh A. Chauhan. "Analysis of TCP Vegas in Linux 2.6.1." In: *International Journal Of Engineering And Science* 2.8 (Mar. 2013), pp. 33–37. URL: http://www.researchinventy.com/papers/v2i8/G028033037.pdf (cit. on p. 40).

[29] David X.Wei. "Known problems in TCP algorithms in Linux 2.6.16.3." In: *Known problems in TCP algorithms in Linux 2.6.16.3 (Last Access: 10/2019)* (June 2006). URL: http://netlab.caltech.edu/projects/ns2tcplinux/ns2linux/known_linux/resolved.html (cit. on p. 42).

A

APPENDIX

A.1 EMULAB CODE

In this section, we show the code to create experiment topology in Emulab

```
set ns [new Simulator]
source tb_compat.tcl

#set 4 nodes
set server [$ns node]
tb-set-node-os $server UBUNTU16-64-STDG
set node [$ns node]
tb-set-node-os $node UBUNTU16-64-STDG
set client1 [$ns node]
tb-set-node-os $client1 UBUNTU16-64-STDG
set client2 [$ns node]
tb-set-node-os $client2 UBUNTU16-64-STDG

#set link between Server to Node, Node to Client1 and Client2

set link0 [$ns duplex-link $server $node 5Mb 10ms DropTail]
set link1 [$ns duplex-link $node $client1 1000Mb 0ms DropTail]
set link2 [$ns duplex-link $node $client2 1000Mb 0ms DropTail]

$ns rtproto Static

$ns run
```

A.2 RUNNING CODE

In this section, we show the syntax to run the experiments.

First we need to ssh to Client1 node using Terminal, go into traffiic-gen folder and run bash to use command for Golang.

There are some explanations for the code:

- -mode : switch between client and server.
- -p the protocol : switch between TCP or Quic.
- -v : this one is used for debugging process, so the output of the packets.

- -a : the address of the server, in this experiment the server address in Emulab is 10.1.2.2
- -cc : change the values of this to access to Vegas or Cubic.
- -t : time for 120 seconds we need value 120000.

The Client1 and Client2 should take difference ports to transmit the data. In our case, Client1 used port 4000 and Client2 used port 4500.

After that, the the syntax has to run automatically for the measurements. So that we set a script to do the process:

- ssh to the nodes
- prepare the servers
- prepare the clients
- run the process
- and get the data back to the client computer

```
Client1:
$ go run traffic-gen.go -mode client -p quic -a server_emulab_ip:
    port_number1 -v -cc (cubic or vegas) -t 120000
```

and Client2:

```
Client2:
$ go run traffic-gen.go -mode client -p quic -a server_emulab_ip:
    port_number2 -v -cc (cubic or vegas) -t 120000

Server1:
$ go run traffic-gen.go -mode server -p quic -a server_emulab_ip:
    port_number1 -v

Server2:
$ go run traffic-gen.go -mode server -p quic -a server_emulab_ip:
    port_number2 -v
```

In this section, we show the hardware and software which are needed to do this experiment.

A.3 HARDWARE REQUIREMENT

A.4 SOFTWARE REQUIREMENTS

ITEM	VERSION	DESCRIPTION
Macbook	early2015	Processor 2.7GHz Intel Core i5, Ram 8Gb
Emulab	unknown	network emulation testbed
IKT lab PC	unknown	IKT lab computer

Table A.1: Hardware requirements

ITEM	VERSION	DESCRIPTION
MacOS	Mojave	not the best OS to do this
Ubuntu		IKT PC OS
Golang	1.12	needed for Golib
quic-go		Github code lib for implementation of Quic
traffic-gen		Github code lib for traffic generation
Wireshark		Get raw data
Iperf		TCP modules checking
Linuxptp		Clock sync
Matlab	R2017b	Process the data, plotting
LibreOff	6.30	Change raw data into table, .csv files
Latex	4.01	TextShop.Editing report
VScode	1.39	Main Coding enviroment

Table A.2: Software requirements