An Evaluation of Multipath TCP with Highly Asymmetric Subflows

J.R.M.D.B. Karunarathne[a], Tharindu Wijethilake[a], C. Keppitiyagama[a]

[a]*University of Colombo School of Computing, 35 Reid Ave, Colombo 00700, Sri Lanka*

***Abstract*— Multipath TCP (MPTCP) is an extension of the Transmission Control Protocol (TCP) that allows the simultaneous use of multiple available network interfaces to transmit and receive data. MPTCP can improve the throughput, lower the latency, and provide higher resilience to network failures. MPTCP creates a number of network connections (subflows) between the destination and presents a single endpoint to the application. MPTCP schedulers multiplexe data over subflows based on their end-to-end path metrics. In this study, we found that the presence of asymmetric links within an MPTCP connection can lead to suboptimal performance. We explored the architecture of the Linux implementation of MPTCP and identified the design choices that lead MPTCP to underperform in the presence of highly asymmetric links. To test the behaviour of MPTCP an emulation testbed was built using the Mininet emulator. We conducted comprehensive experiments in this controlled environment to analyze MPTCP behavior under asymmetric subflows in terms of bandwidth and latency. We designed a novel scheduling algorithm tailored to mitigate the impact of asymmetric subflows and implemented it in the Linux kernel. Building a scheduling algorithm for MPTCP in the Linux kernel is not a straightforward task. Several iterations of the algorithm had to be investigated in order to develop a practically deployable algorithm. The proposed algorithms were implemented in the Linux Kernel and were tested in the testbed. These algorithms were tested for their suitability to be used over highly asymmetric links under several test scenarios. Finally, we proposed the "Extended Dynamic Scheduler Algorithm" which observes the MPTCP connection and adjusts its subflows to limit the effect of asymmetric subflows in the MPTCP connection. The algorithm also has its own kickback policy where the throughput of the connection starts to improve when the asymmetry of the subflows decreases.**

***Keywords*— Multipath TCP, Asymmetric Subflows**

I. Introduction

The Internet has become one of the basic necessities in the 21st century. Most modern smart devices including the refrigerator in the kitchen should connect to the internet to work at its full capacity. There is an estimated amount of 43 billion devices connected to the internet as of 2023[1]. Most devices connecting to the internet like smartphones and computers have more than one network interface which is capable of connecting to the internet (Multihomed devices).

Correspondence: J.R.M.D.B. Karunarathne (e-mail: dinendradb@gmail.com)
Received: 04-03-2024 Revised: 10-03-2024 Accepted: 20-03-2024

J.R.M.D.B. Karunarathne, Tharindu Wijethilake amd C. Keppitiyagama are from University of Colombo School of Computing, Sri Lanka (dinendradb@gmail.com, tnb@ucsc.cmb.ac.lk, cik@ucsc.cmb.ac.lk)

DOI: https://doi.org/10.4038/icter.v17i1.7278

For example, a modern smartphone has the hardware capability of connecting to the internet via WiFi or a mobile network. But the interesting fact is 91.5% of the monitored links on the internet are working on the Transmission Control Protocol (TCP) which was introduced in the 1970s[2]. But TCP is not capable of handling multiple network interfaces simultaneously. Due to this limitation of TCP, readily available hardware resources are not used to their maximum capability in multihomed devices. This issue is called the Multi-homed problem and it has become an interesting avenue for researchers to explore.

There are a number of mechanisms proposed as solutions for the multihomed problem at the application layer, network layer and transport layer. To solve the multihomed problem in the Application Layer, all the existing applications should be improved to handle multiple network interfaces simultaneously. This solution was deemed impractical with the growing number of applications and different kinds of devices with a different number of network interfaces. Several network-level solutions were proposed like shim6[3] and HIP (Host Identity Protocol)[4]. These solutions remain experimental and have not been deployed in a commercial environment. Stream Control Transmission (SCTP)[5] is a promising transport layer solution that allows hosts to use multiple paths at the same time. SCTP is implemented in several operating systems. SCTP is not widely used due to two major reasons. The applications working on SCTP needed to be modified to work with SCTP and the middleboxes such as the firewalls which do not understand the SCTP protocol started to drop the SCTP packets.

Out of all these available solutions, Multipath Transmission Control Protocol (MPTCP) has been a prominent Transport Layer solution. MPTCP is an extension to the current TCP, which was standardized by the Internet Engineering Task Force (IETF) and is presented in RFC 8684[6]. MPTCP enables the simultaneous use of several network interfaces to transmit and receive data. Some organizations such as Apple have taken the initiative to use MPTCP in their voice assistance application, Siri[7] and some Samsung Mobile devices[8]. MPTCP is included in some latest versions of Linux operating systems and is also deployed in several systems such as FreeBSD, several MacOS versions, and in the Google Cloud Engine, Amazon AWS, Raspberry Pi and Android[9]. However, in this paper, we will solely focus our study on the Linux implementation of MPTCP.

An MPTCP connection could consist of one or more than one network links by using the network interfaces available in the devices. These individual links are known as subflows. As an example, consider a connection with two subflows, one subflow can be a Wifi link and the other, a wired link. Both these links may have different network characteristics such as bandwidth, throughput, latency, bandwidth-delay product and

jitter[10]. The performance of a network may depend on the mentioned network characteristics.

Asymmetry of subflows will occur when one link outperforms another drastically in terms of the network characteristics mentioned above. In this paper, we will evaluate the behaviour of an MPTCP connection in the presence of highly asymmetric links in terms of bandwidth and latency in the Linux implementation of MPTCP.

The paper is organized as follows. We will discuss the MPTCP protocol and the Architecture of MPTCP in the Linux Kernel along with the schedulers. Next, we discuss some design choices of MPTCP in the Linux Kernel which may lead to MPTCP underperforming the presence of highly asymmetric links. In the following sections, we discuss the test bed that we have developed and then we present the results of our experimental evaluation in different scenarios to show the effect of the design decisions discussed. Further, we discuss how we have developed the ”Extended Dynamic Scheduling Algorithm” and present the evaluation of the implementation of the algorithm. Then we have presented our Conclusions and Future Work in the final section.

II. MPTCP

The use of multiple network interfaces in MPTCP improves throughput and resilience to network failure over traditional TCP[11]. Though it creates a number of subflows using the available network interfaces, it appears as a single network connection to the application. However, each subflow is treated as a separate TCP connection at the transport layer, as shown in Figure 1. This enables MPTCP to mitigate the problems that occur at the middleboxes such as routers[11]. They will observe the connection as a normal TCP connection, therefore, the packets will not be dropped. Applications need not be modified because an MPTCP connection appears as a single network connection to the application. These design choices have made MPTCP popular over the other solutions to the Multihomed problem. Since each subflow acts as a separate TCP connection, an MPTCP connec tion is established similarly to a TCP connection using a 3-way handshake. But MPTCP has its own set of Option subtypes such as MP CAPABLE, MP ‐ JOIN, ADD ADDR, MP FAIL, and DSS used in connection and subflow management.

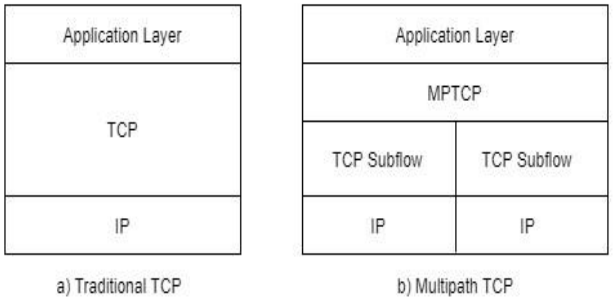


Figure 1: Traditional TCP and MPTCP Stack

Each SYN, SYNACK and ACK packet used in the handshake will contain the MP CAPABLE option and the sender’s and the receiver’s key to be used in the subflow creation as shown in Figure 2.

If any packet received did not contain the MP CAPABLE option it means that either a host or a middlebox is not compatible with MPTCP and the connection will fall back to regular TCP.

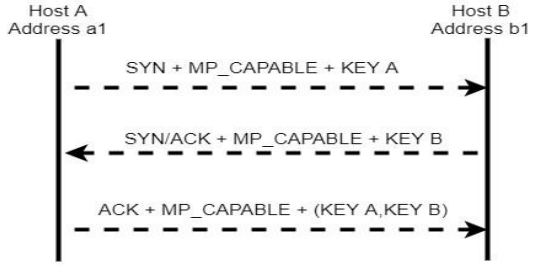


Figure 2: Traditional TCP and MPTCP Stack

After an MPTCP connection is initiated, hosts can create additional subflows by advertising its other network interface addresses with the ADD ADDR option. Similar to the connection initiation, subflow initiation is again done by sharing the connection initiation messages SYN, SYNACK and ACK but this time with the MP JOIN option. Hosts also exchange a nonce and a Hashed Message Authentication (HMAC) code to authenticate the new subflow.

III. MPTCP ARCHITECTURE IN THE LINUX KERNEL

Since this paper focuses on the Linux implementation of MPTCP, we will survey the MPTCP architecture in the Linux Kernel. The current implementation of MPTCP in the Linux Kernel is based on the works of Barre et al.[9],[11]. In this implementation, the architecture consists of three main parts: the master subsocket, the Multipath Control Block (mpcb) and the slave subsocket as depicted in the Figure 3.

The multipath control block (mpcb) supervises the subflows in a connection. It is responsible for running the three decisive algorithms for MPTCP. They are the decision algorithm, the scheduling algorithm and the reordering algorithm. The decision algorithm starts and stops subflows. The scheduling algorithm feeds data from the application layer to a subflow and the reordering algorithm will reorder the incoming data segments before sending them to the application layer.

The master subsocket is the interface between the application and the kernel which is used for TCP communication. If an MPTCP connection cannot be established, only the master subsocket will be used in communication.

The slave subsockets are opened, closed and managed by the mpcb. When the slave subsockets open, it will create a pool of paths with the master subsocket. mpcb will schedule outgoing data segments and receive incoming data segments in the created pool of subsockets. The pool of subsockets is not visible to the application layer.

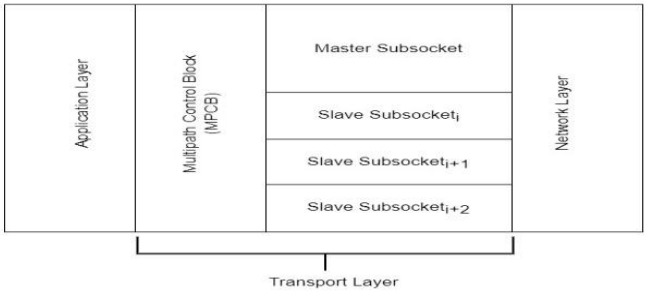


Figure 3: Overview of the main components of the implementation of Multipath TCP in the Linux Kernel

When multiple subflows are created, MPTCP must decide on which subflow it should send the data to. Data packets in the sender queue will be scheduled to the available subflows by the scheduler as shown in the Figure 4. This decision is made by the MPTCP scheduler. The scheduler directly affects the performance of the MPTCP connection as it is responsible for deciding how to distribute the data over the multiple paths available in the connection. A wrong scheduling decision or an irrelevant scheduling algorithm will affect the performance of an MPTCP connection[12].

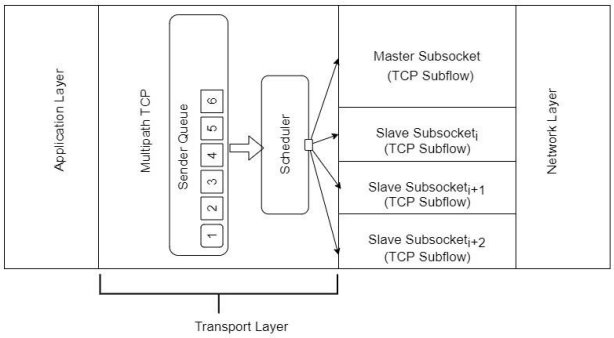


Figure 4: Overview of the main components of the implementation of Multipath TCP in the Linux Kernel

In the current MPTCP implementation, there are three schedulers[11]. They are minRTT scheduler, the round-robin scheduler and the redundant scheduler. The minRTT which is the default scheduler will send packets on the subflow which has the lowest RTT (Round Trip Time). The round-robin scheduler schedules packets on each subflow in a round-robin manner. The redundant scheduler will send the data packets on all subflows. This is done in extreme cases of packet loss to provide better robustness.

IV. Performance of MPTCP with Highly Asymmetric Links

The performance of a network is the measure of the quality of service of the network as seen by the users of the network. It can be measured using several characteristics of the network such as bandwidth, throughput, latency, jitter, packet loss rate and bandwidth-delay product. Depending on one or more characteristics mentioned above, one network connection can have a better performance than another network connection. From this point onwards when a network connection is referred to as a "*good path*" relative to another path, it means that the "*good path*" will have better performance than the other path. The path with the inferior performance will be addressed as the "*bad path*."

There are three main aims of MPTCP[13]. The first goal of MPTCP is to use MPTCP without modifying the application. The second goal is for MPTCP to work in all scenarios where TCP currently works. In this paper, we will focus on the third goal which states that MPTCP should perform at least as well as regular TCP.

There are several design decisions in the Linux implementation of MPTCP which influence an MPTCP connection with high asymmetric subflows to not have the expected performance. In this section, we will discuss these design decisions.

A. Ack Clocked Situation.

When transferring a data segment over an MPTCP connection, once the application fills the congestion window of each subflossw the scheduling algorithm changes. For an instance, let's assume that the minRTT scheduler is in practice and the scheduler prioritizes the subflow with the minimum RTT to send data. Once the congestion window of each path is filled, the scheduler will not prioritize the path with the minimum RTT. As soon as an acknowledgement is received in any individual subflow, the next packet will be scheduled to that subflow irrespective of the RTT. This effect is known as ack-clocked[12].

All the MPTCP schedulers implemented in the Linux Kernel become ack-clocked once each subflow's congestion window is filled in an MPTCP connection[12]. The MPTCP scheduler may schedule packets to all the paths in the MPTCP connection including the *bad path*. Therefore, it is evident that the presence of a *bad path* may have an effect on the MPTCP connection as there is a chance that packets could be sent via the *bad path* as well.

B. Granualtity of Allocations.

The MPTCP scheduler must decide the number of contiguous bytes which will be sent in a single subflow before switching to another subflow. This is the granularity of allocations[12]. For the optimal use of all subflows, the granularity must be of small allocation units. But to send smaller allocation units it would require more scheduler calls, more memory access and more CPU usage. Barre et al.[11] in their implementation of MPTCP in the Linux kernel state that this design of MPTCP would influence the performance of MPTCP in the presence of high bandwidth flows.

C. Maximum Segment Size (MSS).

The maximum segment size is a TCP option that defines the largest amount of data measured in bytes that will travel across a communication channel. With the use of multiple paths in MPTCP, the scheduler must decide on an MSS for each path. But deciding the MSS each time a packet is scheduled to a path is computationally inefficient. Therefore, in the current implementation of MPTCP in the Linux Kernel, the lowest MSS of all paths is used as the MSS of the MPTCP connection. This can affect the performance of MPTCP when using highly asymmetric bandwidth links[11].

D. Receiver Buffer/Queue Constraints.

Data segments at the receiving end are collected at a receiver buffer. Since an MPTCP connection will have several TCP subflows, each subflow will have an individual receiver buffer of its own. But in the Linux implementation of MPTCP, a connection level receiver buffer is maintained which is a connection-level buffer (Reciever Queue)[11] as shown in Figure 5. As soon as a packet is received at an individual subflow buffer, the said packet is pushed to the connection level receiver buffer and the individual subflows are emptied. These phenomena will create the following problems.

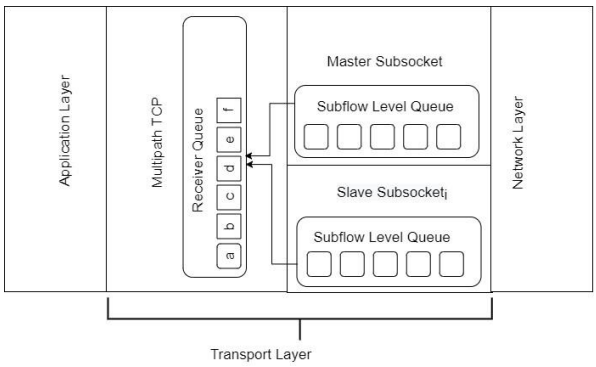


Figure 5: The receiver buffer/queue with the subflow level queues of Multipath TCP in the Linux Kernel.

has two paths, a faster path and a slower path. The faster keeps on sending data packets and they will be held in the receiver queue waiting for the data packets which are on the way in the slower path. The faster path can be blocked due to the receiver buffer being full since MPTCP has to reorder and process the data packets which are being received from the slower path. This phenomenon is called Head-of-Line Blocking[12].

1) Receive-Window Limitations

In a traditional TCP connection, the receiver queue becomes full only if the application stops consuming data. But the receiver queue of an MPTCP connection can be filled due to Head-of-Line Blocking. Due to the Headof-Line blocking problem, it is evident that a significant amount of memory has to be allocated to the receiver buffer to maintain the performance of an MPTCP connection. Paasch et al.[12] in their work have identified the optimum amount of the receiver buffer size as follows,

$$\text{Buffer} = \sum_{i=1}^{n} bw_i * RTT_{max} * 2$$

Where *bw* is the bandwidth of each subflow and RTT_{max} is the highest round-trip time among all the subflows. In a practical scenario, most hosts will not have the capability to allocate this amount of memory. Therefore, out-of-order packets will be dropped and will have to be retransmitted again. Baidya et al.[14] in their work have experimented with an MPTCP connection with two subflows each with a bandwidth of 100Mpbs where an interference was introduced to one subflow. They found that In order for the MPTCP connection to match the performance of a single-path TCP connection with a bandwidth of 100Mpbs, a receiver buffer size of 18MB was required.

Therefore, it is evident that the Linux Implementation of MPTCP was not designed in a way to handle the presence of highly asymmetric paths in the MPTCP connection. In the following sections, we will experimentally evaluate how an MPTCP connection with highly asymmetric links performs.

V. Evaluation

E. Experimental Setup

In this paper, we focus on the behaviour of Linux implementation of MPTCP in highly asymmetric links in terms of bandwidth and latency. We have chosen Mininet[15] to emulate the scenarios. Mininet uses the kernel of the computer that we use and it creates a virtual environment using the kernel's network stack and virtualization technology[16]. This will enable us to realistically evaluate the behaviour of MPTCP in the Linux Kernel in our chosen scenarios. We have currently built the below testbed on Mininet shown in Figure 6.

The testbed we have built is essentially three Linux virtual machines, all running the MPTCP kernel 4.19. "Host-01" acts as the server and "Host-02" acts as the client of the simple topology we have created. "Router-01" is the machine which acts as the router which connects the client and server.

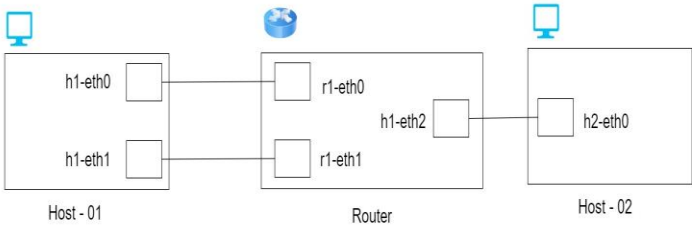


Figure 6: Test Bed developed using Mininet.

The testbed we have built is essentially three Linux virtual machines, all running the MPTCP kernel 4.19. "Host-01" acts as the server and "Host-02" acts as the client of the simple topology we have created. "Router-01" is the machine which acts as the router which connects the client and server.

We ran this testbed on a computer with an Intel® Core™ i7 Processor, with 8GB RAM on an HDD hard drive running the operating system Ubuntu 20.0.4 with the MPTCP Kernel 4.19.

F. Experimental Results and Evaluation.

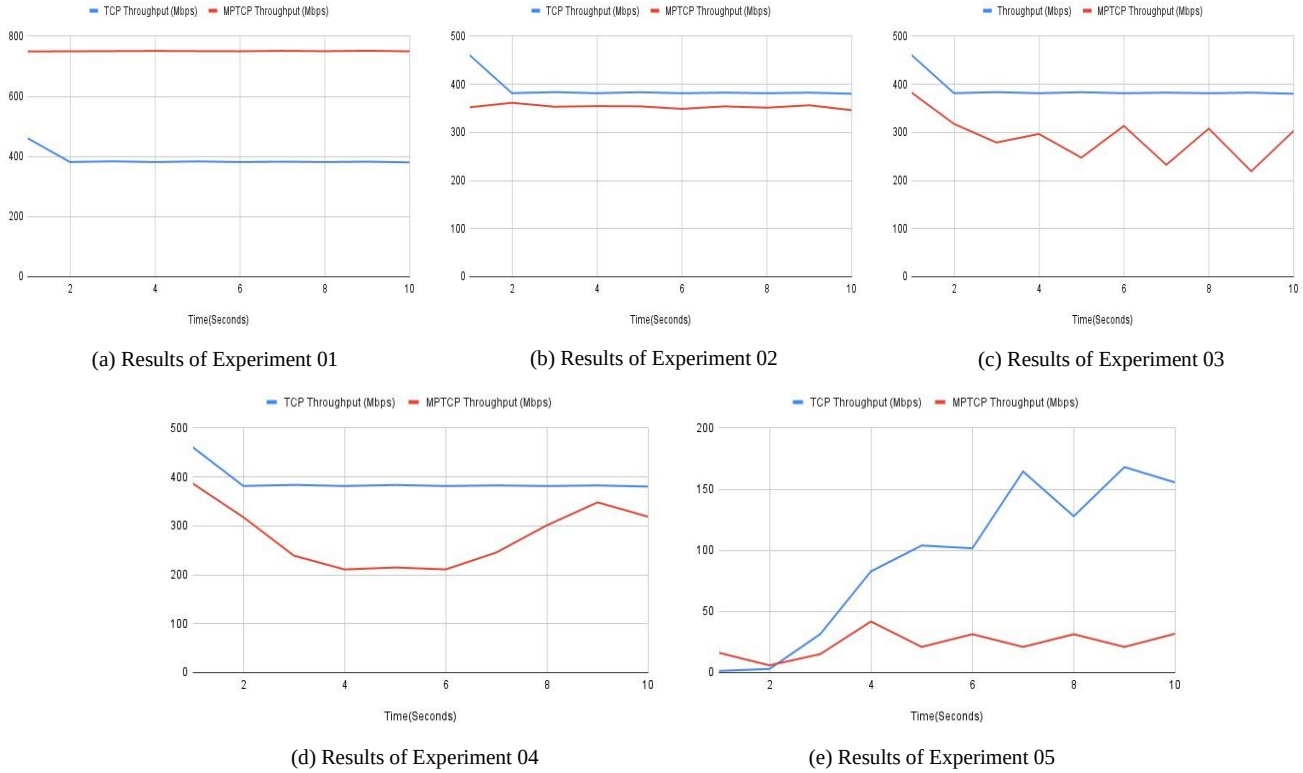
Using the test bed discussed in Section E, we have done the following experiments with the link conditions mentioned in Table 1 to evaluate the behaviour of MPTCP in highly asymmetric links. For this paper, we have chosen the bandwidth and latency of a subflow as the independent variable and throughput as the dependent variable while keeping the other network characteristics as control variables.

Scenario Number	**Single Link TCP Connection {Bandwidth, Latency}**	**Multipath TCP Connection {Bandwidth, Latency }**	
		Link 01 good path	***Link 02 bad path***
01	400Mpbs, 2ms	400Mpbs, 2ms	400Mpbs, 2ms
02	400Mpbs, 2ms	400Mpbs, 2ms	4Mpbs, 2ms
03	400Mpbs, 2ms	400Mpbs, 2ms	400Mpbs, 200ms
04	400Mpbs, 2ms	400Mpbs, 2ms	4Mpbs, 200ms
05	400Mpbs, 2ms	400Mpbs, 200ms	4Mpbs, 2ms

TABLE 1: EXPERIMENTAL SCENARIOS.

In our evaluation, we send a data segment of 1GB over the connection and record the first 10 seconds of the transmission throughput. To increase the accuracy of the results we have run the experiments 10 times and measured the throughput by running iperf[17], a network performance measurement tool.

Then we plotted the graphs using the average of the 10 measurements.



(a) Results of Experiment 01

(b) Results of Experiment 02

(c) Results of Experiment 03

(d) Results of Experiment 04

(e) Results of Experiment 05

Figure 7: Experimental Results of Scenarios in Table 1

In theory, MPTCP should perform well under symmetric paths. To demonstrate this scenario we have first conducted a test with an MPTCP connection with similar subflows which have similar bandwidths and similar latencies. For this test, we have used the bandwidth and latency conditions mentioned in Experiment 01 of Table 1. We can observe, in Figure 7a that the aggregation of the two symmetric connections produces a throughput which is nearly double the throughput of the single-path TCP connection. In this scenario, MPTCP has performed well over symmetric paths.

To demonstrate the effect of subflows with highly asymmetric bandwidths, Experiment 02 of Table 1 is conducted. The latencies of the subflows are kept the same and the *bad path* will have one-tenth of the bandwidth of the *good path*. In this scenario, as seen in Figure 7b, we observe that MPTCP is underperforming as its throughput is less than the TCP throughput. In order to identify whether latency has the same effect on MPTCP we then conduct Experiment 03 of Table 1. The bandwidths of the subflows are now kept the same and the latency of the bad path is now changed to one-hundredth of the *good path*. Similar to Experiment 02, it is again noticed that MPTCP underperforms since its throughput is less than the TCP throughput in Figure 7c.

To observe the effect of the subflow with both high latency and low bandwidth we perform Experiment 04 in Table 1. The bandwidth of the *bad path* is one-tenth of the *good path* and the latency of the *bad path* is a hundred times of the *good path*. As seen in Figure 7d, we again notice MPTCP underperforming with the aggregation of relatively bad bandwidths and high latencies in a single subflow.

Asymmetry of subflows can arise in different manners. In all the experiments discussed up to now, asymmetry of the MPTCP connection was due to one path being the *good path* and the other bad being the path with regards to a network characteristic. But an MPTCP connection can have two asymmetric *bad paths* as well. In Experiment 05 of Table 1, we demonstrate a scenario where one path is better than the other path in terms of bandwidth but has very high latency. The other *bad path* has low bandwidth coupled with relatively better latency. Both paths are *bad paths* but asymmetric due to different network conditions. As seen in Figure 7e, the MPTCP underperforms to a much worse degree compared to the other scenarios. This scenario has a much lower throughput compared to the TCP throughput. Here it is noted that the good network characteristics of neither subflows are appreciated in the MPTCP connection.

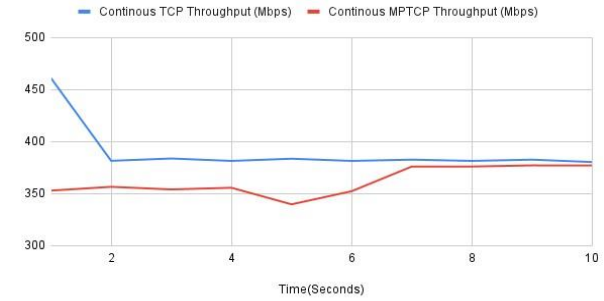
VI. Defining a Scheduler Algorithm for MPTCP

G. Preliminary Experimental Analysis

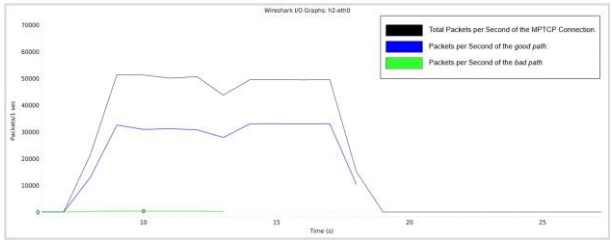
From the results of the experimental scenarios of Table 1, it is observed that MPTCP underperforms when a connection is established with highly asymmetric subflows. As discussed earlier, one of the main goals of MPTCP is to at least perform as well as TCP and we have observed that this goal cannot be accomplished within the tested extreme conditions. It is evident that the presence of the *bad path* in the MPTCP connection has affected the *good path* as well. So the fundamental question has to be asked is "*Will the throughput of an MPTCP connection improve if the bad path is removed?*"

Before implementing a mechanism in the Linux Kernel, we extended our test bed to emulate this scenario. We repeated the experimental scenarios in Table 1 for 10 seconds and removed the *bad path* in the 5^{th} second.

Figures 8, 9, 10, 11 depict the Experiments 2,3,4,5 from Table 1 respectively. These experiments each have been repeated and the average was plotted to increase the accuracy. Except for the scenario in Experiment 5 in Table 1, all the other scenarios have shown a positive response when the *bad link* was removed. The Iperf-Graphs show the increase of the throughput, where it reaches the throughput level of single-path TCP. The WiresharkGraphs depict that when the *bad path* is broken the flow becomes stable and packets-per-second increase in the *good-path*.

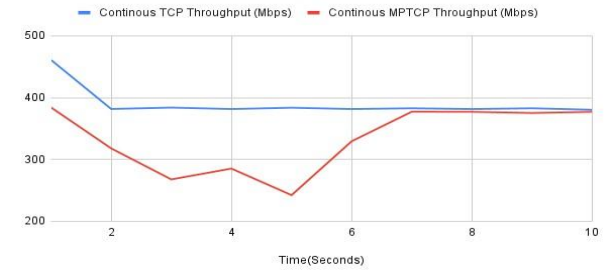


(a) Iperf Graph

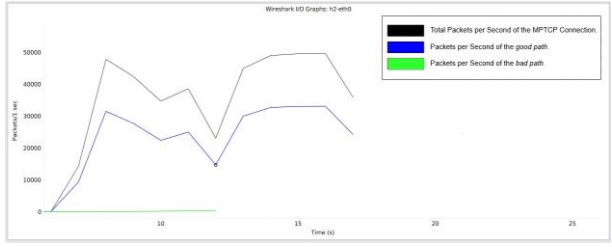


(b) Wireshark Graph

Figure 8: TCP and MPTCP over Highly Asymmetric Bandwidth Links with *bad link* breaking in the 5th second

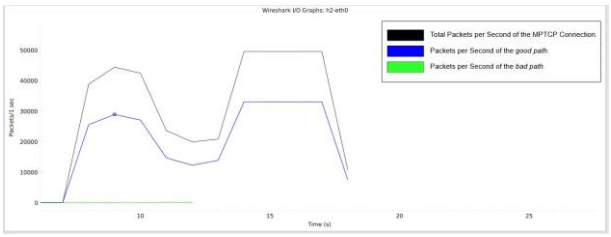


(a) Iperf Graph

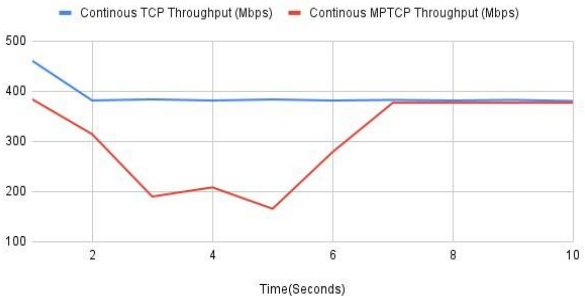


(b) Wireshark Graph

Figure 9: TCP and MPTCP over links with high delay asymmetry with *bad link* breaking in the 5th second

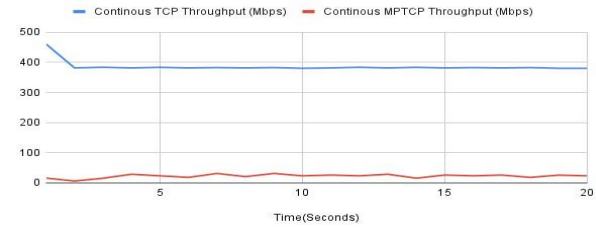


(a) Iperf Graph

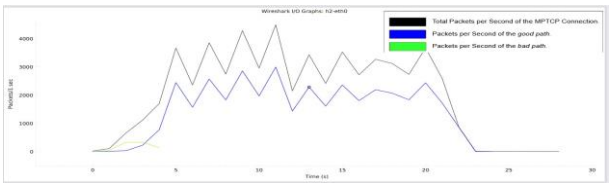


(b) Wireshark Graph

Figure 10: TCP and MPTCP over links with High Bandwidth, Low Delay and Low Bandwidth, High Delay with *bad link* breaking in the 5th second



(a) Iperf Graph



(b) Wireshark Graph

Figure 11: TCP and MPTCP over links with high delay asymmetry with *bad link* breaking in the 5th second

H. Defining the Scheduling Algorithm

The aim of the proposed scheduling algorithm is to minimize the effect on the throughput of an MPTCP connection when it contains highly asymmetric links in terms of bandwidth and latency. But in the context of this research, the aim of the new scheduler should be to minimize or cut off sending packets via the *bad path*. We identified two key elements when that should be defined when defining a new scheduling algorithm.

1. Metric - To identify whether to minimize sending packets via a certainsubflow.
2. Threshold - To identify ”when” to minimize sending packets via a subflow, the metric should

reach a threshold and this threshold should be defined.

The subflow selection algorithm in the current default MPTCP scheduler is depicted in Algorithm 1. The metric used in selecting the suitable subflow is the RTT (Round Trip Time).

Algorithm 1 Subflow selection algorithm of the default MPTCP scheduler

```
for subflow of subflows of MPTCP connection do
    if subflow fails defined checks then
        continue
    end if
    if subflow→rtt < minimum rtt then
        set minimum rtt = subflow→rtt
        best subflow = subflow
    end if
return best subflow
```

The aim of the scheduler design is to minimize the effect on the throughput of the MPTCP connection when the subflows are asymmetric in terms of bandwidth and latency. The RTT of a subflow is the suitable indicator to get a reflection of the latency in the subflow[18].

The impact of the bandwidth of the subflow should also be reflected in the metric. The congestion control mechanism of TCP estimates the available end-to-end bandwidth and uses the estimation to recover from congestion, thus achieving higher throughput[19]. Therefore the size of the Congestion Window of the subflow was selected to account for the influence of bandwidth. It is also important to note that the throughput of a subflow is calculated as follows[20],[21],

$$\text{Throughput of a TCP flow} = \frac{(\text{Congestion window})*(\text{MSS})}{\text{RTT}}$$

Here congestion window is the congestion window of the subflow. MSS and RTT are the maximum segment size and the round trip time of the subflow respectively. Therefore, the metric we define to use in our scheduler(m) is,

$$m = \frac{(\text{Congestion window}) * (\text{MSS})}{\text{RTT}}$$

A threshold is needed for the scheduler to know ”when” to minimize the transmission over a subflow. In the context of this research, the threshold should find a point where the total MPTCP connection does not perform at least as much as the best TCP subflow in the connection.

Implementing this threshold in the Linux implementation of MPTCP is challenging to define due to several reasons. An MPTCP connection does not maintain any aggregate performance metrics. The term aggregate performance here means, the collective performance of all the subflows. Only the performance metrics of the individual subflows are kept on track. As the experiments of the earlier sections revealed, the presence of a *bad path* will also affect the performance of the *good path*. Therefore, it is challenging to define a threshold with reference to the best performance of the *good path* as MPTCP does not store any historical metrics.

Due to the challenging nature of developing the threshold, we examined how other solutions were implemented. Baidya et al.[14] in their work tried to improve the performance of MPTCP with a congestion control algorithm called Slow Path Adaptation depicted in Algorithm 2. Although this algorithm is not a scheduling algorithm, it bares a close resemblance to our design.

Algorithm 2 Slow Path Adaptation Algorithm - Congestion Control

In the Congestion Control for $path_i$,
do the following
if R > Threshold **then**

$$snd\_cwnd_i = \text{min\_cwnd}$$
$$ss_thresh_i = snd_cwnd_i$$

Mark the $path_i$ as a bad path
else
Unmark the $path_i$

In Algorithm 2, ”R” is the metric. Baidya et al.[14] has checked this metric against a ”Threshold”. There is no mention of this ”Threshold” or how it was formed in their work. Therefore we proceeded in developing our own threshold which is defined in Algorithm 3, The Dynamic Threshold. Here the best-recorded throughput of a subflow is recorded. If at any point, the sum of throughputs of all subflows go below the recorded best single path throughput, then the bad can be ignored. Therefore, the threshold here is the best-recorded throughput of a single path which will be calculated dynamically each time a data transfer happens.

Algorithm 3 Algorithm to calculate the Dynamic Threshold for Proposed Scheduler

```
maximum throughput recorded = 0
sum of throughputs = 0

for subflow_i of subflows of MPTCP connection do
    throughput_i = congestion window_i * MSS_i/RTT_i
    if subflow_i→max_throughput < throughput_i then
        Do subflow_i→max throughput =
        throughput_i
    end if
    if maximum_throughput recorded < subflow_i→max
    throughput then
        Do maximum throughput recorded =
        subflow_i→max throughput
    end if
    sum of throughputs+ = throughput_i
end for
if sum of throughputs < maximum throughput recorded then
    Do ignore bad path
end if
```

VII. Implementation and Evaluation

We implemented the proposed Scheduling Algorithm in the Linux implementation of MPTCP by customizing the Linux Kernel source code. In order to evaluate our

implementation we had to extend our experimental setup. Our Test Environment was developed on Mininet does not have the ability to create dynamic link conditions in the middle of transmission. Baidya et al.[14] in their experiments have found an alternate way to create dynamic traffic by introducing a UDP flood to one subflow in the middle of a transmission. We have adopted the same approach to create dynamic traffic as shown in Figure 12.

Figure 12: Test Bed on Mininet with UDP Flooding

I. Evaluation of Proposed Scheduler Implementation on Static TCP Traffic

To demonstrate the effect of static asymmetric subflows, we redid the experiments from Table 1 and compared the results with the default scheduler. Figures 13, 14, 15, 16 depicts the results of Experimental Scenarios 2,3,4,5 of Table 1 respectively. As seen in the graphs MPTCP with the dynamic threshold scheduler has outperformed MPTCP with the dynamic scheduler in all scenarios except Scenario 5 of Table 1. Since Scenario 5 depicts an extreme corner case, it can be concluded that the dynamic threshold scheduler has performed better than the default scheduler of MPTCP for static asymmetric subflows.

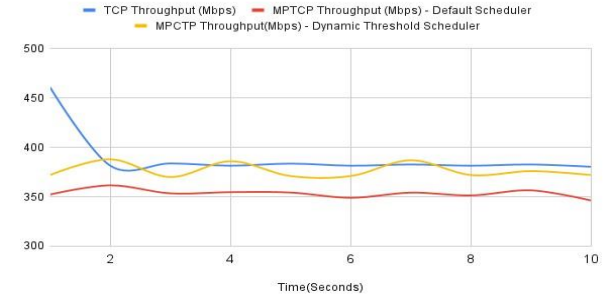


Figure 13: Iperf-Graph: TCP, MPTCP with default scheduler and MPTCP with dynamic threshold scheduler over Highly Asymmetric Bandwidth Links

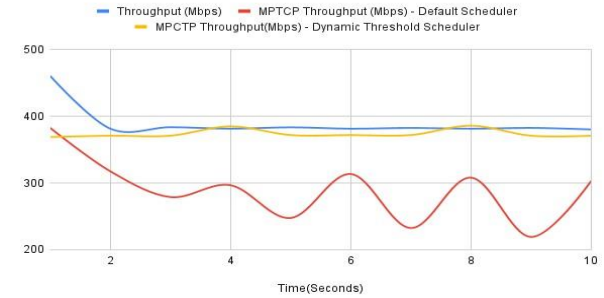


Figure 14: Iperf-Graph: TCP and MPTCP with default scheduler and MPTCP with dynamic threshold scheduler over links with high delay asymmetry

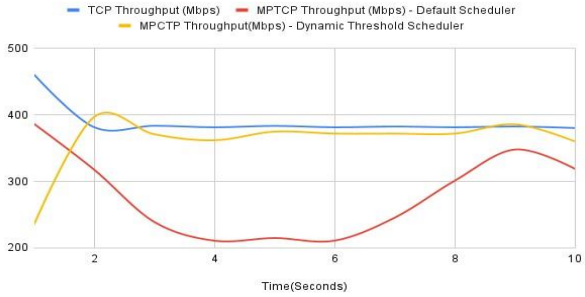


Figure 15: Iperf-Graph: Dynamic threshold scheduler over links with High Bandwidth, Low Delay and Low Bandwidth, High Delay

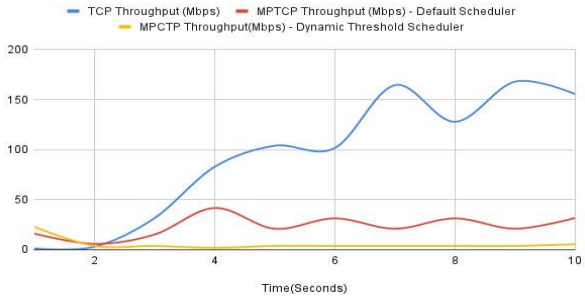


Figure 16: Iperf-Graph: Dynamic threshold scheduler over links with High Bandwidth, High Delay and Low Bandwidth, Low Delay.

J. Evaluation of Proposed Scheduler Implementation on Dynamic TCP traffic

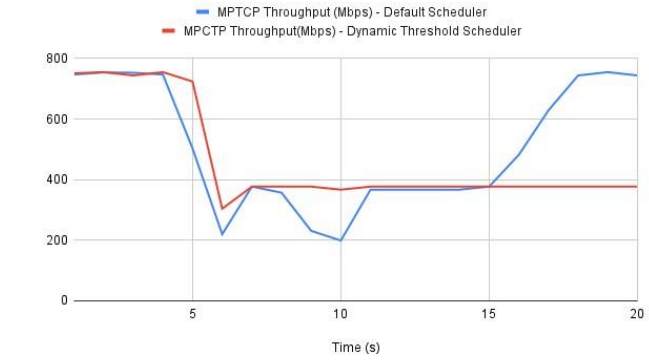


Figure 17: Iperf Graph: Throughput of MPTCP with default scheduler and MPTCP with dynamic threshold scheduler with UDP flooding

Using the setup discussed earlier to evaluate dynamic traffic on mininet, we measure the performance of the dynamic threshold scheduler. In Figure 17, it can be noticed that MPTCP with the dynamic threshold recovers the fastest among all the implementations. The reason for this behaviour is the different design approaches in the scheduling algorithm. The default scheduler is a selection algorithm that selects the best flow to transmit data. The predefined threshold algorithm is a depreciative and selection algorithm that depreciates the *bad path* and selects the *good path* out of the remaining paths. Once the *bad path* is depreciated, it does not have the chance to transmit packets again. But TCP calculates the path characteristics dynamically (while transmitting packets). Therefore, the scheduler will not know

the current situation of the *bad path* and will keep depreciating the path throughout the transmission.

To mitigate this behaviour, we propose periodically sending packets via all subflows despite the link conditions. Then the MPTCP connection can dynamically keep track of the link conditions in each subflow and will be notified once the link conditions improve. Therefore, we extend the dynamic threshold scheduler algorithm by implementing a timer as shown in Algorithm 4 to periodically transmit packets via all the subflows in an MPTCP connection.

Algorithm 4 The Extended Dynamic Threshold Algorithm

for subflow$_i$ *of subflows of MPTCP connection* **do**
throughput$_i$ = *congestion window*$_i * MSS_i/RTT_i$
if *throughput*$_i$ < *dynamic_threshold* **then**
  mark path as *bad path*
end if
if subflow i marked as *bad path* **then**
  if transmit_time == true **then**
    transmit packets over *bad path*
  else ignore subflow **end if**
end if

We conduct the same experiments done earlier to analyze the behaviour of this algorithm in dynamic traffic. Figure 18 marks the throughput of MPTCP with the extended dynamic threshold scheduler with the "△" symbol for clear visibility. We notice that this implementation successfully recovers to the pre-UDP-flood throughput level after the UDP flooding stops.

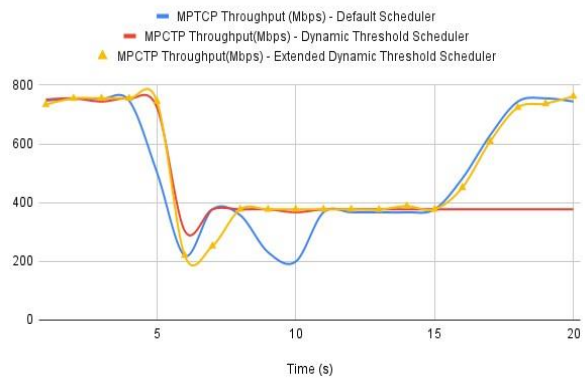


Figure 18: Iperf Graph: Throughput of MPTCP with default scheduler, MPTCP with dynamic threshold scheduler and MPTCP with extended dynamic threshold scheduler with UDP flooding

For further visibility, the Wireshark output of MPTCP with the extended dynamic threshold scheduler is plotted in Figure 19. It is clearly noticed how the subflow affected by the UDP flood recovers after the UDP flow ends.

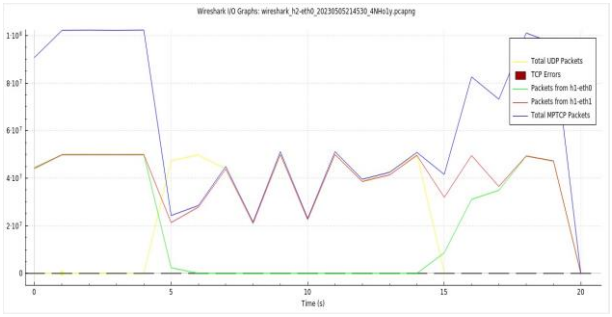


Figure 19: Wireshark Graph: Packets per second of MPTCP with extended dynamic threshold scheduler

It is clear that the reason behind the dynamic threshold scheduling algorithm not recovering to its original potential after the UDP flood ends was the scheduler not getting a chance to rediscover the latest characteristics of the subflow. Therefore, it is evident that periodically transmitting packets of the *bad path* is the solution for the above phenomenon. Hence we have developed a scheduling algorithm capable of handling highly asymmetric subflows in terms of bandwidth and latency in static and dynamic traffic conditions.

VIII. Conclusions and Future Work

Multipath TCP is a promising protocol that will aid in solving the multihomed problem and help utilize unused resources in devices connecting to the internet. However, Multipath TCP may face challenges when deployed on highly heterogeneous links. In this paper, we have discussed the MPTCP protocol and the architecture of MPTCP in the Linux Kernel. We evaluated the implementation of MPTCP to identify design choices that may lead to MPTCP underperforming in the presence of highly asymmetric links. We then evaluated MPTCP in several chosen scenarios and presented the results of how an MPTCP connection with highly asymmetric links in terms of bandwidth and latency has underperformed a single-path link TCP connection.

Based on the results we designed a scheduling algorithm, the Extended Dynamic Threshold Scheduler, which could be deployed in MPTCP environments for steady transmissions despite the asymmetry of subflows. We expect that our experimental results and the evaluation of the design choices of the Linux Kernel which leads to MPTCP underperforming will encourage researchers to present better solutions to improve MPTCP in the Linux Kernel. We have done the evaluation in an emulated environment. In future work, researchers can focus on conducting these experiments on a physical setup. Furthermore, researchers can evaluate the performance of MPTCP with other network characteristics like packet-loss rate and jitter in MPTCP connections with more than two links, since we have restricted our study to MPTCP connections with two links.

References

[1] B. Marr, The top 4 internet of things trends in 2023 (Nov 2022). URL https://www.forbes.com/sites/bernardmarr/2022/11/07/the-top-4-internet-of-things-trends-in-2023/?sh=291c55312aea

[2] L. Schumann, T. V. Doan, T. Shreedhar, R. Mok, V. Bajpai, Impact of evolving protocols and covid-19 on internet traffic shares, arXiv preprint arXiv:2201.00142 (2022).

[3] E. Nordmark, M. Bagnulo, Shim6: Level 3 multihoming shim protocol for ipv6, Tech. rep. (2009).

[4] R. Moskowitz, T. Heer, P. Jokela, T. Henderson, Host identity protocol version 2 (hipv2), Tech. rep. (2015).

[5] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, V. Paxson, Stream control transmission protocol (2007).

[6] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, C. Paasch, TCP Extensions for Multipath Operation with Multiple Addresses, RFC 8684 (Mar. 2020). doi:10.17487/RFC8684.
URL https://www.rfc-editor.org/info/rfc8684

[7] Use multipath tcp to create backup connections for ios (Jan 2022).
URL https://support.apple.com/en-us/HT201373

[8] P. b. O. Bonaventure, Mptcp.
URL : http://blog.multipath-tcp.org/blog/html/2018/12/10/the_first_multipath_tcp_enabled_smartphones.html

[9] Linux kernel multipath tcp project.
URL https://www.multipath-tcp.org/

[10] M. Hassan, R. Jain, High performance TCP/IP networking, Vol. 29, Prentice Hall Upper Saddle River, 2003.

[11] S. Barr´e, C. Paasch, O. Bonaventure, Multipath tcp: from theory to practice, in: NETWORKING 2011: 10th International IFIP TC 6 Networking Conference, Valencia, Spain, May 9-13, 2011, Proceedings, Part I 10, Springer, 2011, pp. 444–457.

[12] C. Paasch, S. Ferlin, O. Alay, O. Bonaventure, Experimental evaluation of multipath tcp schedulers, in: Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop, 2014, pp. 27–32.

[13] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, M. Handley, How hard can it be? designing and implementing a deployable multipath {TCP}, in: 9th {USENIX} symposium on networked systems design and implementation ({NSDI} 12), 2012, pp. 399–412.

[14] S. H. Baidya, R. Prakash, Improving the performance of multipath tcp over heterogeneous paths using slow path adaptation, in: 2014 IEEE international conference on communications (ICC), IEEE, 2014, pp. 3222– 3227.

[15] M. P. Contributors. [link].
URL http://mininet.org/

[16] R. L. S. De Oliveira, C. M. Schweitzer, A. A. Shinoda, L. R. Prete, Using mininet for emulation and prototyping software-defined networks, in: 2014 IEEE Colombian conference on communications and computing (COLCOM), Ieee, 2014, pp. 1–6.

[17] V. GUEANT, Iperf - the ultimate speed test tool for tcp, udp and sctptest the limits of your network + internet neutrality test. URL https://iperf.fr/

[18] K. Obraczka, F. Silva, Network latency metrics for server proximity, in: Globecom'00-IEEE. Global Telecommunications Conference. Conference Record (Cat. No. 00CH37137), Vol. 1, IEEE, 2000, pp. 421–427.

[19] M. Gerla, M. Y. Sanadidi, R. Wang, A. Zanella, C. Casetti, S. Mascolo, Tcp westwood: Congestion window control using bandwidth estimation, in: GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No. 01CH37270), Vol. 3, IEEE, 2001, pp. 1698–1702.

[20] C. Avin, Csci-1680 transport layer iii congestion control strikes back.
URL : https://cs.brown.edu/courses/csci1680/f13/lectures/14-congestion.pdf

[21] Simon s. lam ().
URL : https://www.cs.utexas.edu/users/lam/395t/slides/Congestion\%20Control\%202\%20talks.pdf