# Multipath TCP Prioritized For Ad hoc Networks

by

K. B. Liyanage

2015/CS/078

This dissertation is submitted to the University of Colombo School of Computing

In partial fulfillment of the requirements for the

Degree of Bachelor of Science Honours in Computer Science

University of Colombo School of Computing

35, Reid Avenue, Colombo 07,

Sri Lanka

July 2020

# Multipath TCP Prioritized For Ad hoc Networks

K. B. Liyanage

# Declaration

I, K. B. Liyanage (2015/CS/078) hereby certify that this dissertation entitled "Multipath TCP Prioritized For Ad hoc Networks" is entirely my own work and it has never been submitted nor is currently been submitted for any other degree.

..... .....

Date K. B. Liyanage

I, Dr. C. I. Keppetiyagama, certify that I supervised this dissertation entitled "Multipath TCP Prioritized For Ad hoc Networks" conducted by K. B. Liyanage in partial fulfillment of the requirements for the degree of Bachelor of Science Honours in Computer Science.

..... .....

Date Dr. C. I. Keppetiyagama

I, Mr. P. K. M. Thilakarathne, certify that I supervised this dissertation entitled "Multipath TCP Prioritized For Ad hoc Networks" conducted by K. B. Liyanage in partial fulfillment of the requirements for the degree of Bachelor of Science Honours in Computer Science.

..... .....

Date Mr. P. K. M. Thilakarathne

# Abstract

In 2019, most of the smartphones are equipped with multiple interfaces such as WiFi, Bluetooth and Cellular. However, only one such interface can be used with standard TCP connection. As a solution for this, channel bonding can be used. In a situation where two interfaces connected to two ISPs, channel bonding is not possible. To overcome this problem and to utilize the use of multiple interfaces simultaneously, multipath TCP can be used.

In 2019 the number of mobile users stood at 4.68 billion. With this substantial number of mobile devices, there is an opportunity to create a local ad hoc network between these devices. When considering the utilization of multipath TCP and ad hoc network, there is an opportunity to develop a cost-efficient way to route data through mobile ad hoc network. Ad hoc networking can be used anytime, anywhere with limited or no communication infrastructure. Ad hoc network structure can be used in many real world scenarios as well.

With multipath TCP and relatively cost-efficient paths there is an opportunity to have cost effective routing of data as well as a reliable connection. In this research our goal was to implement a solution to prioritize the subflow that uses the ah hoc network.

In this research we divided the problem into three sub problems. Which are assigning a weight to the subflow to be selected more often, pass a user hint to the kernel level to select the ad hoc network interface and improve the recovery speed by changing the retransmission rate. In this research we managed to provide solutions for all sub problems.

There are other researches that prioritize an interface over the others. One similar research introduced a solution named Delphi, a transport-layer module to choose witch interface to use. One major drawback with this method is not utilizing other interfaces capabilities. We demonstrate that by utilizing multipath TCP this limitation can be bypassed.

# Preface

Basis of this research is to provide a solution to route the data cost effectively and reliably. This research was done by me in conjunction with my supervisor and co-supervisors. I hereby declare implementation of this research and evaluation was all done by me.

# Acknowledgement

In truth, I could not have achieved my current level of success without a strong support from my supervisor Dr.C. I. Keppetiyagama, co supervisors Mr. P. K. M. Thilakarathne and advisor Dr Primal Wijesekera. First of all, my parents, who supported me throughout the research. And I would also like to thank you Mr. T. N. B. Wijethilake, whom has provided patient advice and guidance throughout the research process. Thank you all for your unwavering support.

# Acronym

**MPTCP** Multipath Transmission Control Protocol
**MANET** Mobile Ad Hoc Network
**ISP** Internet Service Provider
**OS** Operating System
**VM** Virtual Machine
**RTO** Retransmission Time Out
**SRTT** Smooth Round Trip Time

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Background to the Research

With the increased number of lightweight devices as well as evolution in wireless communication, the ad hoc networking technology is gaining ground with the increasing number of widespread applications. Ad hoc networking can be used anytime, anywhere with limited or no communication infrastructure.The ad hoc network architecture can be used in real time business applications to increase the productivity and profit[1]. One major concern with this is keeping internet connectivity while connected to an ad hoc network. Standard TCP[2] utilize one interface at a time. As a solution for this, channel bonding[3] can be used. In a situation where two interfaces connected to two ISPs, channel bonding is not possible. To overcome this problem and to utilize the use of multiple interfaces simultaneously, multipath TCP[4] can be used.

In 2019 the number of mobile users stands at 4.68 billion[5]. With this substantial number of mobile devices, there is an opportunity to create a local ad hoc networks between these devices. When considering the utilization of multipath TCP and ad hoc network, there is an opportunity to develop a cost-efficient way to route data through mobile ad hoc network.

We identified three research gaps to address in this research. Three research areas are as follows. How to prioritize multipath TCP for mobile ad hoc network, how to schedule multipath TCP to prioritize the path with ad hoc network, how to provide the user with the ability to choose when to prioritize ad hoc and optimization parameters. In the research, we aim to address above areas.

There are other researches that prioritize an interface over the others. One similar research introduced a solution named Delphi[6], a transport-layer module to choose the interface which smartphone should use. One major drawback with this method is not utilizing other interfaces capabilities. We demonstrate that by utilizing multipath TCP this limitation can be bypassed.

### 1.1.1 Multipath TCP

In the "TCP Extensions for Multipath Operation with Multiple Addresses" request for comments[3], author defined multipath TCP as, set of extensions to regular TCP to provide a Multipath TCP service. Because of this extension transport layer obtain the ability to operate single connection across multiple paths. Because of this ability to create a connection using multiple paths that might use multiple availabel interfaces. MPTCP gain more fault tolerance and high throughput due to use of multiple interfaces. This ability to create a connection using multiple paths is achieved by creating multiple sub-flows to the same connection. These sub-flows holds the characteristics of a standard TCP connection. Therefore there will not be any sequence numbering errors when a middlebox analyse packets.

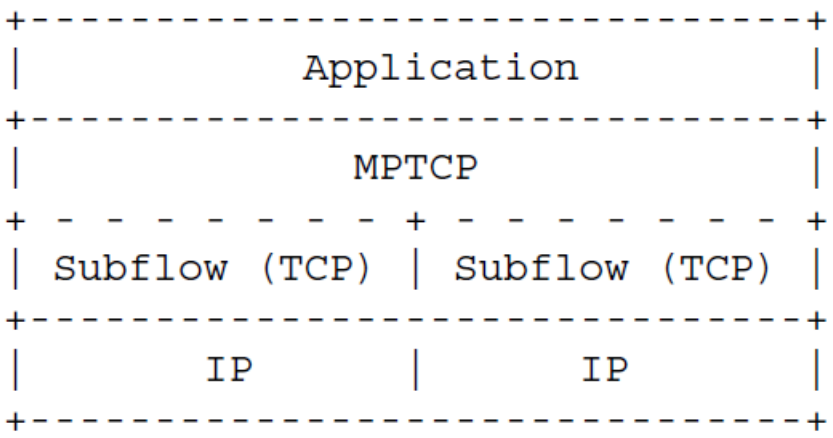


Figure 1.1: Multipath TCP network stack

By using multipath TCP a smartphone can hand over the workload to the other sub-flow seamlessly. This seamless handover of the workload from one sub-flow to another creates mobility opportunities[7]. One of the main decision that MPTCP protocol must take is selecting the path that the next segment should take. This selection is done by the scheduler.

### 1.1.2 Ad hoc network

Ad hoc is an decentralized structure where devices in an ad hoc network are connected to one or more devices in the same network. This allows devices to send data directly to other devices without routing through a centralized access

point. In an ad hoc network devices are not static at any given time. One or more devices are allowed to disconnect from the ad hoc network. Due to this nature ad hoc networks are dynamic. There are technologies associated with ad hoc networks. Such as

- Wireless ad hoc network
- Mobile ad hoc network
- Vehicular ad hoc network
- Smart phone ad hoc network

In this research, we mainly focus on mobile ad hoc networks and smartphone ad hoc networks.

According to Wikipedia[8] mobile ad hoc networks are highly dynamic topologies where each device in MANET maintains information that is required to properly route traffic. Each device forwards traffic that is not unrelated to own use. Each device in MANET is free to move independently in any direction, and will therefore change its links to other devices frequently.

According to Wikipedia[8] smartphone ad hoc network leverage the existing hardware (primarily Bluetooth and Wi-Fi) in commercially available smartphones to create peer-to-peer networks. SPANs use the mechanism behind Wi-Fi ad-hoc mode, which allows phones to talk directly among each other, through a transparent neighbor and route discovery mechanism. SPANs support multi-hop routing (ad-hoc routing) and relays and there is no centralized authority to govern, therefore peers can join and leave at will without destroying the network.

## 1.2 Research Problem

### 1.2.1 Research Questions

This research can be divided in to 4 main sub topics as follows

- How multipath TCP can be prioritized for mobile ad hoc networks?
- How to pass a user hint to the MPTCP kernel level?
- How to speed up the path recovery process?
- What are the impacts on performance when prioritizing multipath TCP for mobile ad hoc network?

### 1.2.2 Objectives

Using an ISP to transfer data is a costly operation. Ad-hoc networks provides a Zero cost data transfer medium. Usability of connection solely depends on the volatility of the ad-hoc network. In this research we intend to address this problem of having a reliable connection with most cost effective routing.

To achieve this we used an implementation (MTCP) that is still in the development phase. Therefore we aim to contribute to this open source project so that everyone can benefit from this. We intend to route majority of data through ad-hoc network but when an ad-hoc network fails we route the data using alternative methods that would cost money. This is a trade between cost and reliability. Therefore users will be able to have a reliable cost effective method of transferring data

### 1.2.3 Project Aim

This research is aims to provide the users a cost effective way to route the TCP data through a mobile ad-hoc network with minimum drawbacks to the performance of transmitting data. We aim to implement a reliable connection with multiple paths. This research is also focused on delivering an implementation that does not degrade the performance.

## 1.3 Justification for the Research

This research mainly contributes to two domains computer science domain and mobile users domain. Due to the significant number of mobile users, this research has a significant impact on society. With this research, mobile users will have the ability to route peer to peer TCP traffic without routing through ISP. This will significantly reduce the cost of communication. Furthermore this will reduce the load from the ISPs giving them an opportunity to expand their customer base without introducing additional infrastructure.

Contribution to the science domain mainly happens in the form of functionalities to be added to the multipath TCP source files. Since this is an open source project, this provide a huge learning opportunity society. MPTCP is relatively new. Therefore there is improvement opportunities in this research domain for further researchers. This can be further developed to be deployed with the vanilla versions of Android operating systems.

## 1.4 Methodology

This research uses existing knowledge in multipath TCP and ad hoc networks to identify a method to prioritize data flow through Ad hoc network. The process of progressing through the research is two parted. The first part of the progression was focused on accumulating knowledge about the multipath TCP implementation and related research. The second part of the progression is focused on making changes to Linux multipath TCP files and evaluating results on a laboratory environment.

In the first part of the progression was to do a preliminary study to identify the similar researches already done and related work carried out by other researches. This mainly focus on multipath TCP solutions that have already researched on. The second step of the progression is to read through multipath TCP documentation to identify, how a solution can be implemented.

In the second part of the progression we set up a laboratory environment for development and evaluation purposes. Setup of this environment was on virtual machines. This virtual machines configured with two interfaces each. Theses interfaces was paired together into separate sub-nets. As the second step we made improvements to Linux kernel files. This step was followed by evaluating the results of the developed Linux kernels' performance on laboratory setup. Step two and three was repeated throughout the research.

To evaluate the performance a test environment was created. In this test

environment performance and the sub-flows of the final solution was evaluated. Overview of the methodology is given in the figure 1.2.

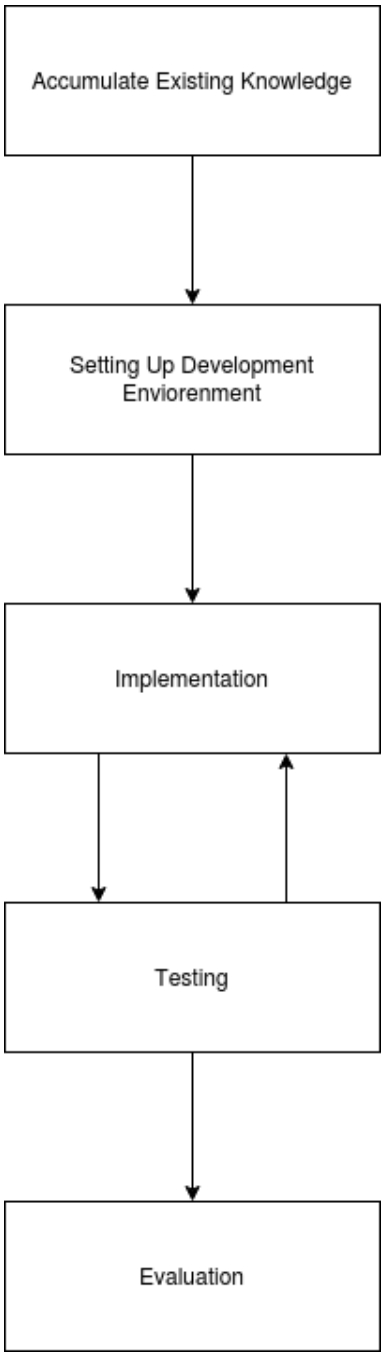


Figure 1.2: Research Methodology Overview

## 1.5 Outline of the Dissertation

This dissertation is structured to start from an overview of the research and step by step move into deeper concepts. Second chapter of this research is designed to give the reader a good grasp of the field. This chapter will explain theories about the domain. This chapter will provide the necessary information to understand the research in depth.

Third chapter of this research How the research was done according to the research methodologies, sources of data, and idea about the instruments used to carry on this research.

Fourth chapter of this research is designed to give the implementation details of the research. Fifth chapter of this dissertation is designed to express the evaluation results of this research. After that conclusion, the remark of this research is presented.

## 1.6 Scope and Delimitations

### 1.6.1 In Scope

In this proof of concept research, the scope is limited to supplementing multipath TCP kernel files with additional functionalities to handle prioritization of ad-hoc sub-flow. To provide means for user to specify when to enable prioritization of ad hoc sub-flow, expose an interface from user level to kernel level with minimum violations to the network layers. Furthermore changes to multipath TCP scheduler was made to choose strategies to optimally acquire which sub-flow use and to speed up the recovery of lost subflows.

### 1.6.2 Out Scope

In this research following details will be considered as out of scope. Due to the time constraint research evaluation on real world environment will consider as out of scope. Also ad-hoc routing will be considered as out of scope.

## 1.7 Conclusion

This chapter laid the foundations for the dissertation. This chapter gives a overview about the research. More details about the research is presented in the following chapters. This chapter introduced the research problem and research questions and hypotheses. Then the research was justified, the methodology was briefly described and justified, the dissertation was outlined, and the limitations were given. On these foundations, the dissertation can proceed with a detailed description of the research.

# Chapter 2

# Literature Review

## 2.1 Multipath TCP

In the "TCP Extensions for Multipath Operation with Multiple Addresses" request for comments[3], the researcher defined multipath TCP as a set of extensions to regular TCP to provide a Multipath TCP service. Because of this extension transport layer obtain the ability to operate a single connection across multiple paths. Because of this ability to create a connection using multiple paths that might utilize the multiple interfaces available, MPTCP gains more fault tolerance and high throughput. This ability to create a connection using multiple paths is achieved by creating sub-flows to the same connection. These sub-flows hold the characteristics of a normal TCP connection. So when a middlebox analyses the packets there will not be any sequence numbering errors.

By using multipath TCP a smartphone can hand over the workload to the other sub-flow seamlessly. This handover of the workload from one sub-flow to another creates mobility opportunities [9]. With MPTCP one of the main decision protocol must take is choosing the path that the next segment should be sent. This is handled by the scheduler.

## 2.2 MPTCP Scheduler

There are a number of schedulers implemented in the Linux MPTCP kernel. Scheduler function is defined as a modular function so different scheduling policies can be plugged into the scheduler. Currently, vanilla MPTCP kernel shipped with four different schedules.

- MPTCP BLEST
- MPTCP Round-Robin

- MPTCP Redundant
- Default MPTCP Scheduler

Scheduler breaks the data into segments that can be transmitted over the subflows and schedule these segments to an available sub-flow. Checking the availability is also done by the scheduler. The scheduler is dependent upon information about the availability of paths exposed by the path management component and then makes use of the subflows to transmit queued segments. In order to make scheduling decisions, a scheduler makes use of various information such as state of each TCP subflow, congestion window, and RTT estimation.

There are few schedulers already implemented to schedule data using different policies. In my research, a new policy mechanism was created to give a subflow a different priority than others according to user hint. In the literature review, there is a solution to define subflows as backup subflows, but the problem with this method is switching into backup subflows only happens when there is no available active subflows. This method does not take other factors into consideration.

## 2.3 TCP Loss Recovery

Standard TCP does not tolerate packet losses. If a packet loss happens, TCP retransmit the data segment that was lost. This retransmission process [10] is repeated until an acknowledgement is received or retransmission retire number exceeds the defined value in the OS. In MPTCP this mechanism is also used to identify if a path is reactivated. There is a drawback to this mechanism when it comes to MPTCP. In the standard TCP after one retransmission, retransmission timeout is doubled. MPTCP relies on this mechanism to see if a path is reactivated. As time passes, the frequency of retransmision drops. Time to recover the path increases.

## 2.4 MPTCP Path Restriction

MPTCP has multiple paths to send data. Therefore MPTCP kernel should know when a subflow level transmitting is not possible. If not, the scheduler might schedule the next packets through the subflow which is not available at that time. This will reduce the throughput and will waste the kernel memory. To overcome

this issue MPTCP kernel Sets a field in socket structure to 1. In the scheduler, this field is checked to identify whether the subflow is available at that time.

In this research, user hints must be used to restrict the paths as well. In the current implementation of path restriction, user-level hints are not considered. This limitation will be addressed in this research.

## 2.5 MPTCP Path Reactivation

MPTCP can restrict a subflow if it is not suitable to send data. These restricted paths can become suitable with time. If a subflow becomes suitable MPTCP reactivates the subflow. MPTCP achieves this by constant TCP retransmission over the fallen subflow. If a retransmitted packet gets an acknowledgment, MPTCP input handler sets the previously set socket field that restricts the use of subflow to 0. Because every time MPTCP scheduler gets a packet to schedule MPTCP check this field to see if the subflow is available, the next packet will be sent on this subflow. Due to the fact that retransmission happens with exponential intervals in between, path recovery time increases exponentially with the amount of time it takes to recover.

In the current method of path reactivations, MPTCP does not consider user-level information. This limitation will be addressed in this research. Current path reactivation methods only consider the fact that the path is completely unavailable or not. But in this research path reactivation and restriction consider the measurements between subflows additionally.

## 2.6 MPTCP Loss Recovery

TCP has three different mechanisms for lost recovery, namely fast recovery, retransmission timeouts and TCP lost probe. TCP lost probe is used to identify tail-end losses. Fast recoveries are significantly faster than retransmission timeouts. How fast recovery works is, when a packet is received, the packet comes with an acknowledgment number, with this acknowledgment number sender can identify the last received in-order packet. This works by identifying duplicate acknowledgments. If the system identifies three duplicate acknowledgments, even before the retransmission timeout happens TCP will send retransmission. This significantly reduces recovery time. RTO is a timer that is set with each packet. If the packet acknowledgment does not reach the sender before the timer expires,

the sender will retransmit the packet again. This is significantly slower than fast retransmission.

MPTCP also uses these mechanisms for loss recovery. However, it is not completely clear how the loss recovery happens in the implementation [11] and which subflow retransmits the lost packets. If the recovery is handled at the meta-level, the lost packet may be rescheduled and retransmitted at the available subflow with the lowest RTT. If the recovery is handled at the flow level, the packet may be retransmitted in the same subflow. In the Linux implementation of MPTCP, a heuristic was used for loss recovery. The heuristic says that if the retransmission is a fast retransmit, then the same subflow is used for retransmission. If the retransmission is for a timeout, then the scheduler would re-evaluate the packet transmit options. In addition, the lost packet is always retransmitted on the original subflow as required by TCP.

# Chapter 3

# Design

The main focus of this research is to prioritize a subflow created by MPTCP connection. Before focusing on the main problem of the research, exploring the MPTCP behavior was necessary due to the fact that MPTCP is still in the development stage and has not been released as a commercial solution. Therefore latest version of the MPTCP at that time was configured and installed. In this research we created a working model after analyzing the behavior and functionalities of MPTCP. In the final stage of the research, the proposed MPTCP solution was tested against standard TCP and original MPTCP (version 0.95) to evaluate the results.

By considering the above-mentioned requirements, research methodology was divided into 5 stages as follows figure 3.1. Inside the methodology stages tools used and justification for using the tool, data source, etc. will be discussed.

## 3.1 Configure MPTCP

Configuring, compiling and installing MPTCP kernel was done as the fist stage of this research. Latest stable version source code was taken to compile the MPTCP kernel. Latest stable version of MPTCP was 0.95 at the time of this research and was taken from the open-source repository of MPTCP [12].

For the compilation of the source code, Ubuntu 16.04 was used. The choice of OS does not affect how the kernel behaves after the compilation. Before the compilation of the source code, configurations were made to the makefile in order to enable the MPTCP functionalities of the kernel. These configurations include enabling MPTCP functionalities, advance congestion control, path manager control and policy routing. Advance congestion control mechanisms were introduced into the MPTCP kernel due to the fact that standard congestion controls were not effective when handling multiple paths. These were introduced to the MPTCP

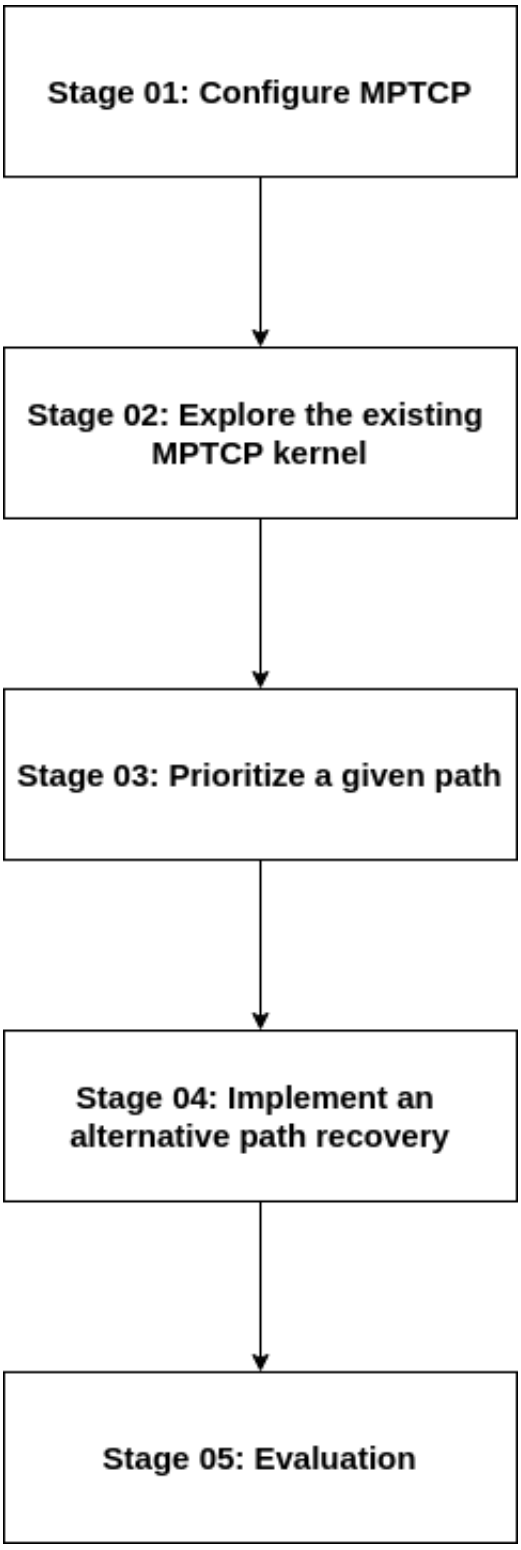


Figure 3.1: Stages of Research

kernel. Path manager is also a loadable module in the MPTCP kernel which was not in the Linux kernel. This module handles the path creation. Compilation was done using GCC [13] after configuring these changes.

The virtual environment used for the analysis was as follows. Two virtual

machines were used and a host-only network was created between these two VMs. These VMs were created using VMware player [14]. One VM was configured as the server and the other VM was configured as the client of this system. Both of these VMs were configured with two interfaces each. Each of these interfaces were given a different submask to represent two different networks in a real-world scenario. Therefore, even though all the interfaces belong to host-only network, one interface of a VM is directly connected to only one interface of the other VM. In the server, VM Iperf [15] is used to listen to a connection and act as a server. On second VM was used as the client. To send requests from client Iperf was used. Client request data from the server, when this occurs two subflows were created between two interface pairs. This was handled by the MPTCP kernel. Transfer rates of each interface were measured using ifstat [16] to ensure both interfaces are active and transmitting.

After all the configuration steps, configuration of the firewall was done. This was done to allow both interfaces to connect with each other without the interference of the firewall. After this step configuration of network interfaces was done. Configuration of the interfaces was as follows.

- Client VM:
  - Interface ens34:
    - Address: 10.222.10.1
    - Netmask: 255.255.255.0
  - Interface ens35:
    - Address: 10.222.20.1
    - Netmask: 255.255.255.0
- Server VM:
  - Interface ens34:
    - Address: 10.222.10.2
    - Netmask: 255.255.255.0
  - Interface ens35:
    - Address: 10.222.20.2
    - Netmask: 255.255.255.0

After the configurations, using these configurations standard behavior of MPTCP was analyzed.

## 3.2 Explore the existing MPTCP kernel

In the second stage of the methodology, code was analyzed to map the behavior with the code. This stage is crucial due to the fact that MPTCP is in the development stage and the documentation provided for the MPTCP kernel is limited. For this, MPTCP kernel source code was taken from the Github repository of the MPTCP researchers. Github link of the repository is available on the official MPTCP website [17]. In this stage, the structures that needed to be altered and the architecture of MPTCP scheduler architecture, lost recovery architecture were identified.

More details about the architectures will be discussed in the next section. Also in the exploration of MPTCP some experiments were done to get the exact idea about the behavior. This includes restricting the use of a path at a subflow creation moment, Tuning congestion windows of a subflow to limit the use of subflow, keeping a linear RTO till the connection expires, etc. This will be discussed in more detail in the implementation chapter.

## 3.3 Prioritize a given path

In subfolw prioratization stage of the research, implementation of the solution began. The main intention of the research was to identify a way to prioritize the interface that is bound to an ad-hoc network. For that, identifying the behavior of the scheduler was necessary. This was necessary because the packet scheduling decision was taken at the scheduler. Just by understanding the behavior of the scheduler is not enough due to the fact, users need to pass a hint to the kernel level, stating the network interface bound to the ad-hoc network. To achieve this functionality MPTCP controller need to be modified.

Both of these functionalities reside in the MPTCP kernel. Therefore modifying the kernel source code is inevitable. As the first step of modifying the scheduler and controller, functions involved in this process were studied. The second step of this stage was to alter the source code to behave in a way that prioritizing the ad-hoc subflow is achieved. In this stage of the research prioritizing the ad-hoc subflow and passing user hint to the kernel level is achieved.

## 3.4 Implement an alternative path recovery

This is the implementation of the lost path recovery. Implementation details of this will be discussed in the next chapter. This function is implemented in the output functionality of MPTCP. This function also resides in the kernel source code. Therefore modifying the kernel source code is inevitable. As the first step of modifying the output, functions involved in this process were studied. The second step of this stage was to alter the source code to achieve a fast recovery.

## 3.5 Evaluation

For the evaluation of this research, Ubuntu 16 was installed to two VMware player virtual machines.The kernel version of this ubuntu installation is 4.4.0 (latest at the time of evaluation). To reduce the effect of external factors, modified MPTCP kernel and standard MPTCP kernel were installed on the same virtual machines.

Evaluation can be divided into 5 categories.

- Same latency on both interfaces evaluation
- Path switching on modified kernel
- High latency on ad hoc, srtt difference between the minimum and the prioritized path is less than minimum
- High latency on ad hoc, srtt difference between the minimum and the prioritized path is greater than minimum
- Prioritized subflow recovery speed evaluation

To create these scenarios command line utilities were used. To introduce a packet loss tc [18] command-line utility was used. To introduce latency, VMware player was used. When measuring the ad hoc subflow recovery speed precision is important. For all the evaluations other than hoc subflow recovery speed, the data transfer speed was measured with one second intervals. This was done using the ifstat command line utility. To measure the ad-hoc subflow recovery speed with millisecond interval, ifpps [19] command-line tool was used. All these evaluations, except the last one, were done against modified MPTCP kernel, standard MPTCP kernel, and standard TCP kernel. The last evaluation was done against modified MPTCP kernels' ad hoc subflow recovery vs standard MPTCP kernels' one subflow. Packet loss was emulated using tc command-line tool

# Chapter 4

# Implementation

As described in the previous sections, there are some issues when it comes to prioritizing a subflow with currently implemented schedulers. Because doing so, MPTCP kernel completely ignores the network conditions when giving the priority to a subflow. The main focus of the research was to prioritize a subflow. While doing so proposed solution address the above mentioned issues by introducing a new scheduler policy that is sensitive to both user and the network conditions.

Before discussing further technical details of the implementation, dividing the implementation into hree sub implementations would be suitable. While discussing the sub-topics there are few architectures to be mentioned. Divided subtopics are as follows.

- Passing user hint to the kernel
- Prioritizing ad hoc subflow
- Implementing alternative loss recovery

## 4.1 Passing User Hint to the Kernel

Passing the hint from user level to kernel level is done through a proc [20] file. Proc is a pseudo file system in Linux operating system which is available on /proc. This is an interface to the kernel data structures. Most proc files are read-only by default. By changing a proc variable a user can pass variables to the kernel level. In this research, the user will write the ad-hoc interface to the proc file. This value will be passed to the kernel as a hint from the user. To achieve this first, a proc file should be created. This was done in mptcp_ctrl.c.

To create a proc file MPTCP has defined a static structure. In this structure, MPTCP holds the proc files that have been defined. Thus we want to add an additional proc file, we include a new object to the mptcp_table array. By

this inclusion, a proc pseudo file can be created in the /proc/net/mptcp folder. The name given for the proc file is adhoc_interface. This proc was created with permissions as follows.

Permissions:

- root: read, write
- group: read
- others: read

To write to this pseudo file, a user can use sysctl command in the Linux kernel.
Eg: sysctl net.mptcp.adhoc_interface=10.222.10.1

Just by creating a proc file would not let users pass values to the user level. When a value is written to a proc file there should be a handler to handle the proc. As the second step of creating proc, proc handler was implemented. Proc handler, handles two major functionalities. Namely,

- Handling proc reading
- Handling writing to proc

Even after writing to the proc, proc reading happens. This means regardless of reading from the proc or writing to the proc, proc reading should happen. Before implementing these functions, there should be a place to hold the value. To store the value char array was created in the mptcp_ctrl named face_addr. This array is shared between functions in mptcp_ctrl.c and my.h file. Justification for sharing variables with my.h will be explained later on. The reason defining this value in the global scope of this file is if defined in the global scope this function will be reset every time a user calls the proc handler. But this value needs to be persistent.

After defining a variable to store the value, as the next step reading from the variable and showing it whenever the proc is called was implemented. To do this a structure will be created with value and length of the string. The structure is as follows.

struct ctl_table tbl = {
.data = val,
.maxlen = 16,
}

Then assigning values to this structure happens. To do this string copy was used. One downside of using string copy function is, it is on the heavy side of the sense of computations. This does not have a huge effect on the performance of MPTCP due to this function is only called when a user reads from proc or writes

to the proc. Reading or writing to the proc does not happen often. After this step, the value of the face_addr will be copied into the data field of the structure. After this step returning the data to the proc pseudo file happens. This was done using the proc_dostring function which is defined in the MPTCP kernel. This is the steps to show the value stored in face_addr to the user.

As explained above before even after writing to the proc, reading from the proc should happen. This is due to the fact, the value should be displayed to the user after writing to the proc. Therefore in the implementation proc_dostring function is called to show the value. To proc_dostring function, integer named write is passed as an argument. From this value proc_dostring determines if it is a write or a read. If it is a write, proc_dostring write to the address of the data field of the struct tbl. This will ultimately write to the face_addr value. After writing, written value will be returned to the proc.

After writing to the face_addr, the value will be stored as a string. This is not optimum when it comes to comparisons. When two strings are compared, it is costly. In order to reduce the computational cost of checking if the user has defined an ad-hoc interface, whenever a user defines a value other than 0.0.0.0 value was stored in an integer name adhoc_avail. But this value is not bound to connections. To bind this to a connection, a field was introduced to the socket structure called is_adhoc_avail. To understand where this value was assigned socket architecture is needed.

As explained in the literature review, user-level programs do not know that the program is connected to a MPTCP socket. This is achieved by defining a standard socket directly connected to a meta socket. All data sent and received on a Multipath TCP connection passes through the meta socket. This meta socket contains the head of the pointers to various structures including the pointer to the MPTCP control block. This is the head of the MPTCP subflow linked list. MPTCP cb points to the first TCP subflow. This socket is wrapped in a MPTCP socket. This MPTCP socket holds the pointer to the next TCP subflow and so on. This is illustrated in the diagram below. Figure 4.1

As explained before mptcp_cb can be used to hold information about the connection. Therefore if there is an ad hoc subflow available, information can be stored in the mptcp_cb structure. In this research, this value is stored in a new field that was introduced to mptcp_cb called is_adhoc_avail. Every time a packet is scheduled to be sent on a subflow this value will be checked. After defining is_adhoc_avail field in the structure, binding adhoc_avail to is_adhoc_avail should happen. This is achieved in the MPTCP socket creation stage.

When a MPTCP connection initiates, at least one subflow must be created.

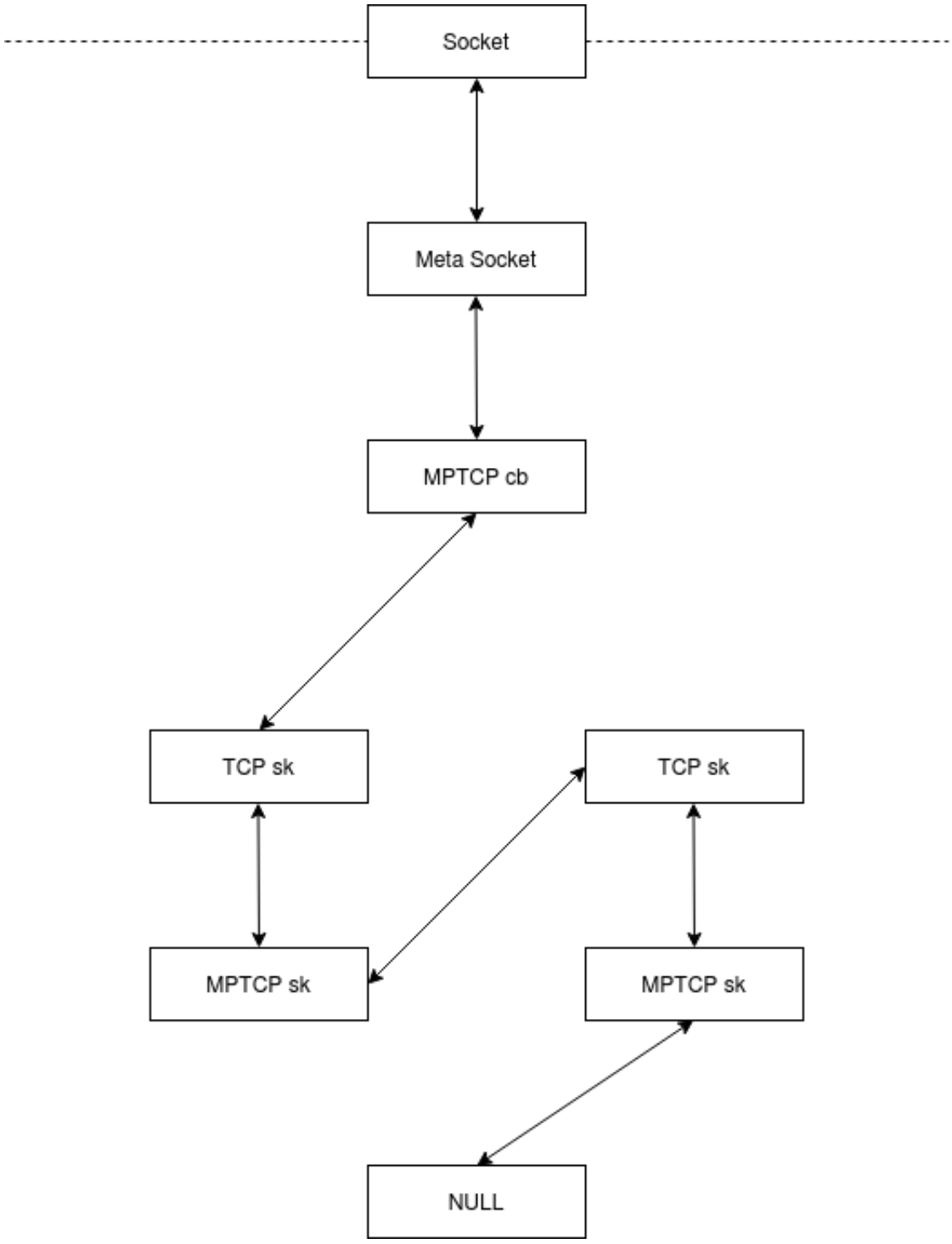


Figure 4.1: MPTCP Socket Structure

For a subflow MPTCP socket wrapper is present. In this socket wrapper pointer to mptcp_cb is present. Using this pointer, value to is_adhoc_avail field in the mptcp_cb is assigned. Adding this MPTCP wrapper socket is also present in mptcp_ctrl.c. Therefore we define adhoc_avail globally to be accessed by the function mptcp_add_sock. In this function, adhoc_avail will be assigned to is_adhoc_avail variable in the mptcp_cb structure. This will conclude the passing

user variable to the kernel level section.

## 4.2 Prioritizing ad-hoc Subflow

After passing the hint and defining a method to identify if a path through an ad hoc network is available, we can move onto the second subtopic that is prioritizing the subflow. This is a scheduling decision. If there are two or more paths that a packet can take to reach the receiver, a decision to be made to selec a path. This decision is taken in the MPTCP scheduler. The following figure gives an abstract idea about schedulers' functionalities.

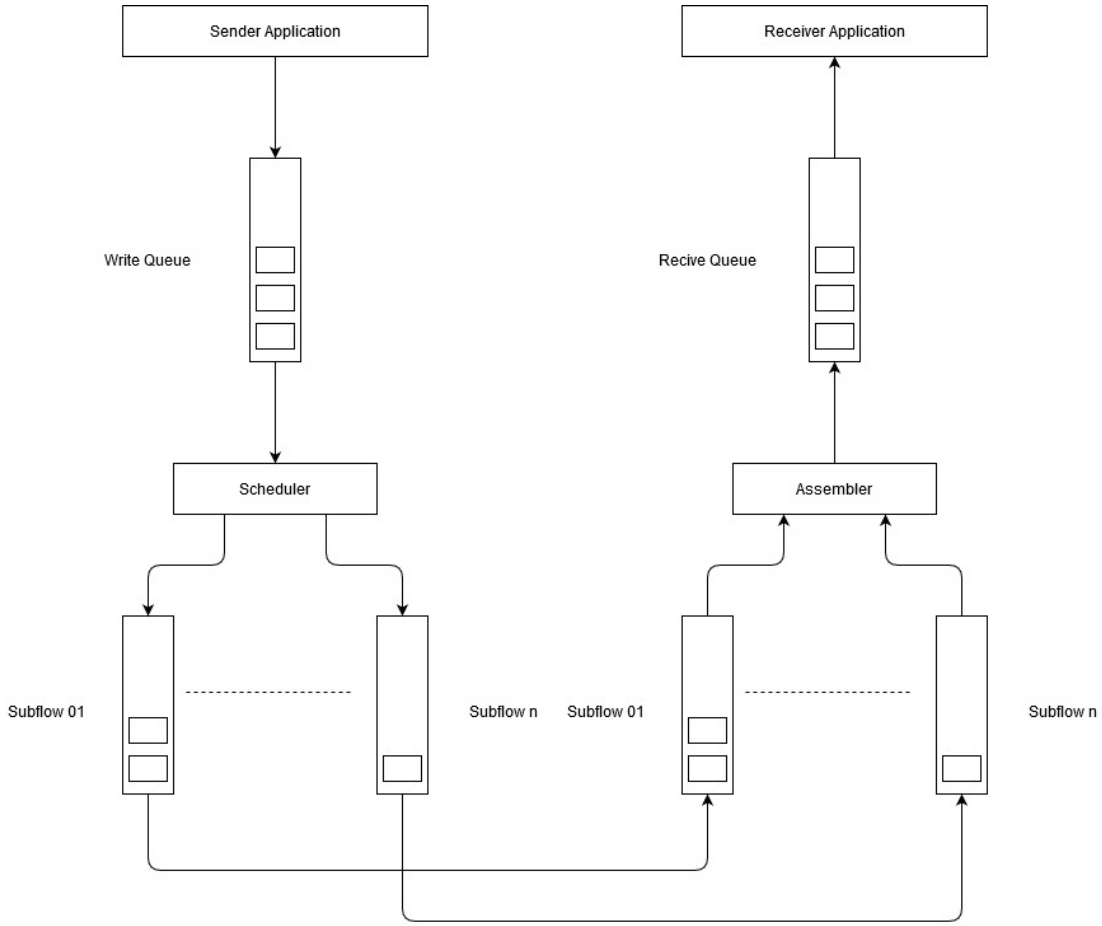


Figure 4.2: Overview of MPTCP Scheduler

In this research scheduler that was implemented is highly influenced by the default scheduler. The default schedulers' policy is to take the subflow if there are no packets scheduled or select the minimum latency path. Here in this scheduler, there is a function that defines what subflow to schedule a packet. In this

research, alterations were made to get_subflow_from_selectors function to achieve the expected functionality.

Before moving into finer details, it is better to give an overview of this function. First, this function takes a pointers to MPTCP control block, scheduled socket buffer, whether searching for a subflow in active paths or backup path, zero window test value, and a variable to express if a path has been found. Then each subflow in the MPTCP control block is checked with escape conditions. This can be illustrated as a state diagram below.

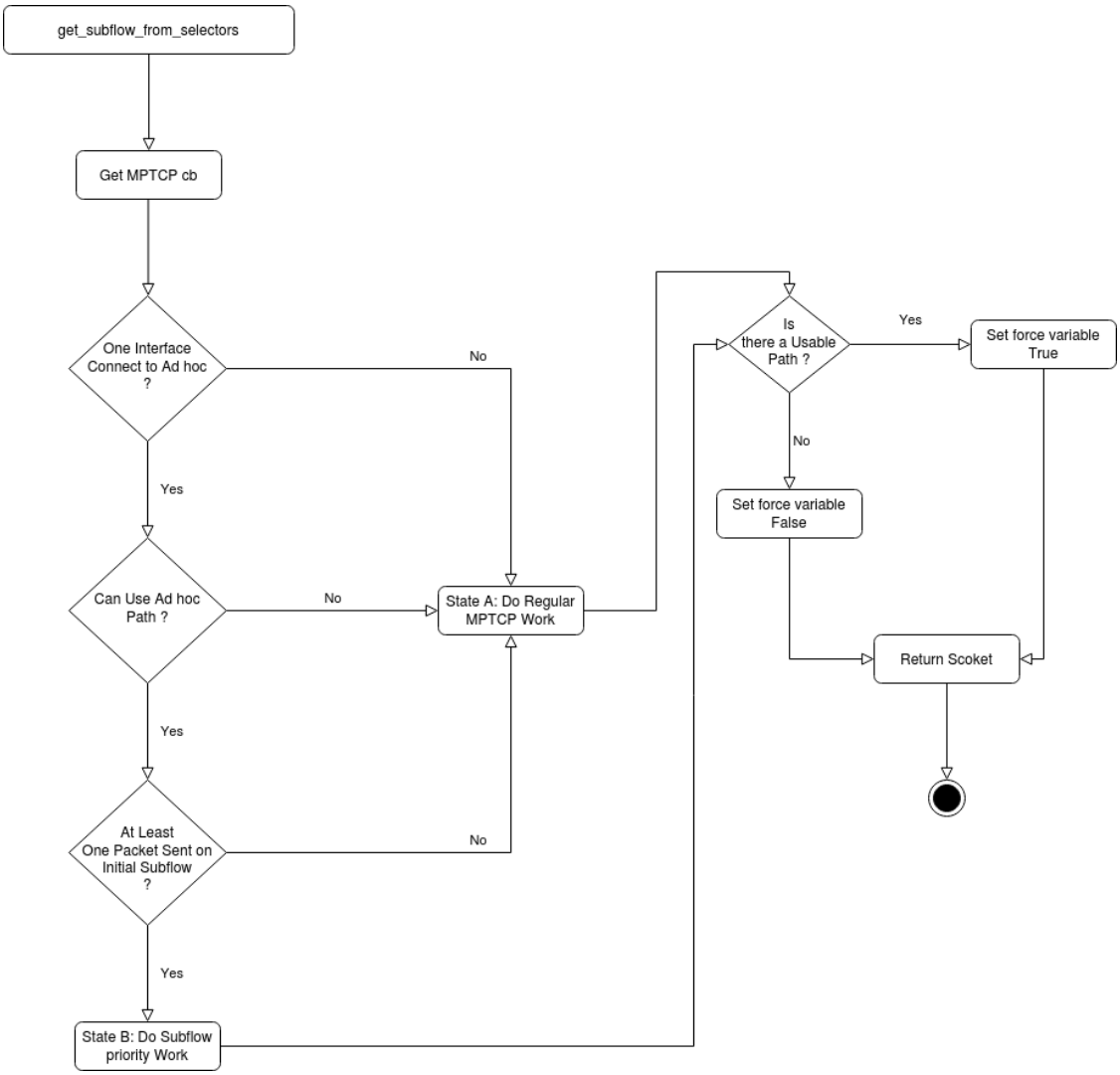


Figure 4.3: State Diagram of MPTCP Scheduler

As explained in the diagram, before beginning to prioritize the subflow, there are some requirements to be checked. These requirements are

- Is there an ad-hoc network connected

- Is this ad hoc subflow path available
- In this connection if at least one packet routed on initial subflow

In this scheduler implementation, first we check if at least one network interface is connected to an Ad hoc. This was done by comparing the value in the is_adhoc_avail variable in MPTCP control block, which was set in the connection creation time with the use of a hint that was passed onto the kernel by the user. If at least one interface is connected to an Ad hoc network, it will continue onto other checks. If there are no interfaces connected to Ad hoc network, the scheduler will switch to state A. More details about state A will follow later on in this chapter.

After checking to see if there is an Ad hoc interface present, a second check will be done to check if the subflows connected to Ad hoc interface are viable. In standard MPTCP kernel viability of path only depends on the availability of the path. In this research viability of the ad hoc subflow depends on the availability of the path as well, the smooth round trip time of the ad hoc subflow compared to the minimum smooth round trip time of all the other subflows. Implementation details about the comparison will be explained later in this chapter. Therefore a new variable is needed to hold this information. This variable is defined in the MPTCP control block structure as adhoc_priority. This will indicate if the ad hoc subflow in this connection is viable to use. This was defined in the MPTCP control block to achieve better performance. If this variable was defined in the MPTCP socket structure, check for this variable has to be done inside the loop that wll be explained in the state transition diagram. This is a costly operation. Therefore this variable is defined in the MPTCP control block structure.

After the check of the path viability of ad-hoc subflow, the scheduler must check if at least one data packet is sent on the connection initiation subflow. This one packet needed to startup the dataflow. If this check is not there and the initial connection happened on a subflow that is not through ad-hoc interface, if kernel schedules the first packet through ad hoc subflow, the data flow will not start. In this research main priority is to route all the data through ad hoc subflow, this will occur. To overcome this, the scheduler will switch to state A. This variable is also defined in the MPTCP control block. This variable was given a name adhoc_tries. If any of these checks fail, the scheduler will switch to state A.

### 4.2.1 State A

State A functionalities are heavily influenced by default TCP scheduler. The functionality of state A is as follows. In state A, the scheduler executes a loop to

traverse all the subflows of MPTCP control block. In each iteration as the initial step scheduler checks if the selected subflow of that iteration is wanted socket. Meaning this function search for if it is defined as an active subflow or a backup subflow. This is due to get_subflow_from_selectors function execute twice, first to find active subflow that is viable, if not a second time to find backup subflow that is viable. Next step of this iteration is to search if the data segment to be transmitted is already scheduled in this socket. After that check scheduler is checking if this socket is definitely unavailable at that time or temporarily unavailable. These conditions are used to skip the iteration if necessary.

After these steps state A store the minimum srtt of the subflows and the pointer to that socket. At this point state A also stores the srtt of ad hoc socket. Identifying an ad hoc socket was done by comparing the source address of sockets. If the source address matches the address that the user passed onto kernel, that socket will be considered as an ad hoc subflow socket and the srtt of this socket will be stored locally in adhoc_srtt. The code segment for this is presented in the Appendix.

Entering to state A decision is taken with the above mentioned comparisons. Whenever an ad hoc path is viable, the scheduler should prioritize the ad hoc path. Therefore once an ad-hoc path becomes viable, the next packet schedule path should switch to the state B to prioritize the ad hoc path. Therefore switching from state A to state B mechanism is needed. As the next step of this execution, this condition is being checked. Switching from state A to state B occurs whenever the following conditions are true. Conditions of this switch are, If ad hoc socket is available and if the srtt difference between ad hoc srtt and minimum srtt is less than minimum srtt. If both of these conditions satisfied adhoc_priority variable is changed to indicate that a viable ad hoc routing path is available.

State A is the state that handles packet scheduling if there is no viable ad hoc subflow. The functions of this state are explained above. If all of the following conditions, is there an Ad hoc network connected, is this ad hoc subflow path available and, in this connection did at least one packet sent on initial subflow are true scheduler will fall into state B.

### 4.2.2 State B

State B is the state that handles the scheduling when an ad hoc route is viable. In this state also as in state A, a loop is executing to traverse all the subflows. As in state A, each subflow is examined to identify if it is an active or backup subflow. Then check whether the data segment is already scheduled on this. Then move onto checking if the subflow is definitely unavailable or temporarily unavailable.

These conditions are used to skip the iteration if necessary.

After these considerations, state B attempts to find a socket that is bound to an ad hoc interface. This is done by comparing the source address of sockets. If the source address matches the address that the user passed onto kernel, that socket will be considered as an ad hoc subflow socket and the srtt of this socket will be stored locally in adhoc_srtt. If there exists a socket that is bound to an ad hoc interface, the viability of this socket should be checked. This is done by comparing with the minimum srtt as explained in state A. If the viability check passes, the socket will be returned. Otherwise adhoc_priority variable in the control block will be changed to express there aren't any viable ad hoc paths available.

## 4.3 Implementing Alternative Loss Recovery

The final section of the implementation forcused on implementing an alternative loss recovery. This is needed due to the fact, lost recovery attempts should happen frequently in an ad hoc network. This is due to the fact that ad-hoc network is a volatile entity. This alternative method was implemented in the tcp_timer.c. This alteration was integrated into the tcp_retransmit_timer function. As explained in the MPTCP path recovery in the literature review, MPTCP reactivates a path when a reply to retransmission happens. By speeding up the retransmission rate, we can achieve faster path recovery.

In standard MPTCP kernel, for each unsuccessful retransmission, retransmission timeout is doubled. This limits the ability of faster recovery. In this research, we implemented a policy to have retransmission keep the double of its initial retransmission timeout for 10 retransmissions and the switch to doubling retransmission timeout. To do so first we check if the socket that retransmitting is an ad hoc socket. Then if it is an ad hoc socket, change the policy to the policy we implemented otherwise keep the normal policy.

## 4.4 Additional Information

## 4.5 Backward Compatibility

Backward compatibility of MPTCP kernels is one of the main concerns when it comes to implementation. This means if either one of the machines involved in the ends were not compatible with MPTCP, MPTCP kernel should change to Standard TCP implementation. Therefore the implementation of this research should be backward compatible as well.

Since the implementation was done within MPTCP boundary and the MPTCP kernel is backward compatible, implementation of this research is also backward compatible. All the changes were made MPTCP socket structures. Therefore backward compatibility is not an issue.

# Chapter 5

# Results and Evaluation

As mentioned in the research design evaluation is done on Ubuntu 16.04 environment with two virtual machines with two interfaces. Rather than using a real-world environment virtual environment was used due to easy maintenance and easily configurable to emulate scenarios. This evaluation was configured into four categories.

- Same latency on both interfaces evaluation
- Path switching on modified kernel
- High latency on ad hoc, srtt difference between the minimum and the prioritized path is less than minimum
- High latency on ad hoc, srtt difference between the minimum and the prioritized path is greater than minimum
- Prioritized subflow recovery speed evaluation

In all the evaluations cost of routing through Ad hoc will be taken as 1 per kB and the cost of routing through other networks will be taken as C per kB for the calculations. Routing through an ad hoc is free. Therefore always C greater than 1.

## 5.1 Same Latency on Both Interfaces Evaluation

Blue Line: Prioritized Interface
Orange Line: Standard Interface

This evaluation was taken to measure the performance impact after the modification to the kernel. This evaluation was emulated on a network that has 4 Mbs maximum throughput to be closer to the real-world environment.

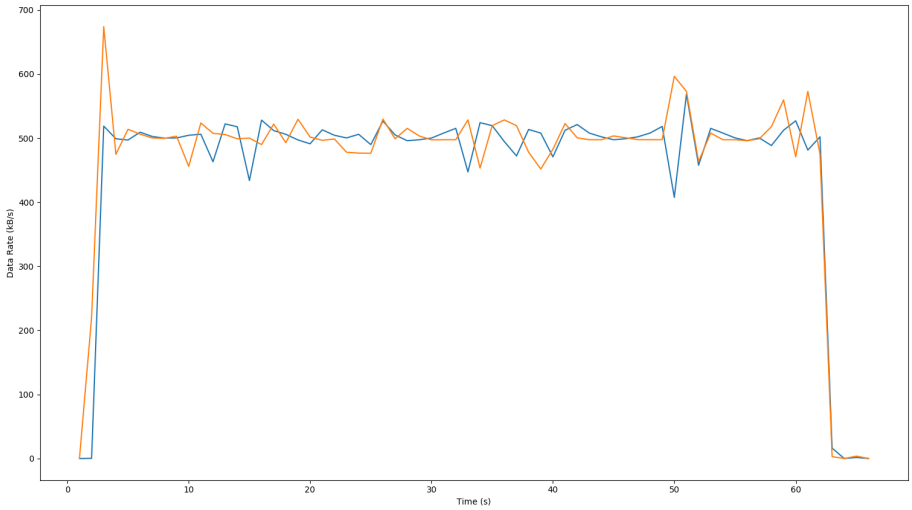


Figure 5.1: Same Latency on Both Interfaces with Modified MPTCP: Without Setting Priority

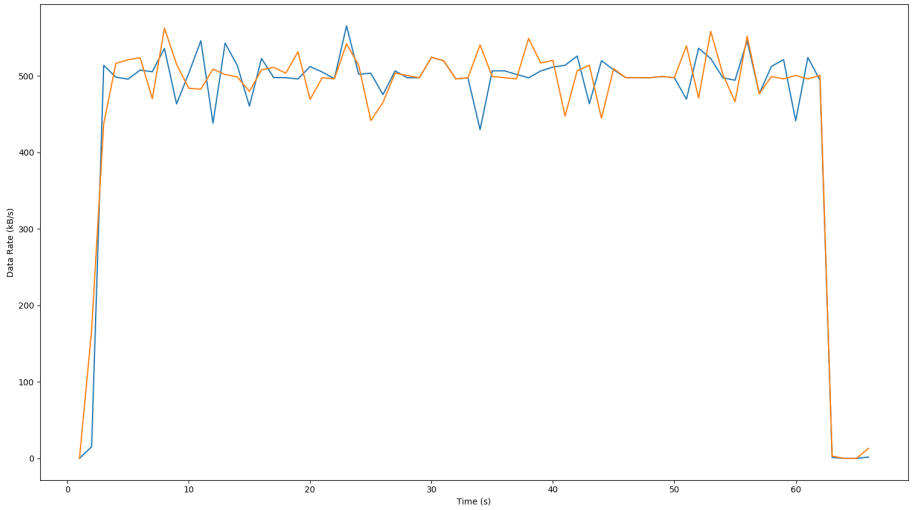


Figure 5.2: Same Latency on Both Interfaces with Standard MPTCP

In the comparison of figure 5.1 and figure 5.2, Both the interfaces on the modified MPTCP kernel reaches 500 kB/s speed. Also in the standard MPTCP kernel, Maximum data rate is 500 kB/s. Therefore significant gain or loss was not recorded when an ad hoc path is not specified.

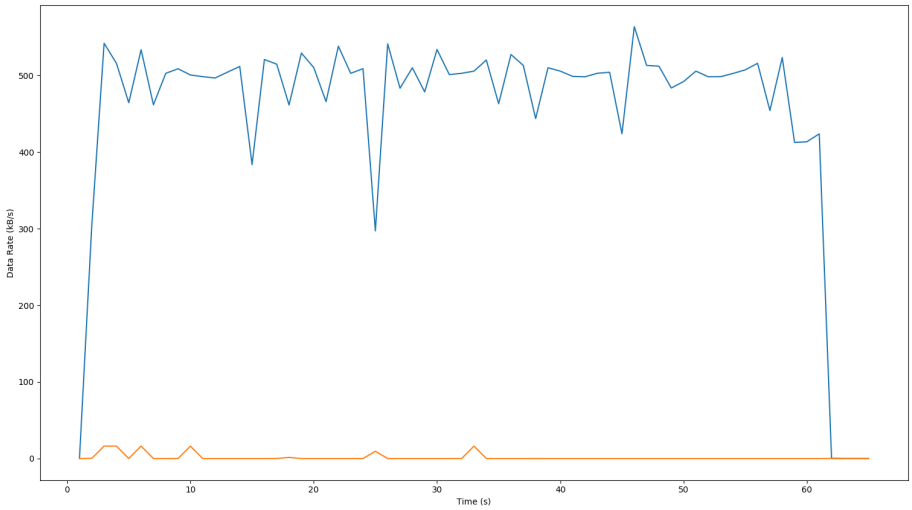


Figure 5.3: Same Latency on Both Interfaces with Modified MPTCP: With Setting Priority and Initial Connection Through the Prioritized Network

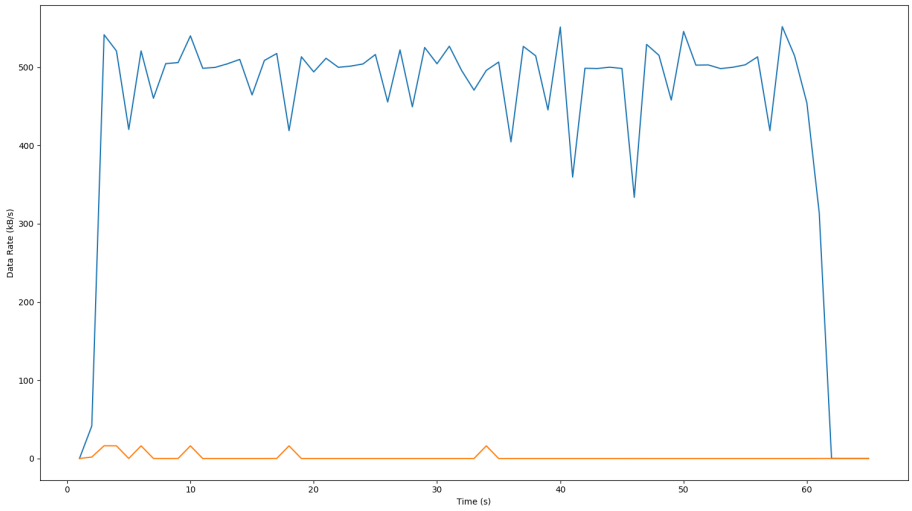


Figure 5.4: Same Latency on Both Interfaces with Modified MPTCP: With Setting Priority and Initial Connection Through the Non-prioritized Network

When comparing figure 5.3 and figure 5.4 together we can see scheduler, schedule packets to that interface. But by doing this data rate will be dropped to half of the data rate that we achieved before. The data rate is not the only factor we consider when we are choosing a route. By forcing data to follow the path that route through the Ad hoc network, we can reduce the cost of transmitting data. Thus if a user is sending 500 kB of data cost of routing with modified MPTCP with set priority will be 500. If a user were to send data using standard MPTCP

kernel cost will be (1 + C)/2. Therefore C is always greater than 1, the cost will be increased.

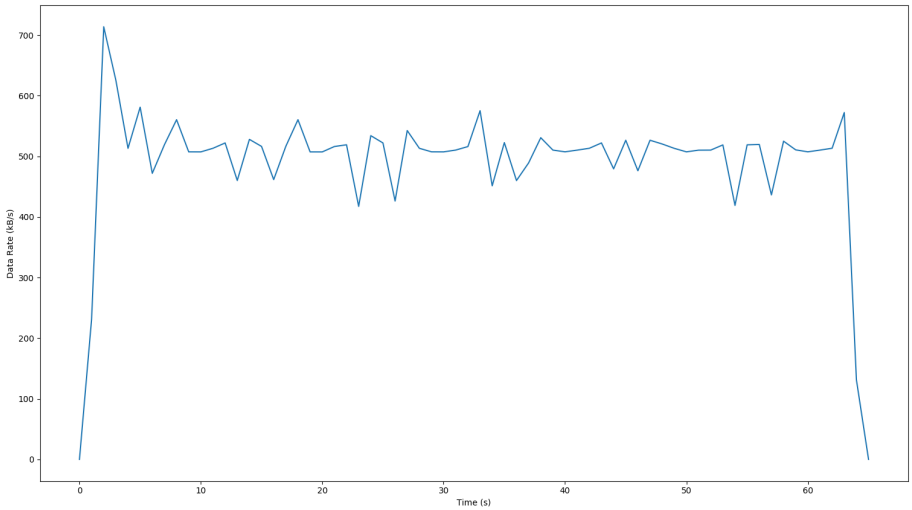


Figure 5.5: Standard TCP Performance

The performance of modified MPTCP kernel can be compared with standard TCP kernel. This comparison can be done by comparing figure 5.5 with figure 5.3 or figure 5.4. In this comparison, one can see the modified MPTCP kernel utilizes the single path as much as standard TCP utilize a path. So comparing the performance gain there isn't any significant loss of performance. But using the Modified MPTCP kernel is reliable. It switches between paths without destroying the connection. Standard TCP kernels lack this ability.

## 5.2 Path Switching on Modified Kernel

Blue Line: Prioritized Interface
Orange Line: Standard Interface

This evaluation was done to illustrate how the path switching was handled in the modified MPTCP kernel. Parameters of this evaluation are 40 ms latency, 4 Mb/s maximum throughput network. Every five seconds the server drops all the packets on ad-hoc interface for a five second duration. This emulates a connection loss and connection reestablishment.

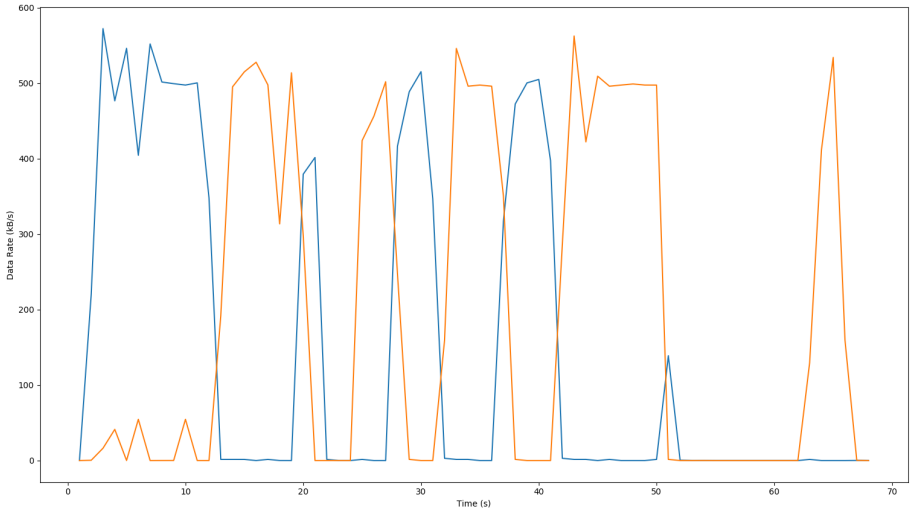


Figure 5.6: Modified MPTCP Path Switching

Switching between paths can be identified from the valleys of the subflows in figure 5.6. Whenever the ad hoc subflow is reactivated, the modified MPTCP scheduler schedules packets to the ad hoc subflow.

## 5.3 High latency on ad hoc, srtt difference between the minimum and the prioritized path is less than minimum

Blue Line: Prioritized Interface
Orange Line: Standard Interface

This evaluation was done to emulate an ad hoc subflow that have higher latency. There is a condition for this to be true. As explained in the implementation chapter, subflow is not always given priority. The difference between the ad hoc subflow srtt and minimum srtt should be less than the minimum srtt. Parameters of this evaluation are, 40 ms latency on non-prioritized subflow and 60 ms latency on prioritized subflow, 4 Mb/s maximum throughput network.

As illustrated in figure 5.7 even though prioritized subflow has a higher latency scheduler choose to send data through the prioritized subflow.

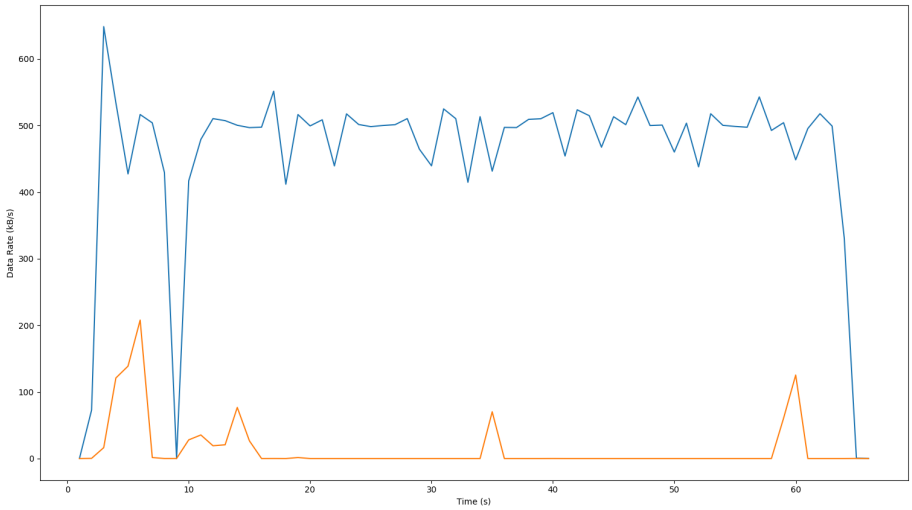


Figure 5.7: High Latency With Path Priority 01

## 5.4 High latency on ad hoc, srtt difference between the minimum and the prioritized path is greater than minimum

Blue Line: Prioritized Interface
Orange Line: Standard Interface

This scenario illustrate how scheduler behaves if users specify to prioritize a subflow and the latency difference between minimum srtt and the prioritized path is greater than minimum srrt. Parameters of this evaluation are, 40 ms latency on non-prioritized subflow and 200 ms latency on prioritized subflow, 4 Mb/s maximum throughput network.

As illustrated in figure 5.8 when the latency is high, the scheduler will fallback to the default behavior. In the default behavior, both paths were given the same priority. This is the intended functionality. This will include network characteristics to the scheduler decision making as well as users' input.

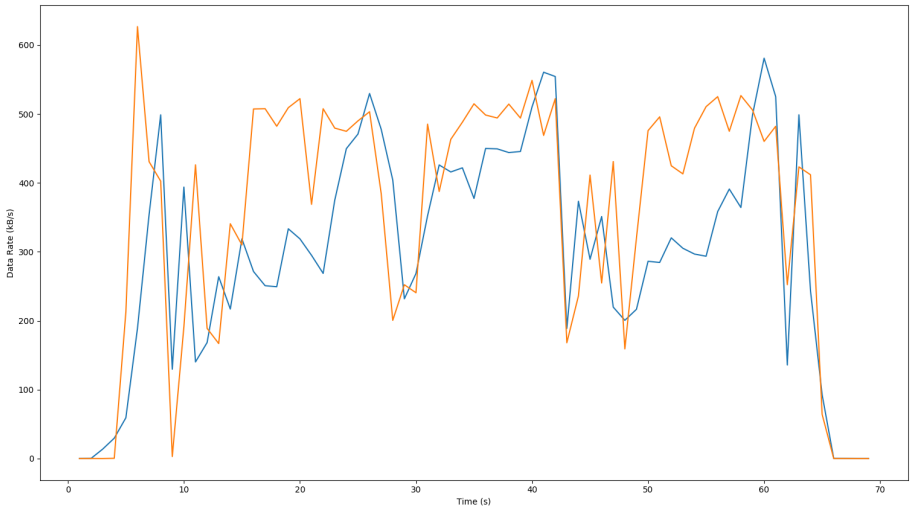


Figure 5.8: High Latency With Path Priority 02

## 5.5 Prioritized Subflow Recovery Speed Evaluation

Blue Line: Standard MPTCP Kernel
Orange Line: Modified MPTCP Kernel

This evaluation was done to evaluate the path recovery speed. This was done by dropping all the packets for 8 seconds and removing the restriction. This was compared against prioritized path recovery of modified MPTCP kernel and normal path recovery of standard MPTCP kernel. Parameters of this evaluation are 40 ms latency on both interfaces on both VMs, 4 Mb/s maximum throughput network.

As illustrated in figure 5.9, prioritized path recovery occurs 40 ms earlier. This is a significant improvement due to the fact that the latency of the interface is also 40 ms.

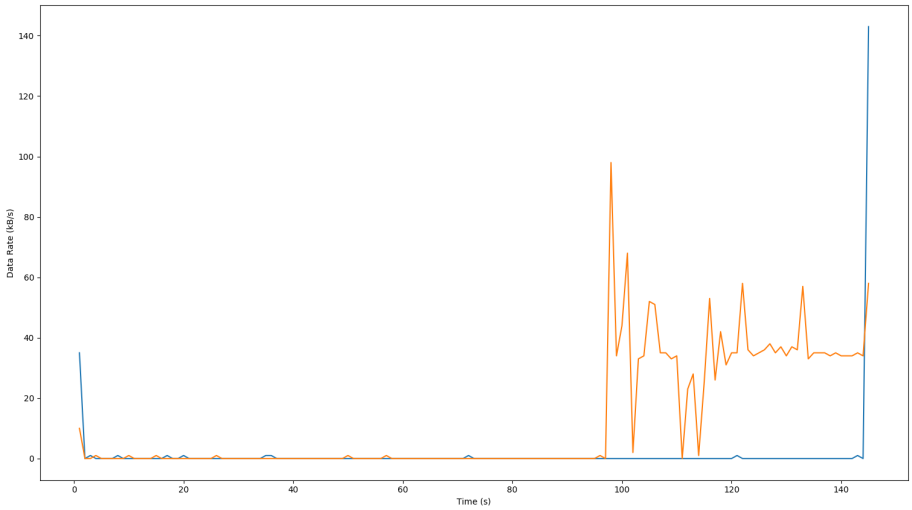


Figure 5.9: Path Recovery Speed Comparison

# Chapter 6

# Conclusions

Transmission Control Protocol is a protocol that have been in use to make a reliable connection between two hosts. One limitation of this protocol is that this protocol only utilizes one interface at a given time. But in this modern society most of the mobile devices are produced with more than one network interface. Under utilization of this multiple network interface was a problem until Multipath Transmission Control Protocols' birth (MPTCP). MPTCP is a cutting edge implementation that is even on this day, it is still in the development stage. This implementation utilizes the multiple interfaces in a device. MPTCP is also backward compatible with the standard TCP. Which means if one of the devices that is involved in communication does not understand MPTCP protocol, MPTCP implementation has the ability to fallback to standard TCP.

With the multiple path utilization arises a problem that the transport layer of OSI model did not have before. This is taking the routing decision on the transport layer. This issue should be addressed without disrupting the OSI model. MPTCP implementation does this by using the data that is visible to the transport layer. However while making this routing decisions we had to import data fom other OSI layers. Therefore MPTCP violates OSI model to some extent. As explained above, MPTCP needs to make decisions on where to route the next packet. Therefore a loadable module is implemented. Meaning in the run time scheduling policy can be changed. This creates an overhead to the packet transmission.

With multiple paths involved, MPTCP has to deal with congestion controls as well. Normal congestion controls are not sufficient enough to provide fair congestion controls. Therefore coupled congestion controls were invented. Then with multiple paths another problem occurs. This is how to restrict path use when the subflow is faulty and how to re-enable the path when it recovers from it. In this research, we had to struggle with all these problems to give the user the desired output, that is giving the user the ability to prioritize a subflow over others. While implementing this we dealt with the assumption, that the path needs to be prioritized is bound to a volatile ad-hoc network. This allowed us to implement additional methods to

identify a reactivated subflow with speed.

## 6.1 Conclusions about research questions

In this research four main research questions were answered. How can an ad-hoc subflow be prioritized? As shown in the evaluation chapter this was achieved by scheduling the data to route through the ad-hoc interface, if the ad-hoc interface is available. Second question answered in this research was how to pass a user hint to the kernel level. This was achieved by writing the value to the proc file. Implementation details on this is present in the implementation chapter. Third question answered in this research is how to recover a subflow quickly. This was achieved by introducing retransmission policy to subflow level. Evaluation chapter, results explained that there is a speed up when the new method is introduced. Fourth question of this research was performance of this implementation. This is thoroughly explained in the evaluation chapter.

## 6.2 Conclusions about research problem

As explained in the above subtopic, this research managed to find a solution for the problems that were defined at the beginning. Therefore this research holds a value to the general population. This implementation can be used to develop a cost effective reliable data transmission.

## 6.3 Limitations

As this was done in a virtual environment evaluation process need to be done in a real world environment to get an absolute idea about the performance. This research is currently limited to linux kernel, this can be extended to Android to get a better use.

## 6.4 Implications for further research

This research was done to prioritize ad-hoc subflows. Identifying if the subflow is bound to an ad-hoc interface is done by string comparison. This is computationally costly. This can be addressed by introducing a variable to the subflow levels and set it at the creation of the subflow to state whether it is an ad-hoc subflow would be faster. Also while a connection is established if the user changes the priority, priority change would not be visible to the established connection. This can be improved. Furthermore modularizing path recovery mechanism to change the policy at the run time would be a good improvement. Lastly implementation of priority levels is encouraged.

# References


[1] J. Hoebeke, I. Moerman, B. Dhoedt, and P. Demeester, "An overview of mobile ad hoc networks: Applications and challenges," *JOURNAL-COMMUNICATIONS NETWORK*, vol. 3, pp. 60–66, 07 2004.

[2] "Rfc 793 - transmission control protocol." `https://tools.ietf.org/html/rfc793`. (Accessed on 03/20/2019).

[3] L. Deek, E. Garcia-Villegas, E. Belding, S.-J. Lee, and K. Almeroth, "The impact of channel bonding on 802.11n network management," 12 2011.

[4] "Rfc 6824 - tcp extensions for multipath operation with multiple addresses." `https://tools.ietf.org/html/rfc6824`. (Accessed on 03/20/2019).

[5] "Number of mobile phone users worldwide 2015-2020 — statista." `https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/`. (Accessed on 03/20/2019).

[6] S. Deng, A. Sivaraman, and H. Balakrishnan, "All your network are belong to us: a transport framework for mobile network selection," 02 2014.

[7] C. Raiciu, D. Niculescu, M. Bagnulo, and M. J. Handley, "Opportunistic mobility with multipath tcp," 2011.

[8] N. Panday, "A comparative study of manet routing protocols," *STM Journal of Journal of Mobile Computing, Communications Mobile Networks*, vol. 1, p. 9, 05 2014.

[9] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and 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)*, (San Jose, CA), pp. 399–412, USENIX Association, Apr. 2012.

[10] M. Lima, N. Fonseca, and J. de Rezende, "On the performance of tcp loss recovery mechanisms," pp. 1812 – 1816 vol.3, 06 2003.

[11] M. Handley, C. Raiciu, A. Ford, J. Iyengar, and S. Barre, "Architectural Guidelines for Multipath TCP Development," Feb 2020. [Online; accessed 19. Feb. 2020].

[12] multipath tcp, "mptcp," Feb 2020. [Online; accessed 19. Feb. 2020].

[13] "GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)," Jan 2020. [Online; accessed 19. Feb. 2020].

[14] "VMware – Cloud, Mobility, Networking & Security Solutions," Feb 2020. [Online; accessed 19. Feb. 2020].

[15] V. Gueant, "iPerf - The TCP, UDP and SCTP network bandwidth measurement tool," Feb 2020. [Online; accessed 19. Feb. 2020].

[16] "ifstat(8) - Linux manual page," Feb 2020. [Online; accessed 19. Feb. 2020].

[17] "MultiPath TCP - Linux Kernel implementation : Main - Home Page browse," Feb 2020. [Online; accessed 19. Feb. 2020].

[18] "tc(8) - Linux manual page," Feb 2020. [Online; accessed 19. Feb. 2020].

[19] "ifpps(8) - Linux manual page," Feb 2020. [Online; accessed 19. Feb. 2020].

[20] "proc(5) - Linux manual page," Feb 2020. [Online; accessed 19. Feb. 2020].

# Appendices

# Appendix A

# Code Modification

This chapter is appended to give information about the code implementation and modification in MPTCP kernel. This chapter is closely related with the implementation chapter. All the alterations done to the kernel source code are listed below.

## A.1 MPTCP Control Block

le containing mptcp_cb structure resides on the /include/net/mptcp.h of MPTCP kernel. Three new variables were introduced to the mptcp_cb structure. Namely adhoc_priority, Is_adhoc_avail, adhoc_tries. These variables are used to store the information if ad-hoc should be given priority, is there an ad-hoc bounded interface, and did at least one packet was sent through the connection initiation subflow.

```
struct mptcp_cb {
        /* list of sockets in this multipath connection */
        struct hlist_head conn_list;
        /* list of sockets that need a call to release_cb */
        struct hlist_head callback_list;

        /* Lock used for protecting the different rcu-lists of mptcp_cb */
        spinlock_t mpcb_list_lock;

        /* kshithija */
        int        adhoc_priority;
        int        is_adhoc_avail;
        int        adhoc_tries;

        /* High-order bits of 64-bit sequence numbers */
        u32 snd_high_order[2];
        u32 rcv_high_order[2];
```

Figure A.1: MPTCP Control Block Code

## A.2 Proc Table

This code segment is located at the /net/mptcp/mptcp_ctrl.c. This is the structure that stores the information about proc that was created.

```
201         {
202                 .procname       = "adhoc_interface",
203                 .mode           = 0644,
204                 .maxlen         = MPTCP_INTERFACE_NAME_MAX,
205                 .proc_handler   = proc_mptcp_adhoc,
206         },
```

Figure A.2: Proc Table Code

In the line number 202 name of the proc was defined. In the line number 203 permission of the proc is defined. In the line number 204 maximum length that allowed to write to the proc is defined. In line number 205 what function should trigger when a operation to the proc happens defined.

## A.3 User Hint information

This code segment is located at the /net/mptcp/mptcp_ctrl.c. This code segment is used to store the user hint.

```
83 char face_addr[16] = "0.0.0.0";
84 int adhoc_avail = 0;
```

Figure A.3: User Hint information

## A.4 Proc Handler

This code segment is located at the /net/mptcp/mptcp_ctrl.c. This is the function that gets triggered when an operation to the proc happens. Functionality of this segment is explained in the implementation chapter.

```
124 static int proc_mptcp_adhoc(struct ctl_table *ctl, int write,
125                                 void __user *buffer, size_t *lenp,
126                                 loff_t *ppos)
127 {
128         char val[MPTCP_SCHED_NAME_MAX];
129         struct ctl_table tbl = {
130                 .data = val,
131                 .maxlen = 16,
132         };
133         int ret;
134
135         strncpy(val, face_addr, 16);
136         ret = proc_dostring(&tbl, write, buffer, lenp, ppos);
137
138         if (write) {
139                 strncpy(face_addr, val, 16);
140                 if(strcmp(face_addr, "0.0.0.0") == 0){
141                         adhoc_avail = 0;
142                 }
143                 else{
144                         adhoc_avail = 1;
145                 }
146                 ret = 0;
147                    }
148         return ret;
149 }
```

Figure A.4: Proc Handler Code

## A.5 Initial Value Assignment

This code segment is located at the /net/mptcp/mptcp_ctrl.c. This code segment is defined inside mptcp_add_sock function. This is where the initial values to the mptcp_cb structure are assigned.

```
1512         mpcb->is_adhoc_avail = adhoc_avail;
1513         mpcb->adhoc_tries = 0;
1514         mpcb->adhoc_priority = 0;
```

Figure A.5: Initial Value Assignment Code

## A.6 Should Prioritize the Path or not Decision

This code segment is located at the /net/mptcp/mptcp_sched.c. This code segment is the decision maker that determines if the scheduler should jump to state A or state B as explained in the implementation.

```
150         if(mpcb->is_adhoc_avail == 1 && adhoc_priority == 0 && mpcb->adhoc_tries == 1){
```

Figure A.6: Should Prioritize the Path or not Decision Code

## A.7 State A

This code segment is located at the /net/mptcp/mptcp_sched.c. This is the code segment of state A as explained in the implementation.

```
196         else{
197                 mptcp_for_each_sub(mpcb, mptcp) {
198                         struct sock *sk = mptcp_to_sock(mptcp);
199                         struct tcp_sock *tp = tcp_sk(sk);
200                         struct inet_connection_sock *icsk = inet_csk(sk);
201                         bool unused = false;
202
203                         /* First, we choose only the wanted sks */
204                         if (!(*selector)(tp))
205                                 continue;
206
207                         if (!mptcp_dont_reinject_skb(tp, skb))
208                                 unused = true;
209
210                         if (mptcp_is_def_unavailable(sk))
211                                 continue;
212
213                         if (mptcp_is_temp_unavailable(sk, skb, zero_wnd_test)) {
214                                 if (unused)
215                                         found_unused_una = true;
216                                 continue;
217                         }
218
219                         if (unused) {
220                                 if (!found_unused) {
221                                         /* It's the first time we encounter an unused
222                                          * sk - thus we reset the bestsk (which might
223                                          * have been set to a used sk).
224                                          */
225                                         min_srtt = 0xffffffff;
226                                         bestsk = NULL;
227                                 }
228                                 found_unused = true;
229                         }
230
231                         interface_type = is_adhoc(tp);
232
233                         if(interface_type == 0){
234                                 adhoc_srtt = tp->srtt_us;
235                                 lenear_retransmits = icsk->lenear_retransmits;
236                                 icsk->lenear_retransmits = 0;
237                         }
238
239                         if (tp->srtt_us < min_srtt) {
240                                 min_srtt = tp->srtt_us;
241                                 bestsk = sk;
242                         }
243                 }
244
245                 mptcp_debug("%s kshithija normal return %u %u %u \n", __func__, min_srtt, adhoc_srtt, (adhoc_srtt - min_srtt));
246                 if(adhoc_srtt != 0xffffffff && ((adhoc_srtt - min_srtt) < min_srtt)){
247                         mpcb->adhoc_priority = 0;
248                 }
249
250         }
```

Figure A.7: State A Code

## A.8 State B

This code segment is located at the /net/mptcp/mptcp_sched.c. This is the code segment of state B as explained in the implementation.

```
151             mptcp_for_each_sub(mpcb, mptcp) {
152                     struct sock *sk = mptcp_to_sock(mptcp);
153                     struct tcp_sock *tp = tcp_sk(sk);
154                     bool unused = false;
155
156                     /* First, we choose only the wanted sks */
157                     if (!(*selector)(tp))
158                             continue;
159
160                     if (!mptcp_dont_reinject_skb(tp, skb))
161                             unused = true;
162
163                     if (mptcp_is_def_unavailable(sk))
164                             continue;
165
166                     if (mptcp_is_temp_unavailable(sk, skb, zero_wnd_test)) {
167                             if (unused)
168                                     found_unused_una = true;
169                             continue;
170                     }
171
172                     interface_type = is_adhoc(tp);
173
174                     if(interface_type == 0){
175
176                             if (unused) {
177                                     if (!found_unused) {
178                                             bestsk = sk;
179                                             adhoc_srtt = tp->srtt_us;
180                                     }
181                                     found_unused = true;
182                             }
183                     }
184
185                     if (tp->srtt_us < min_srtt) {
186                             min_srtt = tp->srtt_us;
187                     }
188             }
189             mptcp_debug("%s kshithija %u %u %u \n", __func__, min_srtt, adhoc_srtt, (adhoc_srtt - min_srtt));
190             if(bestsk && adhoc_srtt > min_srtt && ((adhoc_srtt - min_srtt) > min_srtt)){
191                     mpcb->adhoc_priority = 1;
192             }
```

Figure A.8: State B Code

## A.9 Socket Address Comparison

This code segment is located at the /include/net/my.h. This is the code segment of that compares socket address with the user hint address.

```
1 extern char face_addr[16];
2 //extern int set_my_rto = 0;
3
4 static inline int is_adhoc(struct tcp_sock *tp)
5 {
6         char sip[16];
7         snprintf(sip, 16, "%pI4", &((struct inet_sock *)tp)->inet_saddr);
8         return strcmp(sip, face_addr);
9 }
```

Figure A.9: Socket Address Comparison Code

In this code segment line 01 makes face_addr shared with the my.h. In the function is_adhoc, it takes TCP socket as the argument. In line 07 TCP sockets source address is converted into a string. In line 08 string comparison is done and returns the value.

## A.10 Retransmission Speed up Code

This code segment is located at the /net/ipv4/tcp_timer.c. This code segment is integrated with tcp_retransmit_timer. This is the code segment that implements the steady RTO.

```
586             if(is_adhoc(tp) == 0 && icsk->lenear_retransmits == 1  && icsk->lenear_retransmits < 10){
587                     icsk->icsk_rto = min(icsk->icsk_rto, TCP_RTO_MAX);
588             }
```

Figure A.10: Retransmission Speed up Code