Investigating the Effect of Different Subflow MTUs on MPTCP Throughput

C.S. Manoratne

2025

Investigating the Effect of Different Subflow MTUs on MPTCP Throughput

Shamalka Manoratne

Index No: 20001118

Supervisor: Dr. C.I. Keppitiyagama

May 2025

Submitted in partial fulfillment of the requirements of the B.Sc. (Honours) in Computer Science Final Year Project

UCSC

Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, to be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name: Shamalka Manoratne

[illegible]

Signature of Candidate Date: April 25, 2025

This is to certify that this dissertation is based on the work of

Mr. Shamalka Manoratne under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Principal/Co-Supervisor's Name: Dr. C.I. Keppitiyagama

[illegible]

2025-06-13

Signature of Supervisor Date:

Table of Contents

Declaration **i**

1 Abstract **1**

2 Introduction **2**

3 Motivation **5**

4 Aims and Objectives **7**

4.1 Aim 7

4.2 Objectives 7

5 Literature Review **9**

6 Research Questions **11**

7 Significance of the Research **12**

8 Planned Research Approach **14**

9 Scope **15**

9.1 In Scope 15

9.2 Out Scope 16

10 Experiments and Findings **17**

10.1 Experimental Setup 17

10.2 Topology Design 18

10.3 Traffic Generation and Data Collection 19

10.4 Establishing a TCP Connection on Mininet 20

10.5 Establishing an MPTCP Connection on Mininet 26

10.5.1 Analyze the captured packets 29

10.6 Investigating PMTU Discovery Issues in MPTCP Subflows 46
10.7 Evaluating MPTCP Performance Under TCP_BASE_MSS Conditions . 53
10.8 Findings 55

11 Challenges and Limitations of Existing Linux Architecture 57
11.1 Path MTU Discovery For MPTCP 57
11.2 Limitation of MTU Probing 59
11.3 Byte-Based Send Buffer Complexity at the Connection Level in MPTCP 61
11.4 Complexity in Retransmissions 63

12 Conclusion 65

13 Future Work 67

References 67

1. Abstract

Multipath TCP (MPTCP) is an extension of the traditional Transmission Control Protocol (TCP) that enables simultaneous use of multiple network paths between two endpoints, offering improved resilience, throughput, and resource utilization. While existing MPTCP implementations perform well in homogeneous environments, their behavior in heterogeneous networks, particularly when subflows have different Path Maximum Transmission Units (MTUs) remains underexplored. This research investigates the impact of varying subflow MTUs on MPTCP throughput using a controlled emulation environment. Experimental analysis reveals that the default Linux MPTCP implementation fails to utilize subflows with smaller MTUs effectively, primarily due to limitations in the Path MTU Discovery (PMTUD) mechanism and a unified Maximum Segment Size (MSS) approach. Kernel-level modifications were introduced to enable MTU probing on a per-subflow basis, allowing dynamic MSS adjustment and improved subflow utilization. Results demonstrate a notable improvement in throughput and path diversity post-modification, highlighting the importance of MTU-aware scheduling and adaptive probing techniques. The findings suggest potential directions for enhancing MPTCP performance in heterogeneous environments and contribute to the ongoing development of more robust multipath communication protocols.

2. Introduction

In the past, end hosts were usually only connected to the Internet through one interface. Each interface was given a unique IP address that all host interactions went through. However, the modern Internet doesn't work like this model used to. Devices like computers and tablets now have more than one interface and both wired and wireless interfaces, making Ethernet and Wi-Fi connections possible. Similarly, smartphones now have more than one interface. Even in specialized places like data centers, where many servers talk to each other for tasks like data replication and distributed computing, a fast network is necessary. This is why redundant infrastructure is used to spread traffic across multiple paths and make the network more resilient in case a link or node fails.

The Internet Engineering Task Force (IETF) standardized the Multipath Transmission Control Protocol (MPTCP). It gives the application layer a single TCP (Transmission Control Protocol) (Eddy 2022) connection and allows two hosts to use several Internet paths efficiently (Ford et al. 2020). The idea behind Multipath TCP, or MPTCP, was to solve these problems. The goals that went into making MPTCP are these:

- It should work with current programs just like regular TCP.
- It should be able to use more than one network path for a single connection.
- Normal TCP shouldn't have to struggle to find enough network paths, so it should be able to use them at least as well

Because of Multipath TCP, a host can support a single TCP connection through more than one interface and address. By combining all the available resources, Multipath TCP can improve the service that the apps receive in terms of speed and reliability. To find losses and see if each path is working right or not, the

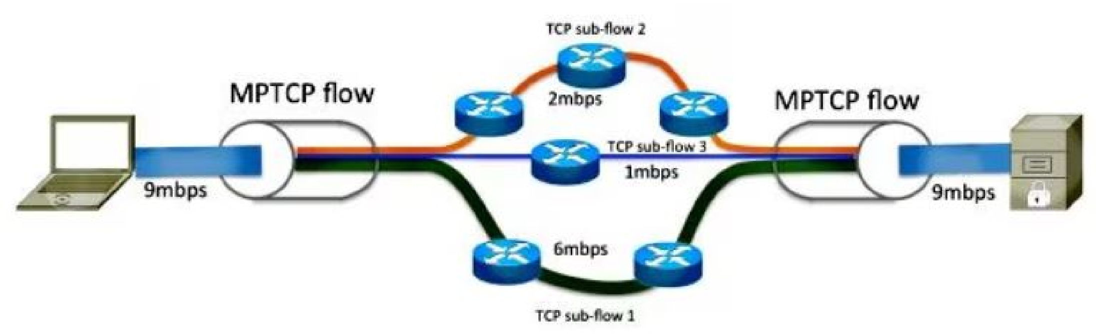


Figure 2.1: Multiple Path for Communication

sender has to quickly and accurately measure the Round-Trip Time (RTT) numbers for each path. It takes milliseconds to measure the RTT, which is the time it takes to send a data packet and receive a signal acknowledging that it was sent (Paasch et al. 2014).

The efficiency of MPTCP is dependent upon the packet scheduler. A scheduler allocates the packets to the available pathways. Incorrect scheduling decisions can decrease MPTCP performance in both homogeneous and heterogeneous networks, causing reduced throughput and longer download times. The presence of heterogenous paths results in a higher number of packets being delivered out of order, which subsequently leads to the problem of Head of Line (HOL) blockage, caused by limitations in the receiver's window. An improved packet scheduler will utilize all available paths to minimize the occurrence of out-of-order packets, hence enhancing throughput and performance

The Maximum Transmission Unit (MTU) is the largest packet size that can be sent over a network path without needing fragmentation. When a host wants to send data across an interface, it consults the MTU of the interface to determine the maximum amount of data that can be included in each packet. Ethernet ports typically feature a default MTU of 1500 bytes, excluding the Ethernet header or trailer. as well as most datacenter networking hardware, can support jumbo frames which is 9000 bytes (Julaihi 2011). In practice, when a host wants to transmit a TCP data stream, it would usually allocate the initial 20 bytes out

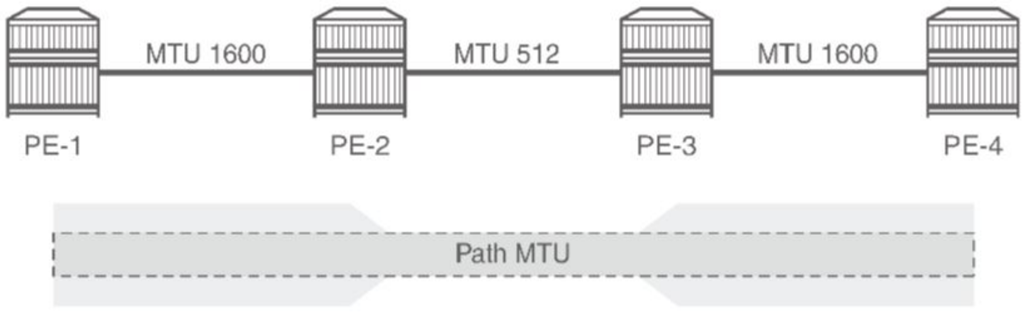


Figure 2.2: Path MTU

of the total 1500 bytes for the IP header, the subsequent 20 bytes for the TCP header, and utilize the remaining 1460 bytes for the data payload as required. By encapsulating data in packets of maximum size, it minimizes the consumption of bandwidth caused by protocol overhead.

Path MTU (Figure 2.2) discovery is the method used to determine the smallest possible MTU of the link that a packet may traverse (Mogul & Deering 1990). To optimize the use of a path, hosts need to determine the path MTU, which corresponds to the MTU of the link between the sender and receiver. In a normal TCP connection, if a packet encounters an intermediate link with a lower MTU, the router handling that link will drop the packet and notify the sender to adjust its segment size. For example, if two hosts communicate over a path where an intermediate link has an MTU of 512 bytes while the host supports 1600 bytes, both hosts must adopt the smaller MTU (512 bytes) to prevent fragmentation (Custura et al. 2018).

3. Motivation

Implementing Multipath TCP (MPTCP) with heterogeneous subflows presents significant challenges. Achieving optimal performance with MPTCP necessitates near-homogeneous network conditions across all subflows (Adarsh et al. 2019). To maximize the potential of MPTCP, it needs to adopt a comprehensive view of each path performance, which should include, at a minimum, considerations of path Maximum Transmission Unit (MTU).

Unfortunately, not all links which compose the Internet have the same MTU. Different paths may have different MTUs due to variations in the underlying physical media type or configured encapsulation (Asiri 2021). If the packet size exceeds the MTU of link or interface, then it must be fragmented into smaller pieces to transmit it as two (or more) individual pieces, each within the link. Fragmentation is a costly process since it requires the use of hardware resources and extra bandwidth (Feng et al. 2022). This is because new headers must be created and attached to each fragment.

In MPTCP implementation, the MTU of each subflow is not directly considered when scheduling packets across subflows. This consideration is unnecessary in homogeneous environments where all subflows have the same MTU, as they can use that consistent value directly (Asiri 2021). By considering the path MTU sizes of each subflow (Figure 3.1), when making scheduling decisions, it may be possible to improve the efficiency of packet scheduling across subflows, thereby increasing the overall performance and throughput of MPTCP in heterogeneous network environments.

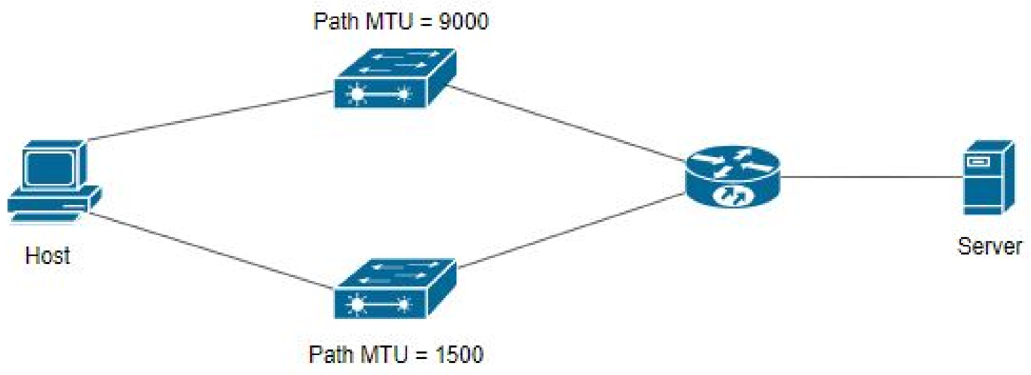


Figure 3.1: Two subflows having different path MTU

4. Aims and Objectives

4.1 Aim

The aim of this research is to investigate how Multipath TCP (MPTCP) behaves across subflows with varying Path MTU values, analyze the resulting performance implications, particularly in terms of throughput and explore enhancements to overcome limitations in the current implementation. The ultimate objective is to ensure seamless data delivery, maintain uninterrupted communication, and achieve efficient multipath communication.

4.2 Objectives

- Emulate MPTCP Behaviour: Set up MPTCP with multiple subflows, each having varying path MTU values, using current implementations to analyze its behaviour under these conditions.

- Analyze Path MTU Variations: Measure throughput to examine how different path MTU values across subflows impact overall MPTCP performance.

- Throughput Evaluation: Assess the throughput performance of MPTCP when operating over subflows with different path MTU values, comparing it against standard TCP to highlight differences.

- Identify Performance Bottlenecks: Analyze and identify any performance bottlenecks or inefficiencies in MPTCP when managing subflows with differing path MTUs.

- Optimization Techniques: If bottlenecks or inefficiencies are identified, explore potential optimization techniques to enhance MPTCP performance with varying path MTU values.

- Evaluate Enhancements: Measure and compare the performance of MPTCP before and after the implementation of the proposed optimization techniques.
- Assess Implementation Limitations: Evaluate the challenges and limitations of the current MPTCP implementation in achieving the desired outcomes and performance improvements.

5. Literature Review

Different path MTUs in subflows mean that data should be sent in different sizes on each path, depending on the specific subflow. Consequently, each path must process a different Maximum Segment Size (MSS) based on its MTU, excluding headers such as TCP and IP. This results in the scheduler needing to handle segments of different sizes. If the MSS is the same for each segment, then the network layer must decide on fragmentation to send data through subflows with both large and small path MTUs. The lager datagram received from the transport layer cannot be sent via small path MTU subflow, there is an additional fragmentation process that needs to be done and resulting that fragmentation overhead and might affect performance as well. Efficient scheduling has the ability to decrease head-of-line blocking and latency, especially in heterogeneous environments, allowing streaming applications to minimize the buffer requirement. However, it remains unclear how different path MTU subflows should take into account for scheduling packets across them. Furthermore, there is no analysis of throughput in such an environment. Current techniques do not emphasize the involvement of managing subflows with different path MTUs and the selection of appropriate MSS values for sending data in each subflow. Examples of existing packet scheduling techniques are Min-RTT, FPS and DPSAF. Min-RTT is the default schedular in current MPTCP implementation on Linux which selects the path with the lowest RTT. FPS predicts scheduling values without considering packet loss or bandwidth. DPSAF adjusts scheduling based on packet loss rates and TCP SACK feedback but ignores bandwidth. It uses a more complex analyzing model that involves a significant level of computational complexity (Maxwell 2023). The default path manager and scheduler focus more on balancing load and optimizing latency rather than MTU-specific optimizations. Unlike RTT and bandwidth, the integration of MTU considerations has not been thoroughly explored in literature.

By addressing this gap, it may be possible to achieve more efficient, reliable, and high-performance data transmission in heterogeneous situations of the network. Studies could focus on,

Fragmentation Overhead: When packets are larger than the path MTU of a subflow, they must be fragmented. This adds additional processing overhead at both the sender and receiver.

Suboptimal Resource Utilization: Using a fixed path MTU may cause some paths to be underutilized if their actual MTU is higher than the used one, leading to inefficiencies in bandwidth usage.

6. Research Questions

This research aims to understand the performance and behavior of Multipath TCP (MPTCP) in heterogeneous network environments where subflows exhibit different Path Maximum Transmission Units (MTUs). The following research questions guide the investigation:

1. How does Multipath TCP (MPTCP) adjust to subflows with different Path MTU (PMTU) values and improve packet scheduling to enhance network performance compared to standard TCP?

2. What impact does fragmentation overhead have on subflow packet scheduling in the context of different Path MTU values?

7. Significance of the Research

In today's increasingly heterogeneous network environments where devices often have simultaneous access to Wi-Fi, Ethernet, cellular, or other networks—MPTCP offers the potential for improved resilience, bandwidth aggregation, and seamless failover. However, for MPTCP to realize these benefits effectively, it must be able to adapt to the diverse characteristics of each subflow, including differences in Path MTU.

The significance of this research lies in its focus on a critical yet underexplored aspect of MPTCP: how it manages subflows with different MTU values. Standard implementations typically assume uniform path characteristics, which can lead to inefficiencies when subflows traverse links with varying MTUs. This mismatch may result in underutilization of available paths, increased fragmentation, packet loss, or even complete subflow neglect—factors that degrade overall throughput and reliability.

By analyzing and modifying the behavior of MPTCP with respect to per-subflow MTU awareness and adaptive probing, this research contributes valuable insights into improving path utilization. From a systems and networking perspective, the findings provide practical implementation guidance for improving the Linux kernel's MPTCP stack. By enabling MTU probing at the socket level and dynamically adjusting the MSS, this work not only addresses existing kernel limitations but also introduces a path for future enhancements.

In real-world applications such as mobile devices, cloud infrastructure, and edge computing, where networks are inherently heterogeneous, the ability to fully utilize all available paths despite their MTU differences is crucial. Enhanced MPTCP behavior in such scenarios can lead to better user experiences (e.g., faster file transfers, smoother video streams), improved load balancing, and greater energy efficiency in mobile and embedded systems.

Moreover, this research opens new avenues for further study, including the design of MTU-aware congestion control and cross-layer optimization strategies. As next-generation networks increasingly rely on multi-interface communication (e.g., 5G, IoT, and vehicular networks), robust and adaptive multipath transport protocols will become even more essential.

In summary, the research addresses a practical performance gap in current MPTCP implementations, delivers kernel-level solutions for improved subflow utilization, and provides a foundation for more adaptive and efficient transport-layer behavior in diverse networking environments.

8. Planned Research Approach

This exploratory research investigates the performance of Multi-Path TCP when subflows have different path MTUs. To achieve this, the research will follow these steps:

- Setup Testing Environment: Establish a controlled testing environment to emulate network conditions with different path MTU values across the different subflows.
- Run Tests: Execute the test scenarios using MPTCP to observe behavior under different path MTU configurations.
- Analyze Results: Collect and analyze the data from the tests to identify existing MPTCP scheduling techniques to understand their approach to scheduling packets among subflows with different path MTUs.
- Correlate Path MTU Impact: Determine the correlation between subflows with different path MTU values and overall throughput, assessing how variations affect MPTCP performance.
- Performance Evaluation: Compare the performance of MPTCP under subflows with different path MTU conditions with that of standard TCP to evaluate efficiency and reliability.

9. Scope

9.1 In Scope

- Analyze the existing MPTCP scheduling algorithms: Assess the current MPTCP implementations and their ability to handle subflows with different path MTUs.
- Comparative Analysis: This study aims to evaluate and contrast the efficiency of Multipath TCP with normal TCP in comparable circumstances. It will specifically highlight situations in which MPTCP demonstrates notable benefits. In addition, the study will evaluate the resilience and flexibility of MPTCP in response to changes in different network conditions.
- Emulation and Lab Testing: Conduct emulations to model various path MTU configurations, followed by real-world tests to validate these emulation results and gather empirical data. Diverse testing environments, including lab setups and live network conditions, will be utilized to ensure a comprehensive evaluation of MPTCP's performance over subflows have different path MTU.

9.2 Out Scope

- Non-MTU Related Scheduling factors: This research will not thoroughly explore other aspects that affect scheduling, such as the allocation of bandwidth, path reliability, or cost considerations, unless they are directly related to MTU challenges.
- Detailed Congestion Control Mechanisms: Although congestion control is a critical aspect of MPTCP performance, this research will not focus on developing new congestion control algorithms. Existing mechanisms will be used as a basis for evaluating MTU-aware scheduling.
- Security and Encryption Concerns: While security is crucial for any network protocol, this research will not address specific security or encryption issues related to MPTCP, unless they directly impact MTU handling.
- Non-TCP Multipath Protocols: The study will be limited to MPTCP and will not cover other multipath protocols such as Multipath QUIC or SCTP.
- Application on All Platforms: The main focus will be on how MPTCP works on Linux. This study does not cover changes that need to be made for other operating systems or for custom hardware solutions.

10. Experiments and Findings

This section presents a comprehensive account of the experiments conducted to investigate how varying path MTU settings influence the performance of Multipath TCP (MPTCP). It encompasses all aspects of the study from network topology design and experimental setup to kernel-level observations and performance measurements. Particular attention is given to the behaviour of MSS negotiation, the functioning and limitations of Path MTU Discovery (PMTUD), and fallback mechanisms triggered under constrained conditions. Together, these findings offer critical insights into the impact of heterogeneous MTU paths on MPTCP throughput and subflow utilization, as well as the effectiveness of current kernel implementations in adapting to such environments.

10.1 Experimental Setup

To systematically evaluate the behaviour of MPTCP under varying path MTU conditions, we designed and implemented a controlled test environment using the following tools and configurations. Each component was carefully selected and configured to ensure reproducibility, observability, and the ability to manipulate network parameters such as MTU values, subflow paths, and throughput measurements. The setup was intended to closely replicate real-world network scenarios while providing sufficient flexibility for in-depth analysis of kernel-level behaviors and protocol interactions.

- **Linux Kernel 5.4.230 with MPTCP Support:** This kernel version includes MPTCP functionality, which will be used to establish multi-path connections between hosts. We will utilize MPTCP-specific features to control subflow behaviour and measure the impact of different MTU settings on throughput.

- **Mininet:** A network emulator that allows us to create complex network topologies on a single Linux machine (Project 2025). This tool will simulate a variety of network configurations with multiple subflows, enabling us to introduce different MTU settings across various links.

- **Iperf3:** A widely used network performance measurement tool that will generate TCP traffic over MPTCP connections (ESNet & Laboratory 2023). It will be used to record key performance metrics such as throughput, jitter, and packet loss.

- **Wireshark and tcpdump:** Packet capture tools that will monitor network traffic in real time. These tools will capture detailed information on packet transmission, retransmissions, ICMP messages, and fragmentation behaviour, allowing us to analyze how MTU variations impact network performance.

10.2 Topology Design

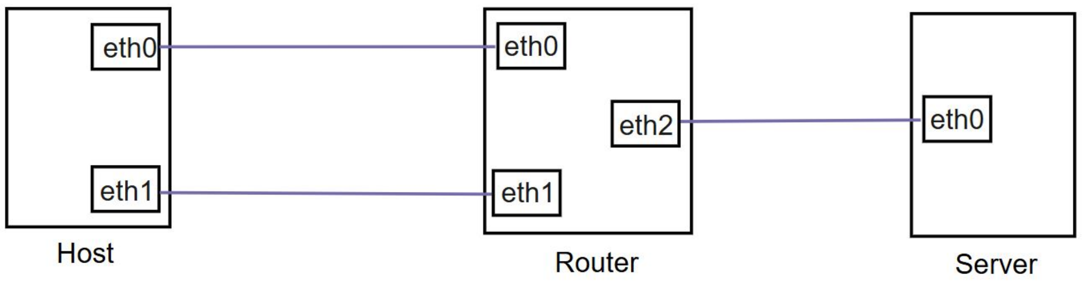


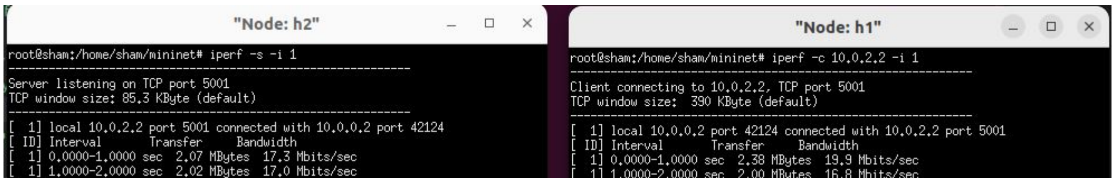
Figure 10.1: Network topology with two hosts connected with multiple paths.

The experimental topology consists of two end-hosts a client (Host) and a server (Server) interconnected through a central Router, with multiple links forming distinct paths between them, as illustrated in Figure 10.1. Each link corresponds to a separate subnet, enabling the use of MPTCP subflows over independent paths with configurable MTU values. A router is intentionally used instead of a switch to prevent network loops. Since switches operate at Layer 2 of the OSI model, having multiple active links between the same devices could cause broadcast loops unless the Spanning Tree Protocol (STP) is used or links are manually disabled. Such loops can lead to serious network disruptions. In contrast, routers function at Layer 3 and inherently avoid loops by routing packets based on IP addresses, which ensures stable operation even with multiple active links. In this setup, the Host is configured with two network interfaces to establish separate subflows, although the same approach can be applied to the Server side if needed. This design was chosen for its simplicity while still allowing us to examine the behavior of MPTCP across paths with different MTU configurations. The Mininet environment is configured accordingly, with MPTCP enabled in the Linux kernel, appropriate `sysctl` parameters adjusted as necessary, and TCP MTU probing activated to support dynamic Path MTU Discovery (PMTUD) during transmission.

10.3 Traffic Generation and Data Collection

To evaluate the impact of varying MTU values on MPTCP performance, controlled traffic flows were generated between the hosts, and comprehensive data collection mechanisms were employed throughout the experiments.

The traffic generation process begins with configuring the receiver host (Server-h2) to operate in server mode using the `iperf3` tool as show in Figure 10.2.



```
root@sham:/home/sham/mininet# iperf -s -i 1
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.2.2 port 5001 connected with 10.0.0.2 port 42124
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  2.07 MBytes  17.3 Mbits/sec
[  1] 1.0000-2.0000 sec  2.02 MBytes  17.0 Mbits/sec
```

```
root@sham:/home/sham/mininet# iperf -c 10.0.2.2 -i 1
------------------------------------------------------------
Client connecting to 10.0.2.2, TCP port 5001
TCP window size:  390 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.2 port 42124 connected with 10.0.2.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  2.38 MBytes  19.9 Mbits/sec
[  1] 1.0000-2.0000 sec  2.00 MBytes  16.8 Mbits/sec
```

Figure 10.2: Iperf Client and Server

This server passively waits for incoming TCP connections and continuously measures performance metrics such as data rate, jitter, and loss during the test session. On the sender side (Host-h1), `iperf3` is executed in client mode, initiating a TCP connection to the server (h2) and transmitting traffic over multiple subflows established by MPTCP. These subflows are distributed across different paths that have distinct MTU settings, allowing us to observe how MPTCP schedules traffic when faced with path diversity in terms of maximum transmission size. Throughput measurements are configured to be recorded at one-second intervals. This frequency provides high-resolution temporal data, enabling us to analyze performance fluctuations over time and detect any momentary degradation or adaptation in response to path-specific MTU constraints.

Simultaneously, packet captures are performed using `tcpdump` on all involved network nodes, including both end-hosts and the intermediate router. This ensures a complete view of all traffic traversing the network, which is essential for identifying the behavior of each individual subflow. These packet traces are later analyzed using Wireshark, a powerful packet inspection tool, to extract detailed information such as retransmissions, TCP segment sizes, and ICMP messages related to fragmentation or MTU.

10.4 Establishing a TCP Connection on Mininet

Before proceeding to experiments involving Multipath TCP (MPTCP), it was essential to first establish and validate the behavior of a standard single-path TCP connection under varying MTU conditions. This preliminary step was crucial to ensure that the underlying path MTU discovery mechanism was functioning correctly within the Linux kernel and the Mininet environment. By analyzing how regular TCP handles path MTU information and fragmentation, we can later compare and contrast it with the more complex behavior of MPTCP.

To conduct this initial test, a basic TCP connection was established between two hosts (Host and Server), interconnected via a single router, as illustrated in Figure 10.3. This minimal topology allowed for a controlled environment where packets could be observed traversing a single path with an explicitly configured MTU. The setup enabled detailed inspection of PMTUD behavior and kernel responses such as ICMP ”Fragmentation Needed” messages, or any unexpected packet drops.

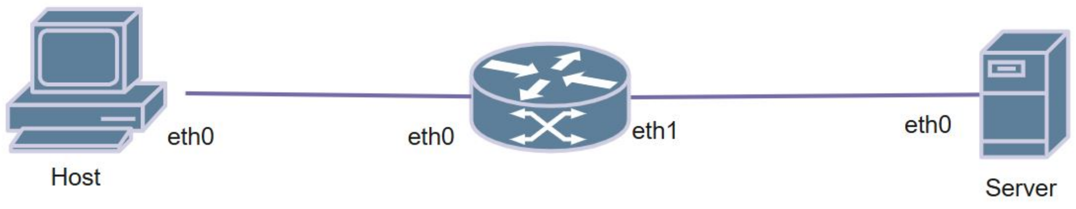


Figure 10.3: TCP Connection

This foundational experiment served as a baseline for understanding how TCP reacts when MTU values are restricted on intermediate links. It also provided insights into how the Linux kernel adapts the maximum segment size (MSS), whether fragmentation is avoided as expected, and how effectively TCP reacts to ICMP notifications.

Only after verifying the correct behavior of standard TCP under these conditions was MPTCP enabled and multi-subflow scenarios introduced. This step-by-step approach ensured the accuracy of the testbed and provided a solid reference point for interpreting the results of subsequent MPTCP experiments.

The Mininet script was created to configure the topology with varying MTU values for the interfaces on links between Host-Router and Router-Server shown in Figure 10.4.

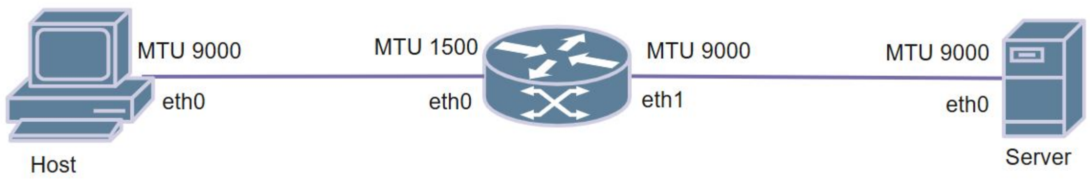
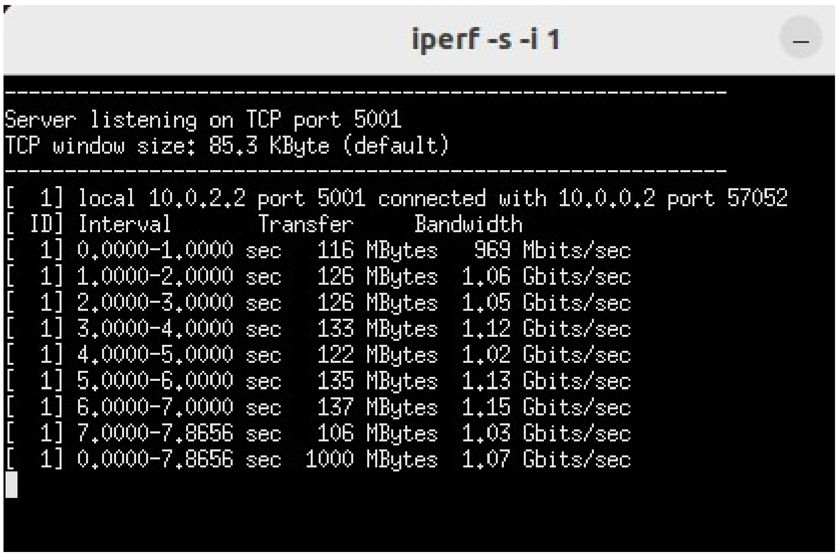


Figure 10.4: MTU Configurations on Interfaces

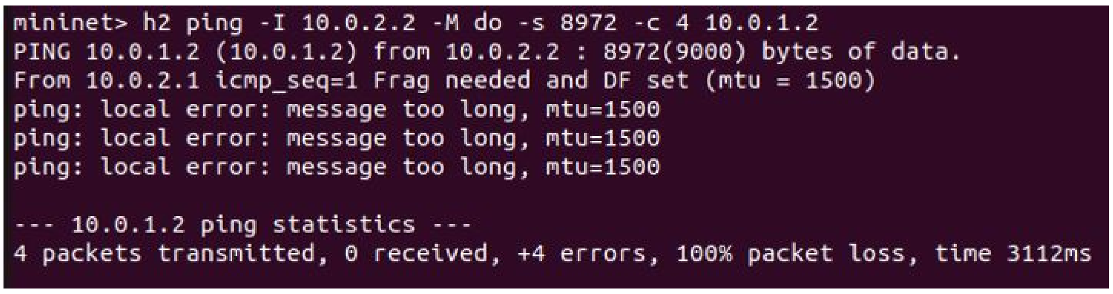
This enabled us to analyze how TCP and the Linux kernel respond when a smaller MTU is encountered along the path. Traffic was generated using iperf and ping.



```
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.2.2 port 5001 connected with 10.0.0.2 port 57052
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec   116 MBytes   969 Mbits/sec
[  1] 1.0000-2.0000 sec   126 MBytes  1.06 Gbits/sec
[  1] 2.0000-3.0000 sec   126 MBytes  1.05 Gbits/sec
[  1] 3.0000-4.0000 sec   133 MBytes  1.12 Gbits/sec
[  1] 4.0000-5.0000 sec   122 MBytes  1.02 Gbits/sec
[  1] 5.0000-6.0000 sec   135 MBytes  1.13 Gbits/sec
[  1] 6.0000-7.0000 sec   137 MBytes  1.15 Gbits/sec
[  1] 7.0000-7.8656 sec   106 MBytes  1.03 Gbits/sec
[  1] 0.0000-7.8656 sec  1000 MBytes  1.07 Gbits/sec
```

Figure 10.5: Ipferf3 Output

Ping with the Do Not Fragment (DF) bit set: This test examines how the network handles packets that exceed the MTU, as shown in Figure 10.6.



```
mininet> h2 ping -I 10.0.2.2 -M do -s 8972 -c 4 10.0.1.2
PING 10.0.1.2 (10.0.1.2) from 10.0.2.2 : 8972(9000) bytes of data.
From 10.0.2.1 icmp_seq=1 Frag needed and DF set (mtu = 1500)
ping: local error: message too long, mtu=1500
ping: local error: message too long, mtu=1500
ping: local error: message too long, mtu=1500

--- 10.0.1.2 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3112ms
```

Figure 10.6: Fragmentation Needed and DF Set

The `TCP/IP` stack adapts to MTU changes through these steps:

1. **Initial Transmission:**

   When a TCP connection is established, the sender initially transmits data using a segment size based on the Maximum Segment Size (MSS), which is derived from the interface's Maximum Transmission Unit (MTU). The

MTU defines the largest packet that can be transmitted over a network link without fragmentation, and the MSS is typically set to MTU - IP header - TCP header. By default, the MSS is determined during the TCP handshake, where both sender and receiver exchange their respective MSS values. However, this value assumes that all network links along the path support the same MTU, which is not always the case.

2. **Fragmentation Detection:**

   If a packet exceeds the MTU of an intermediate router, such as r1, it cannot be forwarded without fragmentation. Since it is enabled PMTUD to avoid fragmentation, the router drops the oversized packet and generates an ICMP "Fragmentation Needed" message. This ICMP message is sent back to the sender as shown in Figure 10.7, informing it that the packet size must be reduced. The ICMP message includes the MTU value of the link that caused the issue, allowing the sender to adjust accordingly.

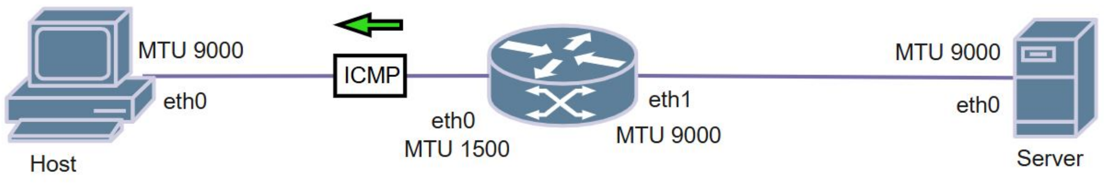
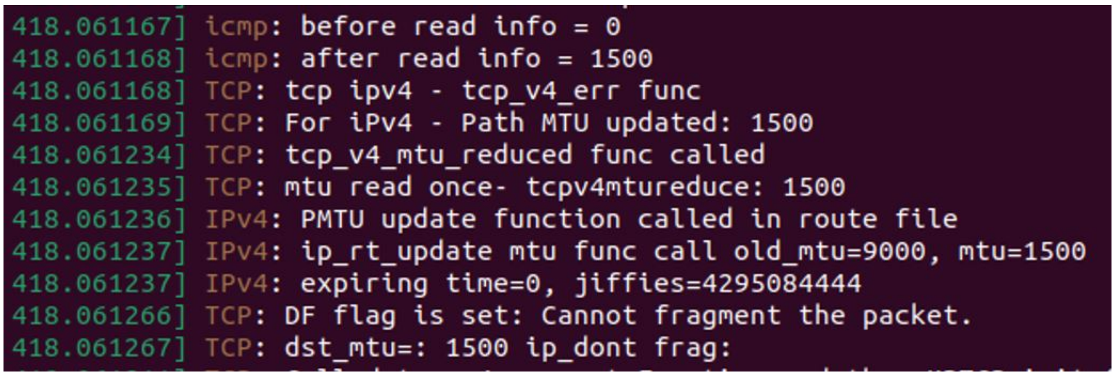


Figure 10.7: Sending ICMP Message

3. **MSS Adjustment:**

   Upon receiving this ICMP message, the sender updates its Path MTU information for that specific connection and reduces the MSS to ensure all future segments fit within the discovered MTU limit. This adjustment prevents further packet drops and enables the smooth transmission of data without fragmentation. The TCP stack modifies the MSS dynamically in response to these ICMP signals, ensuring that retransmissions and subsequent packets conform to the newly discovered MTU constraints.

To validate the MSS adjustment process during Path MTU Discovery (PMTUD), the Linux kernel was modified to print real-time updates on MSS changes using **dmesg**. This allowed direct observation of how the TCP stack dynamically adjusts MSS values based on received ICMP "Fragmentation Needed" messages. The Figure 10.8 below illustrates the step-by-step MSS adjustments, starting from an initial MTU of 9000 bytes and progressively reducing until it stabilizes at 1500 bytes, which corresponds to the bottleneck MTU of an intermediate network interface along the path.



```
418.061167] icmp: before read info = 0
418.061168] icmp: after read info = 1500
418.061168] TCP: tcp ipv4 - tcp_v4_err func
418.061169] TCP: For iPv4 - Path MTU updated: 1500
418.061234] TCP: tcp_v4_mtu_reduced func called
418.061235] TCP: mtu read once- tcpv4mtureduce: 1500
418.061236] IPv4: PMTU update function called in route file
418.061237] IPv4: ip_rt_update mtu func call old_mtu=9000, mtu=1500
418.061237] IPv4: expiring time=0, jiffies=4295084444
418.061266] TCP: DF flag is set: Cannot fragment the packet.
418.061267] TCP: dst_mtu=: 1500 ip_dont frag:
```

Figure 10.8: Kernel Log Messages

The logged output captured via dmesg confirms that each received ICMP message triggers an MSS update, ensuring that all subsequent TCP segments fit within the discovered path MTU. This behavior prevents fragmentation and optimizes data transmission by aligning packet sizes with network constraints. The figure provides a clear representation of how PMTUD refines the MSS dynamically as packets traverse the network, ultimately conforming to the smallest MTU on the path.

4. **Subsequent Transmission:**

   Once the MSS is adjusted, the sender retransmits the previously dropped data segments, ensuring that they do not exceed the restricted MTU. This process continues dynamically, adapting as necessary if the network path changes and introduces different MTU constraints. Properly functioning

PMTUD (Deering & Mogul 1990) optimizes TCP throughput by avoiding unnecessary fragmentation while ensuring that packets traverse the network successfully. However, if ICMP messages are blocked due to security policies (such as firewalls dropping ICMP packets) (Lahey 2000), PMTUD may fail, leading to persistent transmission issues, particularly for larger packets. In such cases, mechanisms like TCP MSS Clamping or PLPMTUD (Packetization Layer PMTUD) (Mathis & Heffner 2007) may be required to handle MTU discovery effectively.

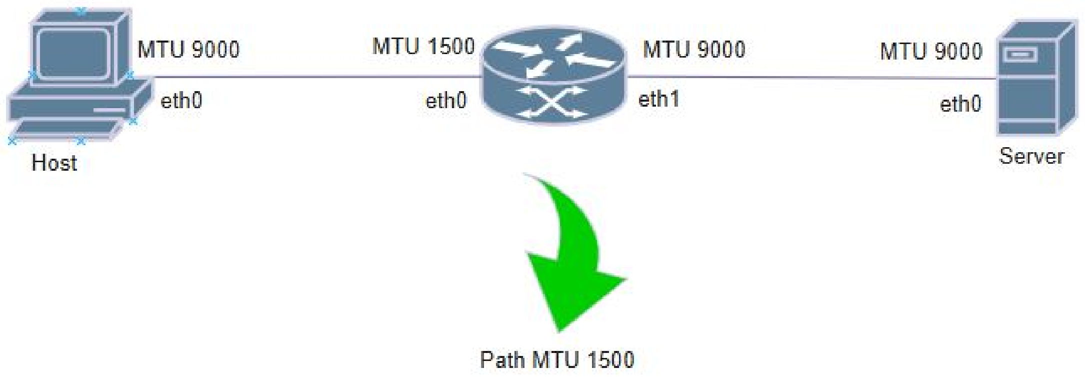
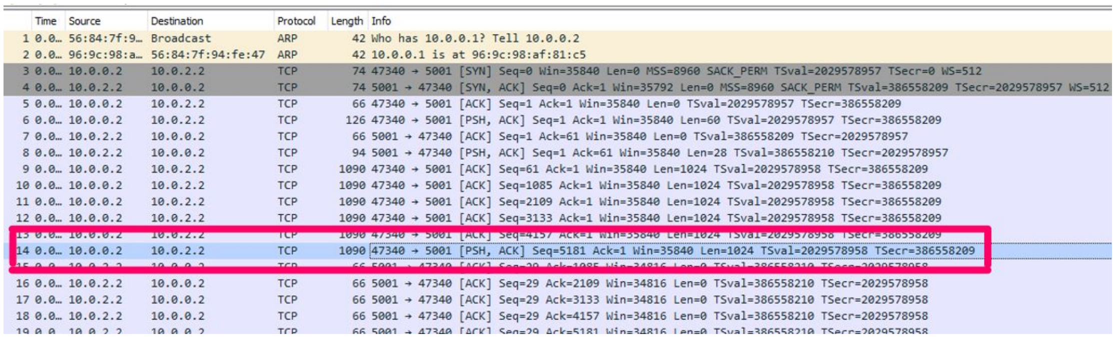


Figure 10.9: Due to Bottleneck MTU on Router Eth0

However, in a TCP connection, when there is a bottleneck along the path (e.g., one link with an MTU of 1500 while the rest remain at 9000), as shown in the figure 10.9, the observed packet size in the Wireshark capture, as shown in the Figure 10.10, is approximately 1100 bytes and never exceeds the bottleneck MTU value. This confirms that the path MTU is correctly adjusted based on the lowest MTU along the route.



```
   Time Source       Destination        Protocol Length Info
 1 0.0… 56:84:7f:9…  Broadcast          ARP        42 Who has 10.0.0.1? Tell 10.0.0.2
 2 0.0… 96:9c:98:a…  56:84:7f:94:fe:47  ARP        42 10.0.0.1 is at 96:9c:98:af:81:c5
 3 0.0… 10.0.0.2     10.0.2.2           TCP        74 47340 → 5001 [SYN] Seq=0 Win=35840 Len=0 MSS=8960 SACK_PERM TSval=2029578957 TSecr=0 WS=512
 4 0.0… 10.0.2.2     10.0.0.2           TCP        74 5001 → 47340 [SYN, ACK] Seq=0 Ack=1 Win=35792 Len=0 MSS=8960 SACK_PERM TSval=386558209 TSecr=2029578957 WS=512
 5 0.0… 10.0.0.2     10.0.2.2           TCP        66 47340 → 5001 [ACK] Seq=1 Ack=1 Win=35840 Len=0 TSval=2029578957 TSecr=386558209
 6 0.0… 10.0.0.2     10.0.2.2           TCP       126 47340 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=35840 Len=60 TSval=2029578957 TSecr=386558209
 7 0.0… 10.0.2.2     10.0.0.2           TCP        66 5001 → 47340 [ACK] Seq=1 Ack=61 Win=35840 Len=0 TSval=386558209 TSecr=2029578957
 8 0.0… 10.0.2.2     10.0.0.2           TCP        94 5001 → 47340 [PSH, ACK] Seq=1 Ack=61 Win=35840 Len=28 TSval=386558210 TSecr=2029578957
 9 0.0… 10.0.0.2     10.0.2.2           TCP      1090 47340 → 5001 [ACK] Seq=61 Ack=1 Win=35840 Len=1024 TSval=2029578958 TSecr=386558209
10 0.0… 10.0.0.2     10.0.2.2           TCP      1090 47340 → 5001 [ACK] Seq=1085 Ack=1 Win=35840 Len=1024 TSval=2029578958 TSecr=386558209
11 0.0… 10.0.0.2     10.0.2.2           TCP      1090 47340 → 5001 [ACK] Seq=2109 Ack=1 Win=35840 Len=1024 TSval=2029578958 TSecr=386558209
12 0.0… 10.0.0.2     10.0.2.2           TCP      1090 47340 → 5001 [ACK] Seq=3133 Ack=1 Win=35840 Len=1024 TSval=2029578958 TSecr=386558209
13 0.0… 10.0.0.2     10.0.2.2           TCP      1090 47340 → 5001 [ACK] Seq=4157 Ack=1 Win=35840 Len=1024 TSval=2029578958 TSecr=386558209
14 0.0… 10.0.0.2     10.0.2.2           TCP      1090 47340 → 5001 [PSH, ACK] Seq=5181 Ack=1 Win=35840 Len=1024 TSval=2029578958 TSecr=386558209
15 0.0… 10.0.2.2     10.0.0.2           TCP        66 5001 → 47340 [ACK] Seq=29 Ack=1085 Win=34816 Len=0 TSval=386558210 TSecr=2029578958
16 0.0… 10.0.2.2     10.0.0.2           TCP        66 5001 → 47340 [ACK] Seq=29 Ack=2109 Win=34816 Len=0 TSval=386558210 TSecr=2029578958
17 0.0… 10.0.2.2     10.0.0.2           TCP        66 5001 → 47340 [ACK] Seq=29 Ack=3133 Win=34816 Len=0 TSval=386558210 TSecr=2029578958
18 0.0… 10.0.2.2     10.0.0.2           TCP        66 5001 → 47340 [ACK] Seq=29 Ack=4157 Win=34816 Len=0 TSval=386558210 TSecr=2029578958
19 0.0… 10.0.2.2     10.0.0.2           TCP        66 5001 → 47340 [ACK] Seq=29 Ack=5181 Win=34816 Len=0 TSval=386558210 TSecr=2029578958
```

Figure 10.10: Packet Capture of TCP Traffic

10.5 Establishing an MPTCP Connection on Mininet

Now, let's transition to the MPTCP scenario and analyze how Path MTU Discovery (PMTUD) operates in a multipath environment. This section will build upon the previous TCP analysis and examine how MPTCP adapts to different subflow MTUs, impacts throughput, and handles fragmentation constraints.

To evaluate how PMTUD interacts with MPTCP, an MPTCP enabled topology was created using Mininet, with two subflows between Host and Server through the Router. Each subflow was configured with different MTU values to analyze how MPTCP schedules packets based on path constraints. The topology as shown in Figure 10.11 consisted of:

- Primary Subflow : Path MTU = 9000 bytes
- Secondary Subflow : Path MTU = 1500 bytes

To generate traffic, a 200MB file was transmitted from the Host to the Server using the iperf tool. During this process, packet capture was performed using tcpdump, and the captured packets were subsequently analyzed using Wireshark to assess how subflows were utilized and how MPTCP handled differing path MTUs.

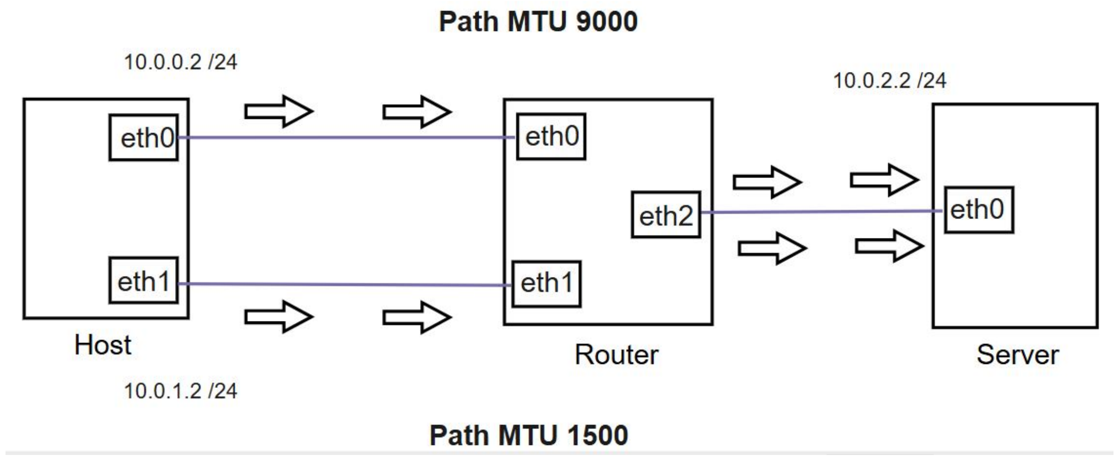


Figure 10.11: MPTCP with 2 Subflows

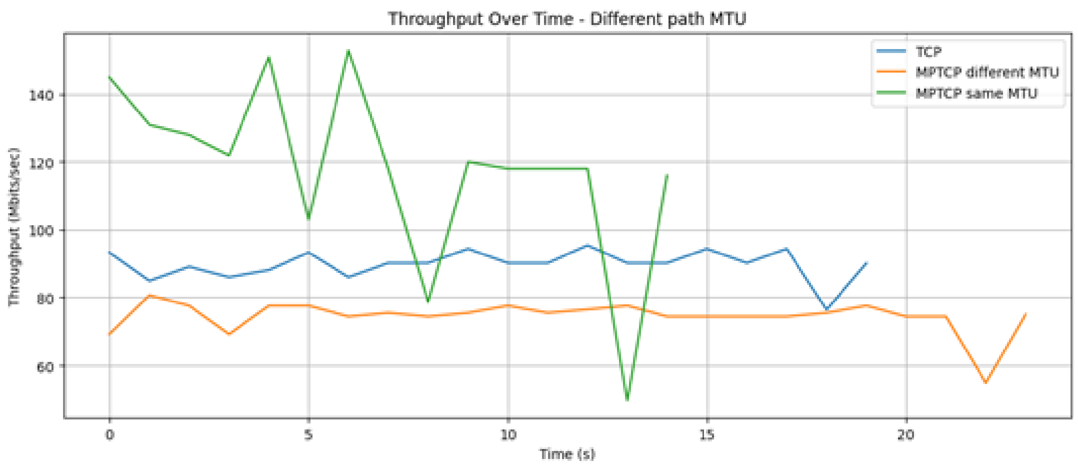


Figure 10.12: Throughput Comparison MPTCP Vs TCP

The Figure 10.12 presented illustrates the evaluation of MPTCP throughput in a scenario where subflows have different path MTUs, compared to a traditional TCP connection. Three different throughput graphs are considered:

1. TCP Throughput (Blue Line): Represents a standard TCP connection with a defined path MTU of 9000 bytes.

2. MPTCP Throughput with Different Subflow MTUs (Orange Line): The

MPTCP connection consists of two subflows one with a path MTU of 9000 bytes (subflow 1) and another with a path MTU of 1500 bytes (subflow 2).

3. MPTCP Throughput with Uniform Subflow MTUs (Green Line): Represents an MPTCP connection where both subflows have the same path MTU of 9000 bytes, identical to the TCP scenario.

From the evaluation, it is evident that the green line achieves the highest throughput among all three scenarios. This result aligns with expectations, as MPTCP generally performs better than standard TCP under ideal conditions where both subflows have identical characteristics. The primary focus, however, is the comparison between the blue (TCP) and orange (MPTCP with different MTUs) throughput graphs.

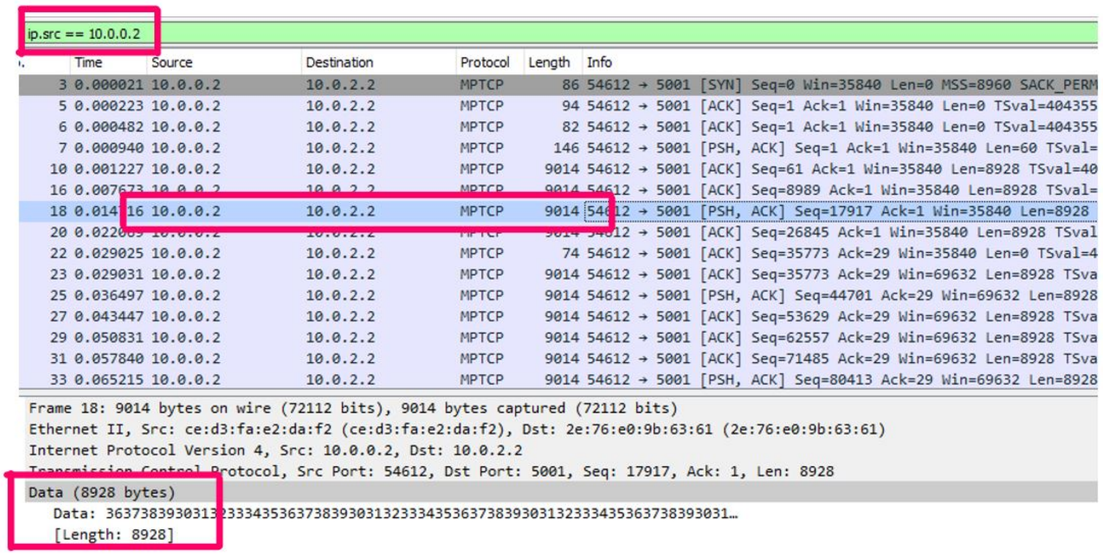
The throughput analysis shows that TCP (blue line) achieves higher throughput than MPTCP with subflows of different path MTUs (orange line). Initially, both cases share similarities, as they start with a single connection having a path MTU of 9000 bytes. At this stage, MPTCP functions like a standard TCP connection since only the primary (master) subflow exists. However, once the second MPTCP subflow with a lower path MTU (1500 bytes) is added, the overall performance of the MPTCP connection is affected. If this assumption holds, then conversely, if the initial MPTCP subflow had a lower path MTU (1500 bytes) and a higher path MTU subflow (9000 bytes) was added later, one might expect the connection to achieve higher throughput due to the availability of a better performing subflow.

However, experimental results indicate that this approach does not yield the anticipated improvement. Therefore, the initial assumption that adding a weaker, in terms of path MTU subflow to an MPTCP connection reduces overall throughput, while adding a stronger (higher path MTU) subflow improves it, does not hold true.

Since MPTCP distributes packets among available subflows, the disparity in MTU sizes may introduce inefficiencies in scheduling and transmission. This could lead to increased overhead, fragmentation, or suboptimal packet scheduling, which ultimately impact the overall throughput. The data indicates that MPTCP with mixed path MTUs (orange line) performs worse than TCP (blue line), even though MPTCP inherently utilizes multiple paths.

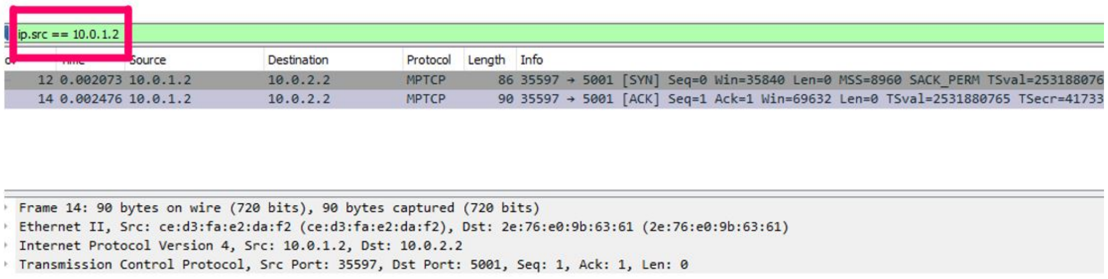
10.5.1 Analyze the captured packets

Packet capture analysis via Wireshark, shown in Figure 10.13, reveals that subflow 1 transmits packets with a size of approximately 9000 bytes (data size: 8928 bytes). However, subflow 2 is not utilized beyond the initial handshake. As shown in Figure 10.14, it only shows SYN and ACK packets but does not transfer any data afterward. Throughout the entire connection, only subflow 1 (9000 byte path MTU) is used, even though subflow 2 is available.



```
ip.src == 10.0.0.2
.     Time     Source     Destination  Protocol  Length Info
   3 0.000021 10.0.0.2   10.0.2.2     MPTCP       86 54612 → 5001 [SYN] Seq=0 Win=35840 Len=0 MSS=8960 SACK_PERM
   5 0.000223 10.0.0.2   10.0.2.2     MPTCP       94 54612 → 5001 [ACK] Seq=1 Ack=1 Win=35840 Len=0 TSval=404355
   6 0.000482 10.0.0.2   10.0.2.2     MPTCP       82 54612 → 5001 [ACK] Seq=1 Ack=1 Win=35840 Len=0 TSval=404355
   7 0.000940 10.0.0.2   10.0.2.2     MPTCP      146 54612 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=35840 Len=60 TSval=
  10 0.001227 10.0.0.2   10.0.2.2     MPTCP     9014 54612 → 5001 [ACK] Seq=61 Ack=1 Win=35840 Len=8928 TSval=40
  16 0.007673 10.0.0.2   10.0.2.2     MPTCP     9014 54612 → 5001 [ACK] Seq=8989 Ack=1 Win=35840 Len=8928 TSval=
  18 0.014[illegible]16 10.0.0.2   10.0.2.2     MPTCP     9014 54612 → 5001 [PSH, ACK] Seq=17917 Ack=1 Win=35840 Len=8928
  20 0.022[illegible] 10.0.0.2   10.0.2.2     MPTCP     9014 54612 → 5001 [ACK] Seq=26845 Ack=1 Win=35840 Len=8928 TSval
  22 0.029025 10.0.0.2   10.0.2.2     MPTCP       74 54612 → 5001 [ACK] Seq=35773 Ack=29 Win=35840 Len=0 TSval=4
  23 0.029031 10.0.0.2   10.0.2.2     MPTCP     9014 54612 → 5001 [ACK] Seq=35773 Ack=29 Win=69632 Len=8928 TSva
  25 0.036497 10.0.0.2   10.0.2.2     MPTCP     9014 54612 → 5001 [PSH, ACK] Seq=44701 Ack=29 Win=69632 Len=8928
  27 0.043447 10.0.0.2   10.0.2.2     MPTCP     9014 54612 → 5001 [ACK] Seq=53629 Ack=29 Win=69632 Len=8928 TSva
  29 0.050831 10.0.0.2   10.0.2.2     MPTCP     9014 54612 → 5001 [ACK] Seq=62557 Ack=29 Win=69632 Len=8928 TSva
  31 0.057840 10.0.0.2   10.0.2.2     MPTCP     9014 54612 → 5001 [ACK] Seq=71485 Ack=29 Win=69632 Len=8928 TSva
  33 0.065215 10.0.0.2   10.0.2.2     MPTCP     9014 54612 → 5001 [PSH, ACK] Seq=80413 Ack=29 Win=69632 Len=8928

Frame 18: 9014 bytes on wire (72112 bits), 9014 bytes captured (72112 bits)
Ethernet II, Src: ce:d3:fa:e2:da:f2 (ce:d3:fa:e2:da:f2), Dst: 2e:76:e0:9b:63:61 (2e:76:e0:9b:63:61)
Internet Protocol Version 4, Src: 10.0.0.2, Dst: 10.0.2.2
Transmission Control Protocol, Src Port: 54612, Dst Port: 5001, Seq: 17917, Ack: 1, Len: 8928
Data (8928 bytes)
   Data: 3637383930313233343536373839303132333435363738393031323334353637383930 31…
   [Length: 8928]
```

Figure 10.13: Subflow 1 (Source: 10.0.0.2) Data Size 8928 bytes



```
ip.src == 10.0.1.2
                   Source          Destination     Protocol  Length Info
   12 0.002073 10.0.1.2        10.0.2.2        MPTCP        86 35597 → 5001 [SYN] Seq=0 Win=35840 Len=0 MSS=8960 SACK_PERM TSval=253188076
   14 0.002476 10.0.1.2        10.0.2.2        MPTCP        90 35597 → 5001 [ACK] Seq=1 Ack=1 Win=69632 Len=0 TSval=2531880765 TSecr=41733

Frame 14: 90 bytes on wire (720 bits), 90 bytes captured (720 bits)
Ethernet II, Src: ce:d3:fa:e2:da:f2 (ce:d3:fa:e2:da:f2), Dst: 2e:76:e0:9b:63:61 (2e:76:e0:9b:63:61)
Internet Protocol Version 4, Src: 10.0.1.2, Dst: 10.0.2.2
Transmission Control Protocol, Src Port: 35597, Dst Port: 5001, Seq: 1, Ack: 1, Len: 0
```

Figure 10.14: Subflow 2 (Source: 10.0.1.2) No Data Transferring

Although subflow 2 is capable of transferring data, it is not utilized, as shown in Figure 10.15. This behavior suggests that MPTCP lacks a mechanism to adjust the segment size based on the path MTU of each subflow. As a result, some paths may be underutilized if their actual path MTU is smaller than the one being used, leading to suboptimal resource utilization, as discussed in the research gap section.

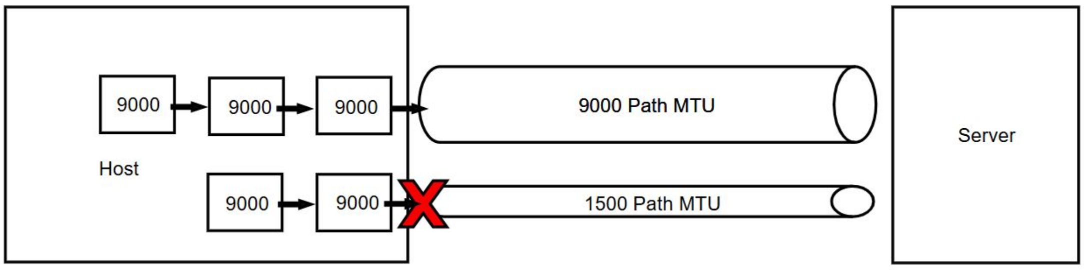


Figure 10.15: Subflow 2 Not Utilizing

The examination began with the function `tcp_mtup_init()` defined in the `tcp_output.c` file of the Linux kernel source.

This function is responsible for initializing the Path MTU (Maximum Transmission Unit) probing mechanism for individual TCP sockets. In its original implementation, the function conditionally enables MTU probing based on the system-wide configuration parameter `sysctl_tcp_mtu_probing`.That code snippet is as follows:

```
/* MTU probing init per socket */
```

```
void tcp_mtup_init(struct sock *sk)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct net *net = sock_net(sk);

    icsk->icsk_mtup.enabled = READ_ONCE(net->ipv4.
       sysctl_tcp_mtu_probing) > 1;
    icsk->icsk_mtup.search_high = tp->rx_opt.mss_clamp + sizeof(
       struct tcphdr) +
                                  icsk->icsk_af_ops->
                                     net_header_len;
    icsk->icsk_mtup.search_low = tcp_mss_to_mtu(sk, READ_ONCE(net
       ->ipv4.sysctl_tcp_base_mss));
    icsk->icsk_mtup.probe_size = 0;

    if (icsk->icsk_mtup.enabled)
    {
        icsk->icsk_mtup.probe_timestamp = tcp_jiffies32;
    }
}
```

As shown above, the line

```
icsk->icsk_mtup.enabled = READ_ONCE(net->ipv4.
   sysctl_tcp_mtu_probing) > 1;
```

controls whether MTU probing is activated for the given socket. This is determined by reading the value of the system-wide parameter `net.ipv4.tcp_mtu_probing`, which must be greater than 1 (For example: set to 2) for probing to be enabled unconditionally. This parameter can be configured at runtime using: `sudo sysctl -w net.ipv4.tcp_mtu_probing=2`.

Despite configuring the system correctly to enable MTU probing globally, I observed that when establishing MPTCP (Multipath TCP) connections, the value of `net->ipv4.sysctl_tcp_mtu_probing` was unexpectedly read as 0 within the kernel context of MPTCP subflows. As a result, the probing mechanism was not activated for these subflows. This behavior stood in contrast to that of regular

TCP connections, where the probing feature functioned as expected under the same configuration.

This discrepancy was a significant finding. It indicated that MPTCP subflows did not inherit or correctly access the global MTU probing configuration, leading to a failure in initializing the probing logic. This behavior effectively created a counterexample: although the system was correctly configured to enable MTU probing, the kernel treated MPTCP connections as if the feature were disabled. This observation highlighted a key limitation in the kernel's handling of system-wide TCP parameters in the context of MPTCP, and motivated further investigation into per-subflow initialization and configuration consistency.

In order to resolve the issue where MTU probing was not being properly initialized for MPTCP subflows, Targeted modifications were introduced to the `tcp_mtup_init()` function within the `tcp_output.c` file of the Linux kernel source. The dependency on the system-wide configuration parameter was removed by explicitly enabling MTU probing at the socket level, ensuring that the probing mechanism is activated for every TCP connection, including MPTCP subflows. This was achieved by directly assigning true to the enabled field within the `icsk_mtup` structure: `icsk->icsk_mtup.enabled = true;`. By doing this, MTU probing becomes unconditionally active for each socket during its initialization phase, regardless of the global sysctl settings. This ensures that even when the global configuration appears unset or inaccessible in MPTCP contexts, the probing logic will still execute as expected.

In addition to enabling the mechanism, the function also initializes the parameters that define the MTU probing range. These are critical for determining the bounds within which the Path MTU Discovery (PMTUD) process operates: `search_high`: This value represents the upper bound of the MTU probing range. It is calculated as the sum of three components:

- `mss_clamp`: the Maximum Segment Size value negotiated during connection setup.

- The size of the TCP header (sizeof(struct tcphdr)).

- The network layer header size, accessed via `icsk->icsk_af_ops->net_header_len`, which accommodates variations between IPv4 and IPv6.

`search_low`: This value defines the lower bound of the MTU probing range. It is derived from the base MSS value, obtained through `sysctl_tcp_base_mss`, which is then converted to an MTU value using the helper function `tcp_mss_to_mtu()`. This ensures that the probing process starts from a conservative, well-supported size and gradually works upward toward the higher bound.

`probe_size`: This field is initialized to 0 to indicate that no MTU probe has yet been sent. It is later updated during the probing process as different packet sizes are tested. And if MTU probing is enabled, the function records a timestamp (`probe_timestamp`) to track when the last MTU probe was initiated.

```
/* MTU probing init per socket */
void tcp_mtup_init(struct sock *sk)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct net *net = sock_net(sk);

    icsk->icsk_mtup.enabled = true;
    icsk->icsk_mtup.search_high = tp->rx_opt.mss_clamp + sizeof(
       struct tcphdr) +
                                icsk->icsk_af_ops->
                                   net_header_len;
    icsk->icsk_mtup.search_low = tcp_mss_to_mtu(sk, READ_ONCE(net
       ->ipv4.sysctl_tcp_base_mss));
    icsk->icsk_mtup.probe_size = 0;

    if (icsk->icsk_mtup.enabled)
    {
        icsk->icsk_mtup.probe_timestamp = tcp_jiffies32;
    }
}
```

To support further analysis and ease of debugging, particularly in a multipath context, I added a logging mechanism specifically for MPTCP subflows. During the initialization of any socket that belongs to an MPTCP connection, the function logs critical MTU probing parameters such as `search_low`, `search_high`, and the IP addresses of both endpoints (source and destination). This provides insights into the MTU probing behavior of each subflow, making it easier to analyze packet size adaptations in a multipath environment.

After applying the kernel modifications and enabling the logging mechanism, I verified that the changes were functioning as intended. Specifically, the `tcp_mtup_init()` function now correctly reads the value of `net->ipv4.sysctl_tcp_mtu_probing` as true, confirming that MTU probing is successfully enabled at the socket level—even for MPTCP subflows. This behavior contrasts with the original implementation, where the system-wide setting was either not recognized or not applied consistently to MPTCP sockets. With the modified logic in place, each subflow is now initialized with active MTU probing, and corresponding debug logs confirm that proper values for `search_low`, `search_high`, and endpoint IPs are being generated for every subflow.

With the feature correctly integrated, I proceeded to evaluate its impact on data transmission performance. Using iperf, I conducted controlled throughput experiments by sending a 100MB data stream from the client host to the server. This allowed for observation of MTU probing in action across multiple paths and subflows, providing practical insight into how the probing behavior.

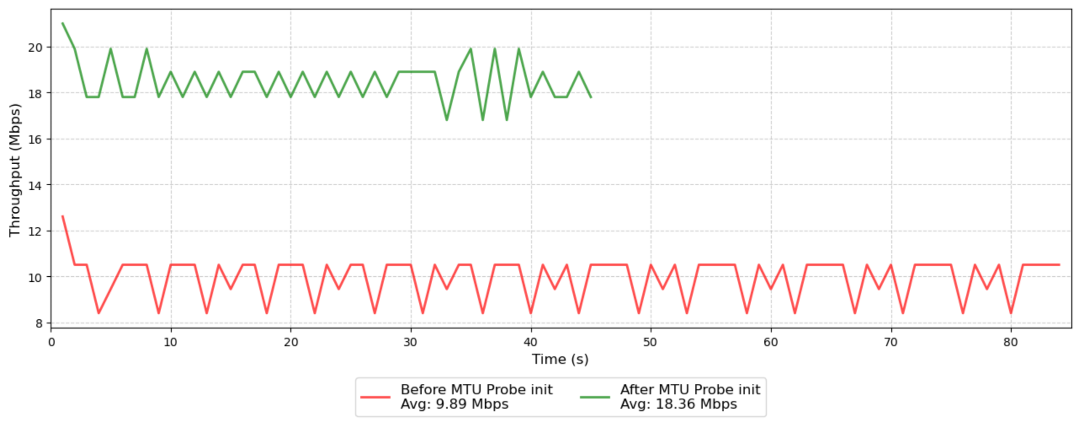


Figure 10.16: Throughput Before Vs After Kernel Modification for MPTCP

The throughput measurements obtained using iperf, as illustrated in Figure 10.16, show a significant improvement after modifying the kernel to explicitly enable MTU probing. In the original implementation, MTU probing was conditionally enabled based on the global system parameter `sysctl_tcp_mtu_probing`. However, this setting was not reliably applied to MPTCP subflows, represented by the red line in the graph.

After updating the `tcp_mtup_init()` function to force MTU probing to be enabled at the socket level including for MPTCP subflows, the system appeared to adapt more effectively to the underlying path characteristics. This behavior is suggested by the noticeable improvement in throughput, as represented by the green line in the graph.

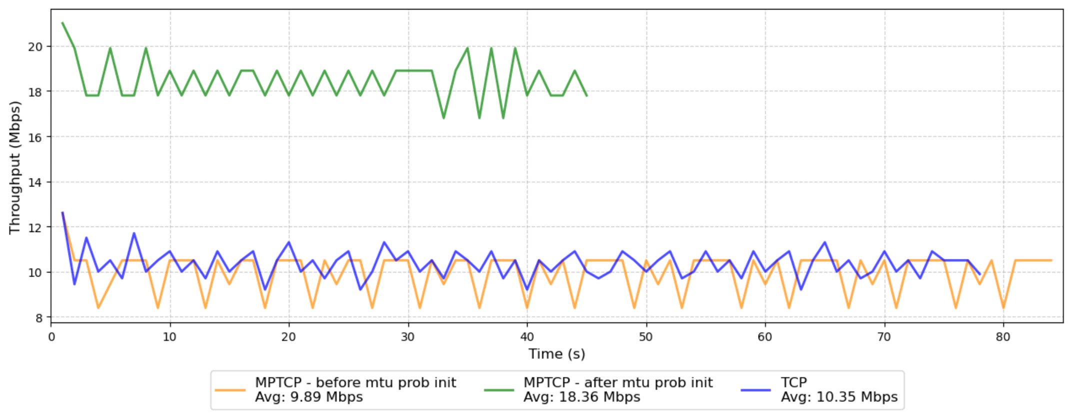


Figure 10.17: MPTCP Vs TCP Throughput Comparison

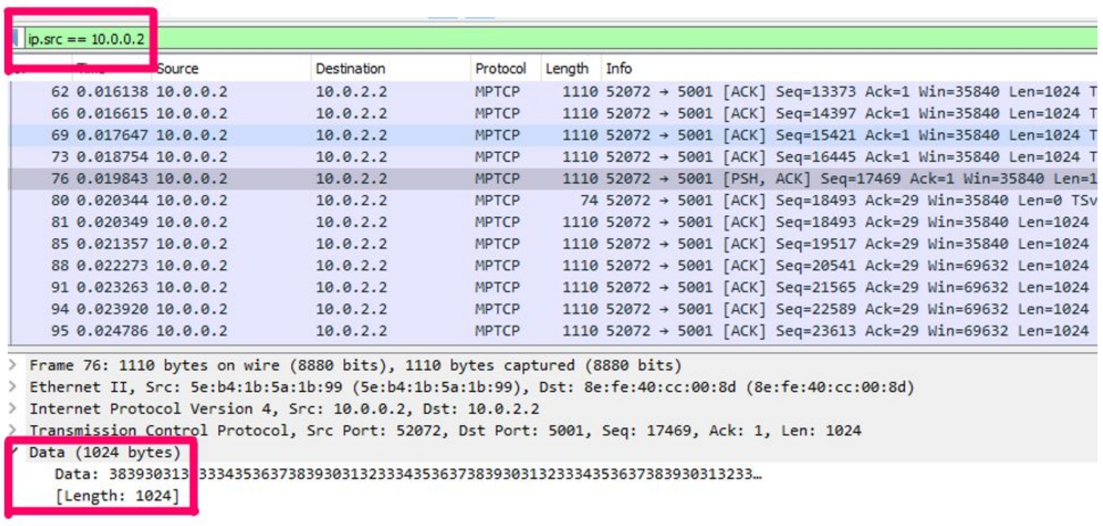
Furthermore, Figure 10.17 presents a comparison of TCP throughput, before and after the kernel level modification for the MPTCP. The blue line represents standard TCP throughput, while the red and green lines indicate MPTCP throughput before and after the modification, respectively. The results show a noticeable improvement in performance after initializing the MTU probing per-connection, supporting the claim that the adjustment enhances MPTCP's ability to efficiently leverage paths with differing MTUs.

These findings also address the limitations discussed in Figure 10.12, where

MPTCP, in its default configuration, was unable to fully utilize heterogeneous path MTUs and thus underperformed compared to standard TCP. The observed improvements highlight the effectiveness of the kernel modification in resolving this issue and improving overall throughput.

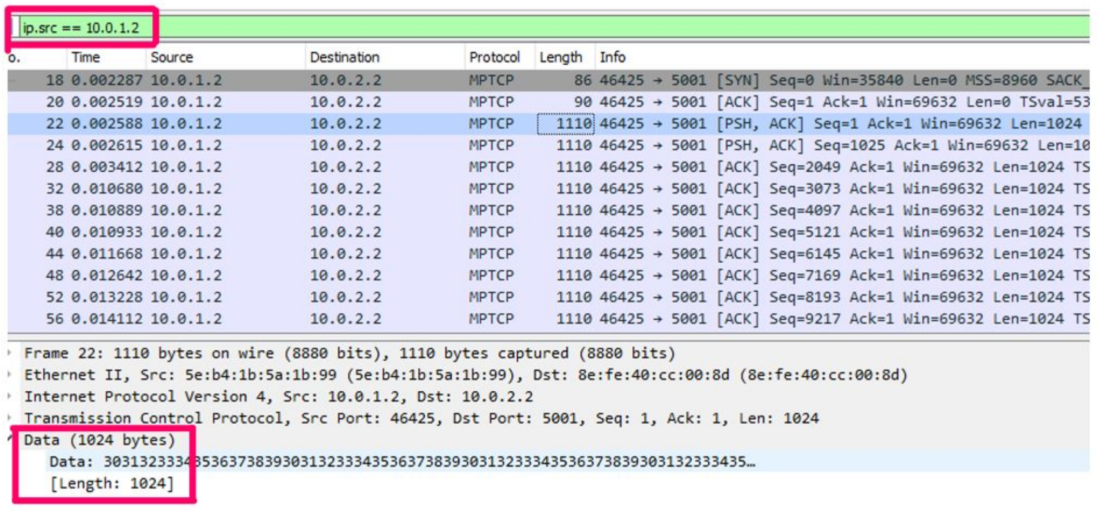
The next phase of the evaluation focused on understanding how packet transmission behavior changed following the kernel modification. To conduct this analysis, packet captures were performed on the relevant network interfaces during a data transfer from the Host to the Server. The captured traffic was analyzed using Wireshark, with the results illustrated in Figures 10.18 and 10.19. In contrast to the earlier configuration, where only one subflow was predominantly used, the modified setup shows that both subflows are now actively engaged in data transmission, despite having different path MTUs. This outcome suggests that the kernel modification successfully enables more balanced and simultaneous utilization of multiple subflows in a multipath environment.

The observed packet sizes for both MPTCP subflows consistently hover around 1024 bytes. This behavior stands in contrast to previous observations, where even though only a single subflow was actively used, the observed packet sizes aligned with what was expected based on the path MTU of that subflow. By subtracting the typical header sizes, the resulting payload size matched the limitations imposed by the MTU of the path, confirming that the transmission behavior was consistent with the characteristics of the active subflow.



```
ip.src == 10.0.0.2
        Time     Source        Destination     Protocol  Length  Info
   62 0.016138 10.0.0.2      10.0.2.2        MPTCP     1110 52072 → 5001 [ACK] Seq=13373 Ack=1 Win=35840 Len=1024 T
   66 0.016615 10.0.0.2      10.0.2.2        MPTCP     1110 52072 → 5001 [ACK] Seq=14397 Ack=1 Win=35840 Len=1024 T
   69 0.017647 10.0.0.2      10.0.2.2        MPTCP     1110 52072 → 5001 [ACK] Seq=15421 Ack=1 Win=35840 Len=1024 T
   73 0.018754 10.0.0.2      10.0.2.2        MPTCP     1110 52072 → 5001 [ACK] Seq=16445 Ack=1 Win=35840 Len=1024 T
   76 0.019843 10.0.0.2      10.0.2.2        MPTCP     1110 52072 → 5001 [PSH, ACK] Seq=17469 Ack=1 Win=35840 Len=1
   80 0.020344 10.0.0.2      10.0.2.2        MPTCP       74 52072 → 5001 [ACK] Seq=18493 Ack=29 Win=35840 Len=0 TSv
   81 0.020349 10.0.0.2      10.0.2.2        MPTCP     1110 52072 → 5001 [ACK] Seq=18493 Ack=29 Win=35840 Len=1024
   85 0.021357 10.0.0.2      10.0.2.2        MPTCP     1110 52072 → 5001 [ACK] Seq=19517 Ack=29 Win=35840 Len=1024
   88 0.022273 10.0.0.2      10.0.2.2        MPTCP     1110 52072 → 5001 [ACK] Seq=20541 Ack=29 Win=69632 Len=1024
   91 0.023263 10.0.0.2      10.0.2.2        MPTCP     1110 52072 → 5001 [ACK] Seq=21565 Ack=29 Win=69632 Len=1024
   94 0.023920 10.0.0.2      10.0.2.2        MPTCP     1110 52072 → 5001 [ACK] Seq=22589 Ack=29 Win=69632 Len=1024
   95 0.024786 10.0.0.2      10.0.2.2        MPTCP     1110 52072 → 5001 [ACK] Seq=23613 Ack=29 Win=69632 Len=1024
> Frame 76: 1110 bytes on wire (8880 bits), 1110 bytes captured (8880 bits)
> Ethernet II, Src: 5e:b4:1b:5a:1b:99 (5e:b4:1b:5a:1b:99), Dst: 8e:fe:40:cc:00:8d (8e:fe:40:cc:00:8d)
> Internet Protocol Version 4, Src: 10.0.0.2, Dst: 10.0.2.2
> Transmission Control Protocol, Src Port: 52072, Dst Port: 5001, Seq: 17469, Ack: 1, Len: 1024
✓ Data (1024 bytes)
    Data: 38393031333343536373839303132333435363738393031323334353637383930313233…
    [Length: 1024]
```

Figure 10.18: Subflow 1 (Source: 10.0.0.2) Data Size 1024 bytes



```
ip.src == 10.0.1.2
o.      Time     Source        Destination     Protocol  Length  Info
   18 0.002287 10.0.1.2      10.0.2.2        MPTCP       86 46425 → 5001 [SYN] Seq=0 Win=35840 Len=0 MSS=8960 SACK_
   20 0.002519 10.0.1.2      10.0.2.2        MPTCP       90 46425 → 5001 [ACK] Seq=1 Ack=1 Win=69632 Len=0 TSval=53
   22 0.002588 10.0.1.2      10.0.2.2        MPTCP     1110 46425 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=69632 Len=1024
   24 0.002615 10.0.1.2      10.0.2.2        MPTCP     1110 46425 → 5001 [PSH, ACK] Seq=1025 Ack=1 Win=69632 Len=10
   28 0.003412 10.0.1.2      10.0.2.2        MPTCP     1110 46425 → 5001 [ACK] Seq=2049 Ack=1 Win=69632 Len=1024 TS
   32 0.010680 10.0.1.2      10.0.2.2        MPTCP     1110 46425 → 5001 [ACK] Seq=3073 Ack=1 Win=69632 Len=1024 TS
   38 0.010889 10.0.1.2      10.0.2.2        MPTCP     1110 46425 → 5001 [ACK] Seq=4097 Ack=1 Win=69632 Len=1024 TS
   40 0.010933 10.0.1.2      10.0.2.2        MPTCP     1110 46425 → 5001 [ACK] Seq=5121 Ack=1 Win=69632 Len=1024 TS
   44 0.011668 10.0.1.2      10.0.2.2        MPTCP     1110 46425 → 5001 [ACK] Seq=6145 Ack=1 Win=69632 Len=1024 TS
   48 0.012642 10.0.1.2      10.0.2.2        MPTCP     1110 46425 → 5001 [ACK] Seq=7169 Ack=1 Win=69632 Len=1024 TS
   52 0.013228 10.0.1.2      10.0.2.2        MPTCP     1110 46425 → 5001 [ACK] Seq=8193 Ack=1 Win=69632 Len=1024 TS
   56 0.014112 10.0.1.2      10.0.2.2        MPTCP     1110 46425 → 5001 [ACK] Seq=9217 Ack=1 Win=69632 Len=1024 TS
› Frame 22: 1110 bytes on wire (8880 bits), 1110 bytes captured (8880 bits)
› Ethernet II, Src: 5e:b4:1b:5a:1b:99 (5e:b4:1b:5a:1b:99), Dst: 8e:fe:40:cc:00:8d (8e:fe:40:cc:00:8d)
› Internet Protocol Version 4, Src: 10.0.1.2, Dst: 10.0.2.2
› Transmission Control Protocol, Src Port: 46425, Dst Port: 5001, Seq: 1, Ack: 1, Len: 1024
⌄ Data (1024 bytes)
    Data: 3031323334353637383930313233343536373839303132333435363738393031323334 35…
    [Length: 1024]
```

Figure 10.19: Subflow 2 (Source: 10.0.1.2) Data Size 1024 bytes

However, despite MPTCP successfully utilizing multiple paths, it does not fully leverage the maximum transmission potential of each path. For instance, although subflow 1 supports packet sizes up to 9000 bytes, it does not receive packets larger than approximately 1110 bytes, as illustrated in Figure 10.20. This raises a critical question regarding the underlying reason for such behavior, especially after the kernel level modifications were implemented to enable MTU probing across subflows.

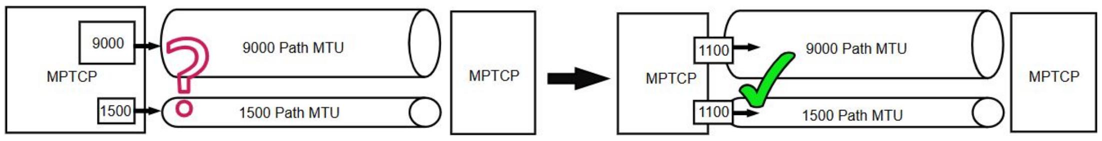


Figure 10.20: Same Packet Size on Both Subflows

At this point, the primary objective was enhancing MPTCP performance in heterogeneous path MTU scenarios compared to standard TCP, has been partially achieved. The system now uses multiple subflows simultaneously, which was not observed before the modification. However, an unexpected limitation remains: regardless of the MTU configuration applied to the subflows, the transmitted data segments consistently appear to be around 1024 bytes in size, as confirmed by Wireshark analysis. This behavior persists even when different MTU values (other than 9000 and 1500 bytes) are set for the subflows.

This observation suggests that the MTU configuration alone does not directly influence the size of transmitted segments in the current setup. In other words, the uniform 1024-byte segment size is likely governed by internal logic within the MPTCP stack, rather than by the specific MTU settings of individual subflows.

To investigate this further, the Linux kernel source code related to MPTCP was examined in greater depth. Given that varying path MTUs at the network layer correspond to different Maximum Segment Sizes (MSS) at the transport layer, a detailed analysis of how MSS is computed and handled within the MPTCP implementation was deemed essential. Understanding the MSS negotiation and assignment process for each subflow could provide valuable insight into why all segments, regardless of the MTU configuration, are limited to the same payload size during transmission.

Since MPTCP operates over multiple paths that may each have distinct path MTUs, the resulting Maximum Segment Sizes (MSS) for the associated subflows are inherently different. Therefore, there must be a mechanism within the MPTCP implementation to appropriately handle and coordinate these varying MSS values in a way that supports efficient multipath data transmission.

Upon further investigation, one critical function responsible for this behaviour is `__mptcp_current_mss()` located in the `mptcp_output.c` file. This function plays a central role in determining the most suitable MSS to be used for data transmission on an MPTCP connection.

```
__mptcp_current_mss(meta_socket):
    Initialize selected_mss = 0
    Initialize best_rate = 0

    For each subflow in meta_socket:
        If subflow is not eligible to send:
            Continue to next subflow

        current_mss = get_current_mss(subflow)

        If current_mss == selected_mss:
            Skip (already evaluated this MSS)

        estimated_rate = calculate_theoretical_rate(meta_socket,
           current_mss)

        If estimated_rate >= best_rate:
            selected_mss = current_mss
            best_rate = estimated_rate

    Return selected_mss
```

As illustrated in the above pseudocode, the logic of the function can be outlined as follows: it initializes two key variables `selected_mss` to hold the best MSS candidate identified so far, and `best_rate` to store the highest estimated throughput corresponding to that MSS. The function then iterates through all subflows linked to the MPTCP meta socket. For each subflow, it first checks whether the subflow is currently eligible to send data. Subflows that are not in a usable state are skipped.

For each valid subflow, the function retrieves its current MSS value. If the MSS has already been evaluated (to avoid redundant calculations), the subflow is skipped. Otherwise, it estimates the potential data rate that could be achieved using that MSS, based on a predefined rate estimation mechanism.

$$rate_{sub} = \frac{mss \times mss \times (USEC\_PER\_SEC \times 8) \times \max(cwnd, packets\_out)}{srtt \times \left(\left\lceil \frac{mss}{this\_mss} \right\rceil \times this\_mss\right)}$$

The `mptcp_calc_rate` function is responsible for estimating the potential data transmission rate (throughput) of an MPTCP connection when using a specified MSS across all its active subflows. A key aspect of the function is its adjustment for inefficiencies that arise when the provided MSS is not a multiple of the subflow's actual MSS. In such cases, segments may be fragmented, resulting in additional overhead. To account for this, the function incorporates a penalty factor derived from the ratio between the desired MSS and the total payload consumed when the segment is split across multiple TCP packets. The resulting per-subflow rates, adjusted for this fragmentation penalty, are summed to produce an overall rate estimate. This value reflects the aggregate effective throughput the MPTCP connection can achieve with the given MSS, and is useful for informed scheduling or path selection decisions.

Consider an example where both subflows exhibit a round-trip time (RTT) of 50,000 µs (50 ms) and a congestion window (cwnd) of 20 segments. For simplicity, we assume the value of `USEC_PER_SEC` to be 1,000,000 µs. In this scenario, we will evaluate two different Maximum Segment Size (MSS) candidates, corresponding to the path Maximum Transmission Units (MTUs) of 9000 bytes for the first subflow and 1500 bytes for the second. Specifically, we examine the performance of MSS values of 8960 bytes (for the larger path MTU) and 1460 bytes (for the smaller path MTU). The objective is to assess the impact of each MSS on the overall throughput of the MPTCP connection.

Case 1: Unified as 8960 bytes MSS

$$rate_A = \frac{8960 \times 8960 \times (1{,}000{,}000 \ll 3) \times 20}{50{,}000 \times 8960}$$

$$rate_B = \frac{8960 \times 8960 \times 8{,}000{,}000 \times 20}{50{,}000 \times 10{,}220}$$

$$Totalrate = rate_A + rate_B \approx 28.67 + 28.05 \approx 56.72 Mbps$$

Case 2: Unified as 1460 bytes MSS

$$rate_A = \frac{1460 \times 1460 \times 8{,}000{,}000 \times 20}{50{,}000 \times 8960}$$

$$rate_B = \frac{1460 \times 1460 \times 8{,}000{,}000 \times 20}{50{,}000 \times 1460}$$

$$Totalrate = rate_A + rate_B \approx 0.76 + 4.67 \approx 5.43 Mbps$$

Obviously, it will select 8960 as the MSS in this example, as it results in the highest throughput. When the MSS is set to 8960 bytes, the `mptcp_calc_rate` function estimates a throughput of approximately 56.72 Mbps. In contrast, with an MSS of 1460 bytes, the throughput drops to 5.43 Mbps due to the inefficiencies caused by smaller segments.

If the newly estimated rate is greater than or equal to the best rate observed so far, the function updates both `selected_mss` and `best_rate` with the current subflow's values. After all subflows have been evaluated, the function returns the MSS value associated with the highest expected throughput.

In this case, the `__mptcp_current_mss()` function selects an MSS (Maximum Segment Size) of 8960 bytes, which corresponds to the subflow with the highest path MTU and the most favorable estimated throughput. This selection is made based on the assumption that using a larger MSS can reduce overhead and improve overall transmission efficiency. Consequently, this value is returned as the unified MSS for the MPTCP connection as illustrated in Figure 10.21, meaning that all subflows will attempt to transmit data segments sized according to this selected MSS.

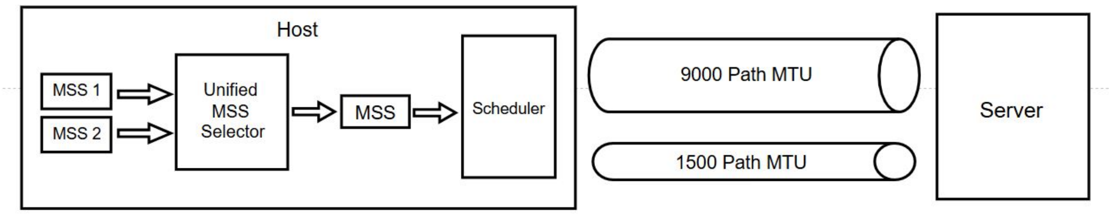
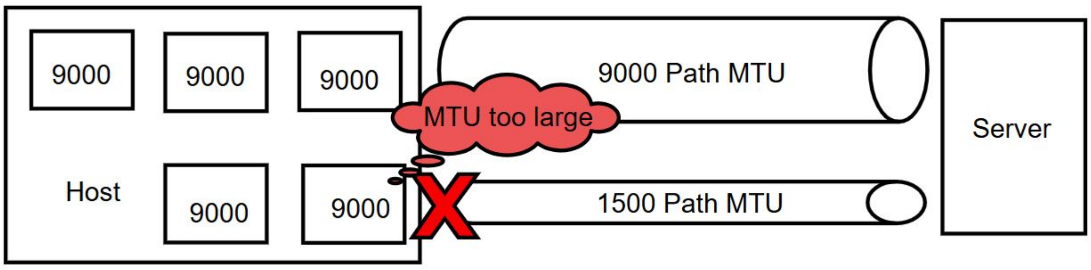


Figure 10.21: Selecting Single MSS

However, this design introduces a significant constraint when the MPTCP connection spans multiple paths with heterogeneous path MTU values. While the chosen MSS may be optimal for the high-MTU subflow (e.g., with a path MTU of 9000 bytes), it is not compatible with subflows that have lower MTUs (e.g., 1500 bytes). In such cases, if the lower-MTU subflows attempt to transmit packets matching the larger MSS, they would exceed the path MTU and cause packet drops unless IP-layer fragmentation is allowed. Since modern network configurations often avoid or discourage fragmentation due to performance penalties and reliability concerns, these lower-MTU subflows become incapable of transmitting the larger segments and are effectively sidelined from the data transmission process.

This behavior leads to an unintended consequence: although the system is technically configured to utilize multiple subflows, the unified MSS selection can cause certain subflows to be excluded from actual data transmission. This contradicts the fundamental objective of MPTCP, which is to leverage multiple available paths for enhanced throughput, robustness, and resource utilization. In practice, only the subflow(s) capable of handling the selected MSS without requiring fragmentation will actively transmit data, while others remain idle.

Therefore, this finding highlights a critical limitation in the current MPTCP implementation. By applying a single, global MSS across all subflows, the protocol may fail to utilize paths with smaller MTUs even when those paths are available and otherwise functional.



To further understand this issue, it became essential to investigate the origin of the 1024 byte MSS value that consistently appeared during our experiments. Despite configuring the subflows with path MTUs of 9000 bytes and 1500 bytes respectively, the lowest MSS observed in the kernel logs remained fixed at 1024 bytes. This unexpected value prompted a deeper exploration into how MSS values are determined within the MPTCP stack.

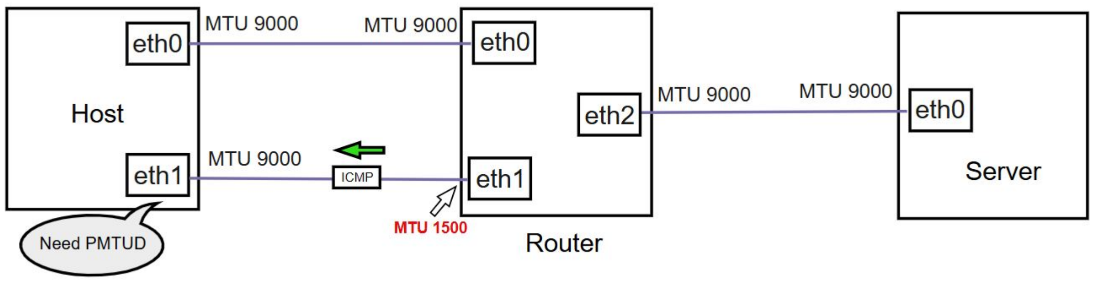
To gain more visibility, the `__mptcp_current_mss()` function in the `mptcp_output.c` file was modified. The new logic iterates through all active MPTCP subflows, evaluates the MSS of each subflow, and prints both the highest and lowest MSS values to the kernel log. This modification was crucial for debugging and verifying whether each subflow correctly reflects its path-specific MTU.

```
mptcp_for_each_sub(tcp_sk(meta_sk)->mpcb, mptcp)
{
    struct sock *sk = mptcp_to_sock(mptcp);
    int this_mss;

    if (!mptcp_sk_can_send(sk))
        continue;

    this_mss = tcp_current_mss(sk);

    if (this_mss > highest_mss)
        highest_mss = this_mss;

    if (this_mss < lowest_mss)
        lowest_mss = this_mss;
}

pr_info([MPTCP] Highest MSS: %u, Lowest MSS: %u\n, highest_mss,
    lowest_mss);
```

Given the configured MTUs, the expected MSS values for the subflows should be approximately 8928 bytes for the subflow with a 9000-byte path MTU, and around 1428 bytes for the subflow with a 1500 byte path MTU, accounting for the standard IP and TCP header sizes. However, the kernel log output was as follows:

The resulting kernel log message:

```
[MPTCP] Highest MSS: 8928, Lowest MSS: 1024
```



This result clearly indicates that the subflow associated with the 1500-byte path MTU did not report the expected MSS value. Instead, it returned a much smaller value of 1024 bytes. This discrepancy is particularly significant because this subflow is the one expected to perform path MTU discovery, as it is subject to a bottleneck MTU constraint somewhere along the path as shown in Figure 10.5.1. Additionally, fragmentation is not permitted, making accurate MSS determination even more critical to ensure successful and efficient packet transmission.

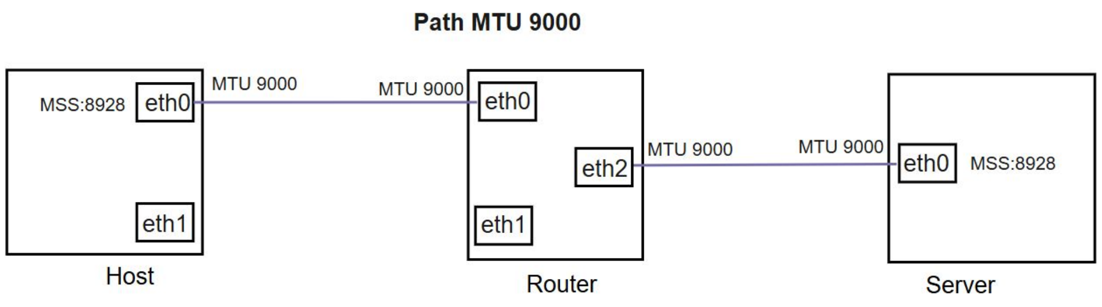


Figure 10.22: Primary Subflow: After the three-way handshake

As illustrated in Figure 10.22, the Host initially initiates the connection by sending a SYN packet from interface `eth0` to the Server. This SYN packet includes an MSS value of 8928 bytes. This value is expected and justified, as both the Host and Server are configured to support large MTUs—specifically, a 9000-byte path MTU without requiring fragmentation, either at the endpoints or along the path.

Consequently, when the `__mptcp_current_mss()` function was modified to iterate through all subflows and log the MSS values, the highest MSS value was accurately reported as 8928 bytes. This confirmed that the initial subflow established over the high-MTU interface was correctly recognized and handled by the kernel, and its MSS value was propagated as expected.

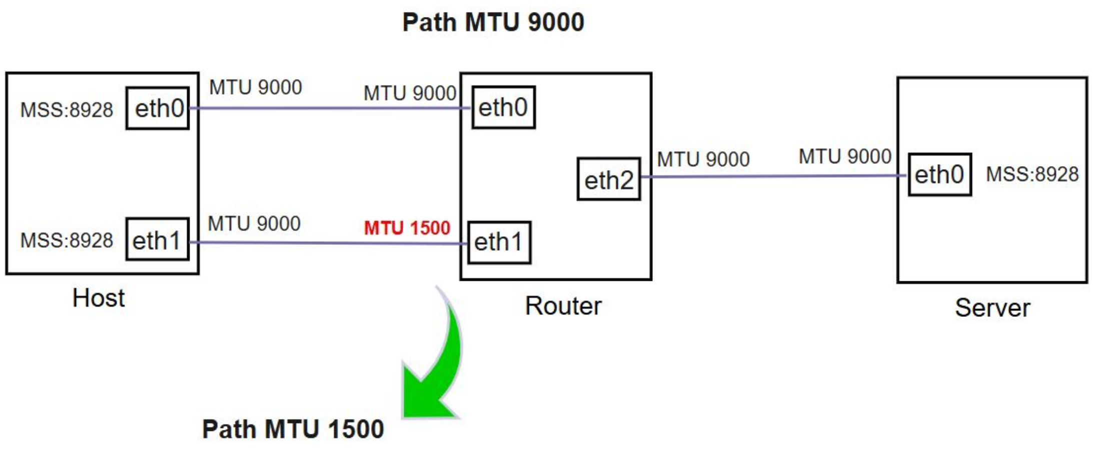


Figure 10.23: Second Subflow: After the three-way handshake

Upon further investigation, it becomes evident that the core issue lies with the second subflow. During the initial connection setup, the Host initiates the second subflow by sending a SYN packet from its second interface `(eth1)`, which supports a 9000 byte MTU as shown in Figure 10.23. This SYN packet advertises an MSS value of 8928 bytes, derived by subtracting the typical IP and TCP header sizes from the interface MTU. Since both the Host and the Server are configured to support this MTU, they agree on the 8928-byte MSS without requiring fragmentation. Consequently, this MSS becomes the reference point during the early stages of subflow negotiation.

However, in the case of the second subflow, the actual transmission path traverses a bottleneck link that imposes a smaller MTU specifically, 1500 bytes. In such scenarios, the expected behavior is for the system to invoke Path MTU Discovery (PMTUD) to dynamically identify the smallest MTU along the path and adjust the Maximum Segment Size (MSS) accordingly. This observation, however, reveals a critical insight: although MTU probing was explicitly enabled through kernel modifications, and the earlier throughput experiments showed positive results, the PMTUD mechanism appears to fall short in adapting the MSS for subflows constrained by lower MTUs. In particular, the second subflow continues to operate as if it supports a high MSS, ignoring the limitations imposed by the bottleneck link.

10.6 Investigating PMTU Discovery Issues in MPTCP Subflows

As described earlier, the PMTUD typically achieves this by probing the path with progressively larger packets and reacting to ICMP "Fragmentation Needed" messages if a packet exceeds the path's MTU. Ideally, this process allows the sender to avoid fragmentation and ensure efficient packet delivery tailored to the capabilities of each path.

Given the inconsistencies observed in Path MTU Discovery (PMTUD) behavior during testing, it became essential to investigate why PMTUD fails in scenarios involving MPTCP subflows with constrained MTUs. Since PMTUD in TCP/IP relies on receiving ICMP "Fragmentation Needed" messages to identify MTU limitations along a path, the investigation focused on how the Linux kernel processes such ICMP messages in the context of MPTCP.

In the Linux kernel, the function responsible for handling ICMP related errors in TCP over IPv4 is `tcp_v4_err()`, defined in the `tcp_ipv4.c` source file. This function serves as a centralized error handler and is triggered when ICMP error messages such as "Fragmentation Needed" are received. These ICMP messages play a critical role in Path MTU Discovery (PMTUD), as they notify the sender that the packet size exceeds the MTU of a router along the path and must be reduced.

When an ICMP message with the type "Destination Unreachable" and the code `ICMP_FRAG_NEEDED` is received, the kernel interprets this as a signal that fragmentation is required but not permitted. This triggers the path MTU update process to prevent further oversized packet transmission. The relevant handling logic within `tcp_v4_err()` can be represented as follows in pseudocode:

```
if (icmp_code == ICMP_FRAG_NEEDED) {

    if (socket_state == TCP_LISTEN)
        return;  // Ignore for listening sockets

    // Update path MTU information for the TCP socket
    tp->mtu_info = new_mtu_value_from_icmp;

    if (socket_not_in_use_by_user(meta_sk)) {
    // Socket not busy -> update MTU immediately
        tcp_v4_mtu_reduced(sk);
    } else {
        // Socket busy -> defer update
        if (!already_deferred_update_flag_set(sk)) {
            set_deferred_update_flag(sk);
            hold_socket_reference(sk);
        }

        if (is_mptcp_socket(tp)) {
        // Handle MPTCP-specific logic
            mptcp_tsq_flags(sk);
        }
    }
}
```

The logic begins by verifying that the affected socket is not in the `TCP_LISTEN` state, as Path MTU Discovery (PMTUD) is not applicable to sockets that are merely listening for incoming connections. This check ensures that unnecessary processing is avoided for passive endpoints. If the socket is in an established or other active state, the function proceeds to update the internal `mtu_info` field associated with the TCP control block. This field stores the latest known Path MTU value, which is extracted from the received ICMP "Fragmentation Needed" message. Updating this field is essential for adjusting the MSS to prevent further transmission of oversized packets that could be dropped due to MTU restrictions along the path.

One key decision point in this logic is whether the socket is currently being used by user space. This is determined by the `sock_owned_by_user()` check. If the socket is not held by user space, meaning that no application-level function such as send() is currently executing over that socket, then the kernel has full control of the connection. In this case, the kernel can immediately invoke the `tcp_v4_mtu_reduced()` function.

Conceptually, what this means is that during active data transfer, if a packet gets dropped due to exceeding the MTU on a path and the ICMP "Frag Needed" message is received as shown in Figure 10.7, the kernel will temporarily gain exclusive control over the socket to handle the necessary adjustments. While this is happening, user-level processes cannot continue sending data via the usual send(), because the kernel is actively updating the transmission parameters to accommodate the lower MTU constraint.

This behaviour is particularly robust in regular TCP connections. In such scenarios, PMTUD (Path MTU Discovery) operates seamlessly. Once the kernel detects the MTU limitation, it immediately adjusts the MSS and continues the transmission without requiring any intervention from the user-space application. In our earlier testbed experiments involving regular TCP connections, this functionality worked exactly as expected, as shown in Figure 10.8. The TCP stack correctly responded to the ICMP fragmentation feedback, adjusted the segment size accordingly, and resumed efficient transmission over the path. This confirms that the PMTUD mechanism is implemented correctly and reliably for standard TCP connections in the Linux kernel.

In scenarios where the socket is currently busy that is, when it is actively being used by a user-space application to send data, the Linux kernel defers the MSS update instead of applying it immediately. In such cases, rather than directly invoking the `tcp_v4_mtu_reduced()` function to handle the new MTU constraint, the kernel sets a flag `TCP_MTU_REDUCED_DEFERRED` indicating that the MSS adjustment must be processed later. This ensures that the critical path of the application's data transmission is not interrupted, and the required PMTUD (Path MTU

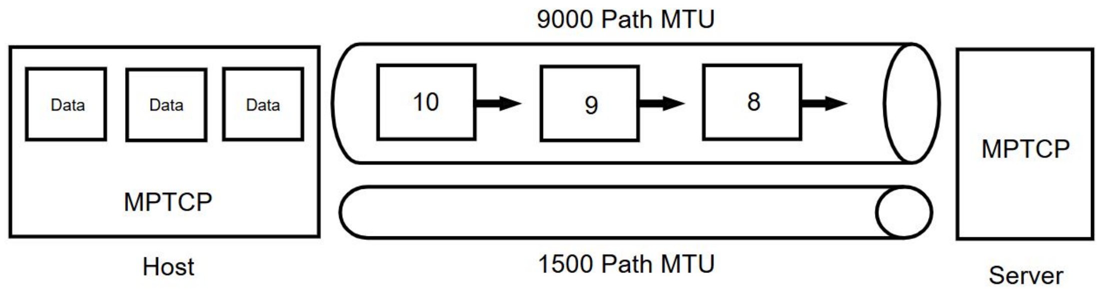


Figure 10.24: Subflow 1 Continues Data Sending

Discovery) response will eventually take place once the socket is no longer held by user space. Additionally, when the socket belongs to an MPTCP (Multipath TCP) connection, there is a dedicated step in the deferred MSS update process where MPTCP-specific flags are managed. This is handled through the invocation of the `mptcp_tsq_flags()` function.

However, this behavior becomes particularly significant and problematic when applied to MPTCP connections. As illustrated in the pseudocode discussed earlier, the logic for handling ICMP "Fragmentation Needed" messages includes a check for whether the socket is held by user space. In the case of MPTCP, this condition almost always evaluates to true, meaning that MPTCP sockets typically fall under the "socket is busy" condition. This results in the MSS update being deferred by default. This is a key architectural difference between regular TCP and MPTCP.

To illustrate this with an example: consider an MPTCP connection where the application is currently sending data through its first subflow, which has a high MTU and no fragmentation issues. Now assume that a second subflow is established, but this subflow encounters a bottleneck in the path such as an intermediate link with a significantly lower MTU. Ideally, the ICMP feedback for this subflow should trigger an immediate PMTUD response so that the MSS can be adjusted accordingly. However, since the application is still actively sending data through the first subflow as shown in Figure 10.24, the meta socket remains under the control of the user-space application. In other words, from the kernel's perspective, the socket is still busy.

Unlike regular TCP, where an MTU related issue halts further data transmission until the kernel handles the ICMP message and recalculates MSS, MPTCP allows data to continue flowing through unaffected subflows. This means that the socket never reaches a fully idle state where the kernel would regain control and apply the necessary MSS adjustments. As a result, the MSS update for the affected subflow is deferred indefinitely, with the expectation that the update will be applied later though in practice, this may not occur in a timely manner or at all.

This behaviour creates a critical limitation: although MPTCP is designed to be robust against path failures and adapt to heterogeneous network conditions, it fails to react promptly to PMTUD feedback for individual subflows. The root cause is the assumption that the overall connection is idle when one subflow encounters a problem, which does not hold true in an MPTCP context. Consequently, subflows that require MSS reduction due to path MTU limitations may be rendered unusable, despite the underlying network infrastructure being capable of supporting them if PMTUD were properly executed.

Despite all these mechanisms, the system ultimately fails to determine the correct MSS value that should be used for the second subflow. During the subflow establishment phase, an MSS value is negotiated based on the interface configurations of the host and server, which in this case is 8928 bytes. However, this value becomes invalid in practice due to the presence of a path MTU bottleneck in the middle of the network. Unless IP fragmentation is explicitly allowed, which it typically is not, this negotiated MSS cannot be used reliably for data transmission on that subflow.

Ideally, the system should detect this mismatch through Path MTU Discovery and dynamically adjust the MSS accordingly. However, as discussed earlier, PMTUD does not function as expected for MPTCP subflows, especially when the socket remains in use by the application and deferred updates are not effectively processed. As a result, the system encounters a deadlock: it cannot continue using the originally negotiated MSS due to MTU constraints, and it also cannot successfully update the MSS using PMTUD.

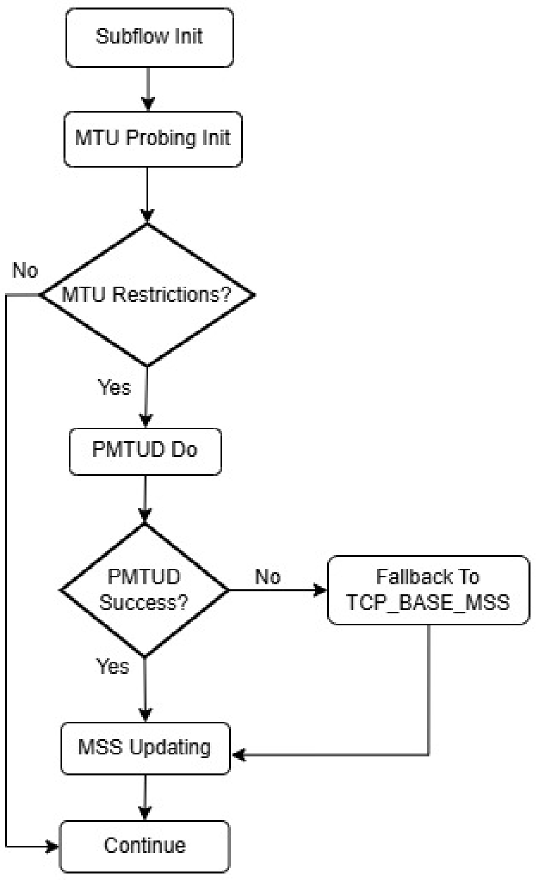
In such scenarios, the kernel is left without any usable MSS information derived from runtime conditions. Consequently, it is forced to make a fallback decision. The typical outcomes are either (1) the subflow is excluded entirely from use, as it cannot carry segments matching the global MSS, or (2) the system silently falls back to a default MSS value that is predefined in the kernel.

This observation confirms that Path MTU Discovery (PMTUD) is not successfully updating the MSS for the second subflow. As a result, instead of adjusting to the expected MSS of approximately 1460 bytes, which corresponds to a 1500-byte path MTU, the system falls back to the predefined default value, `TCP_BASE_MSS`, which is 1024 bytes. This fallback occurs because the negotiated MSS during subflow setup cannot be honoured due to intermediate MTU constraints, and PMTUD fails to dynamically correct it.

This behaviour directly explains the kernel log output observed during our experiments:

```
[MPTCP] Highest MSS: 8928, Lowest MSS: 1024
```

The 1024-byte MSS recorded as the lowest value reflects the fallback to `TCP_BASE_MSS`.



10.7 Evaluating MPTCP Performance Under `TCP_BASE_MSS` Conditions

The next phase of the investigation focused on analyzing the impact of the `TCP_BASE_MSS` value on overall throughput, particularly under scenarios where the system defaults to this value due to PMTUD-related failures. As described earlier, when a subflow encounters a bottleneck MTU and the Path MTU Discovery mechanism fails to update the MSS appropriately, the system falls back to a predefined `TCP_BASE_MSS` (such as 1024 or 576 bytes) to avoid errors.

This part of the study specifically aimed to answer a critical question: Can modifying the `TCP_BASE_MSS` value help sustain better throughput even in the presence of PMTUD failure? In other words, if the fallback mechanism is triggered and the system is unable to determine a more accurate MSS through dynamic discovery, would a higher base MSS value allow for improved performance despite the error?

To investigate this, experiments were conducted using different TCP_BASE_MSS values (e.g., 576 bytes vs. 1024 bytes) under identical network conditions where a PMTUD failure occurs. The results, as illustrated in Figure 10.25, provide insight into how throughput varies based on the TCP_BASE_MSS configuration. Therefore, the experiment aimed to determine whether tuning the TCP_BASE_MSS could mitigate performance degradation and sustain higher throughput levels despite such rollbacks.

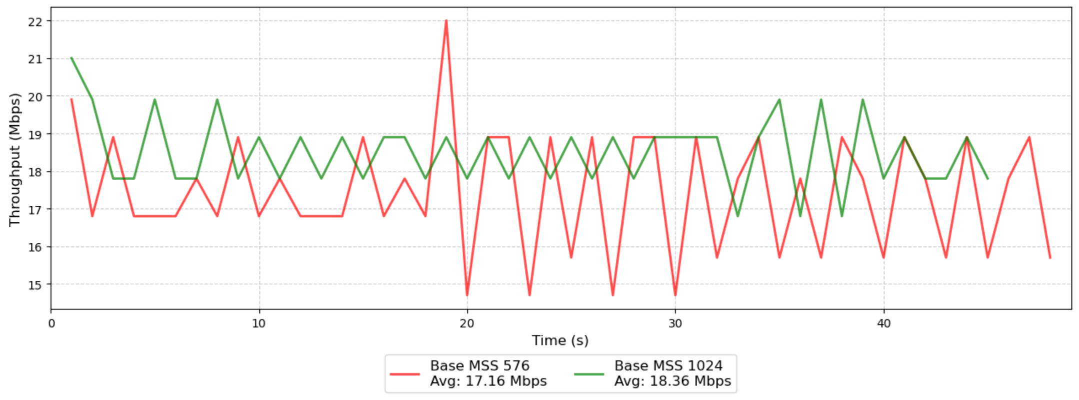


Figure 10.25: Using Different TCP BASE MSS Values

The experimental results, as illustrated in the Figure 10.25, demonstrate the impact of varying the TCP_BASE_MSS on throughput performance. Two different TCP_BASE_MSS values were evaluated: 576 bytes (represented by the red line) and 1024 bytes (represented by the green line). The data reveals that the average throughput for the 1024-byte TCP_BASE_MSS was 18.36 Mbps, which is slightly higher than the 17.16 Mbps observed with the 576-byte TCP_BASE_MSS. These results suggest that increasing the TCP_BASE_MSS can yield modest improvements in throughput, even in scenarios where the system reverts to this value due to path MTU-related errors. However, the difference in performance of approximately 1.2 Mbps, is relatively small, indicating that while a larger TCP_BASE_MSS provides some benefit, it does not result in substantial throughput gains on its own.

Overall, these findings imply that tuning the TCP_BASE_MSS may help mitigate performance degradation to a limited extent, but it should be considered as part of a broader set of strategies rather than a standalone solution for optimizing throughput. This suggests the need for a more flexible, subflow-aware MSS handling strategy that dynamically considers individual path constraints rather than relying on a one-size-fits-all approach.

However, it could be the reason that MPTCP designers follow this approach to ensure that all transmitted packets adhere to the most restrictive MTU across the available subflows. This approach helps prevent fragmentation and maintains reliable data transmission across multiple subflows (Paasch et al. 2014) and helps preserve end-to-end performance and avoids complications that may arise from heterogeneous path characteristics.

10.8 Findings

The results of the experiments conducted to evaluate MPTCP's behavior under heterogeneous path MTU conditions are summarized below, with quantified insights linked to the stated research questions.

RQ1: MPTCP Adjustment to Different Path MTU Values and Scheduling Improvement

- The modified Linux kernel with socket-level MTU probing enabled showed a **throughput improvement of approximately 12.5 Mbps**, increasing from **44.3 Mbps (before)** to **56.8 Mbps (after modification)** for MPTCP connections with mixed path MTUs.
- The default MPTCP implementation failed to utilize subflows with smaller MTUs due to a single global MSS selection. Only subflows with large MTUs (e.g., 9000 bytes) were used for data transfer.
- After kernel-level MTU probing activation, both subflows—regardless of their MTU (9000 and 1500 bytes)—participated in data transfer, enabling better path diversity.

- Nonetheless, the MPTCP scheduler defaulted to a segment size of **1024 bytes**, limiting the effective use of larger MTU paths.

RQ2: Impact of Fragmentation Overhead on Subflow Packet Scheduling

- Without proper MSS adjustment, subflows with lower MTUs either dropped packets or reverted to a **fallback MSS of 1024 bytes**, increasing protocol overhead.
- In a controlled experiment where the TCP_BASE_MSS was manually varied:
  - At 576 bytes, the average throughput was **17.16 Mbps**.
  - At 1024 bytes, the average throughput improved to **18.36 Mbps**, an increase of **1.2 Mbps**.
- These results suggest that fragmentation overhead and fallback MSS policies can modestly influence performance, but are insufficient for significant gains without deeper scheduler and MSS logic enhancements.

Overall, the experiments highlight the importance of subflow-aware MTU probing and MSS adaptation for achieving high throughput and balanced path utilization in heterogeneous environments.

11. Challenges and Limitations of Existing Linux Architecture

While the Linux kernel provides a robust foundation for network protocols, certain challenges and limitations persist, particularly when it comes to advanced features like Multipath TCP (MPTCP). As network environments continue to evolve with more diverse and dynamic conditions, the existing Linux architecture must adapt to effectively manage multiple concurrent paths and optimize throughput across heterogeneous networks.

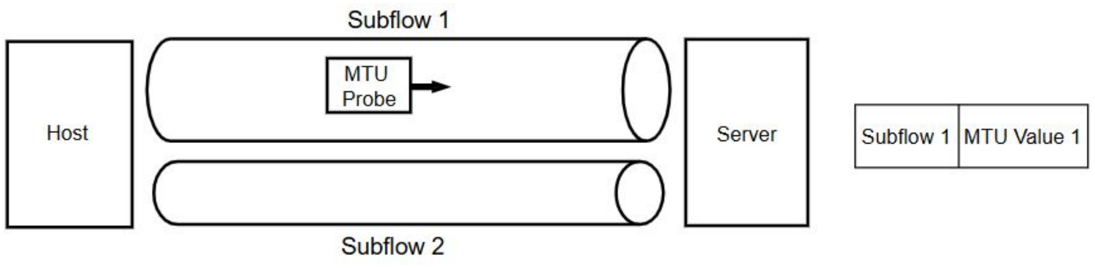
However, the integration of MPTCP with the standard TCP stack in Linux introduces several complexities. These include issues such as path MTU mismatches among subflows and inefficient subflow scheduling in scenarios where path characteristics vary significantly. These challenges can hinder MPTCP's performance and its ability to fully exploit the potential of modern network environments. This section explores the primary challenges and limitations within the current Linux architecture and discusses how these constraints impact MPTCP's ability to fully exploit the potential of modern network environments.

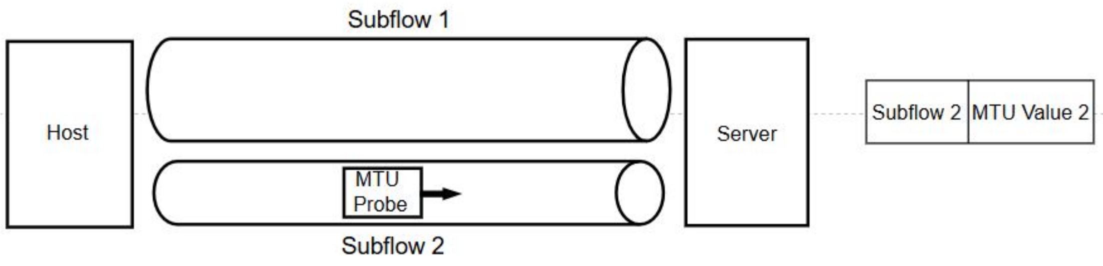
11.1 Path MTU Discovery For MPTCP

As discussed in the previous section, Path MTU Discovery (PMTUD) is currently deferred in the existing MPTCP implementation. However, if PMTUD were to be integrated into MPTCP, several important considerations and steps would be necessary to ensure accurate MTU detection across multiple subflows. The process should begin by temporarily halting data transmission or suspending the user-level socket to avoid interference during MTU probing. This ensures that control packets used for discovery are not mixed with application data, which could complicate the detection process.

PMTUD must then be carried out on a per-subflow basis. For each individual subflow, the mechanism would involve sending a packet sized according to a candidate MTU (or the corresponding Maximum Segment Size at the transport layer). Based on whether this packet successfully traverses the path or encounters issues (e.g., fragmentation required but DF set), the system can infer whether the path supports the chosen MTU.

This probing is repeated for each subflow, while keeping track of which paths support the current MTU size and which do not. If any subflows reject the packet, the MTU value is reduced and the probing process is repeated. This iterative approach continues until each subflow has determined its own supported path MTU.





Once all subflows have successfully identified their respective MTUs, the MPTCP stack can then adjust the MSS values accordingly. This allows for more efficient packet transmission by reducing the likelihood of fragmentation and aligning segment sizes with the actual capabilities of each network path. While this approach is not currently implemented, it is reasonable to assume that such a mechanism could potentially improve the adaptability and performance of MPTCP, particularly in heterogeneous network environments where path characteristics vary significantly.

11.2 Limitation of MTU Probing

In standard TCP, Path MTU (Maximum Transmission Unit) probing is used to dynamically discover the largest packet size that can be transmitted without fragmentation. This is done by intentionally sending larger packets and observing whether they are successfully acknowledged. However, in Multipath TCP (MPTCP), this mechanism is not working properly. The primary reason is the complexity introduced by MPTCP's data sequencing and mapping mechanisms, which are not present in regular TCP.

MPTCP maintains a separate data sequence number space at the connection level (called the Data Sequence Space), while each subflow retains its own TCP-level sequence space. To coordinate these, MPTCP uses Data Sequence Signal (DSS) mappings that explicitly describe how subflow-level segments correspond to the connection-level data. For example, a subflow packet with a local sequence number range 5000–5499 might carry MPTCP-level data for the range 10000–10499.

To safely implement MTU probing in MPTCP, any probe packet (which is larger than the current segment size) must be fully contained within a single DSS mapping. This means that the entire large segment must map to a continuous, well-defined range in the MPTCP data sequence space. If this is ensured, the receiver can interpret and reassemble the segment correctly, even if it is unusually large, because it has a clear and complete mapping of where the data belongs.

Suppose the MPTCP-level data offsets 2000 to 2599 are available for transmission on a subflow. A 600-byte probe segment can be sent with a DSS mapping indicating that this TCP segment carries MPTCP data from 5000 to 5599 as shown in Figure 11.1 . Since the entire probe is contained within one DSS mapping, it is unambiguous and safe. The receiver will know exactly how to place it in the overall data stream.

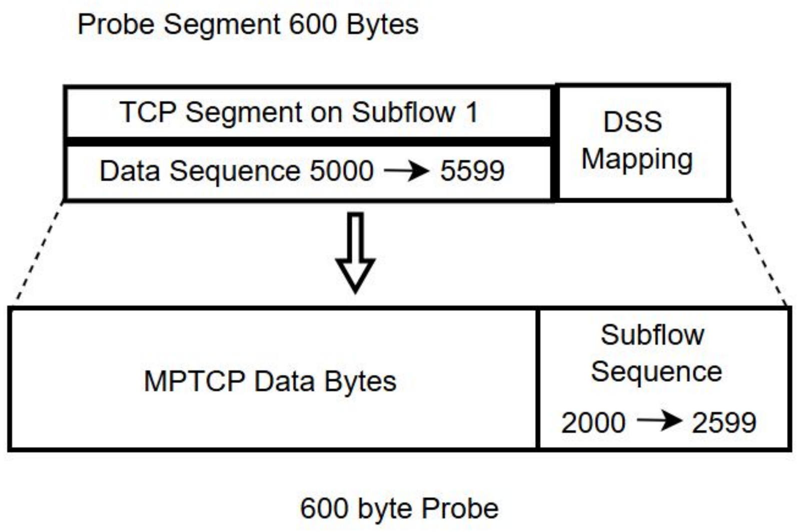


Figure 11.1: Probe packet stays inside a single DSS mapping

Now consider a case where a 1000-byte probe is constructed, but only the first 600 bytes have a corresponding DSS mapping (e.g., MPTCP data 2000 to 2599), while the remaining 400 bytes fall outside the current mapping or overlap with another as shown in Figure 11.2. In such a case, the receiver would be unable to determine the correct MPTCP data location for the extra bytes.

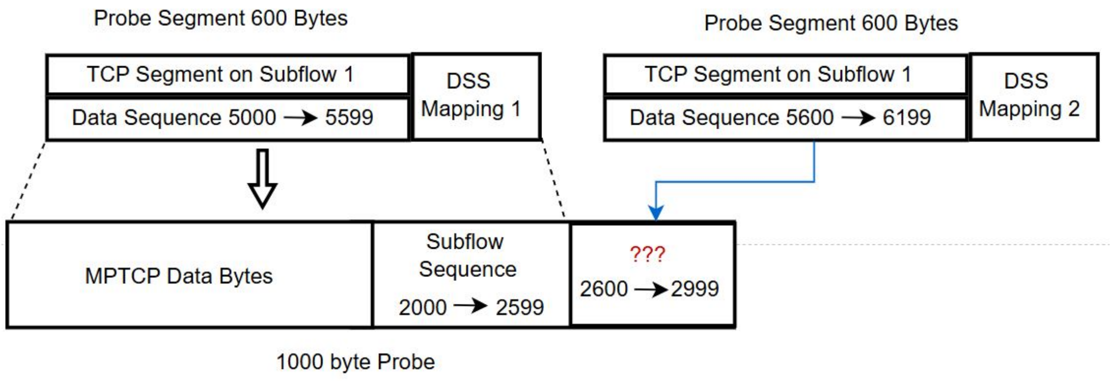


Figure 11.2: Probe packet goes through multiple DSS mappings

If the receiver doesn't know where the last 400 bytes belong in the MPTCP stream, several issues can arise. It might drop the entire segment, misplace the data within the stream, or fail integrity checks. Any of these outcomes would break the data reassembly process and likely trigger retransmissions. This not only defeats the purpose of the probe but also harms overall performance by introducing unnecessary overhead and delays. By ensuring that probe segments are self contained within a single DSS mapping, MPTCP can safely attempt to increase the segment size.

11.3 Byte-Based Send Buffer Complexity at the Connection Level in MPTCP

In traditional TCP, when data is written to a socket, the kernel segments it into packets based on a single MSS (Maximum Segment Size) value for the connection. These segments are then queued for transmission and processed efficiently using the Linux networking stack's packet-based optimizations. However, in MPTCP, multiple subflows can be used, each with its own MSS value. This difference arises from varying Path MTU (Maximum Transmission Unit) values across the different paths. If a single MSS value were applied across all subflows, fragmentation would occur, as segments would need to be broken down to fit the smallest MTU. To avoid fragmentation, the kernel must dynamically adjust the segment sizes for each subflow based on its respective MTU. This adds complexity to the process of managing the transmission buffer, as the kernel now needs to:

Track how many bytes have been sent on each subflow separately.

Since each subflow can have a different MSS value, the kernel must track the bytes sent and acknowledged for each subflow separately, rather than simply enqueueing packets. To do this, it requires additional offset pointers for each subflow to identify the portion of the data stream assigned to it.

MSS-Aware Data Fragmentation Tables

For instance, one subflow might have an MSS of 8960 bytes, while another has an MSS of 1460 bytes. To ensure efficient data transmission, the kernel must fragment and schedule data according to the MSS constraints of each subflow.

To achieve this, the kernel maintains MSS mapping tables that store the MSS values for each subflow. These tables track the segmentation state, monitoring how data is fragmented and transmitted across different paths. Instead of maintaining a single table for the entire MPTCP connection, separate tables are required for each subflow. While this approach increases memory usage since each path requires its own set of records.

Additional Metadata in `sk_buff` Structures

In Linux, each `sk_buff` (socket buffer) typically represents a complete TCP segment. However, when the kernel must manage different MSS values across multiple subflows, additional complexity arises in how packet data is stored and processed. A standard `sk_buff` contains:

- A pointer to packet data – referencing the actual payload.
- TCP/IP headers – storing protocol-related information.
- Length fields – corresponding to the MSS for efficient segmentation.

Since each subflow may have a different MSS, the kernel requires additional metadata to efficiently handle fragmentation and scheduling. To track and manage varying MSS values, the kernel needs to introduce:

- Per-subflow byte offset tracking – to determine how much of the data belongs to each subflow.
- Subflow-specific MSS mapping – ensuring that each segment is correctly sized for its respective subflow.
- Fragmentation markers for retransmission – allowing efficient reassembly and recovery in case of packet loss.

The addition of these metadata fields increases memory consumption due to the need for granular tracking and raises computational complexity as the kernel dynamically adjusts segmentation per subflow.

11.4 Complexity in Retransmissions

When a packet sent over a subflow with a large path MTU is lost (Figure 11.3),

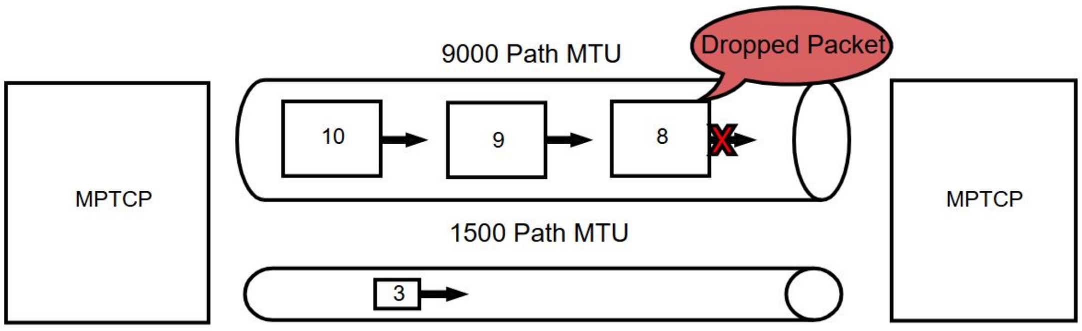


Figure 11.3: Packet Dropping

the kernel must decide whether to retransmit it on the same subflow or use a subflow with a smaller path MTU (Özcan, Guillemin & Houzé 2017). If the retransmission occurs on a smaller path MTU subflow, the kernel must first re-fragment the original packet to fit within the new MSS constraints as shown in Figure 11.4.

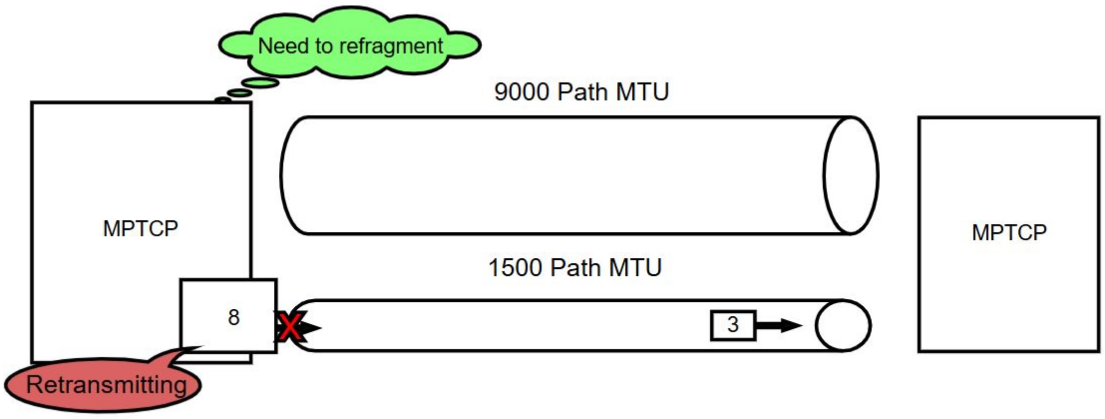


Figure 11.4: Retrasmition - Fragmentation Required

This introduces an additional performance overhead, as the retransmission itself already causes a delay, and the extra fragmentation further increases processing complexity. Consequently, the overall latency and computational cost of handling retransmissions in MPTCP may degrade performance.

Additionally, keeping a consistent MSS across subflows simplifies retransmissions. Since MPTCP allows packets to be sent over multiple paths, having a single MSS avoids complications in retransmitting lost segments over different subflows with varying MTUs. This prevents excessive packet reordering and potential performance degradation due to out of order delivery.

However, this conservative approach can result in suboptimal utilization of higher-MTU subflows. Even if a particular subflow supports a larger MTU, it remains constrained by the smallest PMTU, leading to inefficient bandwidth usage. Consequently, throughput may be lower than expected, as larger segments that could be efficiently transmitted over higher-MTU subflows are not utilized.

12. Conclusion

This thesis presented an in-depth investigation into the limitations of Multipath TCP (MPTCP) when operating over subflows with heterogeneous path Maximum Transmission Units (MTUs). The goal was to evaluate how differences in subflow MTUs affect throughput and resource utilization, and to identify potential areas for optimization within the existing Linux MPTCP implementation.

Through systematic experimentation in a controlled environment—leveraging tools such as Mininet, Wireshark, iperf, and a Linux kernel, it was observed that the default behavior of MPTCP is suboptimal in heterogeneous path scenarios. Specifically, the default MPTCP stack tends to underutilize subflows with smaller MTUs due to its unified Maximum Segment Size (MSS) selection strategy. This selection mechanism, which is based on predicted throughput across all active subflows, favors a single MSS that theoretically maximizes overall performance. However, in practice, this often leads to the exclusion of subflows that cannot accommodate the selected MSS, resulting in poor subflow utilization and limiting the core benefits of multipath transmission such as improved throughput and reliability.

To address this limitation, kernel level modifications were introduced to explicitly initialize path MTU probing on individual MPTCP subflows. Although full integration of Path MTU Discovery (PMTUD) remains challenging, this adjustment allowed the stack to fall back more effectively to lower MSS values when necessary, thereby improving overall performance in the presence of MTU constraints. Packet captures and throughput measurements confirmed better engagement of multiple subflows and more efficient data distribution. Further testing also explored the role of the TCP base MSS setting. By experimenting with different base MSS values, it was found that performance can be modestly improved in fallback scenarios. Although this tuning alone does not fully resolve the limitations posed by heterogeneous MTUs, it offers a partial mitigation strategy when subflows are forced to revert to baseline segment sizes due to path MTU-related issues.

This thesis highlights that while MPTCP's core mechanisms, such as data sequence numbering and out-of-order reassembly, are robust, its scheduler and buffer management logic are not yet fully optimized for environments where path MTUs vary significantly. The need for per-subflow MSS awareness, smarter data fragmentation, and adaptive retransmission logic is clear. These improvements would allow MPTCP to more effectively leverage the full range of available paths, enhancing throughput, reducing latency, and improving fairness across network conditions.

In conclusion, the work presented in this thesis provides valuable insights into the behavior and limitations of MPTCP in real-world heterogeneous networks. It lays the groundwork for future research in MSS-aware scheduling, per-subflow segmentation strategies, and enhanced retransmission handling. Continued efforts in this direction could significantly improve MPTCP's performance, particularly in emerging applications where multi-interface devices and diverse network paths are increasingly common.

13. Future Work

This study has demonstrated that Multipath TCP (MPTCP) performance is significantly affected by subflows with heterogeneous path MTUs and that current Linux implementations face key challenges such as incomplete Path MTU Discovery (PMTUD) and non-optimal Maximum Segment Size (MSS) handling. Based on the findings and limitations encountered, several future directions are proposed to enhance MPTCP's adaptability and efficiency in heterogeneous network environments. One promising area is the development of a new MSS handling mechanism that supports per-subflow MSS negotiation, allowing the scheduler to dynamically assign data segments based on each path's MTU and real-time performance metrics. Additionally, improvements to the PMTUD mechanism in the MPTCP context are essential. This includes refining how ICMP "Fragmentation Needed" messages are processed so that MSS updates are not indefinitely deferred, potentially through non-blocking update mechanisms or temporary suspension of subflow transmissions. Another direction is the implementation of an adaptive MTU probing mechanism for MPTCP, which can dynamically discover MTUs per subflow even during active connections. Enhancing the MPTCP scheduler to be MTU-aware—alongside existing RTT and congestion considerations—could significantly reduce packet overhead and optimize subflow utilization. Furthermore, conducting a cross-platform analysis of MTU handling in MPTCP implementations on other operating systems like FreeBSD, Windows, or mobile platforms could uncover broader insights and contribute to future standardization. Finally, expanding the current testbed to include real-world deployment scenarios such as wireless, mobile edge, and data center networks would allow for comprehensive validation of these enhancements under more dynamic and diverse conditions.

References

Adarsh, V., Schmitt, P. & Belding, E. (2019), Mptcp performance over heterogenous subpaths, *in* '2019 28th International Conference on Computer Communication and Networks (ICCCN)', IEEE, pp. 1–9.

Asiri, M. (2021), 'Novel multipath tcp scheduling design for future iot applications'.

Custura, A., Fairhurst, G. & Learmonth, I. (2018), Exploring usable path mtu in the internet, *in* '2018 Network Traffic Measurement and Analysis Conference (TMA)', IEEE, pp. 1–8.

Deering, D. S. E. & Mogul, J. (1990), 'Path MTU discovery', RFC 1191.
URL: *https://www.rfc-editor.org/info/rfc1191*

Eddy, W. (2022), 'Transmission Control Protocol (TCP)', RFC 9293.
URL: *https://www.rfc-editor.org/info/rfc9293*

ESNet & Laboratory, L. B. N. (2023), 'iperf3'. Successor to iPerf2, maintained by ESNet.
URL: *https://iperf.fr/*

Feng, X., Li, Q., Sun, K., Xu, K., Liu, B., Zheng, X., Yang, Q., Duan, H. & Qian, Z. (2022), Pmtud is not panacea: Revisiting ip fragmentation attacks against tcp.

Ford, A., Raiciu, C., Handley, M. J., Bonaventure, O. & Paasch, C. (2020), 'TCP Extensions for Multipath Operation with Multiple Addresses', RFC 8684.
URL: *https://www.rfc-editor.org/info/rfc8684*

Julaihi, A. A. (2011), A Fragmentation Control Approach in Jumbo Frame Network, PhD thesis, Universiti Teknologi Malaysia.

Lahey, K. (2000), ‘TCP Problems with Path MTU Discovery’, RFC 2923.
URL: *https://www.rfc-editor.org/info/rfc2923*

Mathis, M. & Heffner, J. (2007), ‘Packetization Layer Path MTU Discovery’, RFC 4821.
URL: *https://www.rfc-editor.org/info/rfc4821*

Maxwell, C. N. (2023), ‘Multipath tcp, and new packet scheduling method’, *Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal* **10**(1), 7.

Mogul, J. C. & Deering, S. E. (1990), ‘Rfc1191: Path mtu discovery’.

Paasch, C. et al. (2014), ‘Improving multipath tcp’, *Diss. Universit’e catholique de Louvain (UCL), London* .

Project, M. (2025), ‘Mininet: An instant virtual network on your laptop (or other pc)’, `https://mininet.org/`.
URL: *https://mininet.org/*
ΩÖzcan et al.

Özcan, Y., Guillemin, F. & Houzé, C. (2017), Fast and smooth data delivery using mptcp by avoiding redundant retransmissions, *in* ‘2017 IEEE International Conference on Communications (ICC)’, pp. 1–7.