NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

MULTIPATH TRANSPORT FOR VIRTUAL PRIVATE NETWORKS

by

Daniel Lukaszewski

March 2017

Thesis Co-Advisors: Geoffrey Xie
Justin P. Rohrer

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2017	Master's Thesis 03-19-2015 to 03-31-2017

4. TITLE AND SUBTITLE

MULTIPATH TRANSPORT FOR VIRTUAL PRIVATE NETWORKS

5. FUNDING NUMBERS

6. AUTHOR(S)

Daniel Lukaszewski

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Virtual Private Networks (VPNs) are designed to use the Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) to establish secure communication tunnels over public Internet. Multipath TCP (MPTCP) extends TCP to allow data to be delivered over multiple network paths simultaneously. This thesis first builds a testbed and investigates the potential of using MPTCP tunnels to increase the goodput of VPN communications and support seamless mobility. Based on the empirical results and an analysis of the MPTCP design in Linux kernels, we further introduce a full-multipath kernel, implementing a basic Multipath UDP (MPUDP) protocol into an existing Linux MPTCP kernel. We demonstrate the MPUDP protocol provides performance improvements over single path UDP tunnels and in some cases MPTCP tunnels. The MPUDP kernel should be further developed to include more efficient scheduling algorithms and path managers to allow better performance and mobility benefits seen with MPTCP.

14. SUBJECT TERMS

MPTCP, multipath TCP, MPUDP, multipath UDP, VPN, OpenVPN, mobility

15. NUMBER OF PAGES 81

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

MULTIPATH TRANSPORT FOR VIRTUAL PRIVATE NETWORKS

Daniel Lukaszewski
Lieutenant, United States Navy
B.S., University of Arizona, 2010

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2017

Approved by: Geoffrey Xie
Thesis Co-Advisor

Justin P. Rohrer
Thesis Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Virtual Private Networks (VPNs) are designed to use the Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) to establish secure communication tunnels over public Internet. Multipath TCP (MPTCP) extends TCP to allow data to be delivered over multiple network paths simultaneously. This thesis first builds a testbed and investigates the potential of using MPTCP tunnels to increase the goodput of VPN communications and support seamless mobility. Based on the empirical results and an analysis of the MPTCP design in Linux kernels, we further introduce a full-multipath kernel, implementing a basic Multipath UDP (MPUDP) protocol into an existing Linux MPTCP kernel. We demonstrate the MPUDP protocol provides performance improvements over single path UDP tunnels and in some cases MPTCP tunnels. The MPUDP kernel should be further developed to include more efficient scheduling algorithms and path managers to allow better performance and mobility benefits seen with MPTCP.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

BALIA Balanced Linked Adaptation

DSS Data Sequence Signal

Gbps Gigabits per second

HMAC Keyed-Hash Message Authentication Code

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IP Internet Protocol

IPSec Internet Protocol Security

L2TP Layer 2 Tunneling Protocol

LIA Linked Increase Algorithm

MAC Message Authentication Code

Mbps Megabits per second

MOSH Mobile Shell

MPTCP Multipath Transmission Control Protocol

MPUDP Multipath User Datagram Protocol

MSS Maximum Segment Size

OLIA Opportunistic Linked Increases Algorithm

OSI Open Systems Interconnection

PPTP Point to Point Tunneling Protocol

RTO Retransmission Timeout

RTT	Round Trip Time
SACK	Selective Acknowledgement
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VPN	Virtual Private Network

Acknowledgments

Liz and Dillon: Your patience and understanding were essential to the completion of this thesis. I love you both so much!

Professor Xie: I appreciate all the time you spent with me working out ideas and helping me learn how protocols work at the kernel level. Our weekly meetings really kept me on track and allowed me to finish.

Professor Rohrer: I appreciate all your help with making sure my methodology was correct and helping scope research projects that contributed to my thesis.

Henry: You really helped me get started with using MPTCP and working with Linux. I would have been lost without your guides for installing and using MPTCP.

Friends in the Network Lab: All of you graciously listened to me ramble on about MPTCP. Being able to bounce ideas off you and your assistance in helping me troubleshoot are greatly appreciated.

The Lunch Club: It is easy for me to get wrapped up in my work and work through lunch. Thanks for helping me get my mind off of work and providing stimulating conversations.

P.S. Dillon is awesome 

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

The use of Virtual Private Networks (VPNs) has become common practice in many organizations. This need for VPNs stems from not only the desire to connect remote offices and personnel to local infrastructure without the use of dedicated physical connections, but also a growing need for secure communications to mitigate cyber threats. VPNs are easily built on top of the physical network and can be utilized in various configurations to achieve confidentiality and integrity of communications.

Increased security and the desire to bypass physical connections results in additional overhead in the transport layer. VPNs introduce this additional overhead through the use of secure tunnels [1]. The level of security desired for the tunnel will drive the level of encryption needed, which in turn will affect the amount of overhead added. These tunnels are built using either the User Datagram Protocol (UDP) or Transmission Control Protocol (TCP). UDP is the default protocol used by many VPNs since it provides minimal overhead, but in some instances there may be a need for increased reliability provided by a TCP connection. When a TCP tunnel is used, there exists the potential of performance degradation. The majority of Internet traffic uses TCP so it is safe to assume the traffic flowing through the established VPN tunnel will be TCP traffic. This layering of a TCP tunnel with a TCP connection can lead to drops in performance due to two layers of congestion control acting on the connection. TCP congestion control algorithms are designed to achieve maximal performance by controlling the amount of traffic on a connection. When a packet is lost in transmission or a triple-duplicate ACK is received, TCP relates this to a congested connection and will perform a back-off, resulting in smaller packets being sent. When a TCP connection is established within a TCP tunnel, each layer of TCP establishes Round Trip Time (RTT) timers and sets a Retransmission Timeout (RTO) parameter to aid in determining if packets are lost. Olaf Titz [2] describes a scenario where the outer TCP layer has a shorter RTO than the inner layer, which results in unnecessary timeouts and retransmissions. To counter the perceived congestion caused by these timeouts and retransmissions, smaller packets will be transmitted, which in turn results in a performance drop.

In "Understanding TCP over TCP: Effects of TCP Tunneling on End-to-End Throughput and Latency," the authors discuss TCP parameters that may be specified to reduce TCP over TCP complications [3]. One TCP parameter that has become standard in Linux is the TCP Selective Acknowledgement (SACK) option. This option is designed to allow the client to send an ACK message for specific packets resulting in only needing to retransmit missing packets instead of multiple packets in which some may not have been missing. Honda et al. were able to show that through the use of the TCP SACK option on the tunnel TCP flow, the goodput experienced increased [3]. Another adjusted parameter that can affect TCP connections is the buffer size of the socket being used. By setting a small buffer size, the incoming TCP packets will need to be processed quickly or the buffer will overflow and the packets will be dropped. Having an excessively large buffer could result in packets sitting in the buffer longer than the RTO value set on the link. This would result in unnecessary retransmissions due to a perceived loss that was not present. Honda et al. determined that setting socket buffer sizes of both the end-to-end tunnels and the TCP tunnel to the bandwidth-delay product of the link also produced favorable results [3]. Despite these improvements, the effects of TCP over TCP are still present and provide a hindrance to using TCP with VPNs.

1.1 Problem Statement

The additional overhead introduced when using a VPN often results in decreased performance depending on RTTs and loss rates across the network. To enhance traditional TCP connections, companies such as Apple have started using Multipath Transmission Control Protocol (MPTCP) on iOS devices [4]. Additionally, MPTCP is supported in the latest Linux distributions [5]. MPTCP is an enhancement built into the operating system kernel to allow hosts to use multiple sockets in a TCP session. The number of sockets utilized is a factor of the number of interfaces available to the host as well as the path manager specifications set by the host. Multipath protocols such as MPTCP will allow hosts with multiple interfaces to multiplex traffic and aggregate bandwidth over the additional interfaces. The use of multipath protocols within VPNs has the potential to overcome the decreased performance seen when using standard protocols, thus removing any hindrance in using a VPN to provide secure communications.

1.2 Research Questions

Minimal studies have been conducted regarding the use of multipath transport protocols with VPNs. Additionally, the use of Multipath User Datagram Protocol (MPUDP) within a VPN has yet to be researched to the best of our knowledge. Specifically, this research aims to answer the following question:

- TCP-encapsulated-in-TCP is likely to cause a slow down of data transfer due to redundant applications of congestion control. Will using MPUDP in a VPN improve data transfer rates?

Through our study of VPNs and multipath protocols, we look to answer the following additional research questions:

- How can we integrate MPUDP into an existing VPN with a minimum amount of modifications to the existing configuration?
- Does a plug-and-play solution exist or will it be necessary to modify the VPN framework to support it?
- How does the use of a VPN impact the performance of MPTCP? MPTCP enhances a standard TCP network, but how will MPTCP behave within a VPN?

1.3 Thesis Organization

The organization of this thesis is as follows. Chapter 2 begins by discussing how VPNs are used and the different layers they may be implemented in. Then we discuss a specific VPN, OpenVPN, and how Secure Sockets Layer (SSL)/Transport Layer Security (TLS) is used to accomplish ease of implementation and secure communications. Afterwards, we discuss MPTCP in detail to include how a connection is established, path managers available, scheduler used, and the congestion control algorithms available. Lastly, we discuss related work researching the use of MPTCP and VPN in a gateway-to-gateway environment, using MPTCP to permit mobility, and how MPUDP is being used to improve Secure Shell (SSH) connections. Chapter 3 investigates the implementation of MPTCP in a VPN. Specific emphasis is placed on performance differences with symmetric/asymmetric link conditions and the effect of other link characteristics such as propagation latency and packet loss rate. After performance evaluations are completed, we discuss potential benefits in regards to

mobility when using MPTCP in a VPN. Chapter 4 details the implementation of MPUDP into the MPTCP Linux kernel. A proof of concept evaluation is then performed using MPUDP in a VPN. Lastly, in Chapter 5 we discuss our conclusions and the potential for future work.

CHAPTER 2:
Background

This chapter provides the necessary background information pertaining to VPNs and MPTCP. We begin by providing an overview of how VPNs are commonly used and methods of implementing them. We then discuss one particular VPN, OpenVPN, and how it operates to include providing secure communications and configuration parameters. Once the reader obtains a high-level understanding of OpenVPN, we discuss the MPTCP protocol to include modifications employed to enable behavior similar to the well known TCP protocol, handshake process, available path managers, packet schedulers, the different congestion control algorithms, and how MPTCP interacts with OpenVPN. We provide a brief overview on how to install and use MPTCP as well. Lastly, we introduce related work to include the MPTCP Binder implementation, how MPTCP affects mobility, and a MPUDP application layer implementation.

2.1 VPNs

VPNs are established for various reasons. It might be needed to enable a client to connect to his corporate offices using a hotel Wi-Fi connection while traveling for business. Maybe a small business is branching out to a new geographical location and needs to maintain secure communications between the two locations. These are just two of the many scenarios that may warrant setting up a VPN. There are three common configurations for a VPN:

- gateway to gateway
- host to gateway
- host to host

The gateway-to-gateway model connects one network gateway to another network gateway without the need for a physical connection. This configuration may be used to connect remote offices of a corporation to the local offices and maintain secure communications as if the remote offices were part of the local area network. The host-to-gateway model is very similar. Instead of using a network gateway router, the host machine becomes the gateway to establish the connection with the desired VPN server's gateway. The final and

least employed model utilizes each host machine as the network gateway and establishes a secure connection between the hosts. This method is not very common due to easier methods, such as SSH, to accomplish the same goal. The focus of this thesis is the host to gateway model, but the concepts can be easily abstracted to the alternate configurations.

2.1.1 Network Layer Implementation

There exist several different methods of implementing a VPN. VPNs are built within the existing 7-Layer Open Systems Interconnection (OSI) model and can be implemented in the data link (2), network (3) , or transport (4) layers [1]. Layer 2 Tunneling Protocol (L2TP) and Point to Point Tunneling Protocol (PPTP) VPNs exist at the data link layer. PPTP VPNs are incorporated in base operating systems allowing for ease of implementation, but these protocols have known security vulnerabilities. The L2TP alone does not provide encryption or confidentiality and is typically used with another protocol, such as Internet Protocol Security (IPSec), to provide a secure tunnel [1]. IPSec is implemented at the network layer and provides confidentiality, integrity, and authentication. IPSec VPNs are most commonly used in the gateway-to-gateway model. Additionally, IPSec VPNs are quite complex and require integration into the operating system kernel. This can lead to problems with interoperability since each system must be configured correctly to support the VPN [1]. Transport layer VPNs may be established using the application layer. One such VPN could be made using an SSH tunnel. An SSH tunnel is quick to establish and can be useful for conducting tasks via the command line interface, but its operation at a higher OSI layer leads to limited uses. This thesis will focus on using OpenVPN, which operates in either the data link layer or network layer [1]. OpenVPN is a well-known VPN and should allow the implementation of multipath protocols without the need for changes to the VPN architecture.

2.1.2 OpenVPN

OpenVPN is a well known open sourced VPN. It is a feature-rich data link or network layer VPN that utilizes SSL/TLS for encryption [1]. Unlike the complex network layer IPSec VPN, OpenVPN is easy to install and customize/alter to fit the configuration needed. Since its initial release in 2001, OpenVPN has gone through many updates in order to provide more secure communications, optimize performance, and add additional features.

OpenVPN utilizes the client/server model and can support multiple clients via the same server operating on the same port [1]. Early versions of OpenVPN set default send and receive buffers of 64K bytes. This setting caused performance degradations, which resulted in version 2.3.9 removing the default and allowing the host operating system to set these values on tunnel creation. Depending on the host operating system being used, OpenVPN may install a stable version prior to 2.3.9, thus it is recommended to update to the latest version available. OpenVPN is able to use a variety of encryption standards, such as Blowfish and AES. Users should take note that the choice of certain encryption algorithms can add additional overhead above the defaults and result in slower tunnels. To accomplish this encryption, OpenVPN uses SSL/TLS. SSL/TLS has been utilized since the 1990s, is a well known method of providing secure communications over a public network, and is most commonly used for HTTPS connections. Thus its use in OpenVPN lends to the ease of employment and security.

2.1.3 How OpenVPN Establishes Connections

This section goes into detail on how SSL/TLS is used in OpenVPN to create a secure tunnel connection. An OpenVPN connection begins with a handshake between the VPN client and VPN server. An overview of this process is provided in Figure 2.1.

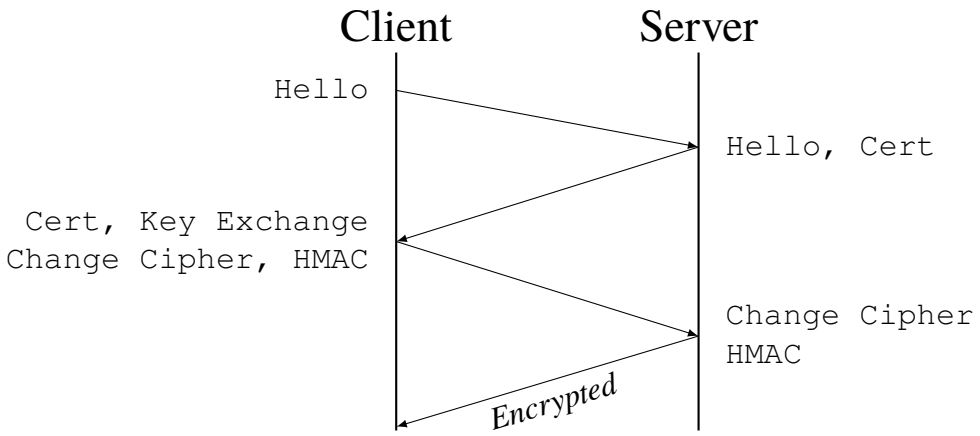


Figure 2.1: Simplified TLS Handshake

The connection begins with the client sending a *hello* message containing a list of cryptographic algorithms it can support as well as a nonce to prevent replay attacks. The server

will then send a *hello* message containing the symmetric algorithm, public key algorithm, and Message Authentication Code (MAC) algorithm to be used. Additionally, the server will provide a digital certificate along with a nonce and request the client's digital certificate. After the client receives the *hello* message, the client will verify the digital certificate of the server, retrieve the server's public key and generate a pre-master secret. Once completed, the client will send the generated secret to the server. At this point the client and server will compute the master secret and generate two encryption and two MAC keys from the secret. From this point on, all further traffic for this connection will be encrypted using the keys. The client will then send a MAC of all the handshake messages, which will be followed by the server doing the same. This final step is used to prevent a man in the middle attack. If the MACs do not match, the connection will be terminated and the handshake will be required to be performed again. After completion of the SSL/TLS handshake, the authentication of the client and encryption parameters have been established for the OpenVPN connection.

2.1.4 OpenVPN Configuration Parameters

For each endpoint of the VPN, configuration files are used to store several parameters for OpenVPN setup. Several of these configuration parameters will be covered in this section. Parameters present in both client and server configurations include the certificates necessary to conduct the initial handshake specified in Section 2.1.3. The server will specify the IP range used for the virtual network, which may be outside the IP range of the local area network of the server. The class A private address space is often used, such as 10.8.0.0/24. The server must also specify the parameters for the keep-alive settings, if being utilized. Keep-alive will send a ping at set intervals and wait a specified amount of time for a response. If no response is received, the tunnel is perceived to be down and the client will need to establish the connection again. The client is often configured to re-establish connections automatically, thus alleviating the need for interaction with the user. The server may also specify the use of a persistent tunnel. When this parameter is set, the server will not tear down the tunnel if a connection restart message is received. This is useful for intermittent links to ensure client VPN addresses are not unnecessarily changed. The last set of parameters often specified by the server is the use of push messages to disable split tunneling and ensure all IP traffic is sent through the tunnel.

2.1.5 OpenVPN Traffic

After the VPN has been established and the desired configuration parameters have been passed, the user may begin using the VPN for various types of traffic to include web traffic. Figure 2.2 illustrates a general timing diagram of traffic between a VPN client and a web server. Each packet in the timing diagram is also illustrated to give a view of the encapsulation and decapsulation throughout the IP, transportation, and application layers.

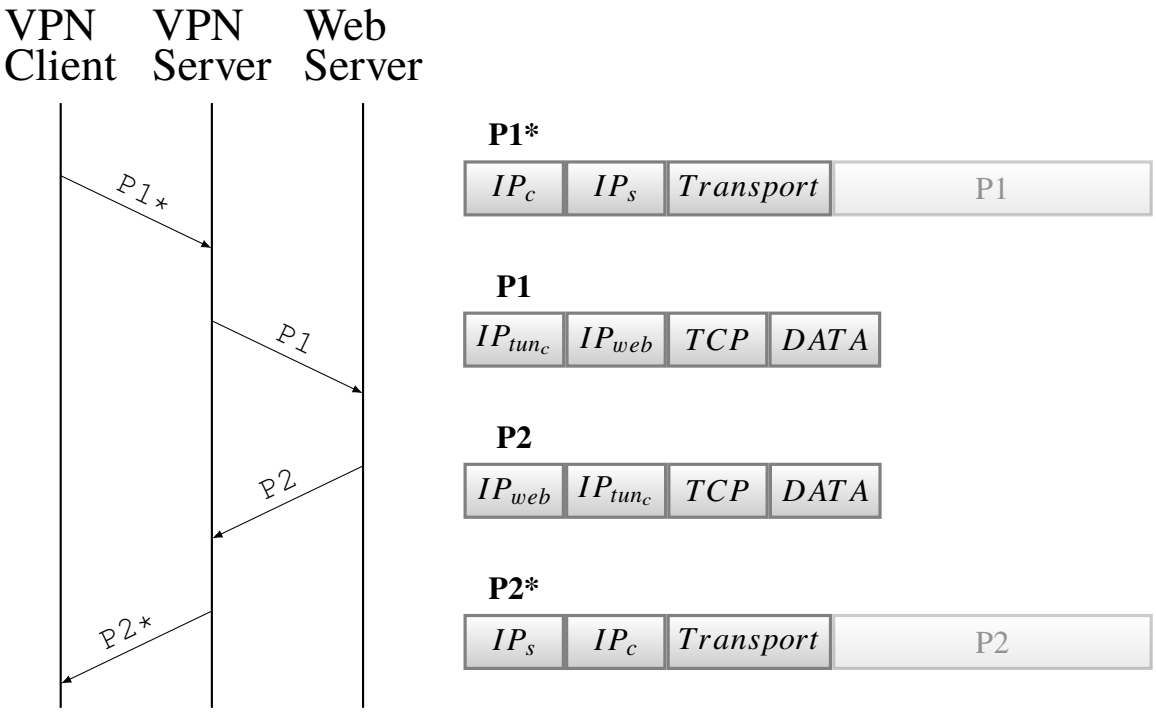


Figure 2.2: Sample VPN Traffic

Packet P1* is the outer packet carrying P1 as a payload and may use UDP or TCP as the transport protocol. In fact, we will even use MPTCP or MPUDP as the transport protocol for P1*. After P1* arrives at the VPN server, the packet is decrypted and P1 is analyzed to determine its final destination. Since P1 is not destined for the VPN server, the packet is then forwarded to its destination with a source address of the VPN client. The reverse process for packet P2 is similar.

2.2 MPTCP

MPTCP is an emerging technology that is still under-utilized in current networking designs. As such, MPTCP was built to be backwards compatible with TCP in order for it to have the best chances possible to be adopted into practice [6]. To achieve backward compatibility as well as application independence, Figure 2.3 illustrates modifications made to the transport protocol stack.

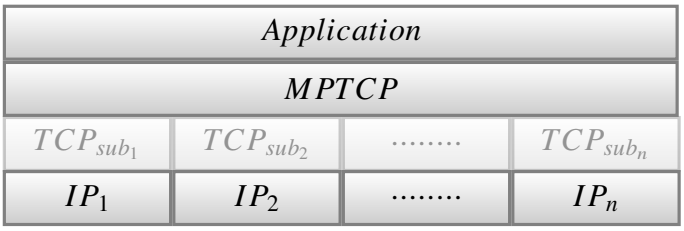


Figure 2.3: MPTCP Protocol Stack

Additional changes were also made to the TCP header-options field to signal the use of MPTCP. Figure 2.4 provides a Wireshark view of the option utilized to enable MPTCP communications, specifically a MP_JOIN connection. A *Kind* field value of 30 has been designated by the Internet Assigned Numbers Authority to indicate the use of MPTCP. The subtype field varies depending on the purpose of the MPTCP message, such as: initiating the connection, joining a connection, providing additional addresses, and data sequence mapping [6].

```
▼ Options: (32 bytes), Maximum segment size, SACK permitted, Timestamps
  ▶ Maximum segment size: 1460 bytes
  ▶ TCP SACK Permitted Option: True
  ▶ Timestamps: TSval 4294920192, TSecr 0
  ▶ No-Operation (NOP)
  ▶ Window scale: 7 (multiply by 128)
  ▼ Multipath TCP: Join Connection
      Kind: Multipath TCP (30)
      Length: 12
      0001 .... = Multipath TCP subtype: Join Connection (1)
    ▶ Multipath TCP flags: 0x10
      Address ID: 3
      Receiver's Token: 4004006579
      Sender's Random Number: 1729002163
```

Figure 2.4: MPTCP Option to TCP Header

This section details the three-way handshake to include establishing additional connections called subflows, path managers, packet schedulers, and congestion control algorithms. OpenVPN interaction with MPTCP will also be discussed.

2.2.1 Connection Process

The initial MPTCP handshake is designed to be like the traditional TCP handshake. The client will initiate a handshake with the server by using the MP_CAPABLE subtype in the MPTCP option section. Along with the subtype being set, the client will also include a 64-bit key that will be used to generate a 32-bit token for use in authenticating all communications after the handshake is completed. Upon receiving the initial handshake message, the server will perform one of two actions. If the server is not MPTCP capable, the handshake will seamlessly fall back to a single path TCP connection. If the server is MPTCP capable, then the server will send a SYN-ACK message along with a 64-bit key. The client then finishes the connection by replying with an ACK message along with both the client and server's keys.

Once the handshake has been completed, additional subflows are created using two different methods. The primary method for establishing additional subflows is through the MP_JOIN option [6]. If the client is multi-homed or wishes to establish additional socket pairs on a single interface, the client will conduct another handshake using the MP_JOIN option in lieu of the MP_CAPABLE option. When establishing this new connection, the token from the initial handshake is used along with nonces and HMAC algorithms to prevent security breaches. Figure 2.5 illustrates the process of initial handshake to additional subflow generation using the MP_JOIN option.

Another method of creating additional subflows is through the use of the ADD_ADDR option [6]. This option, sent by the client or server, advertises additional interfaces available for establishing subflow connections, to include virtual IP addresses. Once the client or server learns of the additional IP addresses, additional subflows will be established using the MP_JOIN process described earlier.

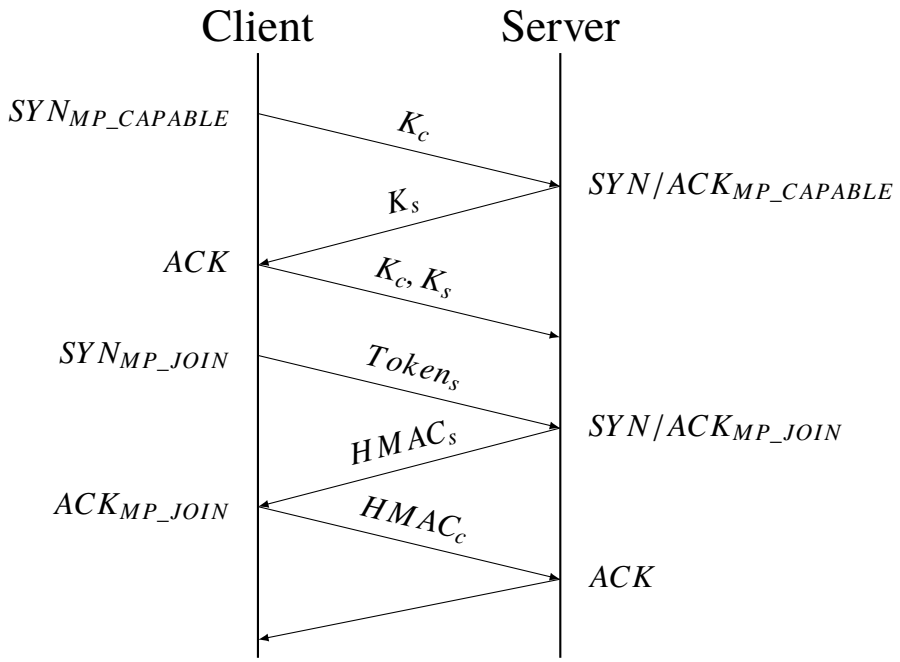


Figure 2.5: MPTCP Initial Connection and Subsequent Join Request

2.2.2 Path Managers

The number of subflows created is determined by the path manager being used by the client. This thesis focuses on two of the existing path managers, ndiffports and fullmesh.

Ndiffports

The ndiffports path manager was developed to "exploit the equal costs multiple paths that are available in a datacenter" [7]. Additionally, ndiffports allows clients with only one interface to be able to utilize MPTCP. Once an initial connection is established, ndiffports will create multiple subflows using the same source and destination IP addresses through the use of different source port numbers. The number of socket pairs to be created is a tunable parameter set by the client. Figure 2.6 provides a simplified illustration of this path manager.

In the paper "Improving Datacenter Performance and Robustness with Multipath TCP," the authors discuss how ndiffports may be implemented in data centers to take advantage of equal cost paths present within the data center [8]. Outside of data center usage, ndiffports is

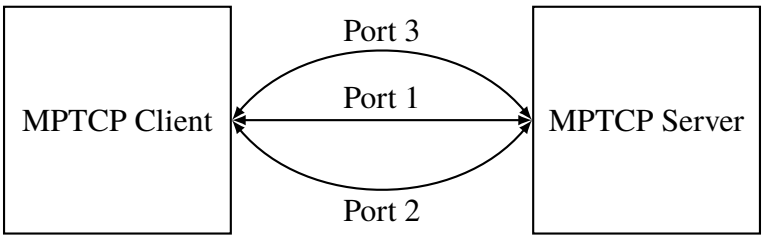


Figure 2.6: Ndiffports Using Three Subflows

not recommended for practical use and should only be used as an example path manager [7]. Contrary to this recommendation, we believe the use of ndiffports in a VPN tunnel may mitigate the TCP over TCP effects and yield performance benefits.

Fullmesh

The fullmesh path manager is designed to create all possible connections between the client and the server by attempting to connect each interface of the client to each interface of the server [7]. Figure 2.7 provides a simplified scenario of a MPTCP client with Wi-Fi and cellular interfaces connected to a MPTCP server with or without the same interfaces.

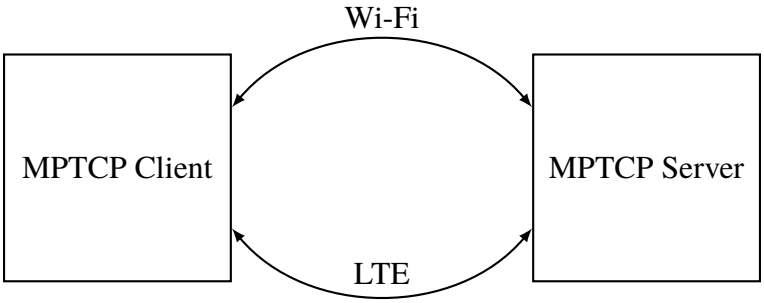


Figure 2.7: Fullmesh Using a Wi-Fi and Cellular Connection

The client is able to tune this path manager to specify the number of subflows to be created on each set of IP addresses, thus combining ndiffports and fullmesh if desired. It should be noted that this path manager will also attempt to use virtual IP addresses to establish additional connections. Although this may be desirable, potential drawbacks exist if not used properly. For instance, when using inside of a VPN, the virtual VPN IP addresses of the client and server will initiate connections to the real IP addresses of the client and server. This may cause routing problems due to the VPN having routes not traditionally used.

2.2.3 Data Sequencing

The use of multiple subflows brings about the need to sequence the data received so that it may be reassembled in the correct order and delivered to the application layer. To accomplish this, MPTCP uses the Data Sequence Signal (DSS) option. Sequence numbers of each subflow are relative to that specific subflow and thus are not useful in determining the proper sequence of the overall flow of data. The DSS option is a 64-bit value used to map subflow sequence numbers to master sequence level or master flow [6]. Besides ensuring proper integration of subflows into the correct order for deliver, the DSS option is also used for acknowledging delivery of packets and signaling the closing of a MPTCP connection [6]. Without the use of this option, MPTCP would not be able to function independently and would cause problems with applications using TCP.

2.2.4 Packet Schedulers

The use of MPTCP also requires a specialized packet scheduler to be used to maximize performance with multiple subflows. The default scheduler, which prioritizes subflows based on RTT, is the best-known option and is the only recommended scheduler for practical use [5]. Another available packet scheduler is the round-robin scheduler. This scheduler sends packets over each subflow in a round-robin fashion in an attempt to balance traffic across subflows. The round-robin scheduler is only useful for academic research and not recommended for practical use [5]. MPTCP v0.91 introduced a new packet scheduler, known as redundant. This scheduler sends packets in a redundant manner across available subflows in order to minimize the latency experienced while sacrificing additional bandwidth that may be available [5]. This thesis utilized the default packet scheduler in order to focus on a practical use of MPTCP within a VPN.

2.2.5 Congestion Control Algorithms

TCP congestion control algorithms aim to achieve the maximum throughput possible on a connection without putting too much traffic on the link as to adversely affect link performance. Well-known single path TCP congestion control algorithms include TCP Reno and Cubic. TCP Reno is an ACK based algorithm in that it increases the congestion window based on receipt of ACK messages. In the event of a triple-duplicate ACK, the congestion window is reduced by one half and the congestion avoidance phase begins again. In the

event of a timeout, more severe restrictions are applied. Instead of dropping by half of the current value, the congestion window is set to 1 Maximum Segment Size (MSS) and the slow start phase begins. Conversely, TCP Cubic does not use the receipt of ACK messages to adjust the congestion window. Cubic will quickly increase the congestion window to the last known threshold before a congestion event occurred. Once a stabilization phase occurs, Cubic will probe the link to determine if additional bandwidth is available. Since Cubic does not rely on ACK messages, performance is improved in links with higher latency as compared to ACK based algorithms. While the default TCP congestion control algorithm in Ubuntu is Cubic, this protocol is not designed for use in MPTCP connections. The reason for this is due to the potential to be unfair to single path TCP sessions. For instance, if a MPTCP user was using Cubic with multiple subflows and there is a shared bottleneck with a single path TCP user, the MPTCP user would obtain a larger fraction of the resources [9], [10]. We decided to evaluate the performance of using the Cubic congestion control algorithm with a testbed that did not have a shared bottleneck between the VPN client and VPN server. The ndiffports path manager was included as well to provide similarity between tests, but using this path manager with Cubic would lead to unfair sharing of resources.

Several congestion control algorithms have been designed to optimize MPTCP while preserving fairness to single path TCP connections by coupling the subflows being used [9]. The default congestion control algorithm for MPTCP is Opportunistic Linked Increases Algorithm (OLIA). OLIA is designed as an improvement to the original Linked Increase Algorithm (LIA). LIA can suffer from performance issues as well as being aggressive to single path TCP users [11]. OLIA was designed to provide congestion balancing and responsiveness simultaneously in order to overcome the limitations of LIA [11]. The OLIA algorithm was not included in this thesis due to suffering performance issues that have been improved upon with the Balanced Linked Adaptation (BALIA) algorithm.

BALIA was chosen as the preferred algorithm for this thesis. OLIA may be slow to respond to network changes, which leads to non-optimal usage of available resources [12]. BALIA was designed to provide a trade-off between responsiveness to network changes and friendliness to other TCP flows [13]. BALIA works similar to TCP Reno with some modifications to take into account the multiple subflows. The BALIA algorithm is ACK based and for connections with a single flow, the algorithm behaves just like TCP Reno with

a minimum MSS of 2. If multiple connections are present, then BALIA sets a minimum MSS of 1 and behaves based on the following formulas [13]:

- Congestion window, w_r, increase for each ACK on path r:

$$w_r = \frac{x_r}{rtt_r * (\sum x_k)^2)} * \frac{1 + \alpha_r}{2} * \frac{4 + \alpha_r}{5} \tag{2.1}$$

- Congestion window decrease for packet loss:

$$w_r = \frac{w_r}{2} * min(\alpha_r, 1.5) \tag{2.2}$$

- $x_r = \frac{w_r}{rtt_r}$ and $\alpha_r = \frac{max(x_r)}{x_r}$

Equations 2.1 and 2.2 demonstrate the meaning of a coupled congestion control algorithms. The congestion window increases and decreases based on the performance of the other subflows being used. With the congestion window being controlled as described, BALIA was shown to be more friendly to single path TCP flows and also more responsive to network changes when compared to other MPTCP congestion algorithms [13].

2.2.6 OpenVPN MPTCP Interaction

As discussed in Section 2.1.1, OpenVPN operates at layer 2 or 3 in the OSI model. This means MPTCP can affect OpenVPN operation in unexpected ways. Figure 2.8 is a high-level view on how OpenVPN interacts with the kernel space.

When an application has data to send it is passed down to the kernel where normal routing table decisions direct the use of the VPN tunnel interface [14]. The data is then passed back up to user space for the necessary encryption and fragmenting done by OpenVPN. From this point the data is passed back down to the kernel space for delivery to the VPN server. Each time the data is passed to the kernel space, MPTCP may adjust the end destination of the data based on path managers and schedulers in use. For instance, if the user's end goal was a MPTCP web server, the first time the data is passed to kernel space the MPTCP protocol

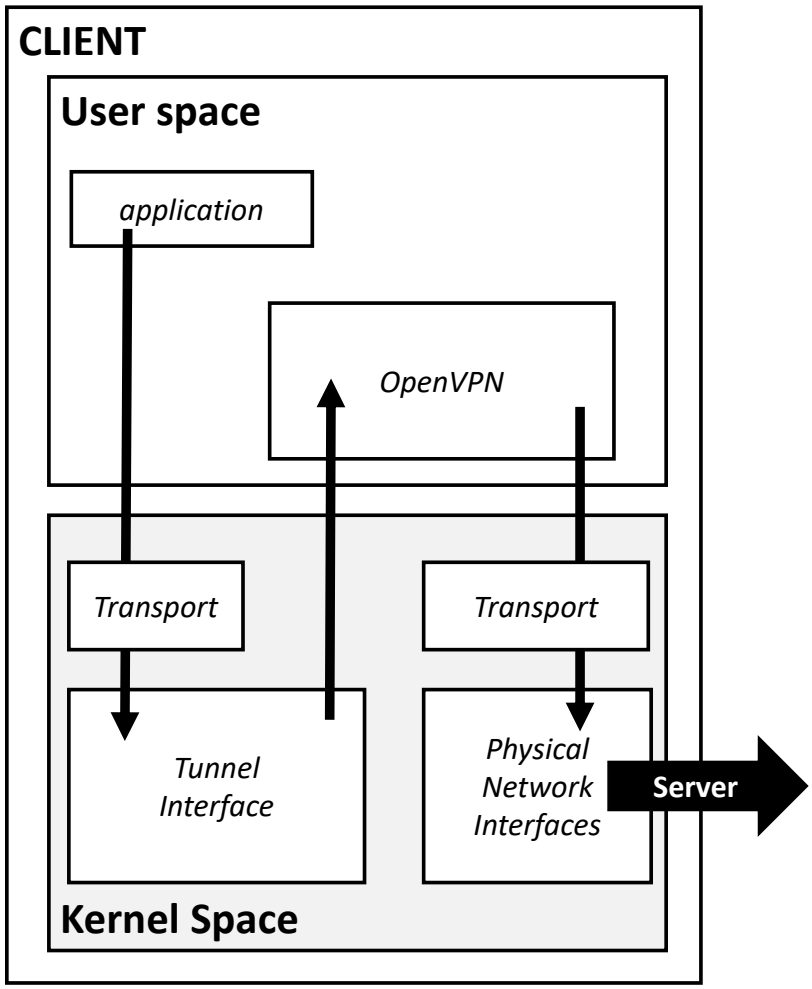


Figure 2.8: OpenVPN Interaction with Kernel. Adapted from [14].

may have a different route for the traffic than through the desired VPN tunnel. This route may have been learned through the ADD_ADDR or MP_JOIN processes. This would be an unwanted side effect of using MPTCP and was discovered to happen during familiarization testing. The ideal situation for using MPTCP with OpenVPN is to have a TCP connection between the user and end host, and a MPTCP connection between the user and VPN server. Since the data from the user enters the kernel space two times, this separation is possible.

2.2.7 Enabling MPTCP on Host Machine

The most common use of MPTCP users can install and customize is the Linux MPTCP kernel [5]. MPTCP is not available in a standard Linux operating system. Instead, MPTCP is built on top of the Linux operating system. Depending on the version of Linux installed, the version of MPTCP supported will vary. Ubuntu 14.04 and MPTCP version 0.90 were used for this thesis. A brief description of how to install and configure MPTCP will be discussed in this section. For a more in depth tutorial, the reader is encouraged to refer to [5] and Appendix B of a thesis written by Henry Foster [15].

In order to install and use MPTCP, the user may follow these basic steps:

1. Install the Ubuntu operating system, such as 14.04.
2. Add the apt-repository and key from [5].
3. Install MPTCP: `sudo apt-get install linux-mptcp`
4. Configure routing following instructions or installing scripts provided by [5].
5. Restart machine and select MPTCP kernel, holding the shift key at startup if necessary.
6. Choose desired MPTCP path manager:
   `sudo sysctl -w net.mptcp.mptcp_path_manger=INSERT_NAME`
7. Choose desired MPTCP scheduler:
   `sudo sysctl -w net.mptcp.mptcp_scheduler=INSERT_NAME`
8. Choose desired TCP congestion control algorithm:
   `sudo sysctl -w net.ipv4.tcp_congestion_control=INSERT_NAME`
9. Launch Wireshark and connect to a MPTCP website, such as [5], to verify proper operation.

2.3 Related Work

Although there has been extensive work on the development of MPTCP and the various congestion control algorithms associated with it, there has been little work involving its use with VPNs. Additionally, there has been some work developing MPUDP applications, but the implementation has not gained traction for use in additional applications. This section discusses a gateway to gateway MPTCP VPN implementation, mobility enhancements using MPTCP, and an application level implementation of MPUDP with Mobile Shell (MOSH).

2.3.1 Gateway-to-Gateway MPTCP VPN Implementation

MPTCP has an additional path manager, Binder, that was designed using the gateway-to-gateway VPN model. Bocassi et al. designed Binder to aggregate paths in a community network to leverage multipath flows between a VPN relay and VPN proxy [16]. Some of the benefits of Binder are the ability to aggregate paths to achieve greater performance and adding redundant paths between the VPN relay and proxy. Binder relies on the use of proxies and loose source routing mechanisms between gateway VPN routers in order to achieve these desired benefits [16]. Bocassi et al. used an emulated testbed (Dummynet) to demonstrate the effectiveness of Binder for this community model. Starting with MPTCP kernel v0.89, this path manager was added as an option. Binder's reliance on loose source routing mechanisms does not lend its use towards a host-to-host or host-to-gateway VPN, which is more closely related to this thesis.

2.3.2 Mobility with MPTCP

MPTCP provides a different method of adding mobility to applications than Mobile IP. Mobile IP is not optimal in the *make-before-break* environment due to its operation at the IP layer [17]. In a mobile environment it would be favorable to the user to have multiple connections established even if the extra connections are rarely utilized. In the event of a connection interruption, such as when leaving a Wi-Fi area, if the user also had a cellular connection pre-established there would be little to no interruption to service. It may not always be possible to have multiple connections established when using MPTCP. Nevertheless, if the user experiences a connection loss, the MPTCP connection will remain open, but will be in a static state [17]. Once a new connection becomes available, the user will transmit a join request and the new connection will proceed were the previous left off. In [17], the authors conducted mobility experiments through the use of simulations comparing MPTCP, Optimal TCP, and single path TCP. Optimal TCP provided an upper bound on how well Mobile IP could perform if it knew in advance the best path to utilize for a set time period [17]. It is clear from the results provided in the paper that the MPTCP implementation allowed for the best performance. This experiment was conducted again using a vehicular speed scenario with similar results [17].

In [18], the authors extend the research of [17] by implementing several different modes of operation for MPTCP. The different modes are as follows [18]:

- Full-MPTCP: Default mode intended for obtaining optimal data transfer rates.
- Backup: Through the use of the MP-PRIO option, the user can specify which interfaces to primarily transfer data. In the event the primary interface goes down, the backup interfaces can immediately respond since their connection was previously established.
- Single-Path: This mode initially establishes one connection. Upon loss of the interface, a new connection is established over another available interface. Unlike Backup mode, Single-Path will take longer before regaining data transfer.

The authors of [18] evaluated these different modes utilizing connections with a Wi-Fi and a 3G network. The experiment established a connection with an HTTP application and then dropped the Wi-Fi connection. As expected the Full-MPTCP mode recovered the fastest from the loss. The Backup mode also recovered quickly, but not as fast due to the backup interface having a lower congestion window [18]. The Single-Path and Application Handover instances took significantly longer to recover since new three-way handshakes were required prior to restoring data transfer [18]. It is expected that utilizing MPTCP with OpenVPN will result in similar enhancements to OpenVPN's mobility.

2.3.3 Multipath-MOSH

MOSH is a project from MIT designed as an alternative to the standard remote SSH shell. Unlike SSH, MOSH utilizes UDP and allows for intermittent connectivity between the client and server without the need to re-establish connections manually. Multipath-MOSH is an enhancement to MOSH to allow for the application to utilize the best flow (similar to MPTCP subflow) for data transfer [19]. The application begins by determining all available addresses for the client and server by setting flags in the MOSH header in order to establish flows for data transfer [19]. To determine which flow to use, the application sends message probes to evaluate network conditions on the link. These probes adds minimal overhead and allow for picking the optimal flow based on RTTs and losses experienced [19]. Multipath-MOSH is a good start towards using MPUDP, but its implementation is specific to the MOSH application. The ideas present in this implementation can be transferred to work

in a MPUDP kernel, which would allow all applications using UDP to take advantage of MPUDP much like the MPTCP kernel. Ideally, the MPTCP kernel should be adapted to also perform MPUDP, thus becoming a more general multipath kernel.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Performance Evaluation of MPTCP-VPN Integration

In this chapter we investigate the use of MPTCP within a VPN, specifically OpenVPN. To better understand how MPTCP interacts with OpenVPN, Figure 3.1 illustrates the connection and tunnel construction process of a VPN client and server that have MPTCP capability.

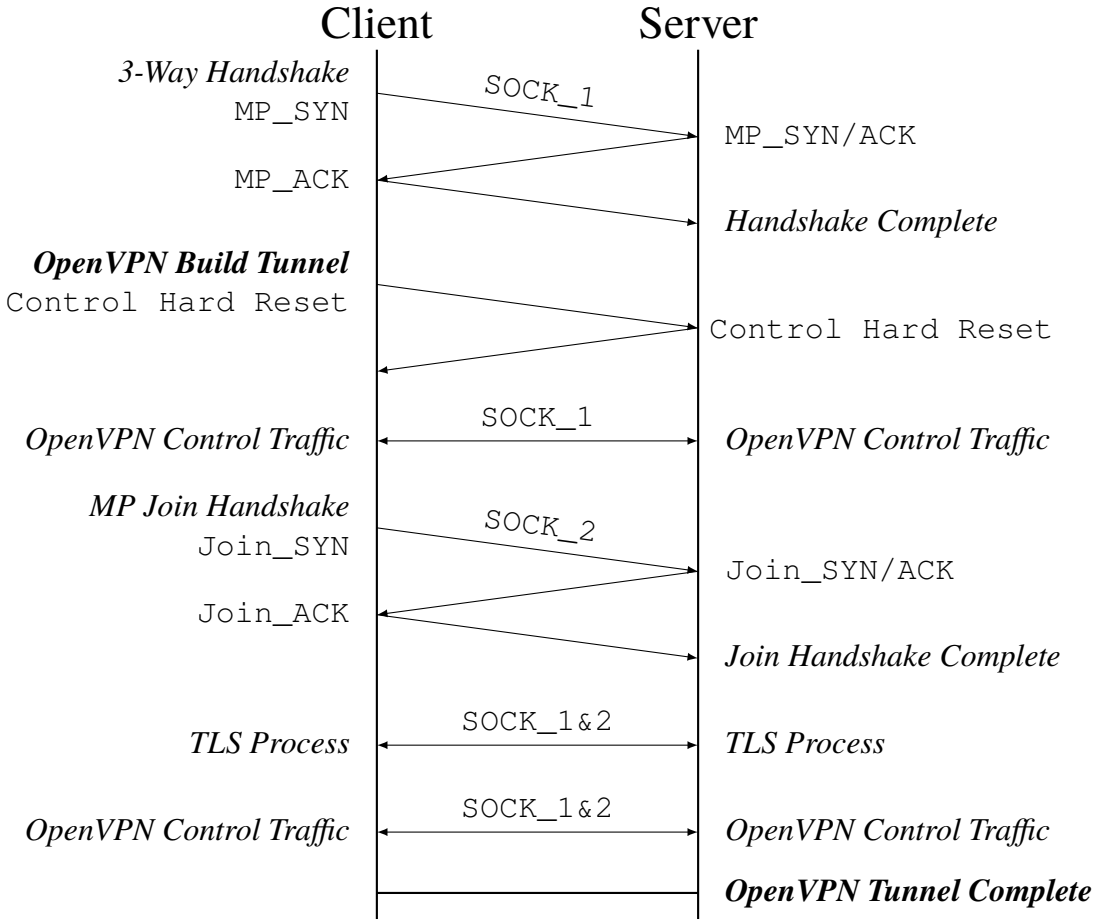


Figure 3.1: MPTCP VPN Connection Process

The VPN client begins with a MP_CAPABLE three-way handshake procedure. Upon completion, the client and server will issue hard reset messages followed by VPN control traffic to establish setup parameters. If the client has additional interfaces or ports to

use, then additional MP_JOIN three-way handshakes will be performed. Once additional subflows are established, the remaining traffic may take place using any subset of subflows. To setup the security of the VPN, the SSL/TLS process described in Section 2.1.3 is performed. The client and server will then exchange the final control traffic to finish the tunnel parameter configurations. The VPN tunnel is now built and available for use with any subset of subflows available.

VPNs are known to have performance degradations when using TCP as the tunnel protocol. We aim to discover if the TCP over TCP problems exist when using MPTCP to establish the tunnel. We begin with a simplified testbed with symmetric link qualities. Using symmetric links simplifies how the scheduler performs since each link will have the same RTTs. The testbed is then modified to have asymmetric links to see the performance differences between symmetric, moderate, and severe asymmetric connections when using the fullmesh path manager. An additional benefit of MPTCP is an increase in mobility, much like Mobile IP. Thus, after evaluating performance, we developed a separate testbed to test the mobility of the VPN when using MPTCP. The degree of mobility added to the VPN is the goal of this evaluation.

3.1 Performance Testing

In this section, we first discuss the simulation testbed topology and baseline system parameters for testing a MPTCP VPN. We then discuss the implementation of the testbed in a virtual environment and then a physical environment. After the testbed implementation is completed, performance is evaluated using fullmesh, ndiffports, and a single TCP connection with symmetric links between the VPN client and server and equal propagation RTTs and packet loss rates. TCP performance is known to suffer from the TCP over TCP effects discussed in [2] and [3], but these effects may not be as severe when using MPTCP. We hypothesize that using MPTCP to establish the VPN tunnel will result in increased performance compared to a standard TCP VPN tunnel. Additionally, we believe tests utilizing the default Linux TCP congestion control algorithm, Cubic, with MPTCP will result in unacceptable aggressive performance.

The typical use of MPTCP will be with links that are asymmetric. For example, a wireless and Ethernet or wireless and cellular connections. We test this asymmetric environment

with moderate and severe asymmetric links between the VPN client and server. This comparison also allows for insight on how the MPTCP default scheduler will perform since the amount of traffic sent on each link will vary as discussed in 2.2.4. We hypothesize the severe asymmetric tests will outperform the moderate tests since the majority of traffic will utilize the primary link and still be able to take advantage of additional bandwidth of the secondary link without suffering significant loss rates.

3.1.1 Simulation Topology

We begin with developing the topology of the testbed to meet the test goals. The conceptual setup is illustrated in Figure 3.2.

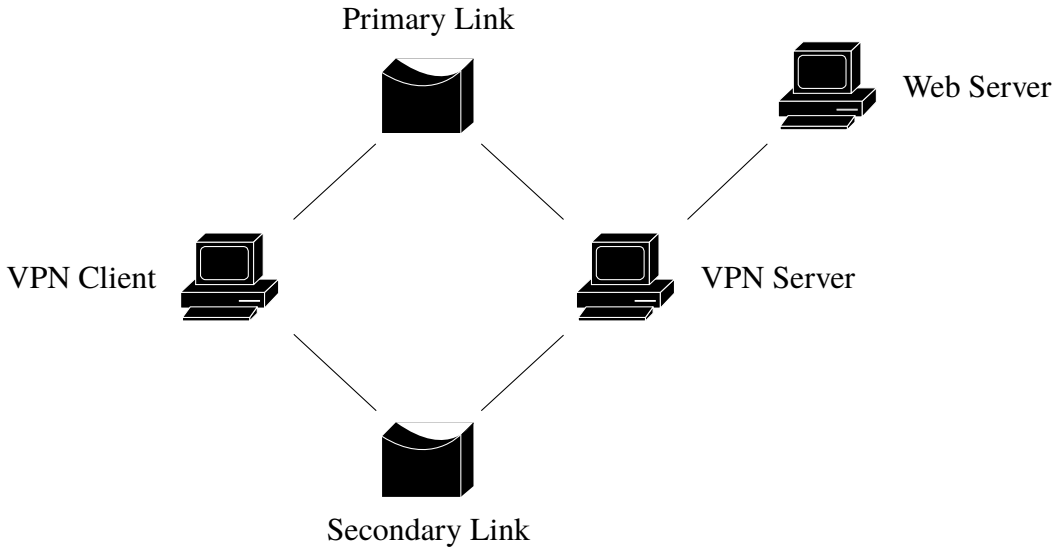


Figure 3.2: Conceptual Simulation Setup

The topology of Figure 3.2 utilizes two paths between the VPN client and VPN server. The use of two paths was determined to allow for sufficient testing all MPTCP path managers without introducing additional complexity that would come with additional paths. Additionally, the choice to have a web server only accessible by the VPN server ensures the VPN client will not be able to send and receive traffic from the web server without using the established VPN tunnel. This was a concern since traffic was observed outside of the VPN tunnel when using the fullmesh path manager during familiarization testing. This was a result of the MP_JOIN and ADD_ADDR options being utilized, which initiated connections between the VPN client and web server without the use of the tunnel.

The base system parameters for the topology of Figure 3.2 is detailed in Table 3.1. MPTCP kernel v0.90 was chosen since it was the stable release at the time of testing. MPTCP v0.91 has since been released, but the features used during this testing were not significantly altered. Additionally, the MPTCP scheduler chosen was the default version for the reasons discussed Section 2.2.4. The OpenVPN version used was also based on the most stable release at the time of the tests. It is important to use an OpenVPN version that is 2.3.9 or greater because prior versions of set default buffer sizes for the VPN tunnel that were insufficient. Since version 2.3.9, these buffer sizes are set by the operating system by default. VPN compression and encryption were disabled in order to simplify testing and analysis. The VPN encryption was enabled during the familiarization phase in order to ensure traffic was using the VPN tunnel. Once this was verified, all further testing disabled encryption. All machines were chosen to run Ubuntu 14.04 based on compatibility with MPTCP v0.90 and user familiarity with the platform. All TCP settings listed were the default settings for Ubuntu 14.04 as well.

Table 3.1: System Configurations

	Parameter	**Setting/Value**
VPN Machines	MPTCP Kernel	v0.90
	MPTCP Scheduler	Default
	VPN Version	OpenVPN 2.3.12
	VPN Ports	443 TCP / 1000 UDP
	VPN Compression	Disabled
	VPN Encryption	Disabled *
Web Server	TCP Congestion Control	Cubic
	Web Server	Apache2
	Test File Size	16331410 Bytes
All Machines	Operating System Kernel	Ubuntu 14.04
	TCP Window Scaling	Enabled
	TCP SACK	Enabled
	TCP Timestamps	Enabled
*Encryption disabled after verification of proper data encryption through VPN tunnel		

3.1.2 Virtual Testbed Implementation

We begin with a testbed using virtual machines within a Virtualbox environment. Link parameters were set using Virtualbox, the Linux traffic controller (tc), and the Linux network emulator (netem). Virtualbox was used to set the same speed of 12 Mbps for all links between the VPN client and server machines. The link speed between the VPN server and web server was set to a 1 Gbps link to simulate a local web server. In order to set the packet loss rates and propagation delays experienced between the VPN client and server, the traffic controller was used on each interface of the Bridge machine of Figure 3.2. For example, if a packet loss rate of 1% and 100ms propagation delay was set, then each interface would have a packet loss rate of 1% and propagation delay of 50ms set. Thus delay was considered to be a RTT adjustment as opposed to a one-way delay.

The performance of the MPTCP VPN tunnel connection was evaluated by measuring the goodput experienced when downloading a test file from the web server. This test was conducted using the standard Linux TCP congestion control algorithm of Cubic. During initial testing to ensure the virtual testbed provided realistic results, performance anomalies were present and can be seen in Figure 3.3.

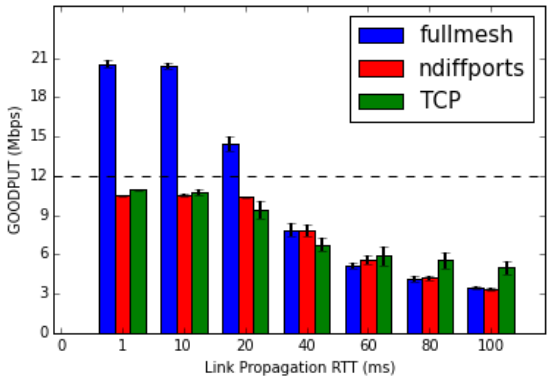


Figure 3.3: Virtual Machine Performance with 0.5% Packet Loss
Error Bars Show 95% Confidence Interval

The first anomaly is the significant drops in performance for MPTCP while TCP performance slightly decreased. The other significant result was MPTCP performance for fullmesh and ndiffports dropped below single path TCP once the delay reached 60ms. To validate these initial results, we decided to build a physical environment.

3.1.3 Physical Testbed Implementation

We began by implementing the physical testbed to match the previous setup seen in Figure 3.2 and Table 3.1 as closely as possible. One notable difference was the physical testbed used 10 Mbps link speeds instead of the 12 Mbps speeds of the virtual testbed. The 10 Mbps link speeds were specified using the Linux ethtool application on the bridge machines instead of the Linux traffic controller. The Linux traffic controller utilizes a token bucket filter in order to control the speed of the link, but this method did not regulate the speed as effectively as the ethtool application. Since the goal was to confirm trends, this difference in speed was not significant enough to be of concern. Once the physical environment was established, we ran the same tests as performed in the virtual environment. Figure 3.4 illustrates the test case of a 0.5% packet loss rate using the Cubic congestion control algorithm.

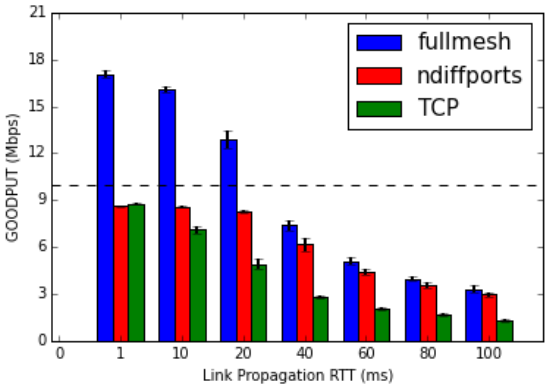


Figure 3.4: Physical Performance with 0.5% Packet Loss
Error Bars Show 95% Confidence Interval

The anomalies seen by the virtual testbed are no longer present in the physical testbed. As the MPTCP performance is affected by the propagation delays added, the TCP connection is also affected. Additionally, MPTCP outperforms the single TCP test and the ndiffports test for all conditions. After confirming the anomalies were no longer present in the physical testbed, we revisited the virtual testbed to determine if incorrect configurations were present. All machines were found to be using a non-default network adapters. Once this setting was returned to the default value of PCnet-FAST III, except for the VPN server to Web server link, the virtual tests were performed again. The results matched the physical environment,

thus validating the network setting as the root cause of the anomalies. For the remainder of the tests, the physical testbed was utilized and verified to be consistent with the virtual testbed.

Physical Testbed Baseline Configuration

The baseline system parameters set by the operating system and link speeds are provided in Table 3.2, followed by a brief description of the parameters.

Table 3.2: Baseline Configurations

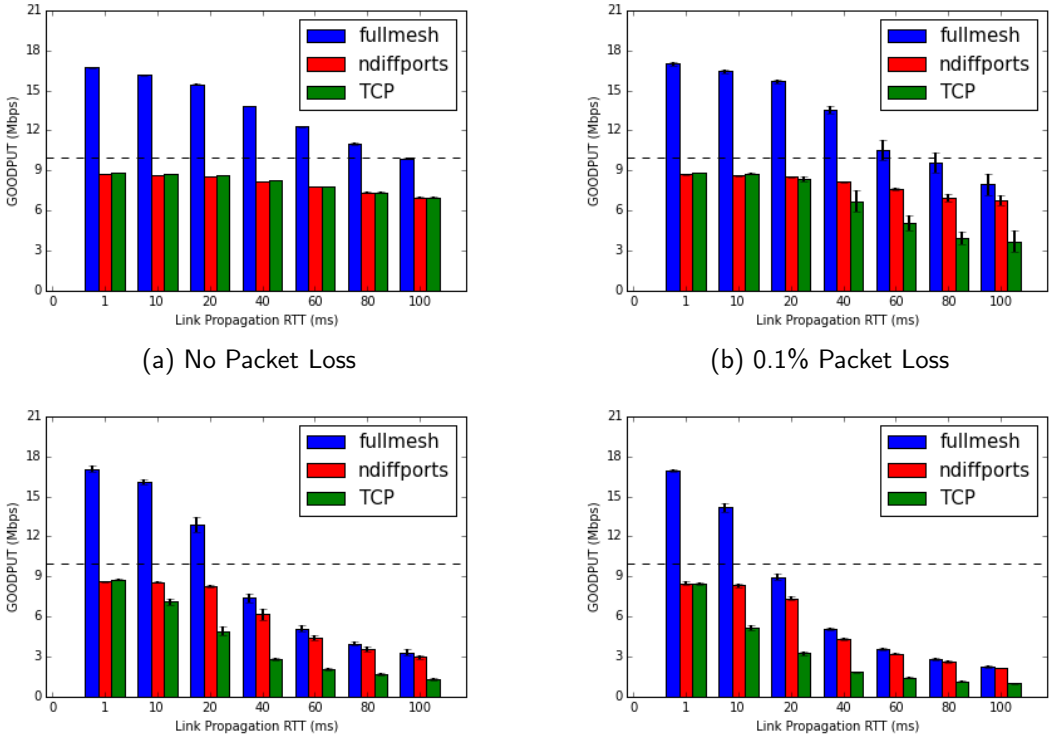
	Parameter	**Value**
Operating System	net.ipv4.tcp_congestion_control	Cubic or BALIA
	net.ipv4.tcp_rmem	4096, 87380, 6291456
	net.ipv4.tcp_wmem	4096, 16384, 4194304
	net.core.rmem_max	212992
	net.core.rmem_default	212992
	net.core.wmem_max	212992
	net.core.wmem_default	212992
	net.ipv4.tcp_no_metrics_save	Enabled
Link Speeds	Primary	10 Mbps
	Secondary	10 Mbps

- **net.ipv4.tcp_congestion_control:** Designates the TCP congestion control algorithm to be utilized. If congestion control algorithms designed for MPTCP are installed in the kernel, this is still the method for setting the desired algorithm.
- **net.ipv4.tcp_rmem & net.ipv4.tcp_wmem (min, default, max):** Sets the size of the TCP socket receive/send buffer in units of bytes. The operating system will assign values based on the memory available, but the user can set these parameters in order to improve performance (e.g., if using high bandwidth links).
- **net.core.rmem & net.core.wmem (max, default):** Sets the operating system receive/send buffer for any type of connection in units of bytes. This is also set since the tcp_rmem or wmem settings do no overwrite these values.
- **net.ipv4.tcp_no_metrics_save:** Sets whether or not connection data will be saved for a short period of time after finishing a TCP connection. This is disabled by default,

meaning that connection information, such as congestion window, will be maintained. The idea is that connecting to the same IP multiple times in a short period can use the previous data to improve performance.

3.1.4 Symmetric Tests with Cubic Congestion Control

The first test scenario performed was via links with symmetric link propagation RTTs and packet loss rates using the congestion control algorithm Cubic. To achieve a relatively small 95% confidence interval, tests were repeated 10 times for each RTT/loss-rate combination. Results for these tests are provided in Figure 3.5.



(a) No Packet Loss

(b) 0.1% Packet Loss

(c) 0.5% Packet Loss

(d) 1% Packet Loss

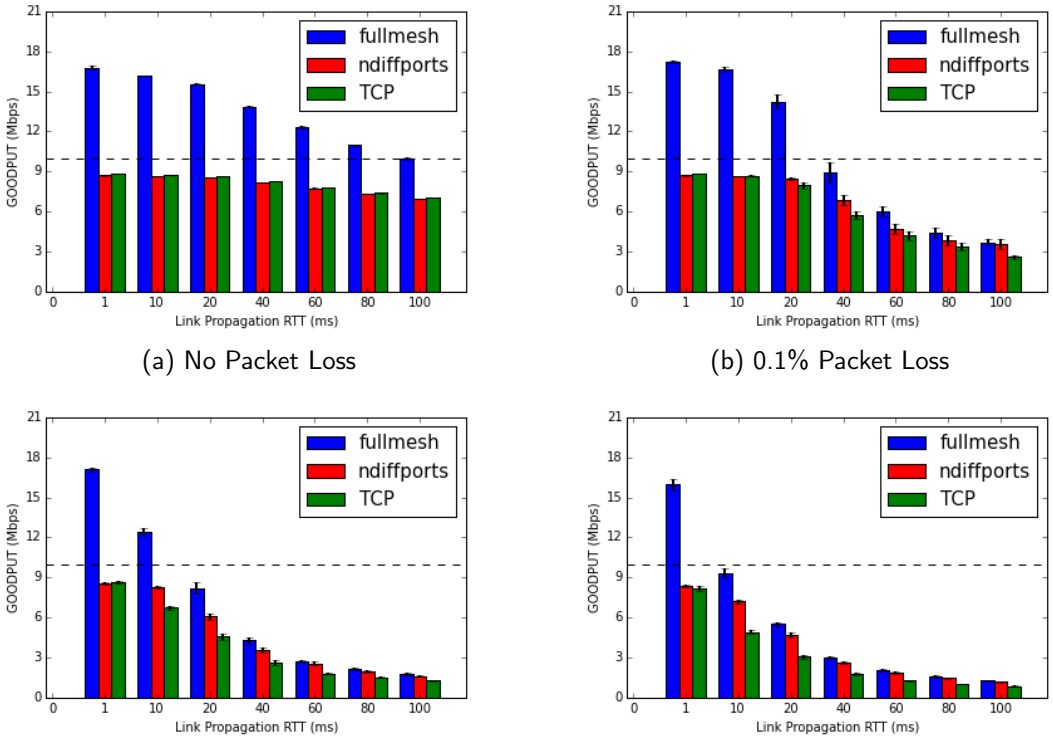
Figure 3.5: Symmetric Cubic Test Results

As seen in Figure 3.5a, when no loss is applied to the links, MPTCP with the fullmesh path manager exceeds the performance of ndiffports and TCP. In Figure 3.5b, when 0.1% loss is applied to both links we can see that fullmesh again has the best performance, but ndiffports also outperforms TCP when the delay reaches 40ms. Figures 3.5c and 3.5d show

similar performance results with the exception that the fullmesh path manager no longer significantly outperforms ndiffports as the delay increases. As expected, once packet loss rates are increased, the fullmesh path manager yields goodput values more than double that of TCP. This behavior is attributed to the aggressive nature of the Cubic congestion control algorithm.

3.1.5 Symmetric Tests with BALIA Congestion Control

The TCP Cubic algorithm was not developed to take advantage of MPTCP while at the same time remaining fair to single path TCP connections. For that reason we also tested performance with the newest of the MPTCP congestion algorithms known as BALIA. The only modified parameter of Table 3.2 was the congestion control algorithm used, all other settings remained the same. Again tests were repeated 10 times for each RTT/loss-rate combination to produce relatively small 95% confidence intervals. The BALIA results can be seen in Figure 3.6.



(a) No Packet Loss

(b) 0.1% Packet Loss

(c) 0.5% Packet Loss

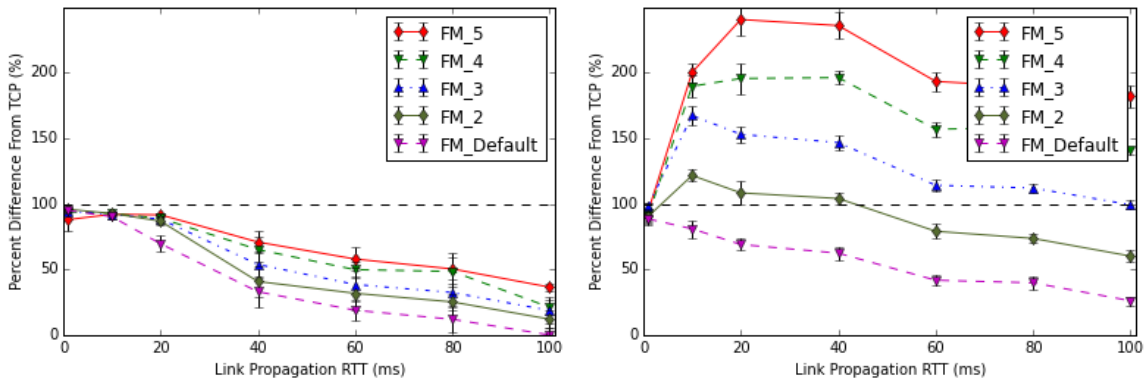
(d) 1% Packet Loss

Figure 3.6: Symmetric BALIA Test Results

Similar to the results of Section 3.1.4, the fullmesh path manager performs significantly better than TCP. Also, the same trend with ndiffports was seen throughout all tests. One significant difference is the fullmesh path manager does not achieve goodputs greater than twice that of TCP. This is expected since the BALIA algorithm is designed to be fair to single TCP connections.

Adding Additional Subflows

The MPTCP kernel allows setting the desired number of subflows to be created on each path for the different path managers. All testing up to this point used the default values for each path manager. The default value for the fullmesh path manager is 1 subflow per path, while the default for ndiffports is 2 subflows. We decided to change this default value to create additional subflows on each path to determine the affect on the performance. We first evaluated the fullmesh path manger with subflow values ranging from the default of 1 to a value of 5 subflows per path. We then normalized the goodputs achieved to that of a single TCP Cubic connection and graphed the percentage increase experienced. TCP Cubic performance was chosen as a comparison since it was shown to have the best performance for single path TCP. The results when normalized to a TCP Cubic connection can be seen in Figure 3.7.

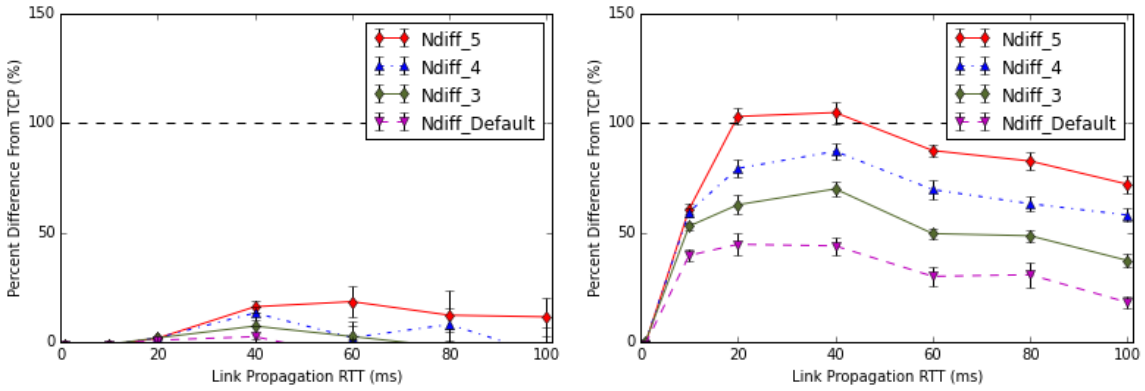


(a) 0.1% Packet Loss (b) 1% Packet Loss

Figure 3.7: Fullmesh Performance with Varying Subflow Settings
Error Bars Show 95% Confidence Interval

As seen in Figure 3.7a, adding additional subflows allows the fullmesh path manager to achieve significant improvements over TCP, but is unable to achieve more than a 100% increase. Alternatively in Figure 3.7b, the fullmesh path manager may greatly surpass the 100% goal marker as the number of subflows is increased. These results were also normalized to TCP Reno for comparison. The performance increases were much larger for the longer RTT cases since the Reno algorithm is dependent on ACK messages to increase its window size.

After seeing the performance benefits of adding additional subflows to the fullmesh path manager, we performed the tests again using the ndiffports path manager with subflow values ranging from the default of 2 to a value of 5 subflows per path. Figure 3.8 illustrates the performance percentage increased experienced with results normalized to that of a single TCP Cubic connection.



(a) 0.1% Packet Loss (b) 1% Packet Loss

Figure 3.8: Ndiffports Varying Subflow Performance
Error Bars Show 95% Confidence Interval

As seen in Figure 3.8a the results of adding additional subflows were less pronounced than with the fullmesh path manager. Nonetheless, the ndiffports was able to gain and sustain an increase when using 5 subflows. Figure 3.8b shows ndiffports outperforming TCP for all number of subflows. Notably, the 5 subflow test was able to achieve just over 100% performance increase. These test were again normalized to TCP Reno for comparison and showed similar results.

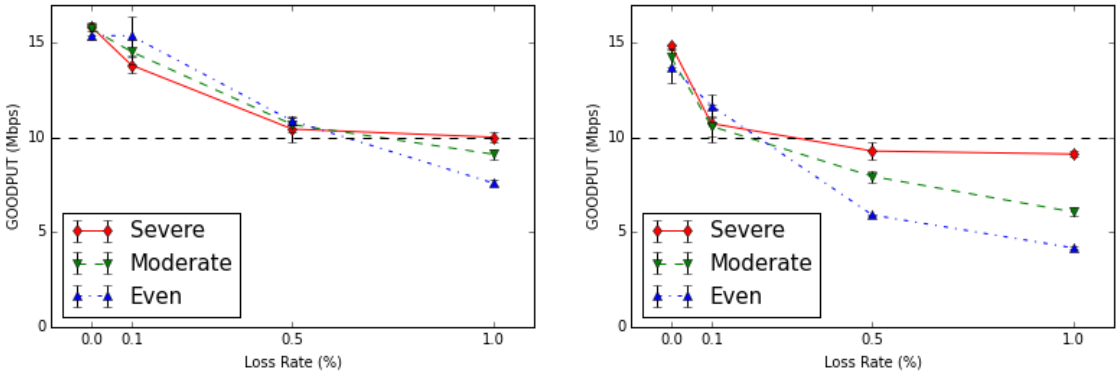
3.1.6 Asymmetric Tests with BALIA Congestion Control

When using MPTCP it is more common that links will not be symmetric. For example, a user may be using a wireless link and a cellular link, which are very likely to have different RTTs and packet loss rates. The same baseline configurations of Table 3.2 were used for the asymmetric tests. Table 3.3 lists the additional parameters used during the asymmetric tests.

Table 3.3: Asymmetric Test Parameters

Sum Delay	**Sum Loss**	**Even**	**Moderate**	**Severe**
20ms	0%	10ms, 0% 10ms, 0%	8ms, 0% 12ms, 0%	5ms, 0% 15ms, 0%
	0.1%	10ms, 0.05% 10ms, 0.05%	8ms, 0.03% 12ms, 0.07%	5ms, 0% 15ms, 0.1%
	0.5%	10ms, 0.25% 10ms, 0.25%	8ms, 0.15% 12ms, 0.35%	5ms, 0% 15ms, 0.5%
	1%	10ms, 0.5% 10ms, 0.5%	8ms, 0.3% 12ms, 0.7%	5ms, 0% 15ms, 1%
40ms	0%	20ms 0% 20ms 0%	15ms 0% 25ms 0%	10ms 0% 30ms 0%
	0.1%	20ms 0.05% 20ms 0.05%	15ms 0.03% 25ms 0.07%	10ms 0% 30ms 0.1%
	0.5%	20ms 0.25% 20ms 0.25%	15ms 0.15% 25ms 0.35%	10ms 0% 30ms 0.5%
	1%	20ms 0.5% 20ms 0.5%	15ms 0.3% 25ms 0.7%	10ms 0% 30ms 1%

We decided to perform the asymmetric tests using the 20ms and 40ms propagation RTTs because these are typical RTTs experienced. For each RTT we ran tests using the same loss rates seen in the symmetric tests. The even tests represent the symmetric tests previously conducted. The moderate tests represent roughly a 40/60 split of the total RTT delay and loss. The severe tests represent roughly a 25/75 split to illustrate two dissimilar links being utilized. The results of this test can be seen in Figure 3.9.



(a) 20ms RTT (b) 40ms RTT

Figure 3.9: Asymmetric Performance with Different Link Conditions
Error Bars Show 95% Confidence Interval

Figures 3.9a and 3.9b show the severe, moderate, and even tests having similar performance for the 0% and 0.1% loss cases. Figure 3.9b begins to show differences at the 0.5% test, where the severe case maintains approximately 10 Mbps. This difference grows as the loss rate reaches 1%, but the severe test case remains unchanged. To evaluate a possible reason for this behavior, we can use throughput models. The BALIA algorithm is reliant on ACK messages to increase window sizes similar to that of TCP Reno, thus a simplified TCP throughput model could be used [20]:

$$T = \frac{MSS * (\sqrt{\frac{3}{2}})}{RTT * (\sqrt{p})} \tag{3.1}$$

For Equation 3.1, p refers to the loss rate of the link. The loss rate can be considered as a constant in terms of each data point in the test. The MSS is also constant across the tests. This results in the RTT being the driving factor for throughput experienced. Using this formula additively for multiple subflows, the even case should yield the lowest results and the strong asymmetric tests the highest results. This behavior can be seen in Figures 3.9a and 3.9b as the packet loss rate increases.

3.2 Mobility Testing

MPTCP has the potential to provide seamless mobility as discussed in Section 2.3.2. We believe seamless mobility with OpenVPN be achieved by using MPTCP. To test this hypothesis, we first develop a new testbed to allow easily changing between different subnets to simulate a mobile client. We begin the mobility test by establishing a frame of reference using a single path TCP connection while changing between subnets. We then compare the results to that of a MPTCP connection.

3.2.1 Mobility Topology

The network testbed, Figure 3.10, was established using Cisco 2800 series routers, personal computers, and a network switch. Virtual machines and additional network traffic were not used during the simulations. When MPTCP was used, MPTCP kernel version 0.90 was installed on an Ubuntu 14.04 computer. The OpenVPN client and server were configured to perform a TLS handshake for initial connection and to use encryption when sending packets over the tunnel. This ensured packets were being sent over the tunnels for all tests.

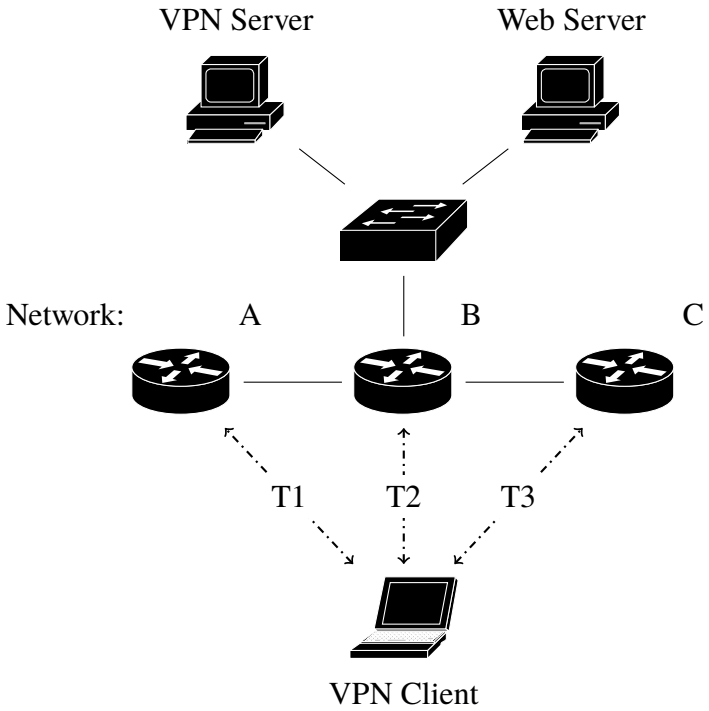


Figure 3.10: Conceptual Mobility Simulation Setup

3.2.2 OpenVPN Configurations

Once OpenVPN is installed, modifications to default settings occur within the configuration file. The purpose of the experiment was to examine and evaluate a mobile solution with OpenVPN, therefore we altered settings that would best facilitate mobility and testing. To enhance mobility, we set the server *keep-alive* parameter to ping every 10 seconds and assume the remote peer is down if there is no return message within a 30 second period. We also set *persistent tunnel* and *persistent key* to yes, thus ensuring the server tunnel remained active with the same parameters even if connections were lost. To simplify testing we decided to not allow split-tunneling. To do this, we tunneled all traffic through the VPN using the *redirect-gateway* push option.

OpenVPN client parameters are similar to the server parameters except there are fewer and they are meant to interact with the server. *Redirect-gateway* remains in the client as well as encryption settings. For the purposes of this experiment, the OpenVPN server created certificates that it would accept. We ensured those certificates were stored on the client configuration file establishing the VPN.

3.2.3 Single Path TCP Test

Prior to testing the effects of MPTCP on the mobility of OpenVPN, we first needed to establish a benchmark level of mobility seen by a single path TCP connection. We began by establishing a VPN connection between the client and server via the T1 link of Figure 3.10. Afterwards, Wireshark was activated on all devices and a download initiated using *wget* from the client and a simple HTTP server running on the web server. Then we disconnected the client's T1 interface and enabled the client's T2 interface. We verified the download was interrupted and observed whether or not the connection was regained. If the connection was not regained, we analyzed our configuration parameters to determine if the correct settings were applied and, if necessary, the test was repeated with the new settings. Data points collected included the amount of time needed to re-establish the tunnel and time before the download recommenced. The test was repeated again using Firefox instead of *wget* to determine web browser support.

TCP Tunnel Results

The first drawback to using a standard TCP connection with a web browser was that web browsers will not repeatedly try a download after sensing a network connection is lost. This resulted in the TCP tunnel not supporting mobility while using a web browser. It was discovered, however, that these tunnel options do provide a degree of mobility for command line downloads, but mobility suffered when improper VPN settings were configured. During the first round of testing, the VPN was configured to allow for duplicate client certificates to be used. When the single interface was dropped and the secondary interface was enabled, the VPN tunnel was rebuilt after the *keep-alive* timer re-initiated the connection, but the client was given a new tunnel IP address that was different than the previous address. This resulted in a failed download via the command line even after the tunnel connection was automatically re-initiated. Follow on testing showed that when the server did not enable duplicate certificates, the client was able to reconnect with the server and receive the same IP address as it previously had. The results of this test are provided in Figure 3.11.

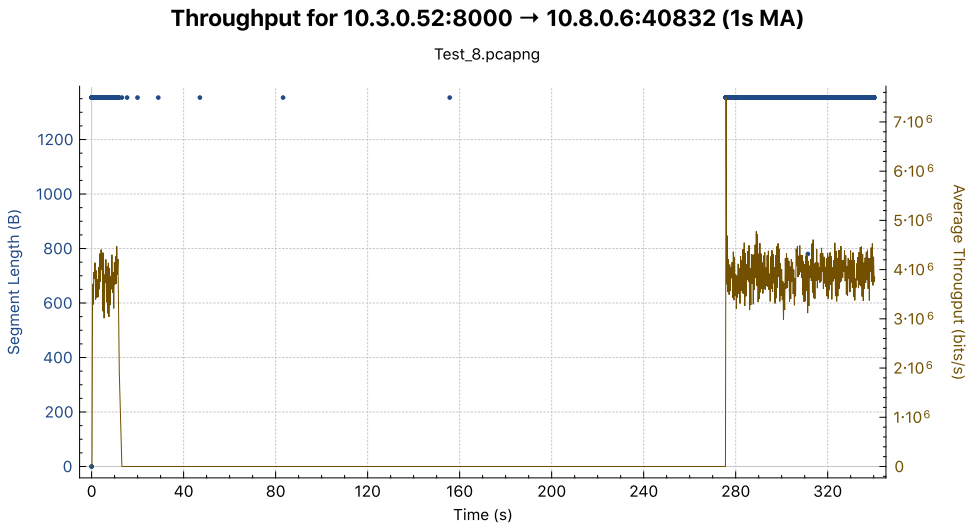


Figure 3.11: Web Server to VPN client TCP Mobility Test

Although the download did not immediately resume, it was eventually able to resume after the tunnel finished reconnecting. It should be noted that the tunnel was able to reconnect automatically after the 30 second *keep-alive* timer expired, but the download took some time before recommencing. Thus some mobility was achieved in the sense that the client did not need to manually reconnect to the VPN server, but was only able to take advantage of changing IP subnets when conducting business via the command line.

3.2.4 MPTCP Test

The MPTCP test outline was very similar to the single path version. The procedure remained the same from establishing the VPN through starting a download. Then the client's T1 interface was disconnected during the download and the T2 interface was connected. Once we verified the download was not interrupted, the T2 link was disconnected and the T3 link was established. Once again we verified if the new connection was able to join the download. After completing these tests, we performed the tests a second time with the exception of establishing the T1 and T2 links prior to dropping T1 and connecting the T3 link.

MPTCP Results

MPTCP with the fullmesh path manager enabled the OpenVPN tunnel to be created with multiple subflows using all available interfaces on the client and server. The use of the multiple subflows allowed for a subset of the interfaces to be disconnected during the download, either via the command line interface or via a web browser, and the download would continue at the capacity of the remaining connections. The results of the initial test are provided in Figure 3.12.

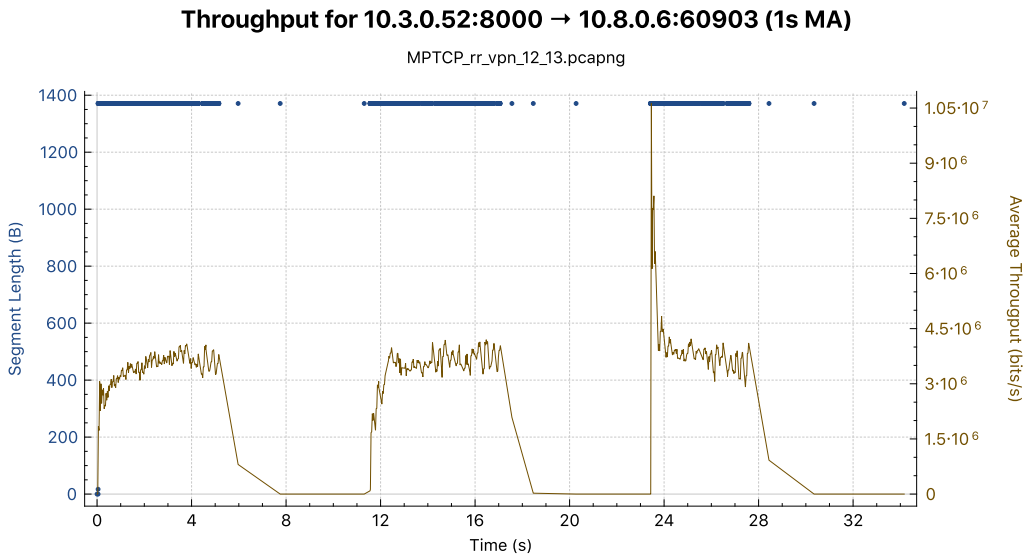


Figure 3.12: Web Server to VPN client MPTCP Mobility Test

The client's T1 interface was disconnected at approximately the 5 second mark followed by the T2 interface being connected. Once the host gained the new IP address, the interface immediately joined the download by use of the MP_JOIN option. Unlike the TCP model, the tunnel did not need to be re-established and the download was able to quickly start again. This same result is seen again when the client's T2 interface is disconnected around the 17 second mark followed by the T3 interface being connected. These tests illustrate using MPTCP with a VPN enables mobile hosts to change subnets without interruptions to the tunnel. This feature is similar to that of Mobile IP, but is able to take advantage of aggregating flows across multiple connections as well as eliminating the need for the home/foreign records required by Mobile IP.

3.3 Chapter Summary

In this chapter we evaluated the performance of using MPTCP to establish an OpenVPN connection. We were able to show that using MPTCP in a VPN yielded superior performance to that of a single path TCP VPN. When the Cubic congestion control algorithm was used with MPTCP, the fullmesh and ndiffports tunnels significantly outperformed the single path TCP tunnel. This algorithm is known to be aggressive to single path TCP connections over a shared bottleneck, thus its use should be limited to connections with disjoint paths. The MPTCP congestion control algorithm, BALIA, was also able to outperform the single path TCP tunnel. Additionally, when the MPTCP subflow parameters were adjusted to perform additional connections in the tunnel, the performance of the MPTCP tunnel increased. Each subflow was able to take advantage of the additional bandwidth available, leading to the aggregate bandwidth utilization to be greater than with fewer flows.

Further testing with asymmetric links showed that all the asymmetric tests yielded better performance than a single TCP tunnel. The severe asymmetric test yielded better results for the majority of the tests since the primary link with lower RTT and packet loss rates could handle the majority of the traffic. The secondary link with the higher RTT and packet loss rates could still be utilized at a lower packet rate to achieve greater cumulative bandwidth. Some tests, particularly where there was a low loss rate and RTT, indicated better performance would be seen with symmetric links. This performance different was not significantly larger and further testing may be useful to determine the root cause for this observation.

Regardless of the performance experienced, users may place more importance on mobility. From the mobility standpoint, we were able to realize benefits of using a MPTCP connection with OpenVPN when focusing on mobile hosts connecting to a VPN gateway. Most notably was the ability to drop single or multiple interfaces and either maintain the download in progress or rapidly regain the download as soon as a new IP address was established. Also notable was the ability to utilize a web browser to conduct downloads while changing subnets without an interruption to the download. Using a MPTCP VPN tunnel was shown to provide performance benefits in terms of achievable bandwidth as well as seamless mobility.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
MPUDP Design and Implementation

Chapter 3 focused on VPN performance when using TCP and MPTCP tunnels. These protocols were subject to the effects of TCP over TCP as discussed in [2], [3]. For this reason the default protocol used for various VPNs is UDP. Using the physical testbed and symmetric link test procedures from Section 3.1.4, we compared single path UDP and TCP tunnel performance. Figure 4.1 compares the performance of a TCP in UDP connection to that of a TCP in TCP connection using the Cubic congestion control algorithm.

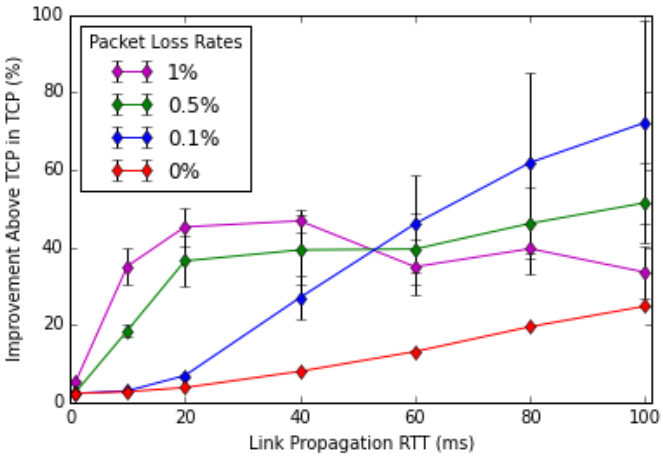


Figure 4.1: TCP in UDP Tunnel Performance
Error Bars Show 95% Confidence Interval

The UDP tunnel consistently outperforms the TCP tunnel since it is not subjected to a second layer of congestion control. The TCP in UDP tunnel connection will rely on the inner TCP connection with the web server to handle the congestion control and loss of packets. This allows for less traffic between the VPN client and server since the UDP tunnel does not require acknowledgement of traffic. We also showed in Chapter 3 how the use of MPTCP in the VPN tunnel could produce significantly better results than a single path TCP tunnel. Unfortunately, a MPUDP Linux kernel implementation to help improve a UDP tunnel does not exist. For this reason, we develop a basic MPUDP kernel in order to determine if similar performance benefits can be seen for a MPUDP VPN tunnel.

As UDP and TCP share several common functions, such as multiplexing and de-multiplexing of traffic of different applications, we will first examine the existing design of MPTCP in Linux to gain the insights required for an expedient prototyping of MPUDP functionality. With this understanding, we discuss the basic implementation of the MPUDP kernel to allow for proof of concept testing. We finish with a discussion of the results of this testing.

4.1 Designing MPUDP

Two of the key principles of MPTCP are the ability for it to fall back to a TCP connection automatically and to minimize the memory footprint needed to add in the MPTCP capability [9]. In TCP/IP Illustrated, the author explains how the various TCP functions interact within the kernel. A modified version of this interaction to include the additional MPTCP aspects is provided in Figure 4.2.

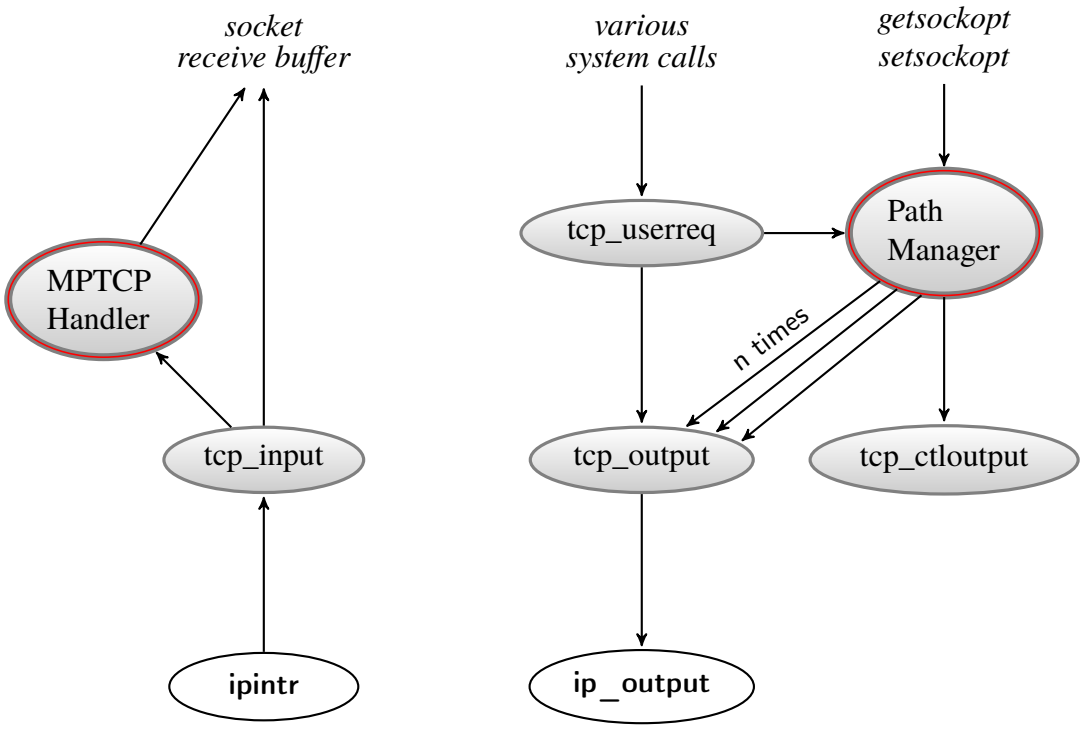


Figure 4.2: MPTCP Relationship to Kernel. Adapted from [21].

Not all systems are capable of using MPTCP, thus it is important for connections to fall back to TCP if necessary. This is the reason for the MPTCP Handler and Path Manger modules of Figure 4.2 to not cause a stoppage in the flow of the graph. If the MPTCP handler is enabled it will adjust how the TCP messages from the tcp_input are assembled and delivered to the application. Additionally, if MPTCP is enabled, the Path Manager will modify the TCP messages generated by the application to fully utilize all interfaces available for MPTCP. Just like MPTCP was able to fall back to a TCP connection, the MPUDP protocol should not cause adverse effects to the single path UDP traffic. With respect to minimizing the memory footprint, we decided to use two kernel modules with minimal kernel code added to the `udp.c` file. Linux kernel modules are able to be loaded and unloaded as needed and are not part of the base Linux kernel. The modules add code to the kernel as needed by the user to provide a function without the need to recompile the kernel with the added code. Kernel modules allow for easily modifying the MPUDP functions without the need to recompile the kernel after each change. This also allows the modifications to the kernel `udp.c` file to be kept to a minimum. This design concept is provided in Figure 4.3.

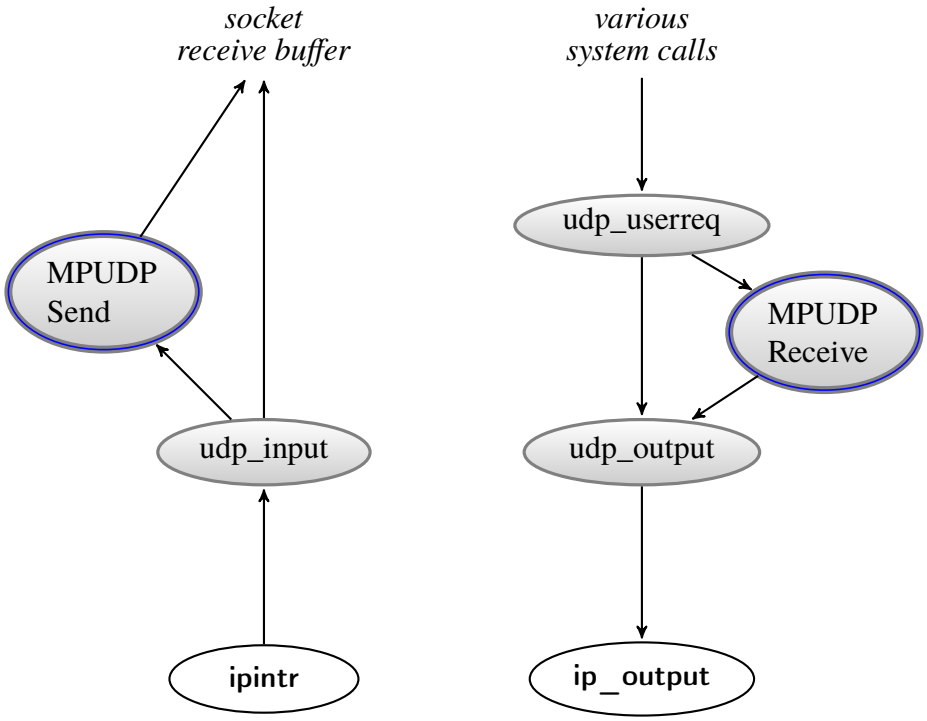
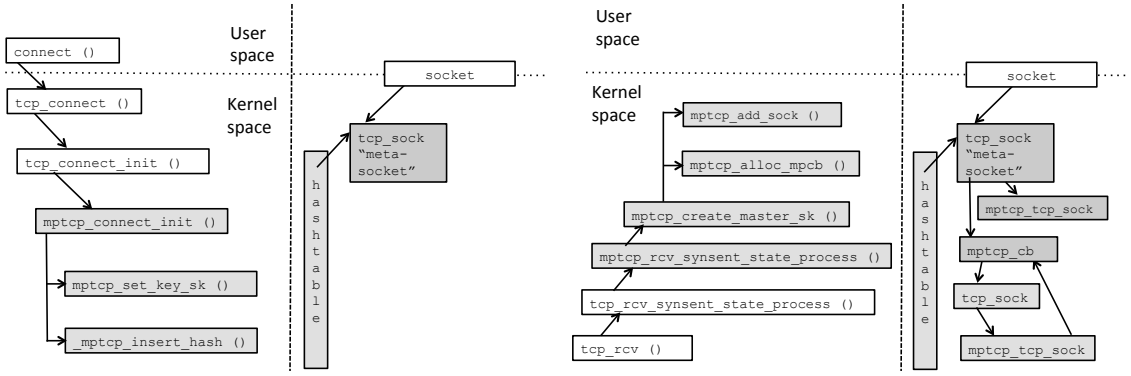


Figure 4.3: MPUDP Relationship to Kernel. Adapted from [21].

Not all Internet traffic will be well suited for MPUDP. MPUDP may provide significant benefits for establishing VPN tunnels, traffic consisting of bulk data transfers, and streaming traffic. There would be little to no benefit in using MPUDP for interactive/transactional traffic, such as Domain Name System queries. For this reason, we decided the user should be able to enable MPUDP and the parameters used through the use of sysctl commands as done for MPTCP.

4.1.1 Establishing a Multipath Connection

MPTCP connection establishment, depicted in Figure 4.4, follows the same three-way handshake process as TCP. Using a client/server framework, the MPTCP client is unable to determine ahead of time if the server will support MPTCP. For this reason, when the client initiates the handshake only the minimal MPTCP information is produced, not the complete MPTCP socket and control buffer. The only additional information generated is the MP_CAPABLE flag and session key [9].



(a) Client SYN (b) Server SYN/ACK

Figure 4.4: MPTCP Connection Process. Adapted from [9].

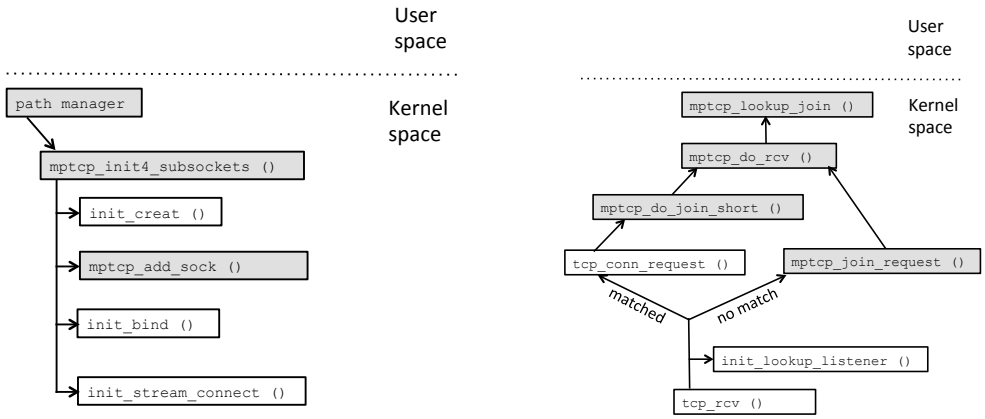
The MPTCP server, upon receiving an MP_CAPABLE SYN message, will completely allocate the required structures for a MPTCP connection. If the client receives a MP_CAPABLE SYN/ACK message from the server, then the client will also allocate the appropriate MPTCP structures [9]. This process will reduce the memory footprint when a MPTCP capable client connects to a single path server. If the client generated all structures needed for a MPTCP

connection that would fall back to a single TCP connection, then memory would be unnecessarily allocated and wasted.

The main challenge with a UDP protocol is that there is no handshake process to establish a connection. This means the same MPTCP connection process to determine if both the client and server were MPTCP capable will not be possible without modification to the protocol. Instead, there will need to be some mechanism for the client to signal the server of its intention to use MPUDP. Our first thought was to use the UDP header to signal the server much like MPTCP sets flags to send signals. However, we determined that the UDP header does not contain room for adding modifications that could survive passing through middle-boxes. Another possibility would be to extend the UDP header to allow for adding signaling information, but this was considered to be too invasive to the UDP protocol. In the end we decided it would be best to use a specified port for listening to incoming MPUDP control traffic. For instance, the MPUDP server could advertise a default listening port of 8080 for MPUDP control traffic. All control traffic could be stored in a multipath control block structure to allow for quick reference and minimal additional memory requirements. The MPUDP client would use a sysctl variable to set this control port and then send initial connection information to learn availability of additional interfaces. Once additional IP addresses are known, both the client and server could begin utilizing extra paths to enhance the performance of an existing connection or future connections. It is also important at this step to initialize a connection key to ensure all additional control information is able to be authenticated, much like the MPTCP kernel does during the initial handshake. If the server was not MPUDP capable, then this control traffic would be dropped and the user connection would default to UDP as desired.

4.1.2 Path Managers

After establishing the initial MPTCP connection, the selected path manager will attempt to establish additional socket connections. Figure 4.5 depicts the process required to establish the additional connections. The path manager will create and bind a new TCP socket and send a SYN message along with the MP_JOIN flag set [9]. If the MPTCP server receives the join request, the request is processed and the subflow added to the *meta-socket* information [9]. With the new connection added to the *meta-socket*, the packet scheduler is able to consider the link for sending traffic.



(a) Path Manager (b) Join Request

Figure 4.5: MPTCP Additional Subflow Process. Adapted from [9].

The MPUDP design could utilize the same concept as MPTCP for path managers. The ndiffports and fullmesh path manager design of allocating additional sockets for use would be the same for the MPUDP protocol. Where MPUDP can differ is the need for a connection to join an existing connection. This process would be controlled using the control port for the MPUDP connection. If additional interfaces are made available after the initial connection process, the user would signal the server of the additional source IP address(es) for the MPUDP connection to allow an update to the server's control structure.

4.1.3 Packet Scheduler

The next step is to determine an appropriate packet scheduler. Unlike MPTCP, MPUDP does not need to collect incoming packets and ensure their delivery in the proper order. Thus, there is no need for DSS to ensure proper delivery of in order packets. Unfortunately, UDP does not track RTTs or network congestion. There must be another method implemented in order to determine the best path for sending data. We examined a couple different methods to determine path conditions. One method would be to use a similar method to Multipath-MOSH discussed in Section 2.3.3. The client and server could periodically probe the links to determine RTT and link congestion. This method would introduce additional overhead, but the overhead in the Multipath-MOSH case was shown to be minimal. Another method would be to infer network conditions by sending statistics over the MPUDP control channel. The client and server could store the information, such as the number of packets delivered

per interface, in the multipath control block structure. Alternatively, a simplified packet scheduler like the round-robin scheduler of MPTCP could be used to split traffic evenly across links.

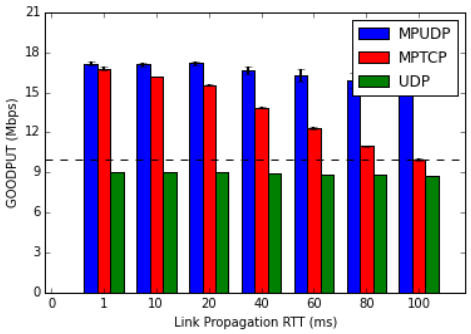
4.2 Implementation and Testing of MPUDP

The MPUDP kernel was implemented into the MPTCP v0.90 kernel used in Chapter 3. We chose to include the MPUDP protocol in the MPTCP kernel to allow for testing performance on the same kernel as well as moving towards an end goal of a true multipath kernel. Although the MPTCP protocol implementation involved adding multiple header and C files, as well as modifications to multiple TCP files, our MPUDP implementation added a minimal amount of code to the `udp.c` file and used two Linux kernel modules to introduce the additional functions needed. For this reason, the MPUDP implementation could be introduced to any version of the MPTCP kernel, not just v0.90. The `udp.c` file was modified to include pointers to functions used in the kernel modules, a branch in the udp_sendmsg function to allow changing the destination IP address, and a branch in the udp_recmsg function to allow changing the source IP of incoming messages.

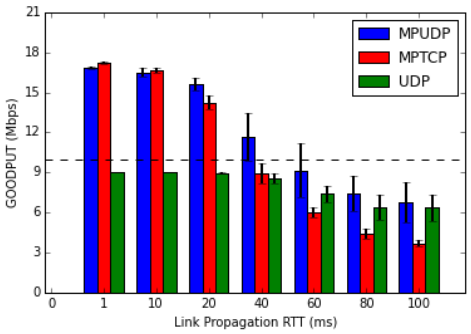
The implementation used for proof of concept testing did not include many of the more complex elements discussed in Section 4.1. Instead we took a standpoint that the client and server already discovered each others additional addresses and a MPUDP control channel was not needed. The use of a path manager was also not required since the addresses for the connection were already known. This simplification was used to facilitate quicker testing of the MPUDP protocol to determine its usefulness to the VPN testbed configuration used in Chapter 3. The Linux kernel `Send` module contains a simplified scheduler to split traffic across the two available subflows. The direction of flow was determined using a random byte generator function available to the kernel to generate a two byte unsigned integer with maximum value of 65535. The number generated was then tested to see if it was less than half the maximum (32767), and if it was the destination address was changed to the alternate path. This scheduler will end up with approximately half the traffic on each path, which is similar to a round-robin approach. Additionally, this scheduler is only useful for testing cases with symmetric links. The kernel `Receive` module contains a function to replace the incoming source address to the address used for establishing the OpenVPN tunnel. This is necessary since OpenVPN will discard the packets if they are not associated with the source

address used during initial setup. The modifications, along with source code information, to the kernel and additional kernel modules are further discussed in Appendix A.

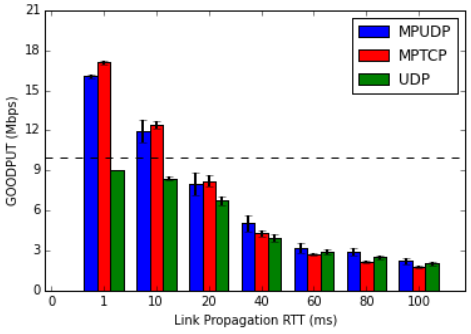
After making the required modifications to the MPTCP kernel, we conducted performance testing using symmetric links as done in Chapter 3. Once again the physical testbed of Figure 3.2, system configurations of Table 3.1, and baseline configurations of Table 3.1.3 were utilized in order to provide similarity between testing. The MPUDP kernel implementation utilized most closely corresponds to the fullmesh MPTCP test using the default number of subflows. Thus, in Figure 4.6, we compare the results of MPUDP and MPTCP using fullmesh to that of a single UDP connection. Tests were repeated 10 times each to achieve the 95% confidence intervals shown.



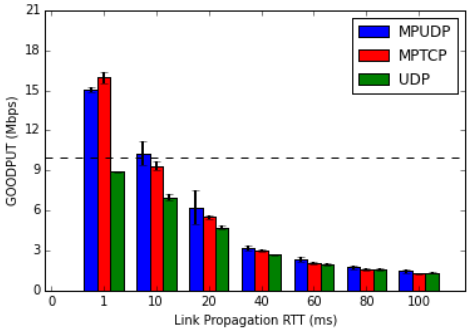
(a) No Packet Loss



(b) 0.1% Packet Loss



(c) 0.5% Packet Loss



(d) 1% Packet Loss

Figure 4.6: Symmetric MPUDP Test Results

In Figure 4.6a, the MPUDP tunnel does not appear to suffer any significant loss as the link propagation RTT increases. This results in a significant performance increase over the MPTCP tunnel and single path UDP tunnel. Interestingly, the UDP and MPUDP tunnels both seem unaffected by the increased link propagation RTT, while the MPTCP performance is significantly decreased. This is expected due to the problems discussed when multiple layers of congestion control are present. It was not expected, however, that the MPTCP tunnel performance would sometimes significantly drop below the single UDP tunnel performance. Figure 4.6b illustrates more interesting results. The MPUDP tunnel continues to outperform the MPTCP tunnel, but approaches the UDP tunnel performance as the link propagation RTT is increased. These trends continue in Figures 4.6c and 4.6d, although with diminished significance.

4.3 Chapter Summary

In this chapter we motivated the need for developing a MPUDP protocol to improve the performance of VPNs. We started by showing that a TCP in UDP tunnel outperformed a TCP in TCP tunnel due to removing the redundant layer of congestion control. Next we discussed a design for a MPUDP kernel. The MPUDP kernel can leverage the previous research conducted to develop and implement the MPTCP kernel. The design principles utilized for both protocols stressed the need for minimal impact to the single path protocols while allowing the connections to default to the single path without interaction from the user. The MPTCP kernel connection process concept was modified to better fit the UDP connectionless design. Instead of utilizing a three-way handshake to setup the connection, the MPUDP protocol would implement a control channel in order to acquire client/server IP addresses as well as other required control information. Additionally the use of path managers and a scheduler are very similar between the two protocols. The main difference is that the MPUDP protocol needs to send additional control traffic in order to establish link conditions to aid the scheduler in determining the best path to use for sending traffic.

The implementation of the MPUDP kernel involved minor additions to the existing UDP protocol to allow the use of Linux kernel modules. These minor additions were applied in the `udp.c` file to have pointers to functions within the Linux kernel modules. The main functions of the kernel modules consisted of a simplified scheduler to split traffic approximately evenly across the two available paths and a function to modify the source

IP of OpenVPN traffic to the IP used to establish the tunnel. The scheduler did not have the benefit of the control traffic determining link conditions, so a probabilistic method was used. If the source IP address was not changed, the traffic would not be associated with the proper source for the tunnel and OpenVPN would not process the packet.

The proof of concept testing showed promising results. The MPUDP tunnel had significant performance increases over the MPTCP tunnel for low loss rates. As the loss rates increased, the performance gain became much less significant. Interestingly, the single path UDP tunnel also performed better than the MPTCP tunnel for tests using higher RTTs. The development of the MPUDP protocol to use more sophisticated scheduling will likely result in the significant performance increases seen in the low loss cases to continue throughout all the tests. Additionally, an improved scheduler will allow for testing asymmetric link conditions as done in Chapter 3. The path manager of the MPUDP kernel was not implemented, which prevented the testing of mobility benefits of using MPUDP. It is safe to assume that once the path manager is implemented, the MPUDP VPN will experience similar mobility benefits seen when using the MPTCP tunnel.

CHAPTER 5:
Conclusion

This thesis investigated the use of multipath transport protocols to establish VPN tunnels. MPTCP is an emerging transport protocol to allow aggregating bandwidth using multiple TCP connections. VPNs are known to suffer performances losses when using TCP tunnels and transferring data also using TCP. MPTCP is able to overcome some of the performance losses seen by these TCP over TCP effects. MPTCP also adds the benefit of mobility to VPNs, which may be important in the host-to-host or host-to-gateway VPN models. The same way that MPTCP provides benefits to TCP, we showed that a MPUDP protocol can enhance a single path UDP tunnel.

The main conclusions of this thesis are as follows:

1. MPUDP should be further developed as a kernel protocol available to the user. Most Internet traffic uses TCP, which limits the potential benefits of using MPUDP. As shown in Chapter 4, however, VPNs benefit from using UDP tunnels since TCP over TCP effects are not present. We showed a MPUDP tunnel may significantly outperform not only a single UDP tunnel, but also a MPTCP tunnel.

2. MPTCP provides significant benefits and should be more widely adopted. The performance and mobility benefits shown in Chapter 3 are unable to be achieved if the VPN server is not MPTCP capable. The limited adoption of MPTCP prevents users from experiencing the full potential of the multipath protocol.

3. MPTCP and MPUDP should be standard kernel protocols, just like TCP and UDP. Additionally, MPTCP and MPUDP should be developed in the same kernel since they do not interfere with one another. Individually each multipath protocol provides benefits to the user. If a full-multipath kernel was utilized, the benefits to the user are greatly increased, as follows. First, a full-multipath kernel may be more appealing than just a partial MPTCP or MPUDP kernel, which can lead to a faster adoption by the community. Second, wide use of the full-multipath kernel will lead to a quicker maturation of the MPUDP protocol, which provides additional benefits to the user.

5.1 Future Work

The majority of future work involves the MPUDP kernel implementation. The proof of concept testing in Section 4.2 involved a simplified implementation of the MPUDP protocol. The implementation of the design characteristics of Section 4.1 would lead to the ability to test additional benefits of using MPUDP VPNs. The features should be implemented in the following order:

- Develop a multipath control block structure to store connection information. This control block should store a session key for the connection, available IP addresses for the client/server and link characteristics necessary for the scheduler.
- Section 4.1.1 discusses a method of establishing a MPUDP connection using a specified port. The client and server should listen on this port for incoming connections and control traffic. The first step would be to implement the ability for the client and server to learn available addresses and store them in the control block.
- Adapt the fullmesh path manager from MPTCP into the MPUDP protocol. It may not be necessary to implement the ndiffports path manager since its use in MPTCP is limited. This development will allow testing mobility benefits. The new path manager, named "redundant," of MPTCP v0.91 may also be useful for MPUDP.
- Section 4.1.3 discusses the scheduler for MPUDP. The first step would be to use the control channel to determine link characteristics, such as RTT, and develop a scheduler that utilizes the best link available for traffic. This development will allow testing asymmetric links.

After the MPUDP kernel is fully implemented, it would be beneficial to conduct kernel performance testing. The MPTCP kernel was tested in [9] to determine the additional load added by implementing MPTCP. This same testing should be done with the additions of the MPUDP protocol to ensure there are minimal effects to the kernel operation.

There is a potential for future work in testing multipath performance with different VPNs. This thesis utilized OpenVPN because of its open-source nature and ease of implementation, but there are various other VPN protocols. As discussed in Section 2.1.1, different VPNs operate at different layers of the OSI model. Testing the multipath protocols of this thesis with different VPNs may yield different performance results. Possibly some VPNs are built in a way that allow optimization with multipath protocols, while some may hinder the use of

multipath protocols. It may also be worthwhile to investigate possible benefits of adjusting OpenVPN to knowingly use multipath protocols. There may exist possible optimizations within OpenVPN to take advantage of using multipath tunnels.

Additional future work related to using MPTCP with VPNs is also possible. There are several additional congestion control algorithms available for use with MPTCP that were not tested. A detailed study of the performance of each algorithm when integrated with VPNs could be interesting. Specifically, each congestion control algorithm claims to have some level of friendliness to single path TCP connections. A detailed study to verify the friendliness of each algorithm when operating in a VPN would be beneficial, especially when utilizing additional subflows as done in Section 3.1.5.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Source Code

This appendix provides links to the source code used while conducting this thesis. Additionally, a break-down of the scripts utilized to conduct and analyze the performance testing is provided. This break-down is provided in hopes of easing future research efforts and any necessary familiarization or validation of the tests conducted. All source code is available at the following github address: https://github.com/danluke2/mpudp_vpn_thesis.

To conduct the performance testing, we utilized bash scripts to allow for automation of the tests. Initially, the majority of testing was done manually, which lead to errors due to forgetting to set certain parameters prior to the test. The bash scripts allowed for reducing errors in the testing since required settings were made within the script. The bash scripts also allowed for conducting multiple tests through the use of command line arguments. This greatly sped up testing, which allowed for more tests to be conducted.

The bash scripts are only part of the performance testing. Once each test was completed, there were multiple excel files to be analyzed to produce the graphs used. Initially, this was also done without the aid of scripts, which resulted in a very slow analysis of the data. Often the analysis of the data would take so long that extra tests were conducted prior to determining a fault in the testing procedure. For this reason, Python scripts were used to quickly parse the data and produce the desired graphs. The use of Python scripts allowed for avoiding unnecessary testing and a deeper analysis. We will now provide a description of the bash and Python scripts used for the testing.

VPN Client Scripts:

- **vpnstart.sh:** Allows for starting the client side VPN remotely from the server using an SSH connection. The script will start a Wireshark capture, initiate the tunnel connection, and then conduct a test download. The Wireshark capture allows for verifying the VPN tunnel behaves as expected for the desired test.
- **download.sh:** Allows for initiating the 10 downloads using *wget* in quiet mode. This script could easily be adjusted to perform any number of downloads required. Quiet

mode was used to prevent unnecessary download statistics being reported to the server via the SSH connection.

VPN Server Scripts:

- **supertest.sh:** This was the master script for conducting the symmetric tests of Sections 3.1.4, 3.1.5, and 4.2. The script allowed for command line arguments to test MPTCP, MPUDP, TCP, or UDP. The user must also specify the desired TCP congestion control algorithm to use. This script allowed for ensuring each test had the same starting parameters and was conducted properly.
- **symm_test.sh:** Called by the supertest.sh script to conduct the required tests.
- **asym_supertest.sh:** Similar to the symmetric supertest, but for conducting asymmetric testing of Section 3.1.6.
- **asym_test.sh** Called by the asym_supertest.sh script to conduct the required tests.
- **subflow_supertest.sh:** This is the master script for conducting the subflow tests of Section 3.1.5.
- **sub_test.sh** Called by the subflow_supertest.sh script to conduct the required tests.
- **tcp_dump.sh:** Starts the required Wireshark captures for the test.

Web Server Scripts:

- **directory_sym.sh and directory_asym.sh:** Creates the required directories automatically for storing the test data. This could be done by the user manually, but problems result if the user forgets to build all required directories.
- **web_tcp_dump.sh:** Starts the required Wireshark captures for the test.

Bridge Machine Scripts:

- **network_start.sh:** This script was run on machine start-up in order to initialize the required bridges between the interfaces. This script also set the required speed limits for the interfaces. To run this script at start-up, a configuration file was used.
- **tc_qdisc.sh:** Show and record the traffic control settings on each interface. Allowed for verifying the proper settings were in place during each round of testing.

- **tc_param.sh:** Used to add, change, or delete traffic control loss rates or delay rates for each interface.
- **bridge_tcp_dump.sh:** Starts the required Wireshark captures for the test. These Wireshark captures were primarily used during initial testing to verify traffic was using the VPN tunnel as desired.

Python Scripts:

- **parser_cubic_balia_mpudp.py:** Python script used to parse csv files and produce bar graphs from Sections 3.1.4, 3.1.5, and 4.2.
- **parser_fullmesh_norm.py and parser_ndiff_norm.py :** Python script used for producing the subflow Figures 3.7 and 3.8.
- **parser_asym.py:** Python script used for producing asymmetric Figure 3.9.

To implement the MPUDP protocol, we chose to use two Linux kernel modules in conjunction with a modification to the UDP kernel file. This method is similar to the MPTCP implementation. We used multiple kernel modules in order to simplify the design and clearly delineate the purpose of each module. We will now provide a description of the C files used for implementing the MPUDP protocol.

MPUDP C Files:

- **`udp.c`:** The `udp.c` file is found in directory $\backslash net \backslash ipv4 \backslash$. This file was modified to include branches in the udp_recmsg and udp_sendmsg functions. Code was also added to export symbols of functions used in the kernel modules.
- **MPUDP_send.c:** This was the kernel module used to implement the MPUDP send function called in the udp_sendmsg function.
- **MPUDP_recv.c:** This was the kernel module used to implement the MPUDP receive function called in the udp_recmsg function.
- **Makefile:** Creates the kernel object files from the MPUDP_send.c and MPUDP_recv.c files. The ".ko" files are the kernel files inserted to allow for the MPUDP functions to work.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] M. Feilner and N. Graf, *Beginning Open VPN 2. 0. 9: Build and Integrate Virtual Private Networks Using OpenVPN*. Packt Publishing Ltd, 2009.

[2] O. Titz. Why TCP over TCP is a bad idea. [Online]. Available: http://sites.inka.de/bigred/devel/tcp-tcp.html. Accessed January 15, 2017.

[3] O. Honda, H. Ohsaki, M. Imase, M. Ishizuka, and J. Murayama, "Understanding TCP over TCP: Effects of TCP tunneling on end-to-end throughput and latency," in *Proc. SPIE*, vol. 6011, 2005, pp. 60 110H–60 110H. [Online]. Available: http://dx.doi.org/10.1117/12.630496

[4] Apple. (2016, August). Use Multipath TCP to create backup connections for iOS. [Online]. Available: https://support.apple.com/en-us/HT201373

[5] C. Paasch and S. Barre. Multipath TCP in the Linux kernel. [Online]. Available: http://www.multipath-tcp.org. Accessed January 15, 2017.

[6] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP extensions for multipath operation with multiple addresses," Internet Requests for Comments, RFC 6824, January 2013. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6824.txt

[7] O. Bonaventure, "Blog entry: Recommended Multipath TCP configuration," 2014. [Online]. Available: http://blog.multipath-tcp.org/blog/html/2014/09/16/recommended_multipath_tcp_configuration.html

[8] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with Multipath TCP," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 266–277.

[9] C. Paasch, "Improving Multipath TCP," Ph.D. dissertation, UCL, London, Nov. 2014. [Online]. Available: http://inl.info.ucl.ac.be/publications/improving-multipath-tcp

[10] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, implementation and evaluation of congestion control for Multipath TCP," in *NSDI*, vol. 11, 2011, pp. 8–8.

[11] R. Khalili, N. Gast, M. Popovic, and J. yves Le Boudec, "Opportunistic linked-increases congestion control algorithm for MPTCP," Internet Engineering Task Force, Internet-Draft, July 2014. [Online]. Available: https://tools.ietf.org/html/draft-khalili-mptcp-congestion-control-05

[12] Q. Peng, A. Walid, J. Hwang, and S. H. Low, "Multipath TCP: Analysis, design, and implementation," *IEEE/ACM Transactions on Networking (ToN)*, vol. 24, no. 1, pp. 596–609, 2016.

[13] A. Walid, Q. Peng, S. H. Low, and J. Hwang, "Balanced linked adaptation congestion control algorithm for MPTCP," Internet Engineering Task Force, Internet-Draft, Jan. 2016. [Online]. Available: https://tools.ietf.org/html/draft-walid-mptcp-congestion-control-04

[14] E. F. Crist and J. J. Keijser, *Mastering OpenVPN*. Packt Publishing Ltd, 2015.

[15] H. Foster, "Why does MPTCP have to make things so complicated?: Cross-path NIDS evasion and countermeasures," M.S. thesis, Naval Postgraduate School, 2016. [Online]. Available: http://calhoun.nps.edu/handle/10945/50546

[16] L. Boccassi, M. M. Fayed, and M. K. Marina, "Binder: A system to aggregate multiple Internet gateways in community networks," in *Proceedings of the 2013 ACM MobiCom Workshop on Lowest Cost Denominator Networking for Universal Access*. ACM, 2013, pp. 3–8.

[17] C. Raiciu, D. Niculescu, M. Bagnulo, and M. J. Handley, "Opportunistic mobility with Multipath TCP," in *Proceedings of the Sixth International Workshop on MobiArch*. ACM, 2011, pp. 7–12.

[18] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure, "Exploring mobile/wifi handover with Multipath TCP," in *Proceedings of the 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design*. ACM, 2012, pp. 31–36.

[19] M. Boutier and J. Chroboczek. User-space Multipath UDP in MOSH. [Online]. Available: http://arxiv.org/abs/1502.02402. Accessed January 15, 2017.

[20] J. F. Kurose, *Computer Networking: A Top-Down Approach Featuring the Internet, 6/E*. Pearson Education, 2012.

[21] G. R. Wright and W. R. Stevens, *TCP/IP Illustrated*. Addison-Wesley Professional, 1995, vol. 2.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California