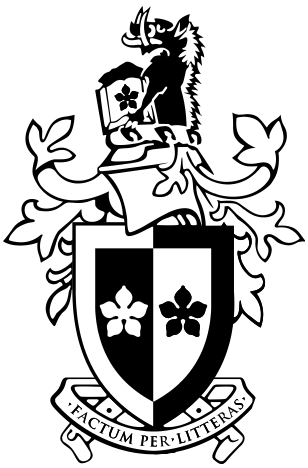
Implementing a Multipath Transmission Control Protocol (MPTCP) stack for FreeBSD with pluggable congestion and scheduling control

A thesis submitted for the degree of
Master of Engineering (Research)

Nigel Williams,
Centre for Advanced Internet Architectures,
Faculty of Science, Engineering and Technology,
Swinburne University of Technology,
Melbourne, Victoria, Australia.



October 24, 2016

Declaration

This thesis contains no material which has been accepted for the award to the candidate of any other degree or diploma, except where due reference is made in the text of the examinable outcome. To the best of the candidate's knowledge this thesis contains no material previously published or written by another person except where due reference is made in the text of the examinable outcome; and where the work is based on joint research or publications, discloses the relative contributions of the respective workers or authors.

Nigel Williams
Centre for Advanced Internet Architectures (CAIA) -
Faculty of Science, Engineering and Technology
Swinburne University of Technology
September, 2017

Acknowledgements

My principal coordinating supervisor Professor Grenville Armitage and coordinating supervisor Dr. Jason But, for exhibiting great patience and providing valuable guidance and feedback throughout the creation of this thesis.

James Mack and Amram Williams for their feedback.

Warren Harrop, for going well beyond in providing feedback and support.

Thanks to Lawrence Stewart for his invaluable technical consultations and advice on FreeBSD kernel programming.

The FreeBSD Foundation, who provided the support funding that allowed me to take on this project.

The Cisco University Research Program Fund, who provided support funding for the initial MPTCP project at CAIA, and support while completing this thesis.

Thanks to friends and family for their support and encouragement. Lastly, thanks to Aeri Lee for her patience, understanding and company.

Contents

Abstract **1**

1 Introduction **3**

2 A Background on Today's Internet **6**

2.1 The Internet Architecture 6

2.1.1 The Internet Protocol Suite 8

2.1.2 TCP: A Ubiquitous Transport Protocol 10

2.1.3 Extending Protocols 17

2.2 The Network Path 18

2.2.1 Defining the End-to-end Path 18

2.2.2 Sources of Dynamic Behaviour 21

2.2.3 Traffic Engineering Mechanisms 23

2.3 A Multihomed Future 27

3 Multihoming at the End-host **28**

3.1 Steering Packets 28

3.2 Multihoming and Path Diversity 30

3.3 Multihoming and Multipath Solutions 32

3.3.1 Link-layer 32

3.3.2 Internet-layer 35

3.3.3 Transport-layer 37

3.4 Conclusion 39

4 Multipath Scheduling and Congestion Control **41**

4.1 Multipath Schedulers 42

4.1.1 Naive approaches . . . 45
4.1.2 Scheduling for the transport-layer . . . 46
4.1.3 Improving loss recovery . . . 50
4.1.4 Assisting the scheduler . . . 51
4.2 Multipath Congestion Control . . . 52
4.2.1 Uncoupled Congestion Control . . . 54
4.2.2 Coupled Congestion Control . . . 55
4.3 Conclusion . . . 58

5 Overview: TCP extensions for Multi-addressed Operation 60
5.1 Key Concepts . . . 60
5.2 MPTCP in Operation . . . 63
5.2.1 Opening a Connection . . . 64
5.2.2 Associating Subflows . . . 66
5.2.3 Transferring Data . . . 66
5.2.4 Closing a Connection . . . 70
5.3 Other Protocol Considerations . . . 70

6 An Architecture for MPTCP in the FreeBSD Kernel 73
6.1 Designing for FreeBSD . . . 74
6.1.1 Data and Control Structures . . . 74
6.1.2 Event-driven Model . . . 79
6.1.3 Leveraging the Modular TCP framework . . . 81
6.1.4 Scheduling and Congestion Control . . . 83
6.2 CPU and Memory Considerations . . . 86
6.2.1 Ensuring Fair CPU Use . . . 86
6.2.2 Shared Memory, Locking and Concurrency . . . 89
6.3 Establishing a Connection . . . 92
6.4 Sending Data . . . 97
6.5 Receiving Data . . . 101
6.6 Closing a Connection . . . 105
6.7 Conclusion . . . 105

7 Experimental Evaluation **107**

7.1 Experimental Design 107

7.1.1 Testbed Topology 108

7.2 Evaluation 109

7.2.1 Basic Conformance 109

7.2.2 Creating Multiple Subflows 112

7.2.3 Retransmissions 118

7.2.4 Scheduling and Congestion Control 121

7.2.5 Performance 127

7.3 Conclusion 130

8 Conclusion **132**

8.1 Summary 133

8.2 Future Work 135

A Design Overview of Multipath TCP version 0.3 for FreeBSD-10 **136**

B Design Overview of Multipath TCP version 0.4 for FreeBSD-11 **143**

References **158**

List of Figures

2.1 The edge and core of the Internet 7
2.2 An Internet mesh topology 8
2.3 Layer-to-layer communication 9
2.4 TCP Header 10
2.5 TCP three-way SYN handshake. 11
2.6 TCP four-way FIN handshake 12
2.7 A simple TCP data exchange. 13
2.8 Examples of TCP retransmission. 13
2.9 Multiple paths between two hosts. 19

3.1 A simplified multipath scheduler. 29
3.2 Data striping across asymmetric paths can lead to packet reordering. 30
3.3 Multilink PPP (MP) 34
3.4 Site-wide and end-host multihoming. A multihomed gateway provides multihoming for an entire site. A multihomed end-host is directly attached to multiple providers. . 35
3.5 IPv4 mobility requires additional infrastructure to tunnel connections. 36

4.1 A basic decomposition of a multipath scheduler 45

5.1 MPTCP sits logically between the socket and standard TCP stack. 61
5.2 Generic MPTCP option format 61
5.3 The 64-bit data sequence is carried over multiple 32-bit TCP sequences. 62
5.4 Negotiating an MPTCP connection. 64

5.5 Examples of joining a subflow. On the left, Host 1 triggers an implicit join to a known address on Host 2. On the right, Host 2 must advertise an unknown address so that Host 1 can initiate a join. 65
5.6 Data Sequence Signal option 67
5.7 MPTCP data exchange using the DSS option. A typical exchange is shown on the left, while on the right we see a retransmission after a subflow fails. 68
5.8 MPTCP connection close can be combined with the subflow-level close. With multiple subflows, the MPTCP connection is closed first. 69

6.1 Datastructures to support MPTCP in the FreeBSD kernel. Preexisting kernel structures are shown in grey. 76
6.2 Changing the protocol switch and user request routines of a socket. 77
6.3 Processing a subflow socket event 80
6.4 Multiple threads are run when receiving a packet. 87
6.5 Receiving a segment in MPTCP. Processing is divided between two SWI threads - when the subflow receives a packet and for MPTCP-layer aggregation. 102
6.6 Subflow and MPTCP-layer receive structures. 104

7.1 Physical and logical representations of the testbed. 108
7.2 Throughput for MPTCP with a single subflow compared with TCP over the same path. 113
7.3 Comparing multiple subflows to TCP, with and without shared bottleneck. 113
7.4 Comparing per-subflow and total throughput for MPTCP flows with TCP. 114
7.5 Comparing per-subflow and total throughput for MPTCP flows with TCP where the router queue is undersized. 116
7.6 Testing fault tolerance. The client is multi-homed and Dummynet and PF provide middlebox emulation. 118
7.7 Random packet loss causing subflow-level retransmission. 119
7.8 Per-subflow and combined throughput during path loss and data-level retransmission. 120
7.9 Failure on SF2 causes a data-level retransmit. On recovery SF2 is able to send new data. 120
7.10 Subflow congestion windows for New Reno and Cubic subflows across a shared bottleneck. 123

7.11 Subflow congestion windows without a shared bottleneck. 124
7.12 Using delay-based and loss-based congestion control on subflows with a shared bottleneck link. 125
7.13 Using delay-based and loss-based congestion control on subflows without a shared bottleneck. 126
7.14 Comparing Iperf goodput to TCP as the link bandwidth is increased. 127
7.15 Average CPU usage by the kernel as bandwidth increases. 128
7.16 Average count of context switches per-second as bandwidth increases. 129

List of Listings

1 Definition of structure `cc_algo` 85

2 The `mp_sched_algo` structure definition 86

3 Release MPTCP data and revert to default TCP stack if MPTCP is not requested by the peer during SYN exchange. 94

4 Using `m_pkthdr` fields to store data-level mappings. 99

5 Identifying and processing MPTCP options. 103

6 Fall back to TCP if `ACK+MP_CAPABLE` is missing 110

7 Successfully establish a MPTCP connection and additional subflow. 111

8 The `simplerr` module definition. 121

List of Tables

2.1 A selection of protocols from layers of the Internet suite. 8
2.2 TCP State Variables 16

4.1 A selection of statistics used by current schedulers. Some must be calculated by the algorithm. Note that some statistics may serve multiple purposes. 44

5.1 MPTCP State Variables 63
5.2 Data Sequence Signal Flags 67

6.1 MPTCP control block sequence variables 76
6.2 Size of MPTCP datastructures, compared with the existing TCP socket structures. . . 78
6.3 Per-connection locks for TCP 89
6.4 Per-connection locks for MPTCP 89

7.1 Example conformance tests. 109
7.2 Linux-endpoint settings required for interoperability. 110
7.3 Mean count of retransmissions (including fast retransmissions, RTOs) when transferring 20MB. 117
7.4 Total-retransmits per-subflow, MPTCP-independent. 117
7.5 Goodput without rate-limiting 128

Abstract

The Internet is a collection of highly interconnected networks. Data sent from an end-host may be routed through multiple networks, and potentially via multiple distinct *paths,* to reach a destination. To facilitate this, the devices connecting networks together are *multihomed,* having attachments in multiple networks. The end-hosts served by these networks have been until only recently predominantly *single-homed* - connected to the Internet through a single point of attachment. It is now common for Internet connected hosts to have multiple network interfaces, for example Wi-Fi and cellular. Multihoming creates a fundamental shift in end-to-end communications, providing an end-host with a choice of alternate paths that can be used for additional capacity, redundancy or mobility.

The protocols responsible for moving data between connected devices have remained largely unchanged since the early days of the Internet. The *Transmission Control Protocol* (TCP) [1], whose genesis was in the 1970s, remains the Internet's default choice for reliable transport. That protocols like TCP have been able to scale is a testament to their design, however a consequence of their age is that they are unable to take full advantage multiple network connections at the end-host. For example, TCP can only utilise a single network path between source and destination per session, and sessions must be broken and reestablished when endpoints shift their network connectivity from one interface to another.

The emergence of multihomed end-hosts has been predicted, and solutions have emerged in various guises over the years. A promising recent proposal is the *TCP Extensions for Multipath Operation with Multiple Addresses,* (MPTCP) [2] specification, which has been standardised by the *Internet Engineering Task Force* (IETF) and is already deployed across the Internet. MPTCP allows existing TCP-based applications to utilise whichever underlying interface is available at any given time, seamlessly maintaining transport sessions when endpoints shift their network connectivity from one interface to another. Though MPTCP has been demonstrated to operate effectively across the Internet, there remains wide scope for exploring how and when data transmission across available paths ought

to be tied together or decoupled through it's primary transmission mechanisms of *congestion control* and *scheduling*. Research into these aspects of MPTCP can currently only be undertaken using the Linux-based reference implementation.

In addition to a comprehensive literature review of technologies relating to multipath communication, this thesis makes two core contributions:

- We design and implement the first ever MPTCP stack for the FreeBSD operating system. Our implementation specifically augments FreeBSD with support for pluggable multipath scheduling and congestion control algorithms, vastly simplifying future research on novel scheduling and congestion control algorithms. Our architecture is extensible, maintainable and has minimal impact on FreeBSDs existing single-path TCP stack.

- We demonstrate a functional prototype, and release our code publicly as a contribution to the FreeBSD community.

Our implementation is shown to achieve basic MPTCP functionality, and we show that scheduling and congestion control modules can be defined to effect different dynamic behaviours of multipath connections. Although our code is currently unoptimised, we show that performance is sufficient to support current experimentation scheduling and congestion control algorithms. Based on early feedback from the 2016 FreeBSD Developer Summit, we expect that further development and improvement of our FreeBSD MPTCP codebase will take place in the FreeBSD community.

Chapter 1

Introduction

Commoditisation has seen computing power distributed into the hands of billions of people worldwide, fueling the rapid expansion of Internet-connected devices. In the early 1990s the Internet reached 1 million connected hosts [3]. A decade later this had grown to 500 million. Current estimates put the number of connected devices at over 15 billion [4].

End-hosts connected to the Internet were until only recently predominantly single-homed - i.e. connected through a single point of attachment. Contemporary computing devices such as smartphones, notebooks or servers are often multihomed, featuring multiple network interfaces that potentially use different link technologies, for example Wi-Fi and cellular.

While the breadth and speed of the Internet has evolved considerably, the *transport protocols* responsible for moving data between connected devices have largely remained the same. A case in point is the *Transmission Control Protocol* (TCP) [1], which first originated in the 1970s [5] to provide reliable transport across the nascent networks of the time. Today, with relatively few extensions, it remains the Internet's default choice for reliable transport, and is used across a range of wired and wireless network links.

In the new multi-homed reality, protocols designed for single-homed hosts have certain drawbacks. For example, TCP has two significant shortcomings – it can only utilise a single network path between source and destination per session, and sessions must be broken and reestablished when endpoints shift their network connectivity from one interface to another. Being bound to a single path also precludes multihomed devices from using any additional capacity - or redundancy - that might exist over alternate paths. To enable multihoming at the end-host, solutions have been proposed that either replace TCP as a transport protocol, as is the case with CMT-SCTP, [6], or attempt to spread a single TCP connection over multiple paths, for instance Multilink PPP [7]. These protocols, though often

deployed successfully in niche use-cases, have not seen widespread deployment as general-use multihoming solutions for a various reasons, including a lack of compatibility with legacy applications and performance.

The IETF's Multipath TCP working group is focused on an idea that has emerged in various forms over recent years: that a single transport session as seen by the application layer might be striped across multiple Internet paths between the session's two end-points. The outcome of this working group is the *TCP extensions for multi-addressed operation* or *Multipath TCP* (MPTCP) standard. When multiple interfaces are concurrently available, MPTCP enables the distribution of an application's traffic across all or some of the available paths (deemed *subflows)* in a manner transparent to the application. MPTCP is already deployed on the Internet [8]. The reference implementation, developed at the Universite catholique de Louvain (UCL) for the Linux kernel, fully implements the MPTCP protocol and is used in a number of production environments [9]. An implementation by Apple Inc. for the XNU kernel provides a subset of MPTCP functionality and is enabled on Apple iOS devices.

As an extension of TCP, an over-arching expectation is that TCP-based applications see the traditional TCP socket API, but gain benefits when their session transparently utilises multiple, potentially divergent network layer paths. These benefits include being able to stripe data over parallel paths for additional speed (where multiple similar paths exist concurrently), or seamlessly maintaining TCP sessions when an individual path fails or as a mobile device's multiple underlying network interfaces change connectivity state. MPTCP allows existing TCP-based applications to utilise whichever underlying interface (network path) is available at any given time, seamlessly maintaining transport sessions when endpoints shift their network connectivity from one interface to another.

Data transmission must be coordinated across the subflows making up the MPTCP session, to both effectively utilise the total capacity of heterogeneous paths and ensure a multipath session does not receive *"...more than its fair share at a bottleneck link traversed by more than one of its subflows"* [10]. Current approaches focus on the use of *schedulers* and *congestion control (CC)* to control how the end-hosts steers data between the available paths. There is wide scope for exploring how and when CC for individual subflows ought to be tied together or decoupled, and how CC and scheduling can be combined for more effective data transmission.

In this thesis we present a comprehensive literature review of technologies relating to multipath communication and survey current Multipath TCP research. When then make two core contributions:

- We design and implement the first ever MPTCP stack for the FreeBSD operating system. Our

implementation specifically augments FreeBSD with support for pluggable multipath scheduling and congestion control algorithms, vastly simplifying future research on novel scheduling and congestion control algorithms. Our architecture is extensible, maintainable and has minimal impact on FreeBSDs existing single-path TCP stack.

- We demonstrate a functional prototype, and release our code publicly [11] as a contribution to the FreeBSD community.

We have selected FreeBSD as it has a rich history as a network experiment platform. A FreeBSD implementation also offers an alternate licensing model to the GPL [12], and the creation of a BSD-licensed MPTCP implementation benefits both the research and vendor community.

The remainder of this thesis is structured as follows. Chapter 2 provides background on the Internet and network paths. Chapter 3 examines existing multi-homing and multi-path solutions. Chapter 4 surveys existing MPTCP research into congestion control and scheduling. Chapter 5 provides brief introduction to the Multipath TCP protocol. Chapter 6 describes the key elements of our design and the rational behind the design choices. Chapter 7 evaluates our prototype. We conclude in Chapter 8 and discuss future work.

Chapter 2

A Background on Today's Internet

This chapter provides an overview of the Internet's architecture and highlights key issues effecting communications sessions between endpoints. Section 2.1 is an overview of the Internet topology and core components. Section 2.1.1 describes the Internet protocol stack, used by end-hosts to communicate over the Internet. Section 2.2 describes what a network path is, sources of dynamic path behaviour and common traffic engineering methods. Section 2.3 discusses the emergence of multi-homed end-hosts.

2.1 The Internet Architecture

The Internet is a network of independently administered, interconnected networks. Connecting these networks are *links* between *gateway nodes* (*routers).* These links and routers provide a path between networks for *packets,* the basic data unit of the Internet. Routers make decisions about which link a packet should be transmitted on a packet-by-packet basis.

Networks that make up the Internet are divided into a *core* and *edge,* shown in Figure 2.1. Core networks provide a pathway, or *transit,* to other networks on the Internet. Edge networks provide Internet access to LANs and *edge devices* such as PCs or mobile devices by connecting directly with other edge networks or obtaining transit from core networks. Edge devices communicate with one-another using a set of protocols defined in the Internet protocol suite [13].

Networks can have multiple links to other networks, making the Internet a *mesh* topology. This means that is it possible for multiple pathways two exist between two endpoints. The mesh topology improves redundancy, as in the event of link failure there may be alternate links available. This topology also leads to performance benefits, as devices can elect to send data over a better performing path

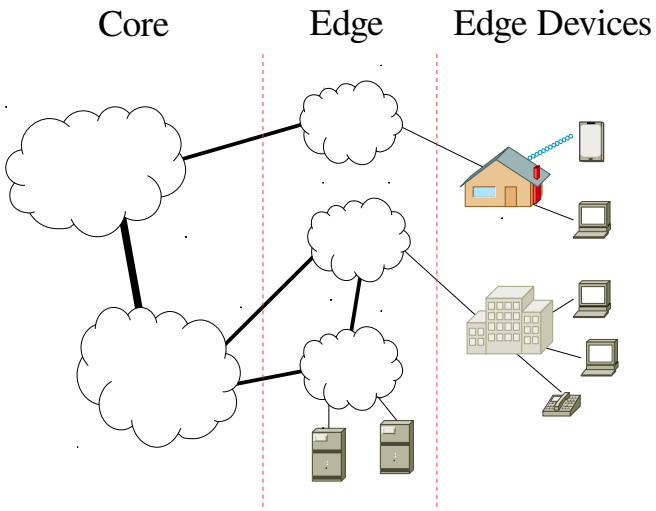


Figure 2.1: The edge and core of the Internet

or stripe data across multiple paths. Figure 2.2 illustrates how independent networks are connected together in a mesh to form the Internet. Hosts within each network can send packets to any other host, even if the networks do not have a direct link, such as between *Network A* and *Network D*.

Packets are delivered via a *single-class, best-effort* service [13]. In a best-effort network packets in transit may be corrupted, re-ordered, duplicated or discarded altogether. There are no guarantees of throughput or on timeliness of delivery. Single-class means that all packets are treated equally regardless of protocol or generating application.

It is the responsibility of end-hosts attached to the Internet to create the context of a communications session. This is referred to as *end-to-end* communications, and together with best-effort delivery is a core principle of the Internet architecture [13]. The Internet is designed such that the intelligence of routing and path discovery lies within the network, and the intelligence of a communications session exists at the end-hosts.

A fundamental part of best-effort networks is the concept of fair resource usage, or *fairness* [14]. Resources are shared and end-to-end protocols are relied upon to divide the available network capacity, typically according to some fairness measure. Endpoint-controlled fairness is feasible only when end-hosts agree to cooperate. If enough end-hosts implement similarly behaved algorithms, approximate fairness can be achieved without network-based management. A single definition of fairness is not widely agreed upon [14], and what constitutes *fair* changes depending on the desired outcomes. The most common form is that of *flow-rate* fairness, where end-to-end protocols attempt to attain an equal division of bandwidth between flows that share a single network link [14].

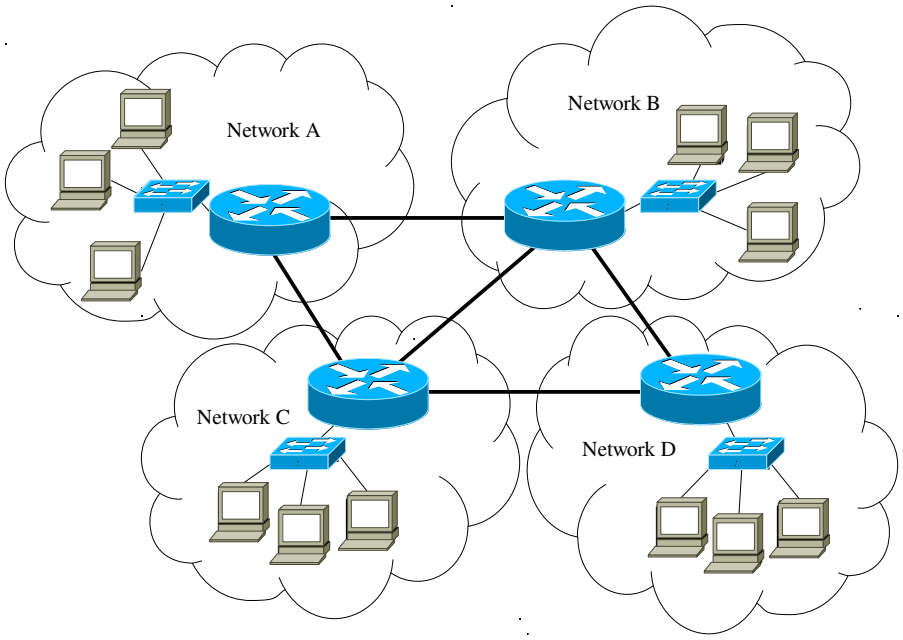


Figure 2.2: An Internet mesh topology

	Layer			
	Link	**Internet**	**Transport**	**Application**
Protocol	ARP, PPP, Ethernet	IP, ICMP	TCP, UDP, SCTP	HTTP, DNS, BGP

Table 2.1: A selection of protocols from layers of the Internet suite.

2.1.1 The Internet Protocol Suite

The Internet protocol suite is a layered stack of protocols that enable a host to communicate over the Internet. Within the stack are protocols that provide functionality such as address resolution, connection setup and reliable transmission. The suite is divided into four layers of abstraction: link, Internet, transport and application. The suite, and the protocols within, are standards defined by the Internet Engineering Task Force (IETF) [15].

Internet stack protocols are grouped within each layer according to the function they perform (Table 2.1). A message originates at the application and is encapsulated by a *protocol header* from each layer before transmission. This process is reversed on reception, such that protocols effectively communicate layer-to-layer, as Figure 2.3 shows. Broadly speaking, the lower layers (link, Internet) provide media access, addressing and routing. The upper layer protocols (transport, application) form the communications channel between end-user applications.

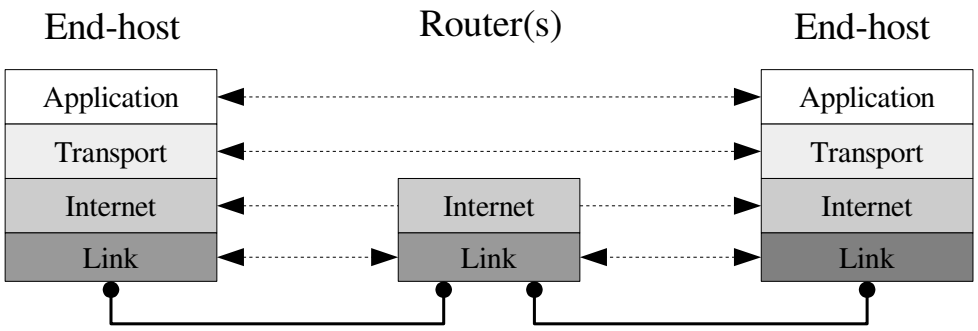


Figure 2.3: Layer-to-layer communication

The application layer is the topmost layer of the Internet suite. From here data originates and is eventually consumed. Application layer protocols can be categorised as end-user and support protocols [13]. End-user protocols provide services to user applications. For example a web browser will communicate with a web server application using *HTTP* [16]. Support protocols are typically engaged by the host system directly. A host might issue a *DNS* lookup [17] in order to resolve the location of a web server that a browser wishes to connect to.

Transport layer protocols provide the end-to-end communications channel and signaling between endpoints. A number of transport protocols have been standardised by the IETF, though the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are the primary protocols used on the Internet [18]. TCP, detailed in Section 2.1.2, is the default choice for reliable, ordered and connection-oriented data service. UDP is connectionless and unreliable, providing only multiplexing and error checking to IP datagrams.

The Internet layer includes the *Internet Protocol* (IP) [19] itself. The Internet Protocol defines the end-to-end datagram format and provides the addressing and routing framework that enables data to be delivered between two end hosts. The transmissible unit of IP is the *IP packet*, and data to be transmitted may be segmented and placed into a sequence of one or more packets. Within each packet is an *IP header* that contains all the meta-data required to reach a destination endpoint. Thus each packet is self-contained and independently routable. Key within the IP header are the source and destination *IP address* fields.

Every host attached to an IP network is identified by an assigned IP address. Decoding an IP address provides a *network address* and *host address*. End-hosts belonging to the same network share a common network address. The host address is always unique on a given network, making each end-host individually addressable and allowing packets to be routed directly between two end-hosts.

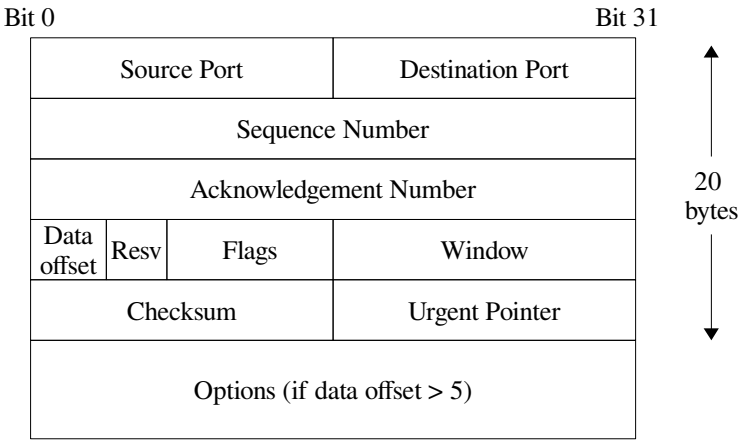


Figure 2.4: TCP Header

Since IP is a stateless protocol, an end-host does not need to signal the IP network explicitly in order to set up a communications channel or transmit new packets. Routing protocols are used to exchange network reachability between ASs. Routers use the destination IP address in the IP header to forward packets through the Internet. Protocols such as ICMP, which provides diagnostics and error notification, are also grouped within the Internet layer.

The lowest layer of the Internet stack is the link layer. Link layer protocols mediate access to the link to which a host is directly connected. The protocols are divided between network-specific medium access control (MAC) protocols, which govern the physical transmission of data over a link, and logical protocols for tasks such as address discovery and configuration.

2.1.2 TCP: A Ubiquitous Transport Protocol

The Transmission Control Protocol (TCP) is the most widely used transport protocol on the Internet. It is used as transport for the on-demand video, web and file sharing applications that account for the majority Internet's traffic volume [20]. Due to it's pervasiveness, the Internet stack is also referred to as the *TCP/IP* stack. TCP provides reliable, sequenced delivery of data between two endpoints over an IP network. As long as a path exists between the source and destination IP addresses, any transmitted data will be received at the destination application in-order and error-free. Once established, a TCP connection is tied to specific IP source and destination endpoints that cannot be changed, even if a host is accessible by multiple addresses. Where a path fails, TCP provides graceful connection closure. Connection management, data segmentation, error checking and retransmission are performed by

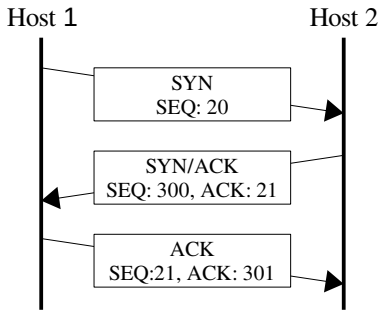


Figure 2.5: TCP three-way SYN handshake.

TCP and hidden from the application.

The TCP header is shown in Figure 2.4. It is added prior to the IP header. The header length is 20 bytes plus up to 40 bytes of options. The *port number* fields allow multiplexing of multiple TCP flows with a single IP address. By convention port numbers are used by servers as an identifier for specific services (E.g. port 80 for web servers) [21]. Together with the IP header protocol field, the *four-tuple* of source and destination IP addresses and port numbers uniquely identifies a TCP flow on the Internet.

A TCP connection is bi-directional and maintains two entirely independent data streams - so that host is able to reliably send data to the other. Thus both the client and server are capable of being a *sender* or *receiver* in a connection. Features such as *retransmission* or *rate-control* operate independently on each stream.

Connection Establishment and Teardown: TCP is connection-oriented and a finite state machine (FSM) is defined for session establishment and termination. There are eleven states [1] within the FSM that, broadly speaking, represent five distinct phases:

- *Closed:* No connection state exists.
- *Listening:* Listening for a connection request.
- *Opening:* Synchronising a connection between two endpoints.
- *Established:* An open connection over which data can be transferred through the socket.
- *Closing:* Dismantling the connection.

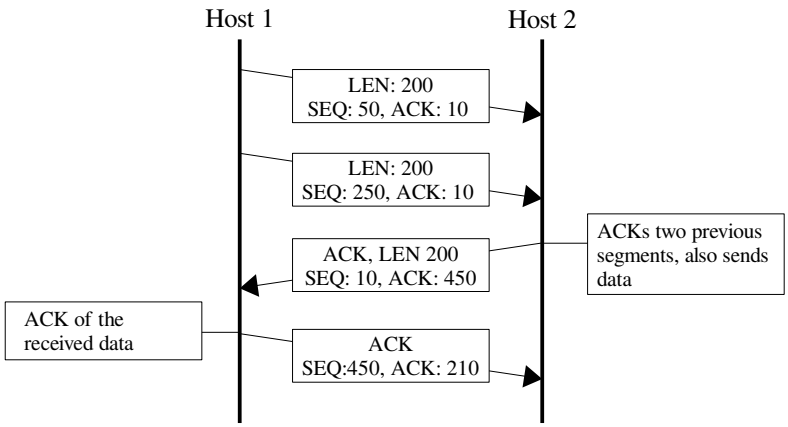


Figure 2.6: TCP four-way FIN handshake

Connections are established by three-way handshake, shown in Figure 2.5. The handshake serves to synchronise the sequence streams and negotiate parameters for the session.

The client host initiates the handshake by sending a packet with the `SYN` flag set (`SYN` *packet*) and a sequence number that will be used to represent the byte stream from the client to the server. The server responds with a packet that has the `SYN` and `ACK` flags set. This `SYN/ACK` packet acknowledges receipt of the `SYN` packet while also including the starting sequence number for the server to client data stream.

Connections are closed by way of a four-way handshake, shown in Figure 2.6. A host indicates that it no longer wishes to send data by sending a packet with the `FIN` flag set (`FIN` *packet*). The remote side must acknowledge all received data up to and including the `FIN` sequence to complete the handshake. Either host may send a `FIN` packet once a TCP connection has been established, however the connection is only *partially* closed until both hosts complete the handshake. A host that has not sent a `FIN` packet may continue to send new data with the expectation that the remote host will receive it.

There are cases where graceful shutdown is not possible. If for example one side of a TCP session loses synchronisation or data is corrupted in-transit, it is necessary to notify the remote host that the connection should be aborted. This is done by sending a packet with the `RST` flag set (`RST` *packet*). A `RST` packet will abruptly close the connection, bypassing the four-way handshake.

Reliable and Ordered Packet delivery: TCP uses sequence numbering to provide ordered and

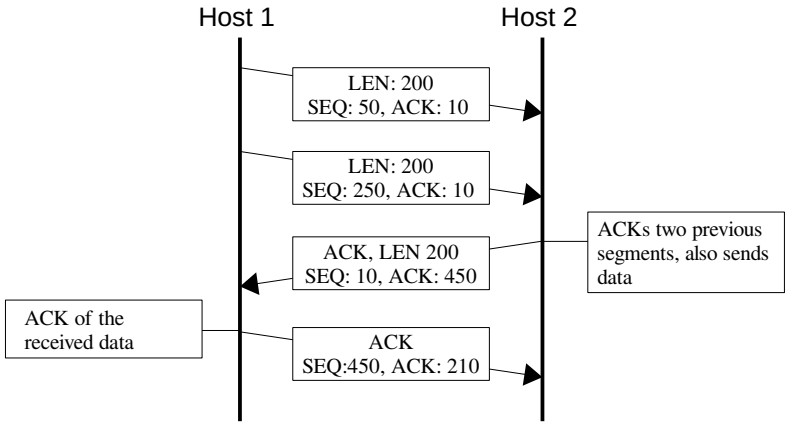


Figure 2.7: A simple TCP data exchange.

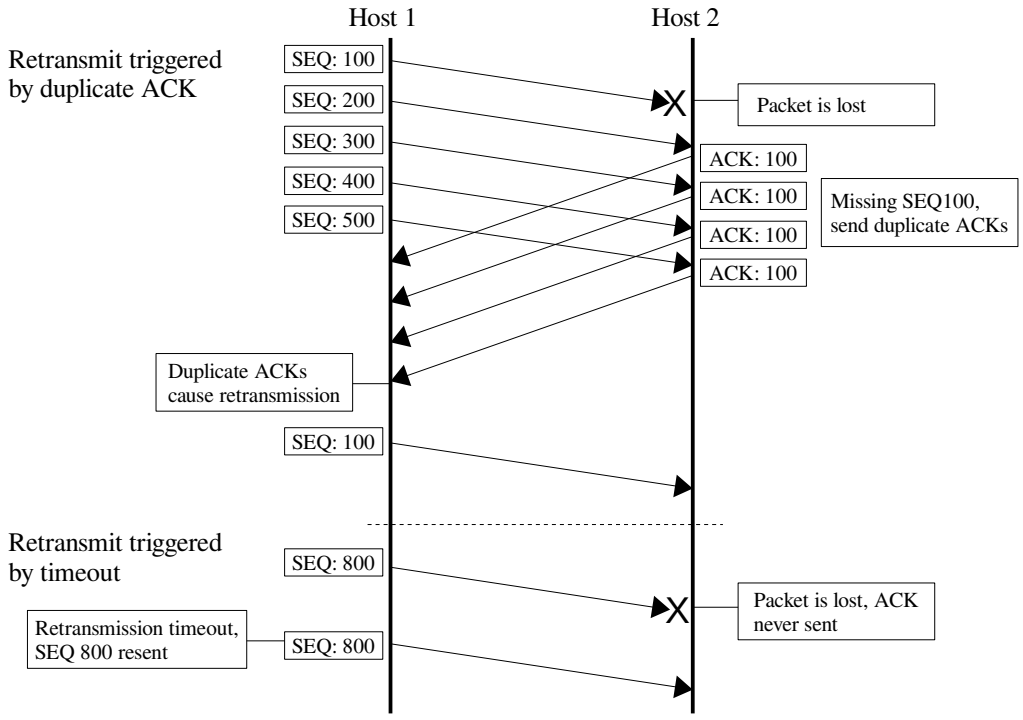


Figure 2.8: Examples of TCP retransmission.

reliable delivery. A sequence number is assigned to each byte in the stream and the receiving host must send positive acknowledgment of each byte received. The acknowledgement informs the sender of how many bytes were received and indicates the next sequence number expected. In the event that the network re-orders packets in transit, the sequence numbers can be used by the receiver to restore the correct byte order. Figure 2.7 shows a basic TCP data exchange on an established connection. In this example Host 1 sends two packets which are acknowledged with the single `ACK`-packet from Host 2. The acknowledgement is piggybacked with a data packet from Host 2 to Host 1. Host 1 then acknowledges this data with an `ACK`-packet that contains no data.

Acknowledgements are also used to signal packet loss. A receiver will send duplicate acknowledgements if packets are received that are not the next expected sequence. After receiving multiple duplicate acknowledgements, the sender will retransmit the missing segment. The absence of acknowledgments is also a potential indicator of packet loss. Each time a packet is sent a retransmission timer is armed at the sender. The duration reflects the time within which an acknowledgement is expected and is based on the current Round Trip Time (RTT) estimates [2] for the path. If the timer elapses before an acknowledgment arrives, the sender assumes the packet was lost and retransmits the unacknowledged segment. Figure 2.8 shows these retransmission scenarios. First is a retransmission caused by duplicate `ACK` packets. Host 1 sends a window of five packets, of which the first is lost (SEQ100). As each of the remaining packets are received, Host 1 sends an `ACK` packet for SEQ100. On detecting the duplicate ACK packets, Host 1 resends SEQ100. In the second example, Host 1 sends a single packet (SEQ800) to Host 2, which is lost. As an acknowledgment is not received the retransmission timer fires, causing SEQ800 to be resent.

Sliding Window and rate-control: The transmission rate of a TCP flow is determined by a dynamically scaled *sliding window.* This represents the window of bytes that can be sent before requiring an acknowledgment. Two mechanisms shape the size of the sender's window: *flow-control* and *congestion control.* The size of the sender's sliding window is the minimum of the flow-control window and congestion control window.

Flow-control prevents the sender from overwhelming a receiver that does not have adequate memory to buffer incoming data. Typically, received TCP segments are not passed directly to the application but are placed into a receive buffer allocated from kernel memory. The application is notified when new data are recorded in the buffer, at which time it may read a portion or all of the contained bytes. The receive buffer has a finite memory limit and the flow-control window is determined by the free space remaining in the buffer (in bytes). The size is shared with the sender via *window updates*

placed in the *window* field of the TCP header (Figure 2.4).

In some instances the capacity of the network path to deliver data is greater than the size of the receive buffer. In such cases the sender's window will be capped at the size of the receive buffer (i.e. the advertised window). It is also possible for data to be delivered at a rate faster than the application reads from the receive buffer. As the buffer fills, window updates are sent to reduce the transmission rate at the sender and prevent a buffer overrun. When the sender's window is constrained by the receiver's window size rather than the capacity of the network, the rate is said to be *receive-window limited*.

Congestion control adjusts the sending window to maximise throughput and reduces the sending rate when the network is experiencing congestion. TCP congestion control has largely been the driver of fairness on the Internet [14]. A *congestion window* (*cwnd*) is calculated based on the current path characteristics, and this determines the amount of data that can be transmitted before an acknowledgement is required. Congestion control for TCP has been studied extensively and there are numerous modifications and enhancements. The following discusses congestion control as outlined in RFC5681 [22].

The size of the congestion window is controlled algorithmically. At the onset of a connection, the *slow start* algorithm is used to discover the approximate capacity of the path. During this phase the congestion window is increased exponentially until either packet loss occurs or a pre-configured *slow start threshold* (*ssthresh*) is reached. At this time the current congestion window is halved and the *congestion avoidance* phase begins. The congestion avoidance algorithm grows the congestion window at a much slower rate then slow start, typically by one segment per RTT. As with slow start, the congestion window is halved when network congestion is detected. The goal of congestion avoidance is to maximise throughput without adding congestion to the network. It does this by increasing the window at a conservative rate when there is no congestion, and reducing the congestion window aggressively when congestion events are detected. If the retransmission timer fires during the congestion avoidance, the sender enters *slow-start recovery*. The `ssthresh` is set to half the current congestion window, the congestion window is set to 1-MSS, and the slow start algorithm again governs congestion window growth. The congestion avoidance algorithm in [22] and derivatives such as *TCP NewReno* are tuned to achieve *flow-rate* fairness.

Fast retransmit and *fast recovery* algorithms are also defined [22] to avoid or improve congestion recovery. These algorithms allow the sender to recover from packet loss without entering slow start recovery. On detecting three duplicate acknowledgements the fast retransmit algorithm resends the

Variable	Description
SND.NXT	Send Sequence: Next sequence number to send
SND.UNA	Send Unacknowledged: Earliest sequence number sent but not acknowledged
SND.WND	Send Window: The window of bytes that the receiver can receive
SND.MAX	Send Sequence Max: The highest sequence number sent
RCV.NXT	Receive Sequence: The next sequence number expected to receive
RCV.WND	Receive Window: Number of bytes that can currently be received

Table 2.2: TCP State Variables

segment beginning at the first sequence number that was sent that has not been acknowledged. This is the segment that is presumed to have been lost. The `ssthresh` is reduced and the fast recovery algorithm controls the congestion window until a positive acknowledgement is received. As a lost packet may have been part of a larger window of packets, further duplicate acknowledgements may be received before the retransmission is acknowledged. Fast recovery takes this into account and inflates the congestion window by one segment for every duplicate acknowledgement received after fast retransmit, allowing the transmission of new data to continue.

Packet loss is not the only signal used by congestion avoidance algorithms to detect congestion, and algorithms may use other indicators, for example delay [23, 24]. The network can also explicitly signal congestion through packet marking schemes. *Explicit congestion notification* [25] is one such scheme that has seen some deployment on the Internet [26].

State variables: Each TCP session must track sequence numbers, state transitions, plus various timers and state flags. Of particular importance are the sequence number variables, through which the core functionality of the protocol (strict ordering, reliability) is derived. The specification [1] thus defines key state-variables for the send and receive sequence spaces, which are summarised in Table 2.2. These variables track the bytes being sent and received by each host, and are used to determine which, and how many, bytes can be sent or received at a given time. We will see later in Chapter 5 that MPTCP inherits the sequence-numbering semantics of TCP.

Options: TCP is extensible, and new functionality is supported via the option field (Figure 2.4), which provides an additional 40 bytes of space for *TCP options*. These are extensions to TCP that enhance the protocol in some way and are typically negotiated during the opening three-way handshake. Some options standardised by the IETF include the *window scale* and *timestamps* options that enable high performance TCP [27], and the *Selective Acknowledgments (SACK)* [28] option that improves

TCP's loss recovery.

2.1.3 Extending Protocols

Protocols typically include some provision for improvement. Networking technologies and applications evolve and extensibility is crucial if a protocol is to remain useful into the future. A common way in which a protocol can be extended is via new header fields. Both IP and TCP have been been improved in this way. In practical terms however the widespread deployment of *middleboxes* since the late 1980s has meant deploying new protocols or even extending existing protocols is difficult [29].

Distinct from routers, middleboxes are not essential for forwarding packets within the Internet, but instead serve specific traffic management purposes by inspecting or altering information in transiting packets. They perform a wide variety of functions at all layers of the Internet protocol stack [30]. Enterprise network requirements for content filtering, security and performance means that a wide range of middleboxes are deployed in such environments [31].

Middleboxes exist to solve a number of problems. Firewalls were developed to provide network security by filtering incoming and outgoing packets. *Network intrusion detection systems* (NIDS) were created in response to a lack of security auditing tools [32]. *Network address and port translators* (NAPTs) allow multiple hosts to share a common publicly-accessible IPv4 address. Performance enhancing proxies may attempt to increase the performance of a connection in a number of different ways - such as optimising `ACKs` or caching segments to expedite retransmissions. A middlebox is not always a distinct appliance.

Middleboxes are deployed to meet specific operational goals for a network and can apply strict bounds on what is considered acceptable traffic. To accomplish this they can break the end-to-end principle by altering packet headers and maintaining connection state independently of the end hosts [29].

Remote network attacks can exploit a wide range of obscure protocol behaviours that may defeat a firewall or remain undetected by a NIDS [33]. In response to this, security policies enforced by middleboxes can be restrictive to the point of preventing common protocol behaviours [33]. It can be easier to apply a restrictive security policy or heavy traffic normalisation[1] rather than attempt to identify all the potential attack vectors.

For example, approximately 50% of the paths tested in [34] discarded packets that included any

[1] A technique of adjusting packet information to fit within a known profile

type of IP options. New TCP options face a similar issue. In [29], an 'unknown' TCP option, representing a new TCP extension, is included in packets sent across the Internet. They find that some middleboxes, particularly application-level proxies, will drop packets containing unknown options. Others forward the packet but strip the unknown option from the TCP header. The paths hostile to IP and TCP options were fewer but non-negligible, and more common in edge networks than core networks. Despite this however [29] concludes that it is still possible to extend protocols such as TCP.

As protocols designed for end-host use must transit through edge networks, interference from middleboxes is a real possibility that must be considered when adding functionality to a protocol. An alternate solution to extending an existing protocol is to tunnel a new protocol over TCP or UDP, such as with QUIC (Quick UDP Internet Connections) [35] or UDP encapsulation of SCTP [36]. This sidesteps the issue of middlebox compatibility, however, existing applications would need to be updated to use the new protocol.

Middleboxes are expensive to purchase, maintain and operate, and adding new capabilities often means refreshing hardware [31]. It can therefore be cost prohibitive to add support for new protocol extensions as they are standardised. The widespread presence of middleboxes means that the Internet is effectively reduced to using only a few existing protocols (TCP, UDP), ports and protocol extensions. This not only severely restricts the ability of designers to create new protocols, but also makes it difficult to extend existing protocols.

2.2 The Network Path

In this section we describe a how a communications path can be established between two endpoints, the characteristics that define a path's behaviour and factors that introduce dynamic behaviour into a path.

2.2.1 Defining the End-to-end Path

The Internet is a packet switched network. Packets are transmitted by an endpoint into the network and contain all the information required for delivery to a destination. *A* path is the collection of network links traversed as a packet moves between routers (*hops*) to the destination. Endpoints can be connected via multiple paths, as shown in Figure 2.9. A stream of packets belonging to the same end-to-end communications session is a *flow.* Packets in the same flow do not have to follow the same path. Links that form a network path are a shared resource. Independent and unrelated flows between different pairs of endpoints are statistically multiplexed together into a flow *aggregate*.

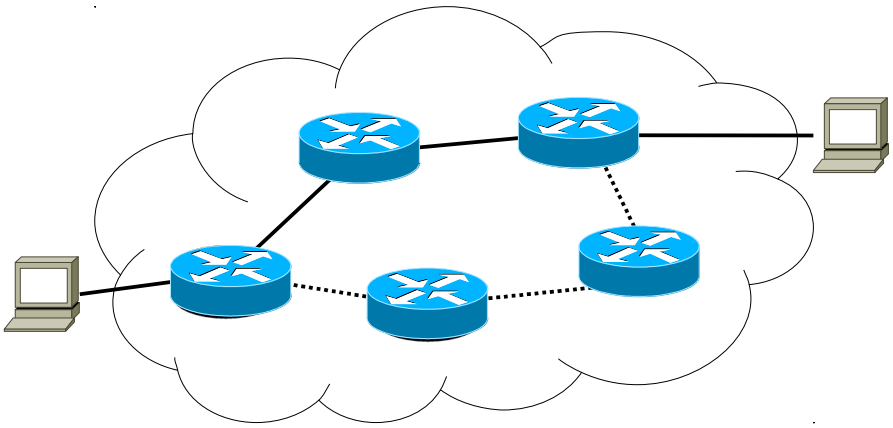


Figure 2.9: Multiple paths between two hosts.

The path itself is established hop-by-hop, with each router deciding what the next hop should be. The next hop is selected via *route table* lookup. Entries in the route table are usually informed by a cost metric, such as number of hops estimated to reach a given destination network. Subsequent connections between the same two endpoints can take a different path, depending on routing decisions in the network. As routers do not maintain per-connection state, paths are not permanent and may change mid-connection, for example in response to link failure. It is common for the forward and return paths to be asymmetric [37].

Paths that exist between two endpoints may be disjoint or share a portion of links. Paths using common links are less redundant, since points of failure are shared. Routers and end-hosts do not know the complete end-to-end path for a flow. If a failure occurs, the endpoints are not aware of the location or even the nature of the failure[2].

Path metrics: Paths and links are described in terms of *bandwidth*, *delay* and *error rate*. Bandwidth is the bit-rate capacity. Delay is the time taken for a packet to reach the receiver (i.e. end-to-end), and is the sum of all the delays incurred along a network path. It is also referred to as *one-way* delay, and includes the propagation, serialisation and processing delays at each hop. The error rate is an expression of how often data bits are altered during a transmission period. The *bandwidth-delay product* (*BDP*) is the product of the bandwidth and end-to-end delay, and represents the amount of data that can exist on the path. If the end-to-end path is abstracted as a pipe, then the BDP represents the amount of data it takes to fill that pipe.

Throughput, delay and *loss rate* are metrics that describe a flow on a path [38, 39]. Throughput is

[2]When permitted, ICMP messages can rely basic diagnostic information to an end-host or router.

the total amount of data received per unit time and is indicative of bandwidth utilisation. Delay can represent one-way delay, *round-trip time (RTT)* and *packet delay variance (jitter)*. RTT is the sum of the forward and reverse end-to-end delays. Jitter represents the variation in end-to-end delay between packets. Loss rate is a measure of the number of packets lost along the path from source to destination over a given time frame. Together these characteristics provide an indication of a path's quality at the time of measurement. Taking measurements of path characteristics can be critical to the performance of traffic management and congestion control algorithms [40].

The definition of a given characteristic can also vary depending on the measurement context, which [39] divides into *network*, *flow* and *end-user*. Network context measurements take the perspective of network devices such as routers. For example delay might measure the forwarding time for a packet through a router. The flow context describes the characteristics of a single flow from the perspective of the endpoints. The end-user context relates characteristics to the end-user experience.

Link technology: The links in a path may combine a mix of underlying network technologies, impacting path characteristics in different ways. The links connecting hops in the path may have different capacities. A slow link can restrict the maximum throughput that can be achieved along the path. Link transmission protocols can effect delay and loss rates. For example flows on DOCSIS access links can experience higher jitter than ADSL links [41]. Wireless networks suffer from errors due to interference, signal attenuation and channel contention. Link-layer behaviours can also interact with higher layer protocols. For example TCP has been shown to obtain poor throughput in wireless networks by falsely interpreting congestion [42, 43].

Queuing: It is possible for the aggregate rate of incoming packets to exceed a router's forwarding capacity. Packet arrival rates fluctuate and aggregate throughput can spike significantly over short timescales [44]. This necessitates the use of *queues* to buffer packets briefly until transmission is possible. A longer queue can result in reduced packet loss and increased link utilisation. A queue should fill only during bursts. The cost of queuing is additional forwarding delay for each packet. Queues add delay and therefore increase the BDP of the path.

Bottlenecks and congestion: A bottleneck is the slowest link in a communications path. The end-to-end throughput of a connection is limited to the bandwidth available at the bottleneck link. Bottlenecks can occur at any point in the network path. A bottleneck can be a slow link, or a link whose aggregate input exceeds output capacity. Bottlenecks also form due to misconfiguration. For example incorrectly configured wireless access points can limit the performance of many home broadband connections [45].

2.2.2 Sources of Dynamic Behaviour

Unlike the provisioned, predictable paths of a traditional telecommunications network, the characteristics of a path across the Internet are not known in advance and may change. Path characteristics are shaped by a number of factors such as geographical distance and the underlying link technology and bandwidth. In addition to these are the more dynamic influences of queuing, link-layer mismatches, cross-traffic from multiple endpoints and adaptive protocols. These interactions drive changes in delay, loss and throughput. In this section we describe some key causes of this dynamic behaviour.

Cross-traffic: Given a flow between two endpoints on a shared link, cross-traffic consists of packets that belong to *other* flows. A flow must compete for network resources against other flows in the aggregate. Cross-traffic comes from a range of source applications that can generate packets of different sizes and at different rates. Flows within the aggregate may change on a hop-by-hop basis, meaning a flow can compete with different cross-traffic at different points in the network.

The duration of cross-traffic flows also varies. The majority of network flows exists for less than a few seconds [46]. These short sessions can be signaling-related UDP flows [18] or TCP-based web-traffic. Short TCP flows do not transition to congestion avoidance [46]. The majority of traffic *volume* in an aggregate belongs to comparatively few long-lived TCP flows, which also consume a higher ratio of resources [47, 48].

Adaptive-rate protocols: Adaptive-rate protocols vary the data transmission rate of a communications session based on the level of bandwidth or congestion perceived in the path. They are essential to resource sharing and in preventing congestion collapse in the Internet [49]. By probing the path to estimate bandwidth and reacting to congestion, an appropriate sending rate can be set that maximises path utilisation. TCP congestion avoidance is the most widely used adaptive-rate transport protocol on the Internet. A popular application-layer adaptive-rate technique is *HTTP adaptive streaming* (*HAS*), used by on-demand video streaming clients [50, 51]. YouTube and Netflix, which now account for a substantial volume of the Internet's total traffic volume [52], use HAS.

When devising new congestion control algorithms, fairness with other flows is considered a key performance metric [39]. Despite this, congestion control appears to be unfair when many flows share a bottleneck [53]. For example long-duration, high-volume or low-RTT connections receive a greater share of bandwidth [54, 55]. TCP variants may not be fairness-compatible with other TCPs. For example TCP Vegas flows have been shown to perform poorly at shared bottlenecks with TCP NewReno [56, 57]. Protocols can also be intentionally modified to more aggressively use the available bandwidth, ignoring fairness principles for higher performance.

Adaptive-rate flows can interact, causing behaviours such as global synchronisation [49] or uneven allocation of bandwidth at bottlenecks. Network conditions can also cause adaptive-rate protocols to react in unexpected ways, for instance rapid changes in RTT can result in spurious retransmissions when the RTO estimate of a TCP connection is not able to adjust in time [40].

There can also be interaction between transport-layer and application-layer adaptive protocols, such as a HAS client using TCP transport. HAS clients uses HTTP requests to fetch sections of video - perhaps tens of seconds at a time - starting with the lowest bitrate encoding. As the TCP congestion window evolves the application can request a higher bitrate encoding. A client may fetch enough video playback that the connection remains idle for a time. If this time is greater than the RTO calculated by TCP, many implementations will reset the congestion window and all estimates of path characteristics. On fetching the next sequence, the connection will have returned to slow start.

Queing: A basic queue services packets in a *first-in first-out* (*FIFO*) basis, dropping packets that arrive when once queue is already full. This type of queuing is referred to as *drop-tail*. FIFO/drop-tail queues are common in edge network routers [41]. A router on a link that is not sufficiently provisioned will queue packets during steady-state loads, adding to end-to-end delay. The ability to absorb bursts is also diminished, leading to increased packet loss and poor throughput. Appropriate sizing of router buffers is critical to flow performance and end-user quality of experience, particularly for applications that are sensitive to delay.

The cost of memory for network devices has decreased over time, meaning input queues can be sized much larger than previously possible. There has been a preference towards using over-sized, unmanaged FIFO/drop-tail buffers in edge networks [41, 58, 59]. If such queues are not properly managed, or fail to provide timely congestion feedback, they can grow to sustain high occupancy without ever draining. Allowing *standing queues* to form adds latency, increases jitter and more importantly strips adaptive-rate protocols of necessary congestion signals.

Inappropriately large queues have been shown to add hundreds of milliseconds (or even *seconds*) of delay [59, 41] without dropping packets. The consequence of this is that a sender will continue to grow their window, in turn adding more delay [60]. In recent years the problems related to over-provisioned and unmanaged queues have coalesced under the term *bufferbloat* [61]. Research is ongoing, with various approaches to buffer sizing [62, 63, 64, 65, 66].

Excess queuing is not restricted to edge and core networks, and can be caused by edge devices such as home routers that are pre-configured with large buffers. Keribich et al. [58] observe that a cable modem may arrive with a buffer configure for 50Mbps upstream, only to be connected to a

1Mbps link. Thus the buffer is vastly over-provisioned and likely to introduce queuing delays.

2.2.3 Traffic Engineering Mechanisms

Traffic engineering aims to simultaneously improve the end-user experience and maximise use of deployed resources. This is typically achieved through a service differentiation model which divides traffic flows for class-specific treatment. This is in contrast to the original Internet service model of a single-class best-effort network, which has largely relied on the use of rate-fair TCP congestion avoidance for controlling flows [49]. Network-based traffic engineering is in-part motivated by perceived shortcomings of the end-to-end best-effort service model:

- The Internet has a wide variety of applications whose flows have diverse characteristics and service requirements. If flows are treated equally, the performance of some applications (such as delay-sensitive or interactive) can be sub-optimal. Providing adequate performance for all traffic types in a best-effort network can require over-provisioning of links.

- Network infrastructure is expensive and best-effort service does not grant operators explicit control of their own resources. Service levels vary as traffic loads change and cannot be guaranteed.

- End-to-end fairness mechanisms typically aim to share resources on a *per-flow* basis. The content, duration and packet profile of a flow are not taken into consideration. It is not possible to provide a higher level of service to certain traffic classes.

- End-hosts are not compelled to act fairly and end-to-end based fairness is open to exploitation. For instance a single TCP application opening multiple parallel streams can easily attain additional bandwidth at a bottleneck. A best-effort network cannot enforce fairness or penalise abusers.

Traffic engineering allows network operators to better manage the available capacity and provide *Quality of Service (QoS).* QoS schemes consider the user experience or performance requirements of flow and can provide provide specific service-level guarantees, such as bounds on delay or minimum throughput. The traffic engineering process can be broken into several important components: classification, steering, queue management and scheduling, which are discussed below.

Packet classification: The first step in providing QoS enhancements is to differentiate packets into service classes. Each service class is defined by the path requirements of the application.

Applications can be divided into several broad categories such as *delay-sensitive*, or *delay-tolerant.* Performance and user-experience are bound by path characteristics. For example a delay-sensitive application will have a maximum tolerable delay or jitter, beyond which the user experience is diminished.

Packets can be classified based on a number of attributes such as transport protocol, payload size or flow duration. The descriptors are typically indicative of the traffic class. For example an delay-sensitive flow could have short, constant rate packets delivered using UDP. Packets may be classified and marked by a dedicated classifier for treatment by downstream nodes or classified and treated by a single device without any wider cooperation.

Traffic steering: The most basic form of traffic engineering is the use of routing to direct traffic aggregates through or between networks. The goal is to efficiently distribute traffic across network resources by using the most effective path. This reduces congestion and improves resource utilisation. Routing within a single network domain is referred to as *interior routing* while routing between networks is *exterior routing.*

Open Shortest-Path First (OSPF) [67] is a popular interior routing protocol. OSPF uses a weighting factor for each link to calculate the lowest-cost next-hop and installs these in the routing table. Weights can be defined manually or algorithmically [68] and can be based on link capacity or similar metric. Protocols like OSPF can be extended with additional traffic engineering functionality [69].

When a network has multiple external gateways, exterior routing protocols like *Border Gateway Protocol (BGP)* are used to select the inbound and outbound links for traffic. A router running BGP learns the path to different destinations based on route advertisements received from neighbouring autonomous systems. Routes or transit to other networks can be announced by sending route advertisements. Path cost is based on a combined set of attributes that can include hop-count, link-availability, administrative policy or an arbitrarily defined metric.

By default routers forward packets according to the destination address. *Policy-based routing (PBR)* allows administrators to configure specific forwarding policies based on information other than destination address. First a PBR filter classifies packets using network/transport header information or packet attributes such as packet size. A matching packet can then be forwarded according to class policy. PBR can be used to separate traffic for QoS, for example routing UDP and TCP flows to the same destination network along different paths. Packets not matching any PBR criteria will rely on a destination-based route table lookup for the next-hop.

Load balancing is the process of distributing traffic loads across links and is closely associated

with routing. As mentioned, routing tables are populated with the lowest-cost routes, meaning packets are always forwarded along the lowest-cost route. If there are multiple paths with the same cost, then the router can dynamically choose between them. It is possible to intentionally engineer multiple equal-cost paths for a destination, allowing for the use of a load balancing algorithm. Several load balancing approaches are described in [70], such as performing a hash on packet head fields and using the outcome to determine the next hop.

Multiprotocol Label Switching (MPLS) is a label encoding and switching framework that simplifies intra-domain routing and provides traffic engineering functionality [71]. An MPLS router at the ingress or egress of a domain classifies a packet into a forwarding equivalence class (FEC) and encodes this into a label attached to the packet. FECs are commonly defined by matching part or all of the destination address, but can also be based on non-header information. Routers within an MPLS domain forward packets based on this label without referring to network layer headers. Labels can be configured to provide an FEC with a fixed path through the domain. The path can be defined by administrators or calculated dynamically from state information taken from routing protocols. A label distribution protocol [72] allows routers to reach agreement as to the meaning of labels. FECs themselves do not explicitly define a class of service but can be used for this purpose, labels can be configured to provide a fixed-path for FEC-based QoS.

Source routing allows a host to specify the route for a packet. IPv4 has two such options, *strict source and record route (SSRR)* and *loose source and record route (LSRR)* [19]. SSRR explicitly specifies each hop in the path. LSRR provides a subset of path hops that must be visited. The *routing header* extension (now deprecated [73]) of IPv6 provided similar functionality to LSRR [74]. The IP source-routing extensions have serious security issues [75, 76] and forwarding packets with these options is discouraged [75]. Several improvements have been proposed [77, 78].

Queue management and packet scheduling: Recall from Section 2.2.2 that unmanaged queues discard packets only once the queue has reached full occupancy. The problems associated with this method of queuing have been known for some time, resulting in the development of *Active Queue Management (AQM)* techniques [49]. The goal of AQM is to maintain low queue occupancy by selectively dropping or congestion-marking packets before the queue fills. Shorter queues can reduce forwarding delay and jitter, improving the end-user experience. AQMs have historically required some tuning and it is unclear how widespread deployment is, though [41] finds evidence of limited deployment in several major ADSL access networks in the United States.

Random Early Detection (RED) [49] is a widely available drop-tail AQM. RED manages queue

length by probabilistically signaling congestion avoidance in response to growing queue occupancy (usually by dropping a random packet as it arrives at the queue). The probability of dropping a packet increases based on the size of the queue and the duration of queuing (bursts do not trigger higher probability of loss). At full occupancy all arriving packets are discarded.

Packet scheduling algorithms select which packet should be sent next from a queue of packets. Scheduling aims to improve the service quality of flows at a bottleneck link through fine-grained control of link bandwidth. Schedulers can be designed for fair bandwidth distribution, rate shaping or prioritisation. Schedulers operate on one or more packet queues. In the case of a single queue the scheduler can control the timing between transmissions. If multiple queues are in use the scheduler selects the next queue from which to transmit from and the number of packets to send. A priority scheduler sends packets from a high-priority queue first. A round-robin scheduler might send a packet from each queue in sequence.

Packet scheduling is often combined with *fair queuing* or *class-based queuing*. In the fair queuing model, packets are sorted into individual drop-tail queues [79] and serviced by a round-robin scheduler so that bandwidth is evenly distributed between flows. *Weighted fair queuing* adds a weighting factor so that bandwidth can be specified per-queue. *FairQueue-CoDel* [80] combines a credit-based fair queuing scheduler with the CoDel [81] AQM algorithm.

Differentiated services (DiffServ) [82] is a class-based preferential scheduling and routing scheme. End-hosts or classifier nodes at the edge of a DiffServ domain mark packets by placing a class code in the *differentiated service* field of the IP header. Classes are coarse-grained and define *per-hop behaviours* (PHB) for DiffServ nodes. The PHB dictates the actual queuing and scheduling treatment required at each node. Several standardised classes exist and include *expedited forwarding* [83] for delay-sensitive traffic and *assured forwarding* [84] for traffic with minimum bandwidth requirements. Schedulers can be priority-based or rate-based (e.g. fair, weighted) [85].

Packet scheduling can also be implemented at the end-host. *Packet pacing* [86, 87] is one technique that aims to reduce the burstiness of TCP. TCP will typically burst back-to-back as many packets as the window allows. Depending on the RTT the link may be left idle between bursts. Packet pacing attempts to spread out the transmission window over the RTT of the connection by spacing the transmission of each packet.

2.3 A Multihomed Future

A host with multiple IP address attachments is multihomed. Multihoming is key to a functioning Internet - after all, routers must be multihomed in order to connect different networks. The relative expense of networking hardware meant that initially the Internet edge was largely without multihomed end-hosts. It is now common for end-hosts to have multiple interfaces. Smartphones are a prominent example of this, having cellular and WiFi interfaces capable of simultaneously connecting to different IP networks. A report [88] by the Internet Society shows that smartphones are already the predominant mode of web access within the United States, and this is expected to be the case worldwide in coming years.

Multihoming creates a fundamental shift in end-to-end communications. Traffic engineering mechanisms that were once the purview of the network can now be implemented at the end host. Consequently, a multihomed host has several advantages over a single-homed host:

- A flow can spread packets across multiple paths simultaneously for greater throughput.
- Paths can be reserved for fault tolerance or load sharing.
- Local policy can influence path selection.

Strictly speaking, end-hosts with multiple IP addresses can be multihomed or *multi-attached*. A multi-attached host has multiple IP addresses on the same subnet [89]. A multihomed host has attachments to multiple networks. In this thesis we include any host that has multiple IP addresses under the definition of multihomed.

Chapter 3

Multihoming at the End-host

We can already see examples of multihoming at the end-host. Datacentre servers are multihomed for greater throughput or failover. 'Wi-Fi assist' [90] in Apple iOS allows a cellular connection to take take over when a WiFi connection fails. However a lack of a multipath aware end-to-end protocol means that we have yet to see many of the potential benefits that multihoming can provide. TCP, the primary transport protocol of the Internet, was designed for single-homed hosts.

There are existing protocols that provide multipath capabilities in some form. A number of end-to-end solutions previously proposed may now be worth revisiting. This leaves a question as to which approach provides the most functionality and stands the best chance for future deployment. In this chapter we first discuss packet steering at the end-host and the impact of path characteristics on multipath connection performance. We then compare a number of existing multihoming and mobility solutions at each layer of the Internet suite.

3.1 Steering Packets

Path selection has traditionally been in the hands of the network. Excepting cases in which a differentiated services model is employed, routing decisions are not typically optimised for the needs of any particular end host or application. A multihomed host has the ability to choose what it considers is the best path available. Path selection can be achieved through localised routing policy or through dynamic scheduling and flow-control mechanisms.

A multipath scheduler allows a host to distribute a data stream from a single application across multiple output interfaces. This is distinct from the network-based schedulers previously discussed, which multiplex packets from different input queues onto a single output link. A simplified multipath

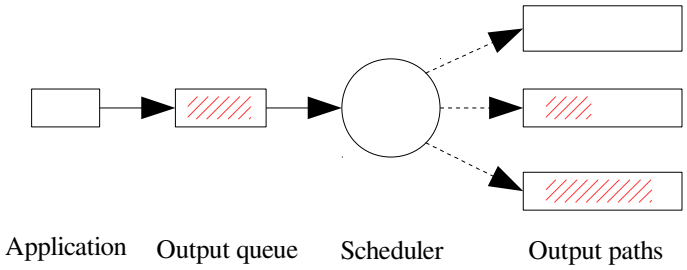


Figure 3.1: A simplified multipath scheduler.

scheduler is shown in Figure 3.1. In this example the application writes data to the output buffer, at which time the scheduling algorithm selects the output path (e.g. sequentially). The data is then transmitted. The scheduler controls how much data is sent, on which path, and when. A scheduler can be combined with a flow-control mechanism for adaptive-rate behaviour. In this arrangement the scheduler only selects the transmission path - the flow-control algorithm determines how much can be sent and the transmission interval.

Previous discussion showed how flow-control algorithms probe the network path and adapt the transmission rate to maximise throughput. In a multipath scenario this can be used to dynamically adjust how much data a scheduler can queue on a particular path. For example, if striping across multiple TCP connections, a scheduler could allocate according to the size of the congestion window, thus sending more data on the faster path. A more advanced form of flow-control can be attained through the application of the resource pooling principle [91]. Pooling treats all paths as a single resource. The flow-control algorithm is aware of conditions on all the paths and can thus more accurately steer where data is to be scheduled by encouraging the use of the best path (e.g. by increasing the transmission window of that path).

Ultimately the benefits of using multiple paths in a connection depends on the paths available. These could be disjoint or share a common bottleneck. A set of disjoint paths might have a large disparity in bandwidth and delay. The end host should therefore attempt to infer path conditions to inform scheduling decisions. The type of information available to the scheduler depends on the layer in which the scheduler operates. A link layer scheduler has a vastly different world view to an application-level scheduler.

To illustrate, consider an application-level scheduler striping data over multiple TCP sockets. The space in the output queue of each socket controls how much data can be allocated to a path. The

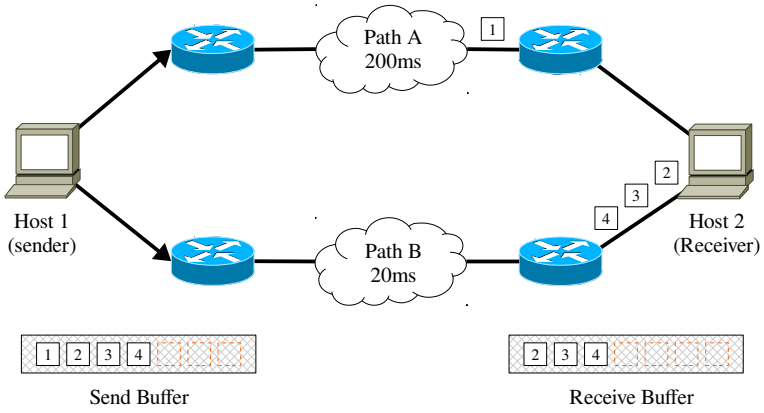


Figure 3.2: Data striping across asymmetric paths can lead to packet reordering.

transmission rate is governed by the TCP *ACK-clock*[1]. A scheduler integrated in the transport layer might look at the available congestion window when selecting a path. Such a scheduler could also use statistics calculated by the flow-control algorithm, such as RTT, when selecting a transmission path.

3.2 Multihoming and Path Diversity

Internet paths are dynamic. At any given time there are a number of mechanisms in play, both at the endpoints and within the network, that interact to alter the characteristics of an end-to-end path. The paths between multihomed hosts are thus very likely to have different characteristics. This is especially true in cases where multi-mode network access technologies are used, for instance the combination of Wi-Fi and cellular interfaces.

For multipath communications, the primary problem arising from diverse paths is out-of-sequence packet arrival. It is possible for packets to be re-ordered along a single path (for instance due to packet loss), however it is more likely to occur if packets are routed or striped across multiple paths. The main cause of packet reordering in multipath communications is differences in path delay caused by RTT or bandwidth asymmetry.

Figure 3.2 shows multihomed end-hosts connected via two disjoint paths, where Path A has higher propagation delay than Path B. Host 1 sends four packets to Host 2: packet 1 is sent on Path A, while packets 2-4 are sent on Path B. Due to the delay difference, packets 2-4 arrive before packet 1. This presents several problems. For example if the receiver requires data in the order transmitted (e.g. using TCP transport), then the packets must be buffered until packet 1 arrives. Alternatively an

[1] Acknowledgements create space in the sender's window, allowing the transmission of new data.

application with timeliness constraints may process packet 2-4 and discard packet 1 when it arrives.

Path loss rate also contributes to packet reordering. Striping data where one path is lossy will result in out-of-sequence delivery. For reliable protocols the time taken to retransmit must be taken into account, effectively acting like additional delay on the path.

Packet reordering can degrade performance in several ways. Foremost it effects the timeliness of delivery, increasing delay and adding jitter. Buffering compresses received segments in time - meaning data is delivered to the application in chunks. Protocol-specific problems can also occur, such as retransmits being falsely triggered. In the following paragraphs we detail some common problems for ordered protocols created by out-of-sequence packets.

Head-of-line blocking occurs when received data cannot be delivered to an application as the next expected packet has not arrived. Packets must be buffered until in-order. Head-of-line blocking is symptomatic of schemes that spread data across multiple paths [92, 93, 94]. Returning to the previous example, we see the receive buffer contents of Host 2 in Figure 3.2. In this example the receiver was expecting packet 1 to arrive but instead received packets 2-4. Packets 2-4 must be held until packet 1 arrives. Note that the buffer has a finite space. There are a number of issues that arise as a result of this. Immediately we can see that the goodput[2] of the connection will be limited by the delay of Path A. The size of the receive buffer is also a consideration, as Host 1 may continue to send packets on Path B. If the buffer is not sized appropriately it may be fully consumed. A full buffer can cause a number of subtle effects on adaptive protocols that use sequenced, flow-controlled windows.

Window blocking is one such problem that results from out-of-sequence data occupying a large portion of the receive buffer. Recall that sliding-window protocols like TCP maintain separate transmission (or congestion) and receive windows. The actual window of data that can be sent at a given time is the minimum of the transmission window and the receive window minus any in-flight data. As the receive buffer fills with out-of-order data the sending window shrinks, since no data is being acknowledged. Once the window is fully transmitted no further data can be sent until an acknowledgement is received. In asymmetric multipath connections this has the effect of throttling the higher performing path, as the window can only be advanced once the data causing head-of-line blocking has been received and acknowledged.

Send buffer blocking is another side-effect of a full receiver buffer. A send buffer is typically large enough to keep the path fed with data. To support reliability, segments must stay in the send buffer until they have been acknowledged. When head-of-line blocking occurs, data that has been

[2]The bytes received per unit time at the application level

successfully received cannot be acknowledged. The result is that the send buffer must hold onto these segments until all the blocking packet is received. Looking again at Figure 3.2, packets 1-4 are retained in the send buffer at Host 1, even though 2-4 have been received and could have conceivably have been acknowledged even before packet 1 reaches Host 2.

Given the increased likelihood of packet reordering on multipath connections and the issues that stem from this, send and receive buffers should be adapted for use with multiple paths. A conventional approach is to size the buffers at least equal to the BDP of the link. A buffer less than the BDP could limit the throughput of the link. This equation increases for multipath transport - the receiver must be able to absorb the BDP of the combined paths, taking into account the delay of the slowest link. A high delay path therefore has to potential to scale the buffer size requirements dramatically. An insufficient buffer will cap the throughput of a connection, with slow paths limiting the utilisation of faster paths. However increasing buffer sizes is not ideal for memory constrained devices or servers with large numbers of concurrent sessions. Optimal performance requires smart scheduling decisions so that buffer sizes can be minimised.

3.3 Multihoming and Multipath Solutions

Internet suite protocols in use today were designed before the multihomed end-hosts became common. Some, such as TCP are fundamentally designed for single-interface operation. For instance a TCP connection is bound directly to specific source and destination IP addresses and ports that cannot change without breaking the connection. There are however protocols within the Internet stack that can leverage a multihomed end-host for multipath transfer in some capacity. In the following sections we discuss current solutions according to abstraction layer.

3.3.1 Link-layer

Link-layer multihoming solutions aggregate the capacity of multiple network interfaces. Typically the interfaces are aggregated under a single virtual interface and share a MAC address. Frames are passed to the virtual interface and a scheduling algorithm determines the physical interface for transmission. Link layer schemes are limited to a single hop (i.e. the next device directly connected at the link layer).

Ethernet Link-Aggregation: Ethernet is a widely supported link-layer protocol. Link-aggregation is the bundling of multiple Ethernet interfaces into a single logical interface. Datacentres and enter-

prise networks can use link aggregation to spread network traffic load, provide redundancy or gain throughput.

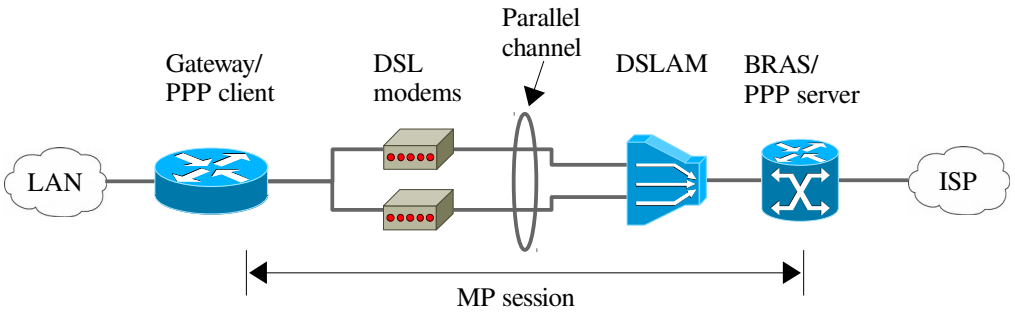
The Link Aggregation Control Protocol (LACP, defined in the IEEE 802.1AX), is an open standard with wide hardware support. LACP simplifies link-aggregation configuration by allowing devices to automatically negotiate with other directly connected LACP devices. In the event of link failure, failover is performed dynamically. Vendor-specific aggregation techniques also exist.

In LACP terminology, data striping is referred to as *frame distribution*. The frame distribution algorithm is implemented in the driver that aggregates the links. The LACP specification does not define a frame distribution algorithm, however discourages schemes that cause frame duplication or re-ordering, since any re-ordered frames will be forwarded to the destination as such. The default behaviour of LACP in FreeBSD and Linux hosts is to provide flow-based load balancing and automatic failover [95, 96].

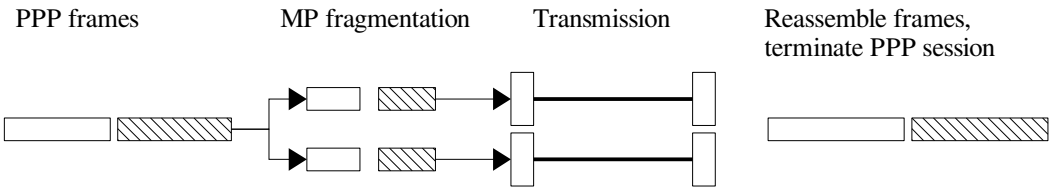
To prevent re-ordering of data frames, link-aggregation devices can identify upper-layer *flows* and assign them to a single interface. The definition of a flow may vary, however will typically at least include the source and destination IP addresses in the packet header. The *lacp* mode of the FreeBSD *lagg* driver identifies flows using a hash of the MAC header and IP source and destination addresses [95]. A basic load balancing algorithm might distribute flows based on the calculated hash or via round-robin selection. More advanced algorithms could consider any number of metrics (such as link load) to more intelligently distribute flows.

Load balancing and failover devices are available by default in operating systems such as FreeBSD and Linux. The *failover* mode of the FreeBSD *lagg* device provides transparent failover between network interfaces. First a virtual interface is created, then physical interfaces are tied to this (for example an Ethernet and wireless 802.11 interface). The MAC address of any additional interface is overridden with that of the master interface. If the master interface fails, the next interface in the hierarchy takes over. Data is sent via the master interface if and when it regains availability. This failover mechanism provides higher-layer persistence (since IP and MAC addresses are constant) and is controlled solely by the host. It does not however provide persistence when changing networks, as a change in IP address would break existing sockets.

Ethernet link-aggregation can be used in networks that require high performance and/or redundancy. However the benefits exist for a single-hop only and LACP must be enabled at both ends. Thus for an end-host there is no guarantee of path diversity beyond this hop. LACP supports redundancy but does not attempt to solve host-mobility.



(a) A site-wide multihoming topology that uses MP at the gateway.



(b) Fragmentation and reassembly of PPP frames.

Figure 3.3: Multilink PPP (MP)

Multilink PPP: Point-to-Point Protocol (PPP) [7] was extended with Multilink PPP (MP) [97] to combine multiple physical access links (e.g. ISDN, voiceband modem). MP is currently supported in enterprise/carrier routers and used for applications such as DSL access links, as shown in Figure 3.3a. MP provides multipath transmission by fragmenting PPP frames and distributing them across the available access links. This process is shown for the MP-over-DSL scenario in Figure 3.3b. The design takes into consideration the byte sequence order of higher-layer protocols and explicitly handles frame re-ordering at the MP termination point.

The standard does not specify a scheduling policy, though features such as data-striping, load balancing and link scaling are typically supported. MP extends the PPP header with an additional header comprising a flag that allows the receiver to detect fragmented frames, and a 24-bit sequence number field to enable reassembly and loss detection. Fragments that arrive out-of-order are held in a buffer and re-ordered according to sequence number. Fragment reassembly is comparable to TCP segment reassembly, however there is no retransmission mechanism and a frame is discarded if it is not reassembled when the first fragment of the next frame arrives [97].

As MP is designed for point-to-point links, the parallel path is likely to exist for only a single hop before being terminated at the PPP server. It is not an end-to-end multipath solution. As the

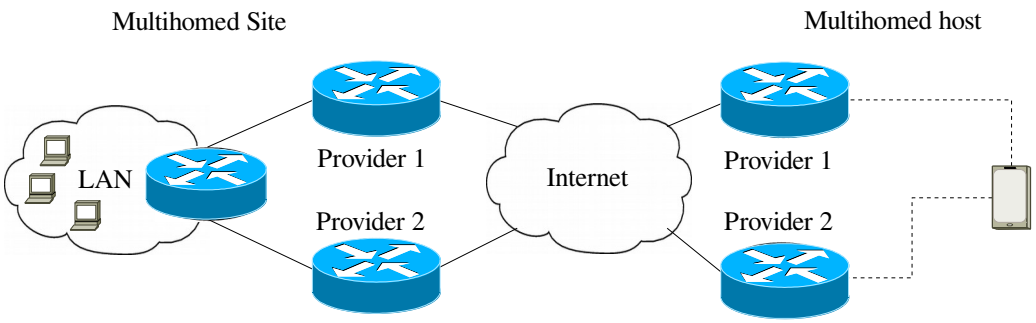


Figure 3.4: Site-wide and end-host multihoming. A multihomed gateway provides multihoming for an entire site. A multihomed end-host is directly attached to multiple providers.

PPP session must be terminated at a PPP server, this becomes a bottleneck that represents a single point of failure. MP must be explicitly enabled and configured on the client and server side. Despite the flexibility in data scheduling and failover, MP does not work when an end-host moves between networks, or if the host interfaces are configured in different networks.

3.3.2 Internet-layer

There are several factors that make multihoming and mobility support at the Internet layer appealing. The IP address is a device's point of attachment to the Internet. The network prefix of this address is the *home* network. A device may have multiple interfaces that may be assigned addresses in distinct networks. A network gateway can be attached to multiple transit providers. Importantly, the Internet layer allows steering of packets independently of higher-layer protocols.

Site-wide Multihoming: Site-wide multihoming describes a LAN or AS connected to the Internet by more than one transit provider. Hosts within the network are typically single-homed, meaning path selection occurs at the gateway router. Path selection is transparent to hosts within the LAN. Figure 3.4 compares site-wide multihoming to end-host multihoming. site-wide multihoming provides redundancy and load balancing for the entire network. It also allows for policy enforcement in the inbound and outbound direction. For example routing updates can be used to influence path selection in the inbound direction, while cost metrics can be applied to steer outbound traffic.

A key benefit of this method of path selection is that it preserves the transport-layer connections of the end-host [89]. For example a multihomed gateway can shift an end-hosts TCP flow between transit providers without breaking the connection. As path selection is performed using source or

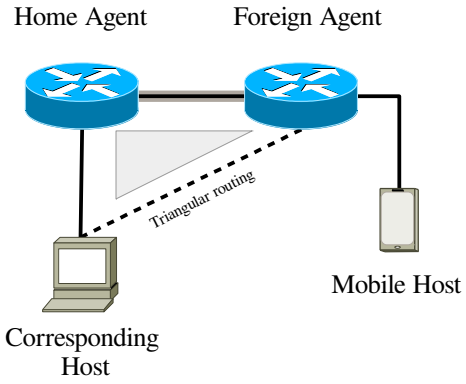


Figure 3.5: IPv4 mobility requires additional infrastructure to tunnel connections.

destination addresses, transport-layer conditions, such as TCP's sliding window, are not taken into account. Scheduling decisions are likely to place more importance on what benefits the network in aggregate rather than what is optimal for an individual end-host.

Mobility for IPv4 and IPv6: Mobility for IPv4 [98] uses a *home agent* and *foreign agent* to tunnel connections to a mobile host, shown in Figure 3.5. Upon detecting that it has entered a foreign network, the mobile host discovers a *care-of* address and registers this address with the home agent. The care-of address is not generally assigned to the host itself[3], but is the logical endpoint of the tunnel between the home and foreign network (typically the foreign agent). The care-of address may be reused for multiple visiting hosts.

Packets to the mobile host are then tunneled via the home agent and foreign agent. Triangular routing occurs on the return path if the foreign agent routes packets directly to the corresponding node. Existing transport-layer sessions can persist through this registration period. New connections to the home address of the mobile host are tunneled by the home agent to the foreign agent, which terminates the tunnel and forwards packets to the mobile host. Where possible the return path from the mobile host is routed directly to the connected endpoint, referred to as *triangular routing,* or tunneled back to the home agent in cases where source address filtering prevents a more direct return route.

After terminating the tunnel, the foreign agent forwards packets to the mobile host with the home address as the destination. A drawback of this method is that although the host IP address remains fixed (and thus preserves existing connections), traffic must *always* pass through the foreign agent and will pass through the home agent in at least one direction.

[3]Though multi-interface devices may obtain a local address via DHCP or some other method.

The IPv6 standard has several improvements over IPv4 in handling multihoming and mobility. Expanded multihoming support means that mobile hosts (even with a single interface) are able to obtain multiple IPv6 care-of addresses. As a result, the foreign agent is only required for per-existing connections, or for connections to the home address of the mobile host. Route-optimisation allows the mobile host to update the corresponding host with the current care-of address. The care-of address is then bound to the home address of the mobile host. The binding is then used to substitute the home address with the current care-of address when sending packets. These are then routed directly to the foreign network.

IPv4 and IPv6 Mobility solutions require additional infrastructure deployment and widespread co-operation amongst network operators to work effectively. They do not support multihoming features such as data striping or failover.

Shim6: Shim6 [99] is an end-host solution to multihoming. It enables end-hosts with multiple IPv6 addresses to directly negotiate additional addresses with the corresponding node for the purpose of failover and load balancing. It does not however support simultaneous use of multiple available paths, as doing so would potentially impact the underlying transport protocol.

Shim6 provides a working solution to end-host controlled multihoming, however it is limited to IPv6 networks only and does not currently support data-striping or host-mobility.

3.3.3 Transport-layer

The transport layer is located between the application and the network interfaces. From here protocols are able to measure the end-to-end characteristics of a path while also having direct access to data in the socket send queue. Multipath data scheduling decisions have been shown to benefit from transport layer information [100]. A number of transport-layer solutions for mobility and multipath have been proposed over the years [101, 102, 103, 104, 105]. The two most promising are SCTP and MPTCP.

Stream Control Transmission Protocol: SCTP [106] is a reliable multihomed transport that transmits data as an ordered, partially-ordered or unordered sequence. This design was motivated in-part by the desire to have reliability without the latency caused by head-of-line blocking. It is suited as a replacement of TCP for application-layer protocols that require reliability but not-strict ordering of data. Typically these are signaling protocols that are used within core networks. It is available in most popular operating systems.

An SCTP connection between two endpoints is known as an *association*. The association encompasses all the potential source and destination IP addresses that exist between the two end hosts. To

reduce head-of-line blocking, the data sequence to be transmitted can be broken down into smaller *regions*. Regions in an association can be further divided into transmitted across multiple sequence streams (opposed to the single sequence stream of TCP). An individual stream may be ordered or unordered. Streams can be transmitted between any of the endpoints available to the association.

Congestion control is based on TCP congestion control [107]. For a given association, all streams that exist between the same IP address endpoints share congestion control parameters. If the association has multiple endpoints, congestion control is maintained separately as paths might differ. Multihoming was initially used for redundancy, providing failover between the available IP addresses. A later extension, Concurrent Multipath Transfer (CMT-SCTP) [6] provides features such as data scheduling and the coupling of congestion control across different IP endpoints.

There are several reasons why SCTP has not seen wider deployment. Firstly, middleboxes must be modified to support SCTP. In particular, SCTP is not able to traverse NAPTs that perform port translation. These must be modified to support SCTP associations [108]. Many middleboxes have not been updated to support SCTP, preventing the use of the protocol on paths with such devices. Legacy NAT traversal can be achieved by encapsulating SCTP in UDP packets [36], though at the time of writing there is no survey indicating how widely this has been adopted. Thus due to the difficulty in using SCTP across the Internet, few end-user applications have been updated with SCTP support.

Multipath TCP: TCP Extensions for Multipath Operation with Multiple Addresses [2] is an extension to TCP that is currently being standardised by the IETF. It is commonly referred to as Multipath TCP (MPTCP), due to the implication that multiple IP addresses assigned to a host may take alternate paths to a given destination IP address. It provides mobility, redundancy and resource pooling between end-hosts.

It is backwards compatible with existing TCP-enabled applications and is designed to work with the Internet as it is today. This is achieved by extending TCP and preserving as much as possible the on-wire behaviour, so as to minimise issues with middleboxes. As with other TCP extensions, MPTCP is implemented within the TCP stack and becomes available to all TCP sockets. A new TCP option kind is assigned for MPTCP signaling and MPTCP is negotiated during TCP's three-way handshake. A high-level summary of the key aspects of MPTCP is provided below. A more thorough discussion can be found in Section 5.1.

Like SCTP-CMT, MPTCP is able to stripe data across any IP source/destination pair that is available to the participant hosts. It does this by transparently establishing multiple streams (called *sub-*

flows) between the transport layer endpoints of the connection, presenting only a single TCP socket to the application layer. A key difference from SCTP-CMT however is that from the perspective of the network, each subflow in a connection looks and behaves like a standard, independent, TCP session. Since subflows are created from within the transport layer, they can be dynamically added (or removed, in the case of failure) from the MPTCP connection without breaking the original TCP socket created by the application. Since subflows are completely decoupled from the application, an MPTCP connection may remain open even if no subflows are currently established.

Importantly, the TCP sequence space (sequence numbers and ACKs) of a subflow is exactly as it would be for a standard TCP connection. This is vital to maintain compatibility with middleboxes. To stripe data between the subflows, then, an additional sequence space is defined to allow the receiver to reassemble data that may have been divided between multiple subflows. This is the data sequence space and represents the true byte-order of the data as written by the application. As will all MPTCP signaling, data sequence numbers are transmitted in the TCP option space. In the event of middlebox interference (for instance removing an MPTCP option from the TCP header), a fall-back mechanism transitions the connection to standard TCP.

MPTCP provides resource pooling of the available subflows through scheduling and coupling congestion control. When an application writes new data to the send queue, the scheduler considers the state of all subflows before deciding which subflow to use for the next allotment of data. Coupled congestion control links the behaviour of the subflow congestion windows, steering transmission over the most effective paths. Head-of-line blocking can occur at both the subflow and data-levels, thus scheduling and congestion control decisions are vitally important. A poor algorithm can lead to severe performance issues, in some cases worse than using standard TCP alone.

MPTCP has been designed to minimise the barrier of entry for deployment while solving issues of mobility and multipath. By modifying TCP it is able to maintain compatibility with applications and function across the Internet.

3.4 Conclusion

Multihomed end-hosts are now commonplace. As a result, benefits such as fault tolerance, load balancing or data striping are available at the end-host. However these benefits have yet to be realised for end-to-end multipath connections, leaving scope for further research and development. This chapter considers the primary challenges of multipath communication and assesses existing solutions for suitability in future multipath research and deployment.

Link-layer solutions do not take higher layer protocol information into account when making scheduling decisions (aside from flow identification). This means scheduling policies must be conservative in order minimise impact on higher-layer protocols. Solutions implemented at the Internet layer would seem a logical choice, however existing methods are compromised in a number of ways. Site-wide solutions do not allow end-hosts to significantly effect path selection, and do not use higher-layer flow state when scheduling. Solutions involving the end-host have not been widely deployed and generally support only one of mobility or multihoming.

There are some significant benefits in solving mobility and multihoming problems in the transport layer. Access to the application send queue, end-to-end path statistics and control over the outbound network interface mean that resource pooling, scheduling, path selection and mobility can be handled at this layer. However there are still some challenges to creating effective and deployable solutions. Scheduling and congestion control decisions rely on accurate path statistics. Middlebox assumptions about transport layer protocols mean that any new solutions must be resilient to interference from such equipment.

Many previous solutions have not gained traction due to lack of support from middleboxes. SCTP has been the most successful of the new transport-layer protocols, though is used predominantly within core networks rather than by end-user applications. MPTCP is a more recent solution that achieves compatibility with existing middleboxes by extending TCP and maintaining as much as possible the on-wire behaviour.

Several functioning implementations of MPTCP are already in use across the Internet [109, 8]. MPTCP retains the connection-oriented and reliable operation of TCP and adds the ability to pool locally connected network resources. Backwards compatibility with TCP and a design that functions within the current Internet improve the chance that MPTCP will be more widely adopted then previously proposed solutions.

Chapter 4

Multipath Scheduling and Congestion Control

This chapter reviews the current state of sender-side mechanisms for multipath transport protocols. The discussion is not implementation specific, however much recent work in this area has focused on MPTCP. As such familiarity with the key concepts of how MPTCP distributes data, as presented in Chapter 5.1, may provide useful context. Though the emphasis is on MPTCP, we include some CMT-SCTP research. Both protocols face similar challenges of scheduling and congestion control across paths with divergent characteristics. By understanding the current scheduling and congestion control approaches - for instance what metrics are required - a suitably flexible software design can be created.

In Chapter 3 we discussed how traffic engineering could be implemented at the end-host. As a transport-layer multipath solution, MPTCP is suitably positioned to provide traffic engineering by combining scheduling and congestion control. These are closely related, and we now revisit the broader connections between scheduling and CC as they relate to MPTCP connections.

The role of the scheduler is to decide which is the best path to transmit on. The best path is that which provides the best application-level performance. This is achieved through several steps. The scheduler first determines which subflows are available to send data. The *best* path is the path that matches the scheduler-specific criteria (e.g. RTT, cwnd size). Once a subflow is selected, the scheduler must decide the granularity (length) of data to allocate to the subflow. For each of these decisions the scheduler must rely on input from the CC algorithm.

Multipath CC can be performed at the connection-level or on an individual per-subflow basis

using standard single-path CC mechanisms. In single-path transport the task of CC is to optimise performance while being sensitive to path congestion. For multipath transport, the aim of CC is not to optimise performance for each subflow, but rather the connection as a whole. It does this by steering traffic away from subflows that do not improve the overall performance of the connection. The main mechanism used for steering is the congestion window of each subflow. To balance the distribution of traffic across the available paths, the CC simply adjusts the congestion window of the paths so that preferred paths are able to transmit more data. Where using subflow-level CC would result in all subflows attempting to optimise there own congestion windows (potentially at the expense of the whole), connection-level CC will only increase the congestion window of a subflow if it benefits the entire connection.

We can now consider a basic characterisation of the scheduler-CC relationship. The scheduler first identifies which subflows are able to send data, perhaps those that have non-zero `cwnd`. The best path selected from these could simply be the path with the largest `cwnd`. Other statistics such as `RTT` or `ssthresh` might be taken into account. Finally, the scheduler allocates data to the subflow, again taking into account factors like `cwnd` or `RTT` when deciding the length of allocation. Both the scheduler and CC algorithm must be in agreement about which is the best path. An inconsistent approach could lead to poor distribution of data across the paths.

A more detailed discussion of scheduling and CC approaches now follows in Sections 4.1 and 4.2.

4.1 Multipath Schedulers

When presented with multiple paths the sender must decide over which path new data is transmitted. The approach to servicing each path is critical, as casual allocation of data to different paths risks decreasing transport-layer performance. What motivates this decision can vary, such as trying to maximise throughput or minimise delay. The context of a connection is important - sending data to a mobile host across the Internet requires a different approach to a bulk transfer over symmetric paths on a private network. Thus the scheduling algorithm should change depending on the use-case. A common theme however is the notion that all available paths should be regularly serviced by the scheduler, ensuring that the available capacity is utilised. After all, a major goal of multipath communication is to leverage additional capacity.

Scheduling approaches in the existing literature can be divided into *naive*, *proactive* or *reactive* methods. Naive algorithms distribute packets with no or little consideration of the impact on

connection-level performance. A naive scheduler may simply fill the capacity of all the available paths, even if this results in re-ordering of sequenced data.

Proactive algorithms actively consider transport protocol performance and employ additional intelligence to, for example, distribute packets so that they arrive in-order at the receiver. These algorithms attempt to trade-off path utilisation and transport-protocol performance. A proactive approach may sacrifice aggregate throughput for better application-level responsiveness.

Reactive approaches focus on scheduling strategies that improve recovery from congestion or packet re-ordering events. This includes when and how much data to retransmit, through to strategies that determine the best path for retransmission.

A scheduling algorithm may also be accompanied by an *optimisation algorithm* that modifies transport-session state in order to influence scheduling decisions. For example an optimisation algorithm may manipulate the congestion window or slow-start threshold of a path, controlling the rate of transmission.

A scheduler can be further categorised based on when it is run and and the way in which data is allocated to a path. Barre [110] and Singh [111] describe three approaches:

- *Push-allocation:* Each path maintains it's own send queue. When the application presents new data to send, the scheduler runs and selects a path, copying the data to the output queue of the selected path. The path may queue the data for a time or transmit immediately.

- *Pull-allocation:* New data is stored in a global send queue shared by all paths. Only when a path has a transmission opportunity is data allocated (say after receiving an `ACK`). Transmission occurs at the time of allocation.

- *Hybrid:* A global shared queue and per-path queues exist, allowing both push and pull allocation.

Each method has trade-offs. Push allocation allows the scheduler to consider path statistics, such as the length of the output queue, when selecting a path. Since each path has a queue, data can also be allocated even if it is not transmitted immediately. However as there is no global queue, data must be transmitted on the path to which it was allocated and cannot be re-assigned after-the-fact.

Pull allocation guarantees that transmission will occur immediately. It is run at the request of the paths, meaning higher capacity paths will be used more often, since these will have more frequent opportunities to transmit. A drawback however is that since scheduling is dictated by the transmission clock of the paths there is less scope for path selection based on other metrics.

Statistic	**Source**	**Used to**
Queue occupancy	Send Queue	Estimate path capacity
Queuing delay	Algorithm	Estimate time-to-transmit
Forward-delay	Algorithm	Estimate delivery time
Round-trip time	TCP layer	Estimate path capacity
Acknowledgement Interval	Algorithm	Estimate path capacity
Slow-start threshold	TCP layer	Estimate path capacity
Congestion Window (cwnd)	TCP layer	Estimate path capacity
Path RTT-difference	Algorithm	Measure of path heterogeneity
RTT-envelope	Algorithm	Detect path RTT inflation

Table 4.1: A selection of statistics used by current schedulers. Some must be calculated by the algorithm. Note that some statistics may serve multiple purposes.

Hybrid allocation inherits the benefits of both push and pull methods. It is the most flexible approach however this comes at the cost of complexity, both for the scheduler and for the network stack scheduling framework. The algorithms discussed in this chapter are a mix of push, pull and hybrid strategies, as one approach is not necessarily better than the next and all should be supported for future research. The scheduling framework in the Linux reference implementation supports each of these approaches to allocation [110].

For all but the most basic algorithms, it is likely that the scheduler will draw upon some statistics to inform path selection. Frequently these statistics are used to estimate path characteristics such as capacity, delay or loss rate. More advanced algorithms may attempt to detect network bottlenecks or anomalies such as bufferbloat. A naive scheduler may consider only a single statistic such as `cwnd`, while a proactive scheduler might implement a novel delay-measurement mechanism. From the literature we can identify two principle sources from which scheduling variables are derived:

- *Queue-state:* These are based on queue capacity and queue occupancy at the sender.
- *Path-state:* These reflect the state of the end-to-end path. This includes round-trip times, capacity, or even if an interface is active or inactive. Transport-layer state variables are usually a source of path measurements, though a scheduler may implement it's own measurement mechanisms.

Algorithms can read state variables as they are or include them in calculations to derive further

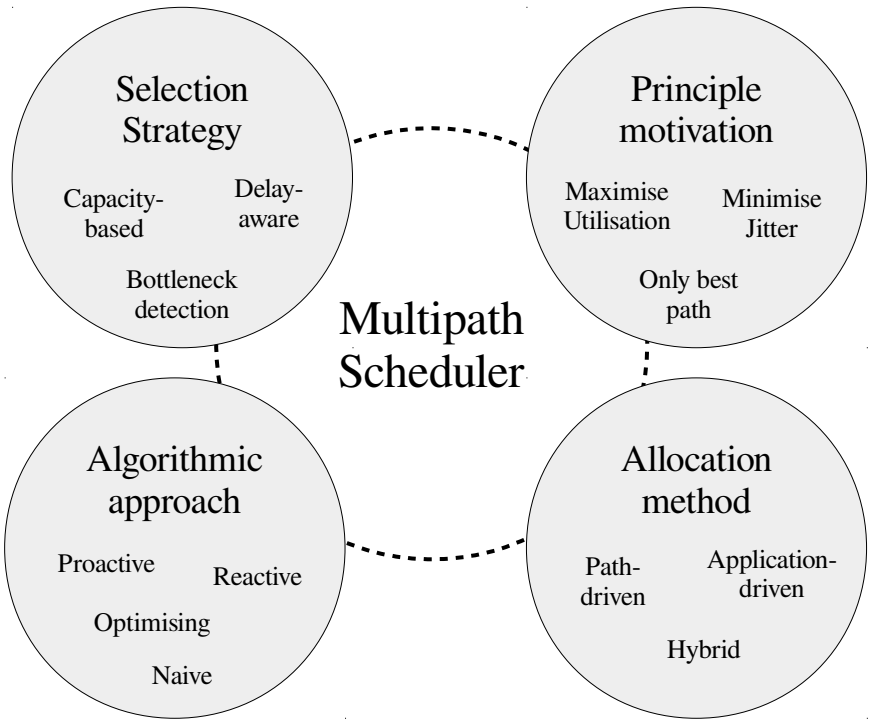


Figure 4.1: A basic decomposition of a multipath scheduler

meaning. Variables are often combined in a single calculation. Table 4.1 lists a selection of statistics obtained from queue-state and path-state variables. Different variables may be combined into a calculation. A scheduler may also aggregate statistics from across multiple paths. In some cases a scheduling algorithm may add tunable variables or new instrumentation if the the desired statistics are not available from the send queue or transport protocol.

Figure 4.1 provides a basic decomposition of a multipath scheduling algorithm. We see here that schedulers can differ greatly in their central motivations, approach to data allocation and in algorithmic complexity. The following subsections discuss a number of existing proposals and concrete implementations that are representative of the different approaches possible.

4.1.1 Naive approaches

A naive scheduling algorithm selects a path considering little or no flow state information. Such an algorithm might attempt to use only the highest capacity paths, or conversely try to fill the available capacity of all the paths. No explicit attempt is made to predict re-ordering at the receiver or perform additional measurements or calculations to infer path characteristics. Naive algorithms such as *ran-*

dom selection (RS) and round-robin (RR) are are found in load balancers [112] but can be applied to multipath scheduling. RS allocation chooses an output path randomly and is completely stateless. A *round-robin* (RR) scheduler chooses output paths sequentially. RS is not used for CMT-SCTP or MPTCP however RR schedulers have been implemented.

ACK-Clocked: A multipath scheduler might simply rely on the transmission clock of the paths to drive allocation. For protocols like MPTCP, this means being driven by the TCP *ACK-clock* of each subflow. Such an algorithm would need to initiate slow-start on each path, which would in turn start the ACK-clock. We did not find an evaluation of such a scheme for MPTCP or CMT-SCTP in the literature, however the existing RR schemes at times operate driven by ACK-clock.

Capacity-based RR attempts to fill the available `cwnd` of the selected path before moving to the next. Capacity-based RR schedulers are available for CMT-SCTP and MPTCP. Path quality (delay, loss rate) is not a consideration and each path is used to capacity. If the `cwnd` for each path is filled, the scheduler becomes ACK-clocked.

Delay-aware RR orders the available paths according to delay. The default Linux MPTCP reference implementation uses a delay-aware RR scheduler, *LowRTT* [113]. LowRTT sends data on the path with the lowest smoothed RTT (`sRTT`) that has `cwnd` available. LowRTT scheduling is motivated by the fact that many multihomed devices are connected to heterogeneous networks - say a smartphone connected to Wi-Fi and cellular - a configuration that results in paths whose RTTs differ significantly [114]. Packet re-ordering can be reduced by allocating more data to the lower RTT path. This is on the surface a proactive approach, however as with capacity-based RR, once the `cwnd` of each path is filled scheduling is driven by the TCP ACK-clock of each path. As a result the scheduler does not try to anticipate re-ordering and simply assigns data to paths as they request it.

Several studies [115, 6, 100] have shown shortcomings of naive RR approaches over heterogeneous paths. Since end-to-end paths across the Internet have such variability it's likely that re-ordering will occur when using RR techniques. A number of optimisation algorithms that limit allocation to poor paths have thus been proposed to augment RR schedulers, which are discussed later in this chapter.

4.1.2 Scheduling for the transport-layer

For strictly sequenced protocols like MPTCP, a scheduler ideally distributes data so that segments reach the receiver in sequence-order. Experiments in simple, controlled topologies [100, 116, 117, 94] have shown naive schedulers to suffer head-of-line blocking issues when paths are asymmetric.

Proactive schedulers offer a means by which to achieve in-order delivery. By calling upon statistics such as forward or end-to-end delay, output queue size or path capacity, a scheduler can attempt to estimate when packets will arrive at the receiver, or exclude paths that do not benefit the connection.

Proactive algorithms differ in the type of statistics used and computational complexity. For instance some algorithms may use only send-queue and transport protocol variables, while others may require more advanced calculations that require new measurement instrumentation. In the following discussion we define and focus on two broad categories of proactive schedulers - those that attempt to maximise concurrency while minimising re-ordering (*all-path*s), and those that try to identify and use an optimal path (*preferred path*).

Using all-paths: These schedulers attempt to use the available paths as much as possible and overcome re-ordering issues by anticipating the arrival order at the receiver. Though essentially trying to achieve the same goal, several distinct approach exist.

In [118], the *Delay-aware Packet Scheduling (DAPS)* algorithm is presented as an alternative to the default capacity-based RR scheduler of CMT-SCTP. DAPS considers the RTT of each path to distribute data so that packets arrive in-order at the receiver. When run, a *schedule* is generated that allocates SCTP 'chunks' to each path based on delay. The lower-sequenced chunks are first allocated to the path with the lowest RTT. If more chunks are available than can be sent over the fastest path, slower paths are used. DAPS then iterates the completed schedule to transmit packets. Estimating path delay requires an accurate timestamp mechanism for best results, and the authors propose a new timestamp scheme for CMT-SCTP, and this would be required for use in MPTCP.

Kim et al. [119] introduce a DAPS-like algorithm for MPTCP (hereafter referred to as Kim-ALG) that approximates the receive time for each segment on each available path by adding send-queue delay and an estimate of forward-delay (TCPs `sRTT/2`). Dividing the sRTT in half is a strong assumption, as measurements have shown asymmetric paths to be common [37]. It is however easily obtained and computed, and used by several algorithms in the literature. A table is created for each path with the estimated delivery time for each segment. The table is then used to compare receive-time estimates and select a path. The receive-time estimate accumulates as multiple segments are added to a particular path. For example if Path A has a forward delay of 10ms and Path B a forward delay of 50ms, then 50ms worth of segments would be queued onto Path A before a segment is queued onto Path B. In this way segments should arrive at the receiver at the same time.

The *Forward Delay-based Packet Scheduling (FDPS)* [120] scheme measures the *delay differences* between paths to determine which path has the shortest forward delay. The delay difference is

the difference between send and receive times for two packets transmitted on two separate paths. The measurement is achieved using the existing TCP timestamp option and without clock synchronisation. The paths are ordered according to delay-difference, and data-segments in the application send buffer are then mapped to each path, starting from the lowest sequence. As FDPS is a pull allocator, segments are fetched using this mapping whenever a path has a transmit opportunity.

To our knowledge the Kim-ALG and FDPS mechanisms are only known to have been tested by simulation. Yang et al. [121] propose an out-of-order transmission scheduler (OOT) and test an implementation in the Linux MPTCP kernel. Each time a packet is enqueued the scheduler is run to determine which path it should be allocated to. It does this by looking at the existing output queue of each subflow and estimating the time to deliver. `sRTT` and queue occupancy are used to estimate the forward delivery time for each path. A key difference from the DAPS algorithm is that segments can be allocated to the send queue of a path even if `cwnd` is currently full. The rationale behind this is that a fast path might deliver a full `cwnd` and backlog of packets before a slow path is able to deliver it right now. As it has been implemented for the Linux MPTCP kernel, a direct initial comparison with the default LowRTT scheduler is made. The authors find that receive buffer requirements are reduced and higher throughput is achieved (particularly for shorter duration flows, as re-ordering is reduced).

The *On-Demand Scheduler* (ODS) [122] presents an alternative scheduling approach of maintaining a single output queue and allocating packets when a path presents a transmission opportunity (i.e. pull scheduling). When transmission of a new packet is possible, the algorithm orders the available paths according to the *estimated time of acknowledgement* (ETA), an indicator of when future transmission opportunities are likely to occur. This includes paths that do not currently have `cwnd` available. Then for each data segment in the buffer the path with the lowest ETA that has space available in the congestion window is selected and the lowest sequence number for that path that would avoid re-ordering at the receiver is transmitted.

Preferred path: A potential issue with out-of-order scheduling methods is that the application must have sufficient data to service the paths. If the available paths have high RTT asymmetry, then a greater number of segments need to be buffered in order to utilise multiple paths. This could potentially extend to several hundred milliseconds worth of segments. High asymmetry can also lead to performance loss if all paths are used.

Approaches have thus been devised that avoid using slow paths altogether if it is anticipated that using the path will impact performance. When applied to MPTCP, these approaches allow connec-

tions to maintain performance at least on-par with single-path TCP when paths are highly heterogeneous. The obvious trade-off in preventing the use of higher-RTT subflows is the potential loss of throughput. The following algorithms attempt to identify the best available path and then schedule data predominantly over this.

The *Bandwidth Aware Scheduler* (BAS) [123] attempts to queue outgoing segments on the path that offers the shortest delivery time as predicted by path bandwidth. A *reception index* is calculated for each path that reflects how quickly the currently queued segments (both transmitted and awaiting transmission) can be delivered and acknowledged. A dedicated queue is assigned for each path, and since scheduling is based on the reception index packets can be queued on paths that do not have congestion window available. This means that segments are always assigned to the best available path, and contrasts the RR approach of using a sub-optimal path as an alternative if the best path has exhausted it's congestion window.

The *Constraint-based Proactive Scheduling* (CP) algorithm in [124] tries to estimate the remaining buffer capacity at the receiver and the required cost (in terms of buffer size) to send segments over a particular path. The CP algorithm has been designed for use with multihomed mobile hosts that have limited memory capacity (i.e. constrained receive buffers) and are connected via heterogeneous paths. The algorithm fills the `cwnd` of the fastest path (lowest RTT) unless a sufficient amount of buffer is available at the receiver to allow use of a slower path. In the event that buffer space is available, an optional *delay constraint* variable enforces a minimum RTT that qualifies a path for use.

One practical consideration is that since slower subflows may be idle for extended periods, active probing is required to maintain up-to-date path statistics[1]. The algorithm includes a mechanism that schedules packets over the idle path to act as probes. An implementation of the algorithm for the Linux kernel is tested using subflows with uncoupled congestion control. The CP algorithm is shown to perform as well as single-path TCP when the receive buffer is constrained and when paths exhibit extreme differences in characteristics. It is one of the more complex schedulers discussed here - featuring system tunables, multiple modes of operation and multiple stages of calculation (ordering paths, estimating re-ordering and receive buffer requirements, path probing). It has, however been implemented for the Linux MPTCP kernel.

Hwang [117]] proposes the *Freeze packet scheduling* scheme (FPS) that aims to preclude the use of subflows whose paths have significantly higher RTT than the lowest RTT path. The algorithm is specifically designed to prevent worse-than-TCP performance for short (stated as less than

[1] Statistics maintained by TCP are invalidated during idle periods

1MB) flows. Several studies have shown MPTCP to perform poorly for short flows [125, 126]. The FPS algorithm simply *freezes* transmission on slow paths if there is a pronounced difference in RTT. The motivation behind the FPS algorithm is that extreme differences in path RTT (e.g. greater than 100ms) have a disproportionately high impact on the completion time of short flows. For example the LowRTT scheduler may decide to schedule data on a subflow with a long RTT if the the congestion window of the lowest RTT path is full. For short flows, RTT asymmetry may be such that waiting for the `cwnd` to open up on the low RTT flow would be significantly faster.

A tunable *thresh_delay* parameter is used to set the delay threshold that determines whether a subflow will be frozen. A packet-count threshold can be used to re-instate use of the slower paths for longer flows (for example reverting to LowRTT scheduling once the count is exceeded). In a preliminary evaluation, the FPS algorithm outperforms a LowRTT algorithm for short flows where subflows have highly asymmetric RTTs.

4.1.3 Improving loss recovery

Retransmission policy has a significant impact on how quickly a connection can recover from the effects of packet re-ordering [115, 100]. A retransmission path should be selected based on how quickly it can clear any head-of-line blocking. Knowing when to retransmit data is also important, as packet re-ordering can falsely trigger retransmission signals.

Several MPTCP retransmission strategies have been adopted from previous CMT-SCTP research. A RR scheduler is used in [6] to study CMT-SCTP performance on diverse-bandwidth paths. The scheduler introduces re-ordering that interacts poorly with the SACK-based acknowledgement method used by SCTP, falsely triggering retransmissions that hamper `cwnd` growth. Further research in [115] uses a RR scheduler to investigate receive-buffer blocking in CMT-SCTP connections with different loss-rates. The authors posit that a more intelligent approach to retransmission should be combined with the RR scheduler to improve performance. Five retransmission policies are defined:

- *RTX-SAME:* Attempt to resend to the the path that experienced loss.
- *RTX-ASAP:* Select the first path that has enough congestion window available for the retransmission.
- *RTX-CWND:* Select the path with the largest congestion window.
- *RTX-SSTHRESH:* Select the path with the highest slow-start threshold

- *RTX-LOSSRATE:* Select the path with the lowest loss rate.

The capacity-estimating policies (RTX-CWND and RTX-SSTHRESH) are shown to be the most effective at reducing delays caused by retransmissions. Retransmission on the original path on which the loss occurred is not recommended for a several reasons, for example due to the increased delay caused in cases of multiple RTOs. Thus the most effective retransmission strategies require the ability to re-schedule data over an alternate path. These policies are a precursor to the retransmission strategies used by the Linux MPTCP reference implementation.

Raiciu et al. [113] compare the goodput of MPTCP connections using LowRTT RR scheduling over Wi-Fi and 3G against a standard TCP connection over Wi-Fi only. They find that goodput for MPTCP connections drops below that of standard TCP when the receive window size is limited. Consequently they introduce the *opportunistic retransmission (OR)* optimisation, which has some similarity to RTX-CWND. The OR algorithm runs when the connection is receive-buffer limited. At this time the first unacknowledged data segment is retransmitted on the first alternate path that has space in it's congestion window. The intention is to clear the head-of-line blocking and create space in the receive window more quickly than waiting for the original path to deliver the segment.

Failover-mode scheduling uses a nominated path until it is no longer functional, at which time data is sent over a backup path. A *failure* state defines when a path is no longer to be used. Failover is specified as part of the MPTCP protocol and available in the Linux and Apple implementations [109]. Failover does not at first glance resemble a scheduler, however the failure state may be defined by threshold conditions such as loss rate. In this way failover-mode resembles both a loss-recovery and preferred-path scheduling. The conditions that constitute a failed path can vary, and guidelines have been proposed to detect these [127].

4.1.4 Assisting the scheduler

Optimisation mechanisms do not explicitly make scheduling decisions but indirectly influence the schedulers selection process, for instance by manipulating state in the transport protocol. A basic example would be reducing the `cwnd` of a poorly performing path, an action that effects path selection.

Raiciu et al. [113] define the *subflow penalisation* (SP) algorithm that attempts to reduce the RTT of poorly performing subflows. The SP algorithm is called after receive-window limiting. Once triggered, it reduces the congestion window and slow-start threshold of the offending path in order to lower the RTT. By lowering the RTT the delivery time of the subflow is reduced, and thus re-ordering is also reduced.

Ferlin-Oliveira et al. [114] identify several instances where the SP algorithm is not effective when paths suffer from bufferbloat. They propose a *Multi-Path Transport Bufferbloat Mitigation (MPT-BM)* mechanism for MPTCP. Like the SP algorithm, BM does not directly make a scheduling decision but instead tries to reduce the occurrence of bufferbloat (and hence re-ordering and buffer-blocking) by capping the congestion window of subflows that show signs of RTT inflation. It does this by comparing minimum `sRTT` against the current `sRTT`. This is an example of a more complex optimisation approach that must perform continuous calculations (in this case once per `sRTT`).

4.2 Multipath Congestion Control

Congestion control (CC) determines how much data can be placed onto a path each round-trip period, how much data can be queued for later transmission and the interval at which data is transmitted. As individual paths are flow-controlled, the congestion window and the functions to manipulate this are vital in determining where data is sent. The congestion window ultimately determines how much data can be allocated to a particular path by the scheduler.

Traditionally CC has been applied to a single path between two endpoints. With multipath transport protocols that bundle multiple endpoints into a single connection, is per-path CC the best approach? Two alternatives are available: *uncoupled*, where each path has it's own CC function, or *coupled* [10], in which a single CC function controls all paths[2].

When using uncoupled CC each subflow behaves as an independent TCP flow. This is analogous to bundling TCP connections at the application layer. Though maximising use of each path there is the undesirable effect of being unfair to cross-traffic. Coupled CC links the `cwnd` evolution of each flow using a single function, abstracting all paths to a single shared resource of higher capacity. A coupled CC algorithm adjusts the `cwnd` on an individual path only after considering the current state of all the paths. By combining the transport-layer response the end-host gains the ability to steer traffic to the best paths available. The ability to apply end-host resource pooling [91] through coupled CC is a core feature of MPTCP.

One further consideration when coupling subflow behaviour is whether `cwnd` is shared between the paths or maintained independently for each. The approach taken by MPTCP and CMT-SCTP is that of a congestion window per-path. This makes sense as there is an expectation that data will be transferred over heterogeneous paths where there is little relationship between the `cwnd`. However

[2]Since the following discussion is primarily MPTCP specific, in this section we use the terms *subflow* and *path* interchangeably.

sharing congestion windows is a valid approach to providing fairness when flows are known to share bottlenecks. Protocols adhering to the *congestion manager* [128] framework or fairness protocols like *Ensemble TCP* [129] use shared congestion windows. And so the approach could conceivably be adapted to MPTCP connections.

Whether coupled or uncoupled, three basic design goals are provided in the specification [2] and must be met by MPTCP CC algorithms intended for Internet deployment:

1. Provide aggregate throughput at least as good as single-path TCP on the best available path.
2. Not use more capacity on a shared bottleneck than if using single-path TCP over the same path.
3. Move traffic away from congested paths, subject to goals (1) and (2).

These goals establish base expectations for what a MPTCP CC algorithm must achieve. Expressed in simple terms, an algorithm must be *as-good-as TCP*, *friendly* and *responsive.* What then are some of the practical considerations in trying to meet these goals?

Goals (1) and (2) require that an algorithm be able to detect when paths are disjoint or when there is a shared bottleneck. When a bottleneck is detected, the algorithm must then be able to divide capacity to that of a standard TCP flow. For this, an accurate estimation of what a single-path TCP would get over the same path is required. Similarly when paths are disjoint the algorithm should provide better-than-TCP performance. Uncoupled CC will very likely achieve goal (1), but cannot achieve goal (2).

Goal (3) depends on how tightly coupled the paths are. An ideal approach would remove all traffic from a path that is experiencing congestion, but also respond quickly when conditions change (e.g. an in-use path becomes congested). This is difficult to achieve in reality. Responsiveness requires up-to-date path information. The only way to know the condition of a path is to probe it by sending packets. And more accurate statistics require more frequent probing, which in turn adds to congestion. Uncoupled subflows provide the fastest response to fluctuating path conditions, but contribute to path congestion. More tightly coupled subflows contribute less congestion but take longer to converge to an optimal state when path conditions change. Thus algorithms exist on a scale between responsiveness and friendliness and must strike a balance between probing paths enough to be responsive but not enough to cause congestion. Another consideration is that a path too sensitive to change may oscillate around an equilibrium, a condition called *'flappiness'* [10].

The ability CC to steer traffic depends heavily on the scheduler being used. Recall that the CC algorithm does not directly choose the output path - a subflow must transmit whatever the scheduler

has assigned to it's output queue. As such the onus of assigning segments to the right path is on the scheduler. How then does a CC algorithm steer data effectively? In cases where the scheduler and CC are not directly integrated, the scheduler must rely on information taken from the transport layer to make decisions. The most obvious approach is to steer by manipulating the congestion window to encourage or discourage a scheduler from using a given path. Although this is not always effective at preventing the scheduler from selecting a congested path [130], it is the approach taken by the coupled CC algorithms discussed in this chapter.

4.2.1 Uncoupled Congestion Control

Existing TCP CC algorithms such as TCP-NewReno [131] or TCP-Cubic [132] can be used independently on each subflow of an MPTCP connection. While effective at increasing throughput and highly responsive to changes in network path conditions, this approach is unfair to TCP cross-traffic on shared bottlenecks. Although Internet-deployed MPTCP hosts are encouraged to use coupled CC *[10, 2]*, there are some cases where uncoupled algorithms might be suitable. There are also several proposals that attempt to adapt existing single-path CC algorithms to multipath operation over the Internet in a way that is fair to cross-traffic.

The Bidimensional-Probe Multipath Congestion Control (BMC) [133] scheme applies a weighting factor to uncoupled subflows to achieve fairness when subflows share a bottleneck with single-path TCP flows. A weighting factor is applied to `cwnd` growth that is proportional to the number of subflows in the connection. The weight of a standard TCP flow is considered *1*. Thus each subflow sharing a bottleneck is weighted so that the combined throughput is equal to 1. A separate algorithm is used to detect if any subflow is on a disjoint path. The weighting for subflows on disjoint paths is increased to better utilise the additional capacity. Although fair to other TCPs, BMC is less efficient at allocating traffic than coupled approaches [134].

Adhari et al. [135] investigates using uncoupled Low Extra Delay Background Traffic (LEDBAT) [136] CC for multipath transport. LEDBAT is a delay-based approach that tries limit the amount of congestion that a flow contributes to the network path by monitoring forward-delay variation. A base-level delay is established, and any increases in this delay due to queuing along the path are interpreted as signs of delay. It is designed to yield bandwidth in the presence of cross traffic (e.g. standard TCP flows). The primary use-case is to allow 'background' applications (e.g. software updates) to perform bulk transfers without interfering with higher-priority traffic flows.

The authors observe that by design LEDBAT will achieve goals (2) and (3), as it's explicit purpose

is to avoid adding congestion. Regarding goal (1), they note that since LEDBAT is designed to use the minimum amount of bandwidth, when compared with a LEDBAT-TCP flow any gains from using LEDBAT-MP satisfy this goal. Their assessment is that LEDBAT-MP is suitable for well-defined network environments, such as datacentres, but can be erratic in uncontrolled networks. Notably very large receive buffers (5MB) are required to handle re-ordering of data-segments effectively. Re-ordering latency is not an issue for background-class transfers, but the required memory footprint might be problematic for memory-constrained devices. The authors are planning to further evaluate and improve LEDBAT-MP for end-user scenarios.

4.2.2 Coupled Congestion Control

MPTCP connections must be friendly toward cross-traffic of different protocols and not contribute to congestion. One way to achieve this is by coupling the congestion control functions of each path. Coupled CC algorithms coordinate `cwnd` evolution across subflows to steer data away from congested bottlenecks. If a bottleneck is unavoidable, subflow congestion windows can be weighted to ensure fairness to other flows. Steering should not only be viewed in terms of providing fairness, and algorithms may disregard fairness altogether in pursuit of other purposes.

Several coupled CC and multipath routing algorithms pre-date those proposed for CMT-SCTP and MPTCP. Particularly relevant is the algorithm and framework of [137], which established that dynamic rate-control and routing functions could be integrated within a congestion control mechanism at the end-host.

Subflows are coupled by linking the functions that drive congestion window changes. What then are the specific functions that can be coupled? Recall from Chapter 2.1.2 that TCP flow-control is divided into distinct phases: *slow-start, congestion avoidance (CA)* and *congestion recovery*. The congestion window is changed in response to *positive acknowledgement* or *congestion notification* signals. The algorithm that responds to these signals varies depending on the current phase. For instance the response to an `ACK` during slow-start is different to that of CA[3], and so on. The coupled algorithm must therefore choose when to link behaviour and when to use independent functions, based on the current phase and the type of signals.

Not all functions need to be coupled. For example the default congestion control of MPTCP [10] only couples the additive increase phase of CA, acting like uncoupled TCP-Reno at all other times. As with scheduling, additional instrumentation may be used by the algorithm to improve accuracy or

[3]For example the classic approach is exponential cwnd grow during slow-start and linear growth during CA.

new variables. For example detecting bottlenecks has been a particular challenge for existing coupled CC [134], so the inclusion of additional algorithms that drive `cwnd` changes should be supported. As with TCP it is expected that different algorithms will be proposed to meet different performance goals. Here we divide a selection of existing algorithms into those designed for general-purpose Internet deployment and those with more specialised goals.

Several general-purpose coupled CC algorithms are included with the base Linux MPTCP distribution. These are the *linked-increases* (LIA) [10, 113], *opportunistic linked-increases* (OLIA) [130] and *balanced linked adaptation* (Balia) [138] algorithms.

LIA was the first coupled CC algorithm standardised for MPTCP. The algorithm couples the congestion window increase function during the congestion avoidance phase. Window decrease, slow start and retransmission behave as per standard TCP-Reno [22]. It should be possible to use the algorithm with different congestion control algorithms such as TCP-Cubic, however at the time of writing no literature has evaluated this.

The algorithm operates as follows. For each `ACK` received on a subflow the `cwnd` is increased as a proportion of the total `cwnd` of all subflows. The increase is based on the estimated bandwidth of the path, with lower loss paths seeing greater increases. Bandwidth is estimated based on loss-rate and RTT. A limit is placed that prevents any individual increase from being greater than what single-path TCP would see on that path. Also considered is the bandwidth that would be achieved by a TCP flow over the best path, and the aggregate rate of the subflows is adjusted to at least this rate.

LIA attempts to continuously probe even high-loss paths, meaning it trades efficient congestion balancing for better responsiveness to path changes. This trade-off means that the stated design goals are not met in some cases [10, 130, 139]. Analysis by [130, 138] shows several instances where LIA provides worse-than-TCP performance or is unfair to other TCP cross-traffic. Critically in some cases using MPTCP reduces the performance of other flows without a corresponding gain. To support probing some level of traffic must always be sent across all the paths. This reduces performance of the best path (and hence the MPTCP connection) while also adding more traffic to congested paths, which impacts cross-traffic on those links.

OLIA attempts to better address goal (3). As with LIA, only the increase-function is coupled between the subflows. OLIA tracks the amount of data successfully transmitted between losses (termed inter-loss distance) and uses this together with RTT to rate which paths are ‘best’. Of the resulting subset of ‘best’ subflows, those with smaller windows grow faster than those with larger windows. Therefore traffic from fully used paths can be allocated to paths with free capacity. Analysis in

[130, 138] shows that OLIA is indeed more friendly and better performing than LIA, however [140] finds that OLIA can be unresponsive to change when path RTTs are similar.

Balia aims to improve on both LIA and OLIA by achieving an optimal balance between friendliness and responsiveness. Evaluation in [140] shows Balia to be friendlier than LIA and more responsive than OLIA. As with the previous algorithms, Balia applies to the AIMD portion of the congestion avoidance phase, however both increases *and* decreases are linked. Balia uses only `RTT` and `cwnd` values of each path in it's calculation.

Each of the approaches discussed so far uses packet loss as an indicator of congestion. Thus changes to traffic distribution balance only occur in response to packet loss on a path. This limits when traffic can be shifted away from a congested path until after loss has occurred. As with loss-based TCP CC the results in a periodic rise and fall of RTT as path queues are filled and emptied as congestion window grows and then collapses. *Weighted Vegas (wVegas)* [141] aims to provide a more fine-grained and fair balancing of traffic using a delay-based approach.

The wVegas algorithm adjusts subflow `cwnd` according to the estimated backlog of packets in queues along the subflow path. If a (user-defined) threshold of queuing is reached, then the `cwnd` is reduced, allowing time for queues to clear. A weighting factor based on an estimate of path capacity is included that increases the window size of higher-capacity paths more quickly, thus acting as a steering mechanism.

Not all proposed algorithms are designed for general-deployment. As with TCP, different deployment scenarios benefit from a different CC approach. *MP-Veno* [142] is one such algorithm that is designed for use in wireless networks. MP-Veno is derived from *TCP-Veno* [143], itself a combination of TCP-Vegas and TCP-Reno. TCP-Veno uses delay-variance to determine whether losses are due to congestion or random packet loss (common in wireless networks). The multiplicative decrease function changes depending on whether the algorithm believes the current state is congested or not congested. The increase function is changed to keep the congestion window within a usable size for a longer period of time.

TCP-Veno uses a bandwidth threshold value B that represents congestion-level queuing in the path. On `ACK` it is compared to an estimate of the current number of packets *backlogged* (queued) along the path. A backlog greater than B indicates that the path is in a congested state and any loss is deemed caused by congestion, and `cwnd` is reduced as with TCP-Reno. Losses that occur with a backlog less than B are treated as random packet loss, ans `cwnd` is reduced by a fraction of the standard TCP-Reno reduction. MPVeno uses a similar `cwnd` growth and reduction mechanisms

as TCP-Veno, however a weighting factor is applied to the backlog threshold B that ensures that if sharing a bottleneck, the sum throughput of the MPVeno subflows does not exceed that of a single TCP-Veno flow. Preliminary similations comparing MPVeno to LIA indicate that MPVeno may be more resilient to random packet losses.

The *Equally-Weighted MPTCP* (EW-MPTCP) [144] proposal attempts to reduce TCP incast [145] issues caused by MPTCP flows in datacentre networks (DCNs). Using MPTCP in a DCN helps to spread connections and reduce path collisions [146] but does not completely eliminate incast. EW-MPTCP is unique in that it adds weightings across multiple MPTCP connections (using LIA CC) so that the aggregate throughput of connections servicing a request is equal to that of a single TCP flow (similar in some ways to E-TCP). The rationale behind this is that within DCN topologies a single request is serviced by multiple concurrent connections, and should therefore be considered part of the same connection. The weighting uses the number of related connections n as a scaling factor and is applied to `cwnd`growth during slow-start and CA phases. A mechanism for learning the value of n in an actual implementation is not provided, and a practical implementation would need to solve this issue. Simulations show that the approach is at least feasible.

4.3 Conclusion

Standard TCP applications are not aware of any underlying multi-path capabilities and cannot influence where data is steered. MPTCP performs these functions through both scheduling and congestion control algorithms. As MPTCP is designed to operate over the Internet, it is expected that these mechanisms should provide good performance across a range of diverse paths whose characteristics can change over time.

CC was developed along with the initial MPTCP specification, and several congestion control choices are already available in the UCL Linux MPTCP distribution. However further research is still required into areas such as bufferbloat detection and bottleneck detection. Bufferbloat is relatively common, yet difficult to detect, and can confuse existing algorithms that rely on packet-loss as a signal of congestion. Current CC algorithms assume that subflows share bottleneck links, even if paths are fully disjoint, which prevents full utilisation of the available capacities [147].

There is, at the time of writing, no standardised or stand-out approach to scheduling. Proactive methods are a popular area of research, however are fairly complex and several proposals have yet to be tested in a real implementation. Naive methods combined with optimisation algorithms have seen more practical implementations, however known issues (such as bufferbloat detection) still exist and

further research is required for solutions.

The existing literature largely focuses on individual schedulers or CC algorithms. The scheduler-CC relationship is vital, though at this point there does not seem to be a survey examining the interaction between different schemes (possibly due in-part to the lack of in-kernel implementations).

Finally, a number of prominent experiments in the existing literature are simulated and do not feature complex 'real-world' topologies or significant cross-traffic. It is critical to be able to test news schemes in real implementations. For example, what is the processing and memory cost of an approach? Is it scalable? Is the performance replicated in a real stack and network? Only few studies [148, 8] have surveyed MPTCP flows in the Internet. In the following chapters we look at the MPTCP protocol itself before describing our MPTCP architecture for the FreeBSD network stack.

Chapter 5

Overview: TCP extensions for Multi-addressed Operation

The core goal of MPTCP is to transparently bring resource pooling to existing TCP-based applications. The resources in this case are IP addresses on the end host, each of which represents a potentially unique network path. By combining the available addresses, a single transport session can gain additional throughput or mobility by sending data over interfaces that would otherwise have been left idle. Only one host in the connection needs to be multihomed to enable MPTCP.

What follows in this chapter is a high-level overview of how MPTCP fits into the network stack, the design decisions that allow it to remain invisible to both the application and network, and an operational overview of an MPTCP connection. This discussion should be considered as introductory only and RFC 6824 [2] should be referred to for greater detail and further reading.

5.1 Key Concepts

From the perspective of the application and network layers, MPTCP looks like TCP. An application creates a TCP stream socket, while the network sees multiple, seemingly unrelated TCP flows. Since there is a presumption that most TCP applications will not be MPTCP aware[1], a MPTCP socket must respond like a TCP socket. Thus the MPTCP socket abstraction is identical to TCP - namely a single, connection-oriented, sequenced, reliable, bi-directional network pipe.

To realise the desired transparency and backwards compatibility goals the protocol designers needed to consider (among other things): where in the network stack MPTCP should reside, what

[1] At least initially. An API could be made available for MPTCP aware applications in the future.

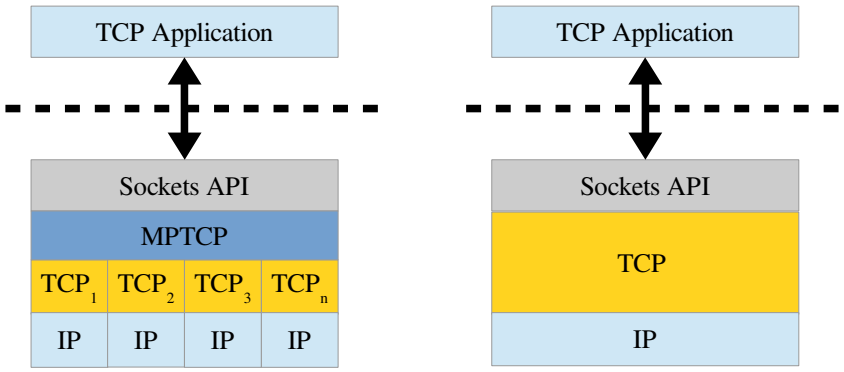


Figure 5.1: MPTCP sits logically between the socket and standard TCP stack.

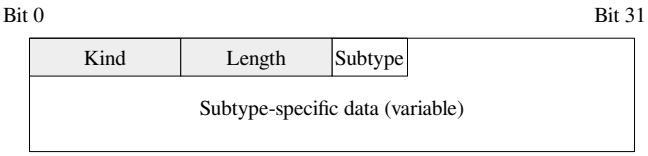


Figure 5.2: Generic MPTCP option format

would be a suitable signaling mechanism and how to aggregate data from multiple TCP-based subflows.

A transport layer shim: MPTCP needs to be available for all TCP applications on a host system, and is therefore implemented within the host's network stack. All operations are embedded within the transport layer. It is here where the multipath connection is terminated, new paths are discovered and data scheduling decisions are made.

Within the transport layer, MPTCP is positioned above TCP. This arrangement allows MPTCP to control the flow of data between the socket and multiple TCP connections that make up the MPTCP connection. Figure 5.1 compares a standard network stack with an MPTCP-enhanced stack. We see that each subflow belonging to a MPTCP connection is effectively a standalone TCP connection bound to a local IP address. MPTCP, in extending across each of the subflows, can decide where data is sent and is also able to aggregate received data for presentation to the TCP application. Though the application 'sees' a TCP socket, in practice TCP is completely decoupled from the application.

Signaling: Chapter 2 discusses the difficulties in deploying new protocols on today's Internet. A key concern of the MTPCP design was how to introduce signaling in a way that was compatible with the current Internet. Research [29] had shown that it was still possible to extend TCP with

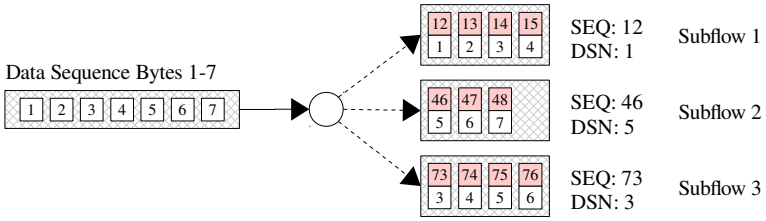


Figure 5.3: The 64-bit data sequence is carried over multiple 32-bit TCP sequences.

new options. Thus all signaling for MPTCP connections occurs via TCP options, carried in the TCP header of constituent subflows.

IANA has assigned a TCP option kind for MPTCP [149], and the MPTCP option is further divided into *subtypes*, each defining a signaling operation. The generic format of a MPTCP option is shown in Figure 5.2. Here the shaded *Kind* and *Length* fields inform TCP that this is an MPTCP option of a given length. The *Subtype* field denotes the particular MPTCP signal that this option represents. The subtype and related data is interpreted at the MPTCP-level and has no meaning at the subflow-level. Several option subtypes are discussed in Section 5.2.

Data-sequence space: MPTCP is not a single TCP sequence space striped across multiple paths. Rather, an individual TCP subflow exists for each available path and has it's own 32-bit TCP sequence space that uses the standard TCP sequence field of the TCP header. Using subflows means that the application byte stream is distributed across multiple independent sequence spaces.

An additional 64-bit *data sequence* space is defined to map the single byte sequence from the application to the subflow sequence spaces. A 64-bit *Data Sequence Number (DSN)* is assigned to each byte being sent over the connection. The DSN is functionally analogous to TCP sequence numbering, and allows MPTCP to provide in-sequence delivery of data through receive-side reassembly, and reliability through positive acknowledgement and retransmission.

A DSN is mapped to a subflow's sequence space at transmission time. The same DSN can be mapped to multiple subflows, for example if retransmission is required. Figure 5.3 provides an example of this mapping. The application has written a sequence of bytes to the send buffer. The scheduler assigns bytes to three subflows. The DSN is now mapped to the sequence space (`SEQ`) of that subflow. The allocation to Subflow 3 shows how the same application-level bytes can be mapped to multiple subflows[2].

[2]This is useful for retransmitting data on alternate paths, or for schemes that require transmitting the same data simultaneously across multiple paths.

Variable	Description
SND.NXT	Data-sequence Send Next: Next sequence number to send
SND.UNA	Data-sequence Send Unacklowledged: Earliest sequence sent but not acknowledged
RCV.NXT	Data-sequence Receive Next: The next sequence number expected to receive
RCV.WND	Data-sequence Receive Window: Receive window to be advertised by all subflows

Table 5.1: MPTCP State Variables

At the receiver the TCP `SEQ` information is used only at the subflow-level. The DSN is used to reassemble the byte stream arriving across multiple subflows. Any data stream bytes that have been duplicated (due to say parallel transmission such as in the previous example) is discarded. How the DSN to subflow mapping is conveyed to the receiver is discussed in Section 5.2.3.

Aside from middlebox compatibility, maintaining an individual TCP sequence space for each subflow has several advantages. Crucially it means that subflows can take advantage of TCP's ordered delivery and reliably at the subflow level. For example a packet loss on a particular subflow is handled by standard TCP retransmission on that subflow, without requiring intervention from the MPTCP layer.

Data Sequence Variables: MPTCP state-variables follow closely the model set out by TCP (see Chapter 2.1.2). The data-level sequence variables duplicate the semantics of TCP entirely, and are shown in Table 5.1. It is worth re-iterating that each subflow in a connection continues to maintain its own send and and receive sequence space variables (e.g. SND.NXT). There is one exception - since the receive window is defined at the connection level (i.e. the receive buffer space available to the application) all subflows advertise the MPTCP-level RCV.WND rather than advertising individual windows.

5.2 MPTCP in Operation

TCP is connection oriented, with distinct initiation, established and closing stages. MPTCP also has separate connecting, established and closed states. The following subsections show the different MPTCP concepts - transparency, TCP option signaling and the data sequence space - come together at various stages of the MPTCP connection life cycle.

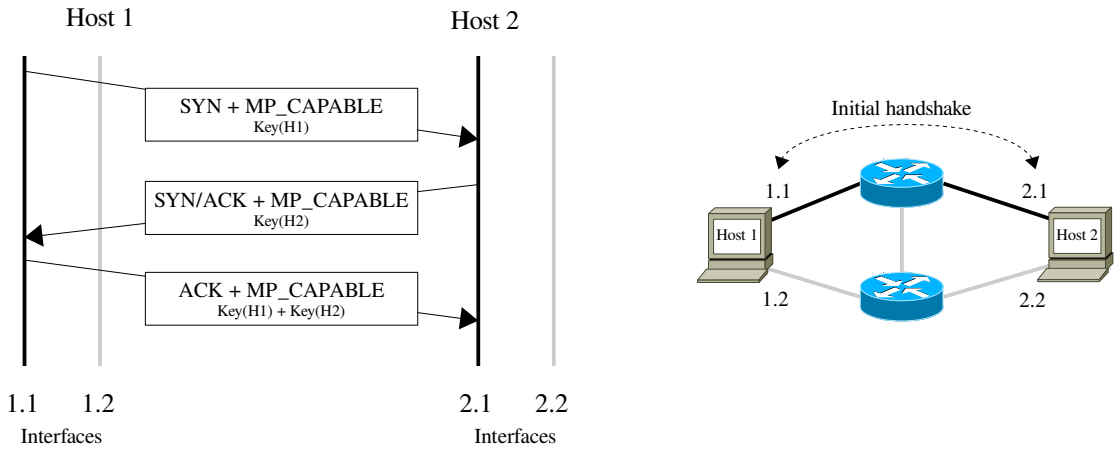


Figure 5.4: Negotiating an MPTCP connection.

5.2.1 Opening a Connection

MPTCP functionality is negotiated during the TCP handshake in much the same way as other TCP options (e.g. SACK). If MPTCP cannot be negotiated the connection continues as a standard TCP session. The `MP_CAPABLE` option subtype is used to establish MPTCP connections. This option has duel purpose: (1) To probe whether the remote host is MPTCP enabled and if so (2) synchronise a MPTCP connection. Synchronisation involves the exchange of unique *keys*, from which initial sequence numbers and session tokens are derived.

The `MP_CAPABLE` sequence is shown in Figure 5.4. Negotiation takes place over the endpoint addresses bound by the application, in this case the address `1.1` on the client (Host 1) and the address `2.1` on the server (Host 2). The client application creates a TCP socket and initiates a connection. An `MP_CAPABLE` option is included on the `SYN` packet. A server that does not have MPTCP enabled will ignore the `MP_CAPABLE` option when the packet is received, and the connection will continue as standard TCP. If, as in our example, the server is MPTCP enabled then a `MP_CAPABLE` is included on the returning `SYN/ACK`.

Also included in the `SYN` packets are session keys that have been generated by each of the hosts. A hash function of the keys is used to derive a session token and the initial data sequence numbers. For example Host 2 uses `Key(H1)` to obtain the session identifier token and the data sequence number of the first byte from Host 1. The data sequence numbers are used for byte stream synchonisation, while the token is used to uniquely identify an MPTCP session at a host.

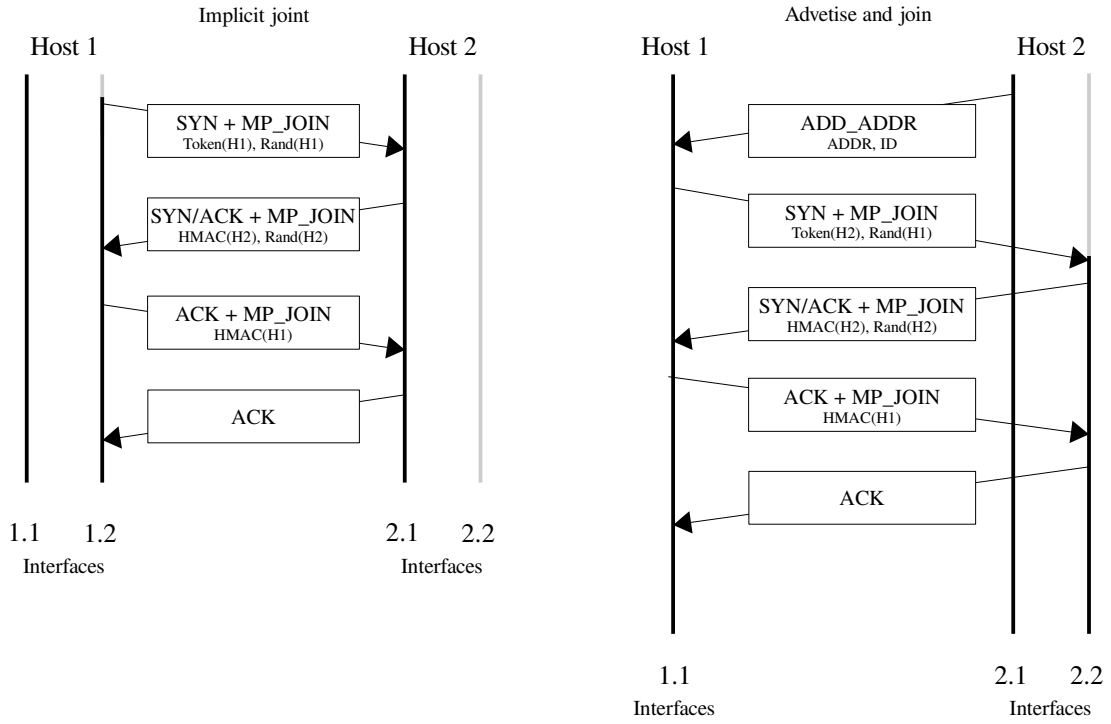


Figure 5.5: Examples of joining a subflow. On the left, Host 1 triggers an implicit join to a known address on Host 2. On the right, Host 2 must advertise an unknown address so that Host 1 can initiate a join.

5.2.2 Associating Subflows

MPTCP connections begin with a single subflow. Additional subflows added only after synchonisation. Again, the standard TCP handshake is the basis of connecting a new subflow. The MP_JOIN option is included during this handshake. Figure 5.5(a) shows a basic joining scenario, where Host 1 initiates the addition of a new subflow between addresses 1.2 and 2.1.

The MP_JOIN uses information exchanged during the MP_CAPABLE handshake for session identification and host verification. The opening MP_JOIN includes the session token exchanged during the MP_CAPABLE handshake. This identifies the SYN packet with an existing MPTCP connection. A random number is included with the token, and is used together with the session keys to create a HMAC[3]. The purpose of the HMAC is to allow both hosts to verify that each was indeed involved in the initial connection setup. The SYN/ACK includes the calculated HMAC(H2) and a random number, but *not* a token as Host 1 has already associated this 5-tuple with the MPTCP connection. Host 1 then calculates HMAC(H1) for inclusion in the final ACK/MP_JOIN. Host 2 verifies HMAC(H1) from Host 1, responding with an ACK. Following this sequence the subflow is now attached to the MPTCP connection and data may carry data. In Figure 5.5(a) we see that Host 2 has an additional interface that is not used. This raises a question as to why Host 1 would not attempt to join a subflow with address 2.2. The reason is that Host 1 does not yet know about this address.

The ADD_ADDR option allows an MPTCP endpoint to advertise an additional IPv4 or IPv6 address that the receiving host can choose to connect to. This is useful in instances where one host is behind a firewall or NAT and cannot receive implicit joins. This scenario can be applied to Figure 5.5(b). Assuming Host 1 is behind a NAT, Host 2 cannot issue an implicit MP_JOIN from the address 2.2. Host 1 cannot connect a new subflow to 2.2 since it is not aware of the address. The ADD_ADDR option is used to inform Host 1 about the additional interface, after which Host 1 can establish a new subflow at the advertised address.

It is worth noting that the specification does not mandate which host should attempt to add subflows. Either the client or the server may initiate a MP_JOIN to known addresses.

5.2.3 Transferring Data

As previously discussed, the 64-bit sequence space ensures that data is delivered in-order and reliably. Each byte in the stream is assigned a data-sequence number, and during transmission this DSN is mapped to the sequence space of the selected subflow. The data must be acked at both the subflow

[3] keyed-hash message authentication code

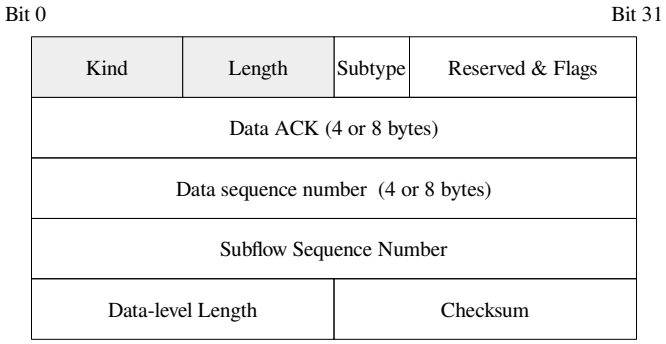


Figure 5.6: Data Sequence Signal option

Flag	When Set
A	Data-ACK present
a	Data-ACK is 64-bit
M	DSN present
m	DSN is 64-bit
F	Data-FIN is present

Table 5.2: Data Sequence Signal Flags

sequence level (i.e. standard TCP `ACK`) and at the data sequence level. The *Data Sequence Signal* (DSS) option, shown in Figure 5.6, carries the mapping between the data sequence and subflow sequence and is used to acknowledge DSNs. The DSS is used for several different signals, and the type of signal is indicated to the receiver by the *flags* field. The definition of each flag is provided in Table 5.2.

When sending data the DSS is populated with the DSN, subflow sequence number (SSN), data-level length and checksum fields. The SSN is a 32-bit sequence number relative to the start of the subflow sequence. A relative value is used for the SSN as firewalls are known to randomise sequence numbers, changing the TCP sequence numbers seen at each end-host. The DSN, SSN and data-level length represent the actual mapping between the data sequence and subflow sequence. A DSS can be used per-packet, or a single DSN can cover multiple packets.

As an example consider a DSS with the values {`DSN:200, SSN:500, Len:1000`}. This tells the receiver that the next 1000 bytes of this subflow (`SSN:500-1499`) represent data-level bytes `DSN:200-1199`. Suppose that the mapping of 1000 bytes is sent in a single packet and that

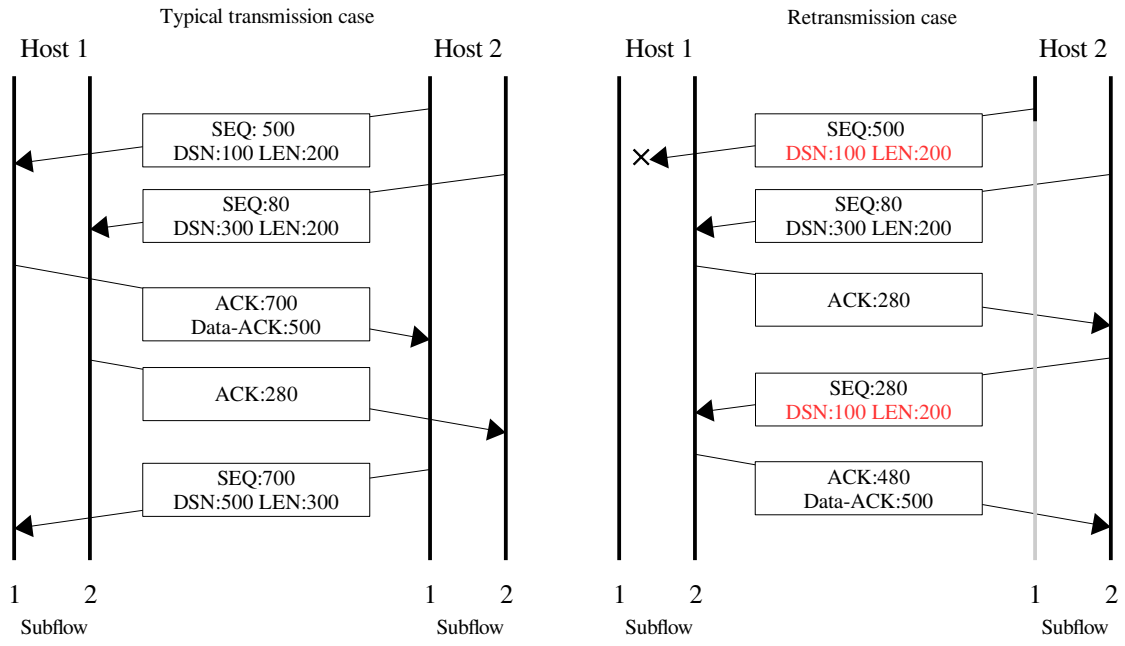


Figure 5.7: MPTCP data exchange using the DSS option. A typical exchange is shown on the left, while on the right we see a retransmission after a subflow fails.

the packet is fragmented into two packets in transit. Even if the DSS only replicated on the first packet, the receiver can use the mapping to recover the DSN for the bytes in the second packet from the TCP sequence number.

Like TCP, MPTCP uses a sliding transmission window. The right edge of the window (i.e. new data) is progressed when sent data is acknowledged. The size of the window is governed by the congestion window and receiver's advertised window. In MPTCP the receiver advertises a receive window using the *window* field in the TCP header of the subflows. Since subflows share a receive buffer, all subflows advertise the same window (receive buffer specifics are discussed further in Chapter 4). As mentioned data must be acknowledged twice - at the subflow level using a standard TCP `ACK` and at the data level using a DSS `data-ACK`. However only the `data-ACK` confirms that data has actually been received by the application, so it is this acknowledgement that allows the window to progress. Figure 5.7 shows the DSS in use when sending and acknowledging data.

The left of Figure 5.7 shows a typical transmission case, where two 200-byte mappings are transmitted across two subflows. Assume the transmission window was 400-bytes, meaning a `data-ACK` is required before new data can be sent. The data is received successfully on both subflows. Subflow 1 combines a subflow-level `ACK` and a `data-ACK` covering the data sent on both subflows. Sub-

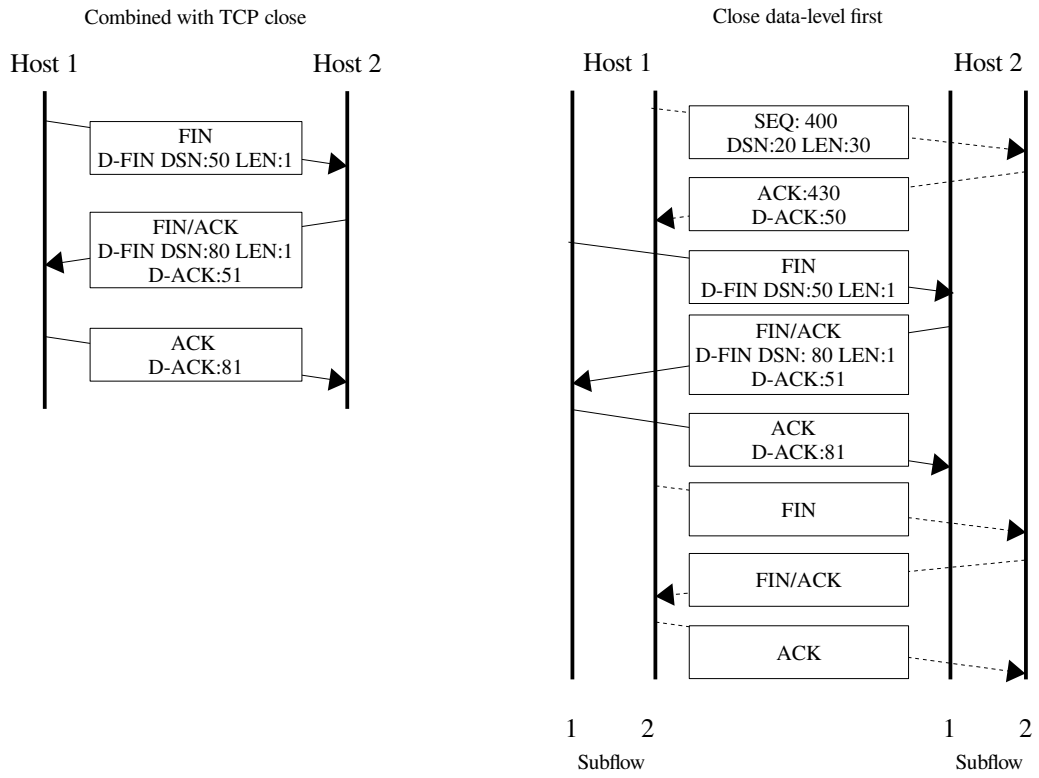


Figure 5.8: MPTCP connection close can be combined with the subflow-level close. With multiple subflows, the MPTCP connection is closed first.

flow 2 only `ACKs` at the subflow level. How `data-ACKs` are transmitted is implementation specific. For instance it would also have been equally valid to `data-ACK` on both subflows, or Subflow 2 only. Having received a `data-ACK`, the right edge of the window progresses and new data can be transmitted.

On the right of Figure 5.7 is a slightly different scenario in which packet loss occurs. Again two 200-bytes mapping are sent. This time however Subflow 1 fails soon after transmission, meaning data sequence bytes `100-299` are not delivered. Subflow 2 does not have any issues and delivers the data sequence bytes `300-499`. The 200 bytes are acknowledged at the subflow level (`ACK:280`) however a `data-ACK` is not transmitted, since data sequence acknowledgements are cumulative and bytes `100-299` were not received. After a timeout period bytes `100-299` are remapped onto Subflow 2 and retransmitted. Data sequence bytes `100-499` are cumulatively acknowledged and transmission may advance.

5.2.4 Closing a Connection

In standard TCP the `FIN` flag is used to inform the receiver that data transmission has completed, typically the result of an application calling `close()` or `shutdown()` on a socket. Both endpoints must independently send and acknowledge a `FIN` before the connection is closed. A standard TCP `FIN` in an MPTCP connection applies only to the subflow on which it is sent. This allows the closure of subflows independently of the overall connection. The MPTCP connection must be closed at the data sequence level. For this, the TCP `FIN` semantics are replicated using the `data-FIN` flag and `data-ACK` field of the DSS option. Thus for MPTCP-enabled connections, closing a socket initiates a MPTCP closing handshake.

Figure 5.8(a) shows a MPTCP connection close combined with a TCP subflow close. In this example all data has been transmitted and acknowledged and Host 1 wishes to close the connection. The sequence plays out much like a standard TCP closing sequence.

Figure 5.8(b) presents a more complex close for a connection that has two subflows. At the beginning of this sequence 40 bytes are being sent from Host 1 to Host 2 on Subflow 2. Soon after the data is acknowledged and Host 1 calls `close()` on the socket. The scheduler decides to close the connection by sending a `Data-FIN` on Subflow 1. Since there are no data outstanding a TCP-level close is also initiated for Subflow 1. Host 2 sends a `data-ACK` that acknowledges the `data-FIN` sent on Subflow 1, as well as a data-FIN. Host 1 then sends acknowledgements that close Subflow 1 and the MPTCP connection. Although the MPTCP connection has now been closed, Subflow 2 is still an active TCP connection. The MPTCP implementation can simply initiate a TCP `FIN` sequence on this subflow at this time.

It is important to note that the `data-FIN` is not acknowledged until all previously transmitted data has been acknowledged by `data-ACK` (even if the data has been acknowledged at the subflow level). This ensures that all data transmitted has been received before the connection is closed. As the examples have shown, subflow-level and connection-level closes are decoupled. Thus the MPTCP connection is typically closed first to confirm the delivery of data, followed by the subflows.

5.3 Other Protocol Considerations

Reliability: MPTCP must support reliability at the data sequence level. Subflows already benefit from TCP's existing recovery mechanisms, though retransmissions are subflow-local. Data sequence retransmission involves remapping of a DSN to a new subflow when the original subflow is unable to

deliver the data (perhaps after TCP-level recovery fails). Figure 5.7(b) provided an example of how the data sequence can be remapped for transmission on an alternate subflow.

The MPTCP specification does not define a particular mechanism for performing data level retransmissions, and the how and when of retransmitting data is implementation specific. Data sequence retransmission might be triggered based on a retransmission timer, or upon detecting a subflow in retransmission. For example, an aggressive scheme could perform data sequence retransmission for every packet loss detected on a subflow. This would reduce delay in retransmitting data at the expense of efficiency (packet loss is not uncommon so duplicate transmissions would be likely). A more conservative approach may wait for a subflow to experience multiple TCP-level retransmission timeouts before resending the data on a new subflows. This approach would reduce the amount of data duplicated over multiple subflows (since data would have a greater chance of being recovered at the subflow-level) at the expense of responsiveness - it would take longer for data level retransmits to occur.

Selecting a subflow on which to retransmit data (when multiple choices are available) is also important, and has been an active area of multipath research. Subflow selection strategies for retransmission are discussed further in Chapter 4.

Falling Back to TCP: The MPTCP design is heavily influenced by the need to maintain compatibility with today's Internet. Specifically this means being resilient to interference from middleboxes. But what is the correct response if such interference is detected? In most cases MPTCP will aim to 'fallback' to a standard TCP connection. This behaviour is transparent and appropriate given the application was at minimum expecting a TCP socket.

Middlebox resilience is built into signaling procedures. The `MP_CAPABLE` handshake first confirms that MPTCP can be indeed be used along the path. Failures (such as the removal of an option) during at this stage will result in the connection simply continuing as TCP. During design testing it was found that some middleboxes allow new options on `SYN` packets but not standard data packets [2]. Thus a DSS option must be acknowledged by another DSS `data-ACK`. Failure to carry DSS options again results in fallback to TCP. Even if a path is known to carry MPTCP options, there are still instances where the connection must respond by terminating a subflow or falling back to TCP, for instance if a middlebox alters payload data.

The DSS option, introduced in Section 5.2.3, contains a checksum field when sending mapped data. This checksum detects if the payload data has been altered in any way. It is important to detect these changes as any alteration in the payload breaks the mapping between the subflow and data

sequence spaces. If sending data on multiple subflows, it is not possible to recover the data or determine how the changes impact the overall data stream. In this case the affected subflow is terminated with the `MP_FAIL` signal, which describes the portion of data that will need to be retransmitted. A checksum failure on a single subflow connection results in an effective fallback to standard TCP.

Chapter 6

An Architecture for MPTCP in the FreeBSD Kernel

The UCL implementation of MPTCP for Linux has demonstrated one approach to creating an extensible, modular architecture [110, 150]. Although generalised for adaptation to other operating systems [110], differences between the Linux and FreeBSD kernels mean that a close translation of this design is difficult. In this chapter we describe an architecture that is suitable for integration within the FreeBSD TCP stack. The implementation was designed to meet the following goals:

- *Extensibility:* Further experimentation into multipath scheduling and congestion control must be supported through modular scheduling and congestion control frameworks. The in-kernel components of the implementation should also be easy to modify or improve.
- *Maintainability:* The FreeBSD stack is constantly changing. The implementation should be designed with ease of maintenance in mind so that it can stay compatible with the kernel into the future.
- *Minimal impact on TCP stack performance:* Adding MPTCP support requires inserting code into existing TCP code-paths. Impact on the performance of the native FreeBSD TCP stack when running standard TCP connections needs to be minimised.

Our MPTCP stack is completely implemented within the FreeBSD kernel space, utilising aspects of the modular TCP framework introduced in FreeBSD-head r292309 [151]. There have been several significant design evolutions during the development process. These previous designs are documented in two technical reports [152, 153]. These are also included as appendixes of this thesis.

6.1 Designing for FreeBSD

Implementing MPTCP for the FreeBSD kernel is a significant undertaking. The API is expected to present the single-session socket of conventional TCP, while underneath, the kernel is expected to support the learning and use of multiple IP-addresses for session endpoints. This creates a non-trivial implementation challenge to retrofit such functionality into existing, stable TCP stacks. For example:

- New shared data structures must be initialised appropriately and protected against corruption.
- Additional data-level processing (such as reassembly) must be deferred so as to not delay TCP-level responses.
- The mainline TCP stack is constantly evolving and MPTCP code must stay compatible.

These are only a few of the challenges in creating an implementation. In this section we discuss how each of these issues was considered in our design.

6.1.1 Data and Control Structures

In Section 5.1 we described MPTCP as a logical layer between the socket and TCP stack. This layering separates data-level and subflow-level operations. The MPTCP layer provides the primary send and receive buffers for the application. It is from these buffers that the scheduler allocates data and reassembles the data stream from the multiple subflows. The MPTCP layer also has direct access to the TCP subflows, allowing it to coordinate scheduling and congestion control. To provide these capabilities, new MPTCP-specific structures are required and the interface between the socket and transport protocol must be modified.

A standard TCP socket in FreeBSD is composed of three linked datastructures:

- *Socket*: The `socket` structure is the interface between the application and the protocol. It contains a pointer to the protocol-specific datastructures and functions. The send and receive socket buffers are also held in the socket structure.
- *Internet protocol control block*: The`inpcb` stores IP-layer information such as addresses and ports, and holds the TCP control block. The kernel maintains a hash list of `inpcbs` that is used to map incoming packets to TCP sessions.
- *Transmission control protocol control block*: The `tcpcb` holds state variables required for the

TCP protocol. In addition to maintaining variables for the receive and send data sequences (see Chapter 2.1.2), the `tcpcb` tracks the finite state machine and maintains session timers.

MPTCP introduces five new datastructures:

- *Multipath protocol control block*: The `mppcb` is the interface between the socket and the MPTCP protocol and is initialised upon successful MPTCP handshake. It contains endpoint information (addresses, ports), the MPTCP protocol function and a reference to the *Multipath connection* block.

- *Multipath connection*: The `mpconn` maintains connection-level state. This includes a list of *subflow subsockets*, and settings for the desired scheduling and congestion control algorithms. The current implementation also maintains variables for path management.

- *Multipath subsocket*: There is an `mp_subsock` for each subflow. Within are connection-level related variables for subflow, such as address ID. Linked to this is a standard TCP socket (consisting of a *socket-inpcb-tcpcb*) that represents the subflow.

- *Multipath control block*: The`mpcb` contains the core state for the MPTCP protocol. Within are variables for tracking data-sequence numbers, timers and the finite state machine for the connection. Table 6.1 defines several key `mpcb` variables referenced throughout the remainder of this chapter.

- *TCP Multipath connection*: The `t_mpconn` is the point of connection between the TCP control block and MPTCP session. It stores subflow-specific state and a back-pointer to the `mpcb`. It is attached to the `tcpcb` via the *function block pointer*, one of two pointers within the `tcpcb` that are used to attach auxiliary datastructure or functions to the standard TCP stack (provided as part of FreeBSD's modular TCP stack support, discussed in Section 6.1.3).

Figure 6.1 shows the relationship with the existing TCP structures. Preexisting structures are shaded grey, with newly defined structures in white. The arrows represent pointers between structures. The left portion of the figure represents the data-level, while on the right is the subflow-level. We see here how the standard *socket-inpcb-tcpcb* arrangement is preserved for each subflow.

The size of each new structure is given in Table 6.2. The total space required for MPTCP datastructures is 2176 bytes without a subflow. Each subflow is 1976 byte, the size of a standard TCP socket. A single subflow MPTCP connection thus requires 4156 bytes (2176+1976) for protocol blocks, roughly the equivalent of two standard TCP connections.

Variable	Description
ds_snd_nxt	Data-level SND.NXT
ds_snd_una	Data-level SND.UNA
ds_snd_una	Data-level RCV.UNA
ds_rcv_wnd	Data-level RCV.WND

Table 6.1: MPTCP control block sequence variables

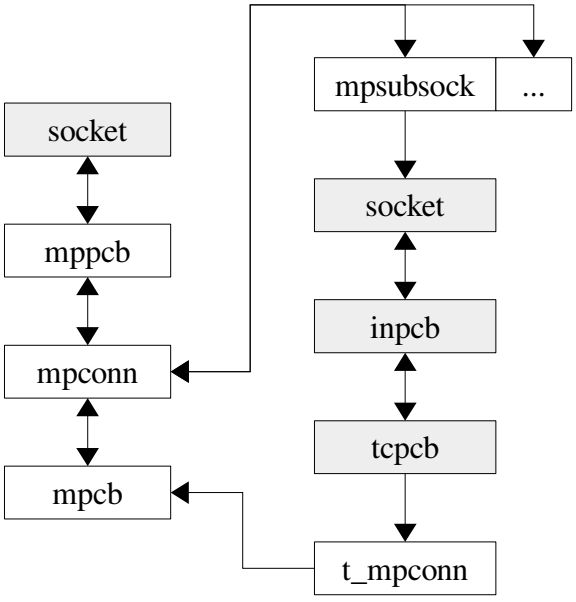


Figure 6.1: Datastructures to support MPTCP in the FreeBSD kernel. Preexisting kernel structures are shown in grey.

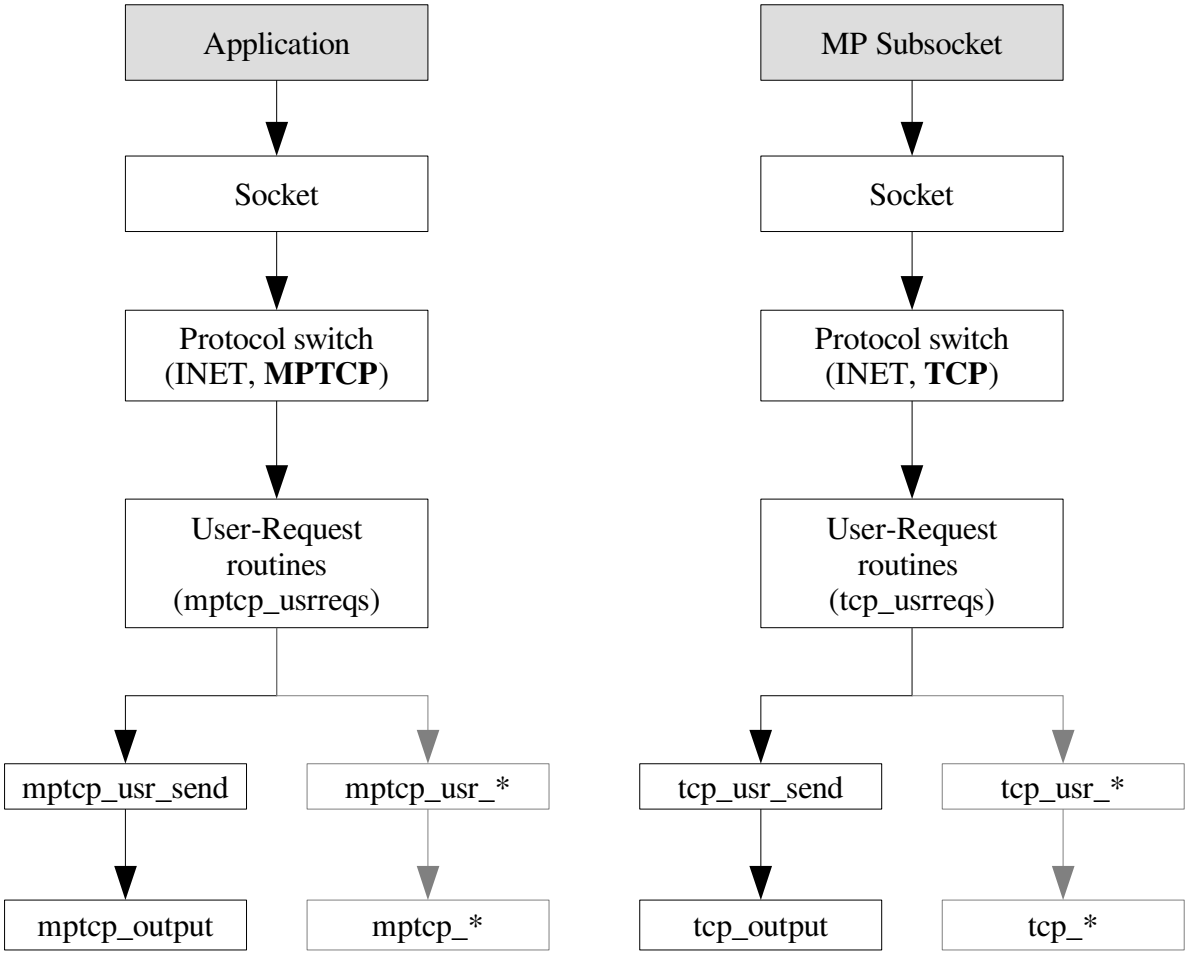


Figure 6.2: Changing the protocol switch and user request routines of a socket.

Structure	Size (bytes)
inpcb	464
tcpcb	648
socket	864
mppcb	120
mp_conn	616
mpcb	240
t_mpconn	336

Table 6.2: Size of MPTCP datastructures, compared with the existing TCP socket structures.

The current control-block structure of the implementation differs significantly from earlier iterations, in which the inpcb and tcpcb structures were directly modified to accommodate MPTCP state. The move away from this earlier approach was motivated by several factors, though principally:

- The desire to avoid modifying the existing tcpcb and inpcb structures.
- To enforce a more logical delineation between subflow-level and data-level variables, and remove connection-level state from the subflows. For example, creating a single mpcb to store data-level state makes more sense than having the variables replicated across multiple subflow tcpcbs.
- To simplify the implementation. Early versions of the implementation attached multiple inpcb/tcpcb pairs to a 'master' socket. This took away the ability to re-use existing socket calls for each subflow, and additional complexity resulted from having multiple subflows share single send and receive socket buffers. Creating separate MPTCP structure to interface with the socket and encapsulate the subflows allowed for a simpler design and greater re-use of existing functions.

The current PCB structure could be further refined, for instance by merging the mppcb and mpconn. This will need to be explored in future work.

Socket-protocol interface: A socket contains a *protocol switch* structure that identifies the attached network protocol and the entry and exit points between the protocol and the socket. This includes details of initialisation and control functions and protocol identifiers (e.g INET, TCP). Of particular interest in the MPTCP case is the *user-request routines* field, which points to a table of functions that specify the interface between the socket and the protocol. User-request functions are

called in response to system calls such as `send` or `connect`. Figure 6.2 shows how the protocol switch applies to an MPTCP connection.

We see that subflow sockets retain the user-request routines as per a standard TCP socket, while the primary socket calls the MPTCP interface. As MPTCP sits between the MPTCP and TCP layers, a new protocol switch block is defined for MPTCP sockets. This allows control signals and data written by the application to be handled by MPTCP before being passed to the subflows. Crucially, the protocol switch can be changed *after* the `MP_CAPABLE` handshake, meaning applications can still request a standard TCP stream socket.

6.1.2 Event-driven Model

The decoupled nature of the implementation and the desire to offload processing to another thread (discussed in Section 6.2) means that an event-driven approach was devised. In the event-driven model, MPTCP operations occur in response to messages from subflow or socket events. For example, a subflow might append a new in-order segment of data to it's receive buffer. This results in a *socket upcall* that queues an message that is later processed in a dedicated event-processing thread. This approach has some similarities to TCP architectures where connections are queued on a dedicated *protocol worker-thread* for processing [154, 155]. We use an event-driven design as it allows the more computational aspects of MPTCP processing (e.g. data-level aggregation) to be performed independently of the subflow thread context.

Tasks are handled by the `mp_event_handler` routine. An `mp_event` structure is used to hold information about the event that has occurred and is parsed by the event handler. The procedure for creating and processing MPTCP events is shown in Figure 6.3 and can be summarised as follows:

1. When an event occurs an `mp_event` is created and inserted into the `mp_event_list`. This list is shared by all MPTCP connections.

2. A task is enqueued on an event processing queue, along with a pointer to the `mp_event_list`.

3. The task thread runs, dequeues an event from the `mp_event_list` and calls the MPTCP event handler.

4. The event is decoded and an appropriate handler function is called to take action.

Some MPTCP operations benefit from being performed in the same execution thread as TCP processing. As discussed in Section 6.1.1, the TCP control block (`tcpcb`) should have access to the MPTCP

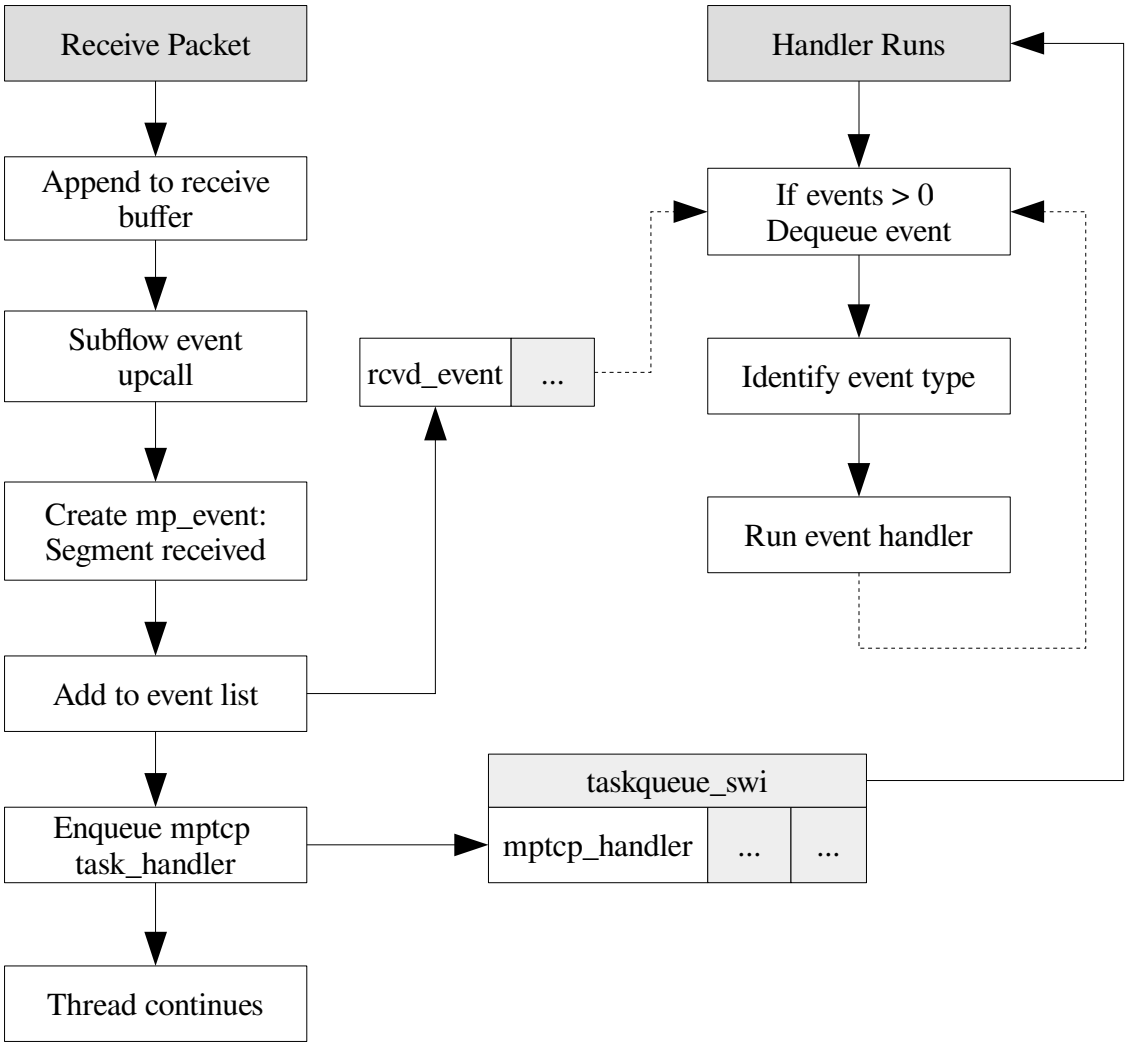


Figure 6.3: Processing a subflow socket event

control block (mpcb). This is so that operations that requires a fast turn-around, such as a data-level acknowledgement, can be performed more quickly. In the case of Data-ACKs, these can therefore be piggybacked on TCP ACKs without waiting for MPTCP-level processing. Other MPTCP supporting code, such as MPTCP option-processing are also executed within the TCP-processing thread[1].

The use of a single queue to process events effects the processing parallelism afforded to MPTCP connections. Critically, this approach serialises event-processing for all connections (even those with a single subflow) onto a single thread. In effect, this means events on different MPTCP connections are serviced by a single FIFO queue, with each connection waiting for events on the preceding connection to be processed. This is in contrast to FreeBSDs TCP stack implementation, where each message for each TCP connection is processed entirely within it's own thread, allowing for extensive parallel processing if multiple CPUs are available.

A second consequence relates to scalability. A single taskqueue for all processing MPTCP events will act as a bottleneck as the number of concurrent MPTCP connections increases. The trade-offs between a serialised and parallel approach to protocol processing are covered further in Chapter 6.2.

6.1.3 Leveraging the Modular TCP framework

The implementation re-uses much of the existing TCP stack code. An important design consideration was how tightly the MPTCP code should be coupled with the existing stack. A tightly integrated approach benefits from greater flexibility and may ultimately be better performing. The trade off is implementation complexity and a high maintenance overhead (as code must continually be updated as the TCP stack changes). A less integrated approach may result in less efficient code paths, and has less direct access to the subflows. This could potentially limit performance in the long-term. Maintenance is easier however as code is not directly affected by the frequent changes to the TCP stack.

In our initial prototype releases [152, 153] we took a highly integrated approach. This provided some benefits such as scheduling flexibility but added significant code to the TCP stack and proved difficult to maintain and debug. Version 0.5 of the implementation was a ground-up redesign that decoupled the majority of MPTCP operations from the TCP stack. The current version further separates MPTCP by using features of the modular TCP framework.

It is not possible to altogether avoid modifying the TCP stack. The kernel must be patched and rebuilt to support MPTCP. It is possible however to minimise the changes by using a combination

[1] MPTCP is a standardised TCP option and is thus processed alongside options such as TCP-SACK.

of asynchronous task execution and the modular TCP framework. The framework provides a way to experiment with new TCP functionality without having to change the mainline TCP stack. As discussed earlier, some procedures - such as option processing - are best integrated with the TCP stack. Where code must be added to the existing stack it is enclosed within a `#ifdef MPTCP` directive, allowing the inclusion or exclusion of MPTCP using the kernel configuration file.

A benefit of combining in-stack and modular approaches is that it allows for a simpler, largely decoupled implementation that can be further refined over time into a parallel, MPTCP-specific stack. We now discuss some of the specific benefits of using the modular TCP framework.

Redefining TCP functions: Existing TCP functions such as `tcp_output()` and `tcp_do_segment()` have been replaced by function pointers and are specified by the `tcp_function_block`. This is the essence of the modular stack and the function pointers allow new functions to be substituted in place of the defaults.

Dynamic stack selection: The modular framework allows the TCP stack to be changed even after a call to `connect()` or `listen()` has opened a connection. Thus it is possible to attempt to negotiate an MPTCP connection and switch to the default TCP stack if MPTCP is not supported. Ideally MPTCP infrastructure is only allocated once we are certain that an MPTCP session will take place (i.e. on completion of the `MP_CAPABLE` handshake). If MPTCP is not negotiated for a connection, then the default TCP stack should be used.

Extending the TCP control block: The `t_fb_ptr` pointer in the `tcpcb` can be used to attach arbitrary data associated with a modular TCP stack. As we discussed previously, this pointer allows MPTCP data to be associated with a subflow `tcpcb` without having to modify the `tcpcb` itself. It is preferable to extend TCP-related structures rather than change them, as changing kernel structures has various side-effects. For example increasing the size of the `tcpcb` would increase the memory use of *all* TCP connections, even if MPTCP is not active. Perhaps more importantly, modifying kernel structures modifies the *Application Binary Interface* (ABI). Changing the ABI is discouraged as device drivers or modules implemented outside of the kernel may read a kernel structure directly (rather than via a call to an API). Changes to a structure may potentially break functionality (unless the external code is recompiled). The end result is that changing the ABI may lead to unpredictable behaviour on a system with the MPTCP patch applied.

6.1.4 Scheduling and Congestion Control

We have discussed ongoing MPTCP research in Chapter 4. Much research still remains in the areas of CC and scheduling for MPTCP. Supporting this research is the key motivation for our implementation. Therefore scheduling and CC components are designed as kernel modules that can be dynamically loaded and configured by the user, an approach that eases the path to practical experimentation.

The scheduler component is largely independent of existing TCP mechanisms and operates within the MPTCP-layer. The scheduling module only needs to provide access into MPTCP control block and input/output paths. Unlike scheduling, MPTCP flow-control must respond to events at the subflow level (e.g. the reception of a `Data-ACK` does not precipitate a change in the congestion window). The major involvement of the MPTCP-layer is in setting the CC algorithm for each subflow and allowing the congestion control algorithm to consider the state of multiple subflows when processing an `ACK`. The modular scheduling component was therefore designed to compliment the existing TCP-CC framework to support both coupled and uncoupled CC approaches.

Modular TCP-CC framework: FreeBSD already features a modular CC framework for single-path TCP ('ModCC'). It was initially developed at Swinburne University [156] and was modeled on the SCTP modular CC framework. The framework was designed to provide a platform for conducting TCP-CC research by allowing CC algorithms to be built as dynamically loadable kernel modules[2]. We use ModCC as a template for our modular components and adopt the design approach and usage semantics for the MPTCP scheduling and CC frameworks. Some key features of ModCC:

- CC routines are replaced with calls to a set of function pointers defined in the structure `cc_algo`. New algorithms are created by implementing these functions.
- Algorithms can allocate additional arbitrary per-connection storage via the `cc_data` pointer.
- The `cc_var` structure is defined to pass related variables between the TCP stack and the new functions. This includes a pointer to the TCP control block, state flags and `cc_data`
- Pointers to `cc_algo` and `cc_data` in the TCP control block allow CC to be specified per-connection. If required these can be be changed during a connection.
- Infrastructure to register algorithms and select from a list of registered algorithms.

The `cc_algo` structure shown in Listing 1 encompasses the entirety of an algorithm from the initialisation function and name to the actual CC functions, of which some or all can be implemented.

[2]Previously, every CC change required patching and rebuilding the entire kernel

For example, an algorithm may replace only `ack_received` and use the default[3] routine for other CC events. Routines are provided to register and unload algorithms as kernel modules. Registered algorithms form a global linked-list and can be viewed and selected via the sysctl interface variables `net.inet.tcp.cc.available` and `net.inet.tcp.cc.algorithm`. Any algorithm specific parameters can be configured via this interface.

Scheduler: The modular scheduling framework is closely modeled on ModCC. Schedulers are defined by the structure `mp_sched_algo`, shown in Listing 2. The scheduler is an MPTCP-layer mechanism and is attached to the `mpconn` via the pointer `sched_algo`. Currently only a single function is required of the scheduler - `(*get_subflow)(struct mp_sched_var *sch_var)`. This function invokes the scheduling algorithm and returns the `mp_subsock` of the selected subflow. The structure `mp_sched_var` stores pointers to `mpconn` and `mp_sched_data`. Through these pointers the scheduler is able to access all subflows and any variables within. Persistent storage is provided through `mp_sched_data`, which is initialised at the beginning of the connection. Additional functions may be added in future versions if required. The registration and configuration approach are identical to that of ModCC. In this case scheduling variables are attached to the `sysctl` tree `net.inet.tcp.mptcp.sched`.

Congestion control: ModCC conveniently implements all the hooks required for MPTCP-CC and we only needed to include mechanisms at the MPTCP-level for registering and configuring MPTCP-CCs. Thus all functions defined in Listing 1 can be implemented for a MPTCP-CC. As with single-path ModCC, an algorithm can override some or all of the CC functions. In the case where no multi-path function is defined then subflows use the default single-path function. A MPTCP-CC may also assign single-path CCs on a per-subflow basis mid-connection, by initialising (if necessary) the new algorithm and changing the `cc_algo` pointer of the subflow TCP control block.

As with the scheduler, a sysctl tree `net.inet.tcp.mptcp.cc` is created for selecting and configuring algorithm parameters. MPTCP-CC is kept separate so as to not pollute the single-path sysctl tree and allow separate defaults to be set for single-path and multi-path TCP connections.

Though MPTCP-CC algorithms are defined using the `cc_algo` structure, they are initialised along with the MPTCP control block. On initialisation the `cc_algo` for the first subflow is changed to newly initialised MPTCP-specific algorithm. A single memory structure pointed to by `cc_data` is also initialised and is shared between the subflows. Subsequent subflows are also set to use the CC instance created by the MPTCP-layer.

[3]TCP-New Reno in FreeBSD

Listing 1 Definition of structure `cc_algo`

```
struct cc_algo {
      char  name[TCP_CA_NAME_MAX];
      /* Init global module state on kldload. */
      int   (*mod_init)(void);
      /* Cleanup global module state on kldunload. */
      int   (*mod_destroy)(void);
      /* Init CC state for a new control block. */
      int   (*cb_init)(struct cc_var *ccv);
      /* Cleanup CC state for a terminating control block. */
      void  (*cb_destroy)(struct cc_var *ccv);
      /* Init variables for a newly established connection. */
      void  (*conn_init)(struct cc_var *ccv);
      /* Called on receipt of an ack. */
      void  (*ack_received)(struct cc_var *ccv, uint16_t type);
      /* Called on detection of a congestion signal. */
      void  (*cong_signal)(struct cc_var *ccv, uint32_t type);
      /* Called after exiting congestion recovery. */
      void  (*post_recovery)(struct cc_var *ccv);
      /* Called when data transfer resumes after an idle period. */
      void  (*after_idle)(struct cc_var *ccv);
      /* Called for an additional ECN processing apart from RFC3168. */
      void  (*ecnpkt_handler)(struct cc_var *ccv);
      /* Called for {get|set}sockopt() on a TCP socket with
         TCP_CCALGOOPT. */
      int   (*ctl_output)(struct cc_var *, struct sockopt *, void *);
      STAILQ_ENTRY (cc_algo) entries;
};
```

Listing 2 The `mp_sched_algo` structure definition

```
struct mp_sched_algo {
  char  name[MPTCP_SCHED_NAME_MAX];
  int   (*mod_init)(void);
  int   (*mod_destroy)(void);
  int   (*mpcb_init)(struct mpconn);
  void  (*mpcb_destroy)(struct mpconn);
  void  (*conn_init)(struct mpconn);
  void  (*get_subflow)(struct mpconn);
  STAILQ_ENTRY (mp_sched_algo) entries;
};
```

6.2 CPU and Memory Considerations

We have previously discussed the event-driven approach in Section 6.1.2. This is one part of an overall effort to reduce the impact of running MPTCP on a system as a whole. In this section we discuss our approach to charging MPTCP for CPU time and managing and protecting memory.

6.2.1 Ensuring Fair CPU Use

A CPU must perform many tasks simultaneously. Not all tasks are of equal importance, therefore a hierarchy of priorities exist to allow the task scheduler to address the most time-critical tasks first. Hardware interrupts are the highest priority threads available in FreeBSD, and are run in advance of other threads in the system. Unlike other threads, hardware interrupts are not scheduled and execute asynchronously, preempting an already executing thread if necessary. Given the high priority, operations executed after such an interrupt are typically kept short so that the CPU can return to other tasks. Hardware interrupt handlers typically queue tasks for later processing by software interrupt (SWI) handlers. These software handers run in the SWI context and are the highest priority *scheduled* kernel thread.

MPTCP adds additional processing steps on top of TCP. When sending a packet, the scheduler must be invoked to select a subflow. On the receive side, segments must be aggregated and re-ordered as they arrive from different subflows. The sharing of MPTCP datastructures between multiple subflows creates overhead when accessed by simultaneously executing threads (this is discussed further in Section 6.2.2). Each new subflow added to an MPTCP connection increases the amount of CPU

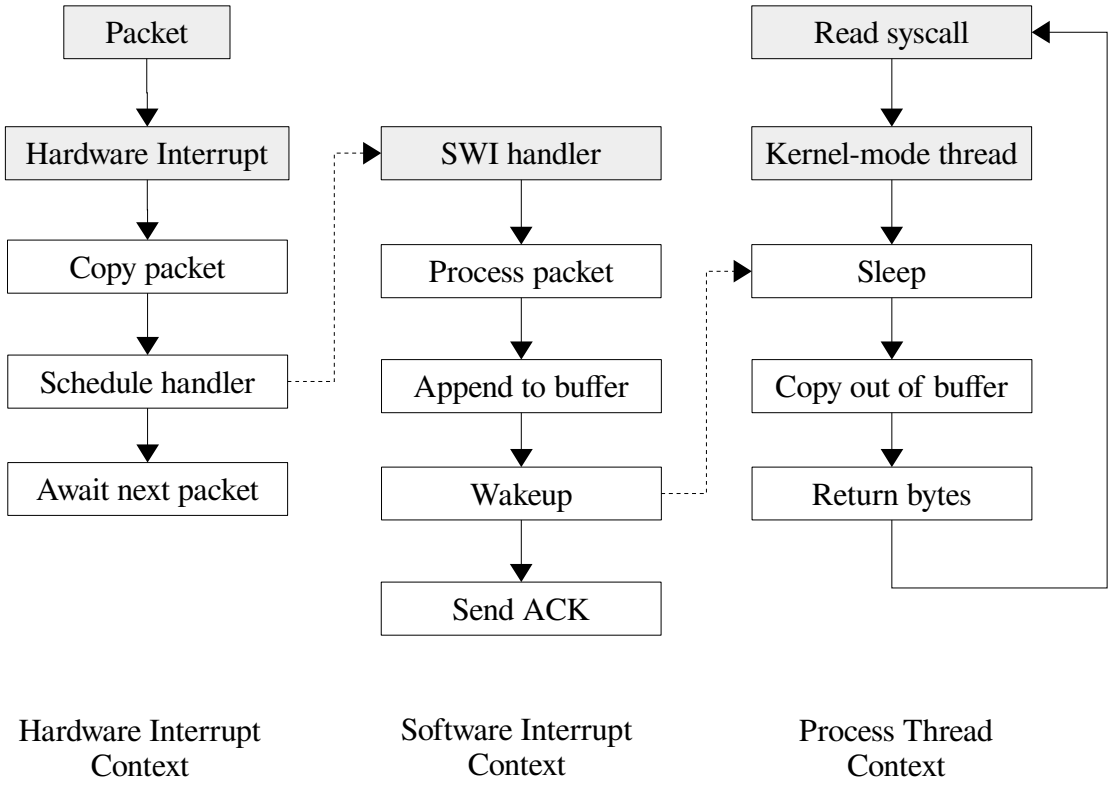


Figure 6.4: Multiple threads are run when receiving a packet.

time consumed by that connection, which raises the separate question as to whether an application that uses MPTCP transport should be able to consume more kernel CPU time as compared to applications using standard TCP.

This ties in to how much MPTCP processing should be performed in the kernel interrupt context. On the send-path, for example, the thread context is borrowed from the application that performed the syscall. The user thread is is thus responsible for copying data to the socket and executing the protocol-specific functions to send the packet. On the recevie-side, however, protocol operations up to and including data-level reassembly and receive occur in the SWI context. Given consuming a disproportionate amount of high-priority CPU time. Ideally MPTCP processing should be deferred to another thread (for instance in the user context [157]), and charged to the application receiving the data. This would allow the TCP processing-path to be kept as short as possible, and minimise the execution time of the software interrupt.

Figure 6.4 shows how processing a received TCP packet is divided between different thread contexts. An incoming packet triggers a hardware interrupt. This causes a SWI to be scheduled that,

when run, handles processing of the packet up to the top of the transport-layer. Code executed in the SWI is directly tied to processing the current packet (parsing options, updating sequence numbers, etc). This includes appending or dropping data from the receive or send buffers. The scheduler charges the user-space application for CPU time only when data is read out of the receive buffer. This processing model, where a packet is processed entirely within the calling SWI context, is referred to as *direct dispatch* [158] and was introduced as the FreeBSD TCP stack transitioned to multi-threaded operation.

MPTCP sits logically between the SWI thread that processes the packet and the application thread that reads out the received data. A fair (to other processes) approach is to defer MPTCP processing, such as re-ordering segments, to the application context. This is the the approach taken by the UCL Linux implementation (an advantage of the Linux kernel is that TCP segment reassembly is already charged to the user process). The FreeBSD kernel does not currently work like this, so for simplicity we use the taskqueue interface [159] to defer MPTCP processing to another kernel thread. This *deferred dispatch* approach more closely resembles the pre-multiprocessing FreeBSD releases that serviced network protocols (such as TCP) via a single software interrupt context [160, 161].

Taskqueue threads enable asynchronous code execution within the kernel. Several global taskqueues exist within the FreeBSD kernel and additional taskqueues can be instantiated if required. Enqueuing and executing a task on a taskqueue works as follows:

1. On an initial thread, a task is placed onto the tail of the desired taskqueue. If the task is already enqueued then a variable is incremented to reflect how many times the task has been enqueued. The taskqueue thread is scheduled for a wakeup and the original thread continues to execute.

2. After an unspecified time the taskqueue thread is set to run and the first task on the list is dequeued. The function associated with the task is called, passing a pointer to related data structure and a count of how many times it was enqueued.

3. The function processing the task is run. In our case this function would locate the MPPCB for the connection and perform data-level processing.

We currently use the taskqueue *taskqueue_swi* [159] for MPTCP. Tasks executed in `taskqueue_swi` are in the same priority class as TCP interrupt threads. However as it is delayed to another thread TCP-level processing can complete and return without waiting for data-level processing.

As discussed in Chapter 6.1.2, the use of a single taskqueue thread means that receiver-side processing for MPTCP connections is serialised. This has an advantage in simplifying lock-order (dis-

Lock	Description
INPCB_WLOCK	Protects the inpcb and tcpcb.
SOCKBUF_LOCK	Protects send and receive buffers.
SOCK_LOCK	Protects socket state.

Table 6.3: Per-connection locks for TCP

Lock	Description
MPP_LOCK	Protects the mppcb and mpconn structures.
MP_LOCK	Protects the mpcb.

Table 6.4: Per-connection locks for MPTCP

cussed in Section 6.2.2 below) but comes with a performance cost to individual connections. Primarily this impacts the data-receive path to the master socket. For example, subflows cannot append data directly to the master socket buffer from the SWI thread, even if data is in-order at the data-level. This means waiting for a context switch and for the MPTCP task thread to be scheduled, even in a best-case receive scenario. At the connection-level, using a single queue means that multiple connections cannot be processed in parallel, reducing efficiency in multiprocessor systems.

Using the SWI task queue means that data-level operations still consume software interrupt CPU time. A near-term improvement might be to use a lower-priority taskqueue dedicated to processing receive-side MPTCP tasks.

The use of a single taskqueue was largely motivated by ease of implementation, and the ability to support an event-driven model. Ultimately it would be desirable to transition to an approach that allows subflows to append data (whether in-order or not) to the master-socket receive queue. MPTCP-level processing would then be left to complete in the user-space thread belonging to the application. Improvements to this design, whether by increasing the number of worker threads or adopting a direct-dispatch approach, must be considered in future work.

6.2.2 Shared Memory, Locking and Concurrency

Multiprocessor CPU architectures are common. For a multi-threaded operating system, this means that tasks can occur in parallel when threads are scheduled to run on different CPUs concurrently. This improves system performance by making more effective use of the available CPUs, though careful consideration must be given to:

- *Scheduling:* Threads belonging to the same application may be scheduled to run on different CPUs. Designs must consider when and where threads might run, and what data might be accessed when they do run.

- *Accessing shared memory:* Multiple concurrent threads may need to access the same memory and this must be synchronised. Failure to correctly synchronise access to shared memory can lead to data corruption and non-deterministic behaviour. Knowing whether a thread might require access to shared memory is also important to ensure that memory isn't freed prematurely.

- *Memory consistency:* Threads executing on different CPUs may change the same datastructure, and CPU caches must be kept consistent.

The FreeBSD kernel (and TCP stack) has been optimised over a number of years [158, 161] to provide a high-level of concurrency on multiprocessor systems. The design of the MPTCP stack was therefore heavily influence by the need to work within this structure. In this section we discuss the use of *locks* and *reference counting* to ensure data integrity and safe access to shared protocol state. We also touch on some important considerations when designing for multi-processor architectures.

Locking strategy: As we have seen in Section 6.1.1, an MPTCP connection must share protocol data structures between multiple subflows that are accessed via multiple thread contexts. FreeBSD's reader/writer lock (`rwlock`) macros are used to synchronise threads that access shared structures. The `rwlock` allows shared or exclusive access to data. Multiple threads may acquire a read lock for simultaneous read-only access. The writer-lock is held exclusively, preventing other threads from reading or writing to the protected memory.

Acquiring and holding a lock incurs a processing/performance cost. Holding a lock for an extended period causes lock contention - if other threads require access to the structure they will have to wait for the current thread to release the lock. If locks aren't acquired or released in a consistent order, than *deadlocks*[4] may occur.

There are a number of locks required during processing for a standard TCP connection. Three locks significant to packet processing are given in Table 6.3. Once a packet has been mapped to a corresponding protocol block, the `INPCB` lock is held throughout protocol processing. The socket buffer locks are acquired only when dereferencing either buffer. The socket lock is held only when

[4]Where two threads block waiting for a lock held by the other thread to be released. E.g. Thread 1 holds lock A and needs to acquire lock B. Thread 2 holds lock B and needs to acquire lock A.

changing socket state[5]. The INPCB lock is ordered before the socket locks, which allows TCP processing (e.g. the input path) gain access to the socket structures (for example appending data to the receive buffer, or changing the socket connection status). As it comes first in the locking order, the INPCB lock cannot be taken while the socket lock is held. Two additional protocol locks are added for MPTCP connections, given in Table 6.4. The existing TCP locks are retained and used to protect subflows.

The MPP_LOCK is analogous to the INPCB lock, and comes first in the locking order. This lock is held when performing connection-level operations (such as running the scheduler or appending data to the master send buffer). As it sits at the top of the locking hierarchy, no subflows can acquire this lock. The lock is held only in the send-path and event-processing threads (a direct-dispatch approach would make the use of this lock difficult, since it cannot be acquired while holding an INPCB lock).

As the mpcb contains the core protocol state for MPTCP (e.g. send sequence numbers, state machine), the MP_LOCK needs to be accessed by the subflows. It therefore comes after the INPCB lock in the locking order, and can be held by any subflow thread, as well as by MPTCP connection-level threads.

Reference counting: Due to the use of the taskqueue interface to perform delayed processing of events, we need to ensure that data structure memory is stable between releasing and acquiring locks. Given the asynchronous nature of the network stack it is entirely possible that the connection state may change while an event is pending, for instance a MPTCP connection may close, freeing associated control blocks. Stability is achieved through reference counting. Reference counts are used for MPTCP protocol blocks and mp_events. A reference is also held on subflow sockets to prevent premature release when a subflow transitions to the closing phase of the state machine.

Addressing concurrency: Network protocol architectures can vary in the approach to concurrency. Willman et al. [154] define three distinct approaches to network stack parallelism - *message-based, connection-serialised by threads,* and *connection-serialised by locks.* The *connection*-oriented approaches dedicate protocol threads or locks to specific connection groups. Connections within a group are processed serially. Message-based architectures focus on processing a message (packet) entirely within one thread, without dedicated worker threads. This is the approach used by FreeBSD and Linux, and allows for the direct-dispatch of received packets to the receive buffer. To synchronise memory-access, the FreeBSD TCP stack uses a *fine-grained* [161] locking strategy in which locks

[5]Though logically distinct, it is worth noting that in implementation the socket lock is actually just the receive-buffer lock.

are placed around important data structures only, allowing for greater parallel execution. The trade-off here is that threads that perform processing may be created in a number of ways, and execute in different contexts, for example:

- A userspace process performs a write and transmission occurs in a user context thread.
- Alternatively, a packet may be transmitted in a SWI context as a result of a timer firing in the kernel.
- When receiving a packet, a hardware interrupt causes processing in the SWI context to both receive and potentially transmit in response.

In the above examples, threads may be scheduled to run on different cores. In the MPTCP case, multiple subflows may have threads running on multiple core that need to access the `mpcb`. Conversely, the single MPTCP event-task thread may need to access memory for subflows that have generated interrupts on different CPUs. Network cards that support multiple hardware queues and operating system features such as Receive Side Scaling (RSS)[6] make it very likely that subflows will be spread across multiple CPUs, increasing lock contention. In addition to this is the cost of cache misses and memory synchonisation.

The implementation does not take into account where threads will be scheduled to run, though it is likely that future work will consider for example assigning a MPTCP connection and subflows to a single CPU.

6.3 Establishing a Connection

Like other TCP extensions MPTCP is negotiated during the three-way handshake. The MPTCP structures described in Section 6.1.1 are therefore only required if MPTCP is successfully negotiated. There are two obvious approaches as to when to allocate these structures:

- Allocate all MPTCP-related structures immediately on calling `socket()`. When a new TCP socket is created, MPTCP structures are also initialised. This was the approach taken in the v0.4 implementation [153].
- Delay creation of the MPTCP structures until the completion of the `MP_CAPABLE` handshake. In such a scenario only the minimum additional infrastructure to synchronise MPTCP is required during the handshake.

[6](RSS distributes and assigns TCP sessions to different CPU cores [162])

Immediate allocation is easier from an implementation standpoint though adds overhead for non-MPTCP connections. An MPTCP connection adds at least six additional structures to a TCP socket, increasing the per-connection memory footprint. These could conceivably be freed if MPTCP was not negotiated, however if MPTCP is not expected to be used for the majority of connections it may be inefficient to continually allocate and free these structures.

Where standard TCP sessions are the majority an alternative approach is to delay allocation of MPTCP structures until the completion of the `MP_CAPABLE` handshake. In this case only the bare minimum structures required to enable MPTCP are allocated for the initial handshake. The current approach follows the delayed allocation method.

If MPTCP is enabled, new TCP sockets will be initialised with the *MPTCP stack* loaded. This assigns certain TCP functions to the MPTCP stack equivalents, though does not initially allocate all MPTCP structures (in particular `mppcb`, `mpconn` and `mpcb` are not initialised). Defaulting to the MPTCP stack initially provides two benefits:

- For the client, `t_mpconn` can be attached to the `tcpcb`, providing storage for MPTCP nonces and tokens during the `MP_CAPABLE` or `MP_JOIN` handshakes. On the server side, the `syncache` structure is extended with a pointer to the `mptcp_syncache` structure (discussed further below).

- The function `tcp_do_segment` can be replaced with `mptcp_do_segment`. This function can be used to determine whether MPTCP has been successfully negotiated.

. A check for the inclusion of `MP_CAPABLE` during the handshake determines if the remote host supports MPTCP. In the case where MPTCP is supported by the peer, then the handshake can be completed and remaining structures allocated and attached to the socket. The TCP connection used for the handshake then becomes the first subflow. If allocation of the MPTCP structures fails, the connection reverts to the default TCP stack.

The test for MPTCP support can be performed easily whether the host is *active opener* (client) or *passive opener* (server). For the client this is performed on the `SYN/ACK`, while on the server it is performed on the first received `SYN` and final `ACK` of the handshake. If `MP_CAPABLE` is missing during any of the checks then the `t_mpconn` structure is freed and the TCP function block `tp_fb` is set to the default TCP stack (and thus the connection falls back to standard TCP).

The check for MPTCP options is performed in `mptcp_do_segment()`. The pseudo-code is

Listing 3 Release MPTCP data and revert to default TCP stack if MPTCP is not requested by the peer during SYN exchange.

```
mptcp_do_segment()
   /* If MPTCP not yet enabled */
  if (mptcp_state == 0)
    process_tcp_options(tcphdr)
    /* MP_CAPABLE not on SYN */
    if ((opt_flags & MP_CAP) == 0))
      mptcp_ended(tcpcb)
      tcpcb->tp_fb = tcp_def_fb
      tcpcb->function_block->tcp_def_do_segment()
      return
```

given in Listing 3. The server also tests for `MP_CAPABLE` in `syncache_add()`, the *syncache*[7] [163] routine for servers to handle incoming `SYN` packets. The `syncache` structure (resembling a pared-down TCP control block) maintains state for embryonic connections (sequence numbers, options) and is used to handle incoming connections before they become established. Syncache functions cannot currently be replaced using the modular TCP stack, so we must add a pointer `sc_mptcp` to the `syncache` structure. This approach is consistent with other extensions such as TCP Fast Open that need to store data during the connection handshake.

It is worth touching on stateless connection establishment, where end-hosts perform connection setup without allocating state variables. In the case that the syncache is exhausted, FreeBSD will fall back to stateless connection establishment using TCP SYN-cookies [164]. A system administrator may also configure the server to forgo the syncache and use SYN-cookies exclusively. SYN-cookies allow connection establishment state to be encoded within the TCP initial sequence numbers of the handshake. The server therefore does not need to maintain a syncache entry for embryonic connections, and the connection state is reconstructed and validated once the handshake is complete. SYN-cookies do not encode state for TCP options such as window scaling (or MPTCP), so state must be retained outside of the SYN-cookie to support stateless open.

The MPTCP specification includes support for stateless MPTCP connection establishment by echoing the keys in the third `MP_CAPABLE ACK`. More recently a modification to the MP_CAPABLE

[7] TCP SYN caching for servers to protect against SYN-flood attacks. Aside from using less memory than a `tcpcb`, only a limited pool of `syncache` structures can be allocated at a given time, preventing memory exhaustion.

handshake was proposed to make stateless establishment more robust [165]. As long as the MP_CAPABLE options are included during the handshake, all data-level state can be reconstructed. MPTCP SYN-cookies are not currently supported in the implementation, and the connection will negotiate as standard TCP if a stateless open is used.

The subflow case: Joining a subflow represents a unique problem since it differs from the traditional socket connection semantics, breaking assumptions used by the TCP stack when associating packets to connections. As previously mentioned sockets are created using the `socket()` call, which initialises a `socket` structure and attaches an `inpcb` and `tcpcb`. On the client side a call to `connect()` causes transmission of a `SYN` packet. The server instead calls `listen()` to set the socket and protocol to a listening state before calling `accept()` and waiting for a incoming `SYN` packet. When establishing subflows the client-side process is much the same. Connecting a subflow on the server side is somewhat more complicated. We now discuses in more depth how a *listening* socket creates a new connection and how the subflow join changes this process.

When addresses are bound a reference to the Internet PCB (`inpcb`) is placed into a global hash table. The key is based on the local address:port (server) or the local and foreign address:port for clients. This table is used to map incoming packets to their respective `socket` (via the `inpcb`).

Whenever a packet is received a lookup is performed on the `inpcb` hash list to retrieve the `incpb` and `socket` for the connection. In the case of a `SYN` packet the associated socket must have the `SO_ACCEPTCONN` flag set (i.e. accepting connections). If a socket is accepting connections the `syncache` is used to temporarily store transport state and the pending connection is queued on the listening socket for the handshake. Once complete a new socket and PCBs are created and returned to the application via the the earlier `accept()` call. The listen socket continues to accept new connections while the newly created socket carries out the actual communication. Incoming subflow connections change this process in several ways:

- Subflows must be joined to an already existing and connected socket, *not* the listen socket - i.e. subflows are queued on the established MPTCP socket rather than on the `LISTEN` socket. This means that the listen socket (as created by the application) has no involvement in the joining of subflows.

- Joins can be sent to ports or addresses not bound in the original connection, so a standard PCB hash table lookup may not return the MPTCP socket.

- The application does not `accept()` a subflow, rather the new socket is connected to the

existing MPTCP connection (since logically they are connecting to the MPTCP layer and not the application).

The biggest difference however is the way in which the socket is retrieved for MP_JOIN packets. As mentioned it is not possible to use the existing Internet PCB hash list lookup to find the socket associated with an incoming SYN. Trying to do so would result in creating a separate single-path TCP connection (if the join connects to the original listening address:port) or discarding the packet as an unsolicited SYN. The server must connect an incoming SYN + MP_JOIN to an existing MPTCP connection rather than a socket.

A global hash table of multipath PCBs is created that uses MPTCP session tokens as the key. The connection token, included with each packet during the mp_join exchange, is used to find the appropriate MPTCP PCB and socket. This way incoming joins can be associated with the correct MPTCP session (and socket) without requiring an explicit LISTEN socket bound for the destination IP:port of the packet.

A side effect of this is that all incoming SYNs must be checked for an MP_JOIN option *before* standard SYN processing takes place. By default in the FreeBSD stack options for incoming SYNs are parsed only *after* checking whether a socket is accepting connections. Rather than calling tcp_dooptions() to check for a MP_JOIN on all incoming SYNs, the existing option processing is used where a listening socket already exists, and an explicit check is used when an Internet PCB lookup fails to find an entry.

In the case where the SYN + MP_JOIN destination matches an existing listening socket, a valid inpcb and socket will be returned and the join option can be processed by the existing call to tcp_dooptions(). At this time the join may be processed and a session token-based lookup can be used to retrieve the socket of the MPTCP session. A SYN + MP_JOIN to an address that is not currently listening for connections will fail to locate a socket. In this case tcp_dooptions() is called explicitly prior to discarding the packet. In both cases the discovery of an MP_JOIN results in a new entry in the syncache table with the 5-tuple as a key.

Processing the final ACK + MP_JOIN follows a similar strategy. However since creating a syncache entry does not create a new socket[8], tcp_dooptions() and syncache_lookup() must be called explicitly for incoming ACKs that fail to produce a socket after lookup but contain an MP_JOIN and session token.

Unlike when performing the MP_CAPABLE handshake, we do not need to consider falling back

[8] Or an inpcb. Both are created at the end of the handshake.

to the default TCP stack. An MP_JOIN is either successful and connects using MPTCP or fails and all state is discarded. This means that on the client side it is possible to allocate all related MPTCP structures prior to initiating a join. Lastly we adopt the joining policy of UCL-MPTCP and iOS-MPTCP of only issuing joins from the client host (i.e. the host that initiated the connection). This may be to the first known address or additional addresses advertised by the server.

Path management: The current implementation contains basic mechanisms for joining subflows and subflow/connection termination. A more fully featured Path Manager is left for future versions. For hosts that have multiple addresses, an address can be manually specified for inclusion in MPTCP connections by setting a sysctl variable[9]. Once added, the address is available to all MPTCP connections on the host. Currently only the server-side will advertise additional addresses, and the ADD_ADDR is sent as soon as possible once the connection is established. Subflow joining behaviour is static, and only performed by the client. A client will attempt to send an MP_JOIN immediately on receiving addresses via the ADD_ADDR option. If the client is multi-homed, it will send an implicit MP_JOIN once the connection is established.

6.4 Sending Data

An application writes new data to a TCP socket using a syscall such as send() or sendfile(). In the case of send() this results in a call to a user-request routine which copies the data from userspace to kernel memory address space and appends the copy to the socket *send buffer*. When using sendfile() [166], memory is mapped directly from a disk or a shared-memory object to the send buffer, eliminating the need for a copy. This optimisation is known as *zero-copy* and is frequently used when transmitting a large amount of data from disk (e.g. Netflix use sendfile() to achieve high-throughput transmission of video content [167]).

Memory in the kernel is handled via *mbufs* - a memory management unit that can be linked together in a *chain* to represent a raw byte-buffer, or a more structured packet/record. In addition to the data itself[10], the mbuf header describes the type of data and the length. For instance if an mbuf chain represents a packet, the head mbuf of the chain may contain a pkthdr structure that holds information about the packet. Multiple packets can be linked together into a queue of records.

Once data is copied or mapped to the send buffer, the tcp_output() routine is called to add

[9] sysctl net.inet.tcp.mptcp.mp_addresses

[10] The mbuf may be too small to fit the required data. Aside from internally storing data an *mbuf cluster* can be use as external storage.

TCP headers and send the data. If required, `tcp_output()` will segment the data to conform to the *maximum segment size* (MSS) of the transmission interface. The TCP MSS is the maximum number of bytes that can be carried in a TCP segment (data payload plus options). This value does not include the TCP or IP header.

The transmission rate of a connection is of course primarily governed by the CC algorithm. A number of other algorithms such as *silly window avoidance* adjust the sending rate in more subtle ways. These largely remain as-is for the subflow context. Therefore in this section we focus on the broader issues caused by the introduction of the layered MPTCP architecture and the need to take a data stream as written by the application and distribute it amongst several subflows.

Global and per-subflow send buffers: In Section 3.1 we discussed pull and push scheduling. To support both approaches the implementation uses a global queue as well as per-subflow queues. In this arrangement, the scheduler selects a subflow and then copies the data to be sent to the send-queue of the selected subflow (while retaining a copy in the global queue).

This presented several challenges. In FreeBSD the TCP send buffer is *byte-oriented*. This means that data from the application is appended in-order to the send-buffer as a single mbuf chain. This is a raw byte-stream lacking protocol-level meta-data, and so the responsibility of assigning TCP sequence numbers and creating MSS-sized segments is given to the `tcp_output()` routine[11]. In an MPTCP connection data-level bytes can be striped across multiple subflows, thus consecutive bytes in a subflow send-buffer may not be contiguous at the data-level. This means that the subflow queue cannot be treated as a traditional, byte-oriented queue. Although bytes are always transmitted in-sequence at the subflow-level, consecutive packets in a single TCP stream may not represent consecutive bytes sent by the application[12].

The second challenge that having multiple send-queues introduces is that of how to propagate the data-level sequence numbers to the subflow level so that `tcp_output()` can build the DSS option. In standard TCP, `tcp_output()` is responsible for segmenting the byte stream and building the TCP header. As part of building the TCP header it must also determine TCP sequence numbers. As mentioned, sequence numbers are *not* attached to the data in the send buffer but are assigned for each segment based on the `snd_nxt` field of the TCP control block. A pointer to the first byte of each segment is set using an offset from the start of the send buffer to the current sequence number to be transmitted. The segment to be transmitted is copied from this offset when assembling a packet. The

[11] If TCP Segmentation Offload is enabled, `tcp_output()` simply passes a *send-window sized* TCP segment to the IP layer.

[12] A single packet never contains discontiguous data, though.

Listing 4 Using `m_pkthdr` fields to store data-level mappings.

```
/* Access DS mapping values */
#define MP_DSN(mb) (mb)->m_pkthdr.PH_per.sixtyfour[0]
#define MP_SSN(mb) (mb)->m_pkthdr.PH_loc.thirtytwo[0]
#define MP_MAPLEN(mb) (mb)->m_pkthdr.PH_loc.thirtytwo[1]
```

stack does not keep any record of individual transmitted segments, so the segment and headers are rebuilt during retransmission.

These issues are solved by adopting a *record-oriented* approach when writing data to a subflow send buffer. With a record-oriented approach, multiple mbuf chains are written to the send buffer, with each chain being a contiguous series of data level bytes. The subflow send buffer thus holds multiple records, each of which is a contiguous region of the data-level byte sequence. Meta-data is embedded in the mbuf header to help differentiate the records. The global buffer in our implementation remains byte-oriented, since it always represents the in-sequence byte stream as-written by the application.

Since each record written to a subflow send buffer contains an mbuf header, data-level mappings are embedded here. The `pkthdr` structure contains two 64-bit unions (`PH_per` and `PH_loc`[13]) to pass protocol-specific data within and between layers of the stack. These are used to attach mapping meta-data (data-sequence number, subflow sequence number and record length) to the mbuf. A flag is also defined to identify a `pkthdr` as an MPTCP mapping. The mapping fields and flag are shown in Listing 4. `MP_DSN` is set when the mbuf header is created by the packet scheduler and is constant until the header is freed. `MP_SSN` and `MP_MAPLEN` are written when the region is assigned to a subflow, whether as new data or as a retransmission. From this meta-data `tcp_output()` is able to construct the DSS option, even after the mapping is segmented or in the event of a subflow-level retransmit.

Using the spare space in the packet header prevents having to allocate memory for an additional map structure. However the space for meta-data in the mbuf packet header is limited. If further meta-data is required in the future it would be possible to attach an mbuf tag [168] to the packet header, or alternatively embed the meta-data into the first mbuf of the chain.

Packet Scheduling: The packet scheduler is responsible for determining which subflows are able to send data from the global send buffer, and the size of the data allocation. The scheduler can be run

[13] As defined, `PH_per` variables persist between layers of the network stack. `PH_loc` values are cleared between layers. In this case PH_per is used for the data-level and PH_loc for the subflow-level. MPTCP packet headers do not traverse into other layers of the stack.

in three instances:

1. The application writes new data to the global send queue
2. A subflow receives an ACK.
3. A data-level retransmission is triggered.

In the current design subflows do not pull directly from the global send queue but instead request that the scheduler run by registering a subflow event. If the subflow does not have any data to send (ACK == SND_MAX) then it must wait for the packet scheduler thread to run and allocate new data. It is possible that the packet scheduler may not allocate new data to the subflow that requested it to run.

Selecting a subflow is only one part of the scheduling process. The following important steps are also performed:

- Copy the data to a new mbuf chain.
- Attach meta-data to the mbuf packet header.
- Append the mbuf chain to the selected subflow.

The m_copym() function is used when copying data from the global send buffer. This produces a new mbuf pointer and increments the reference count of the data pointed to by to original mbuf chain. The use of reference counts means that the sent data is only freed once it has been acknowledged at the subflow and data levels. For instance a mapping sent on two subflows can be acknowledged at the data-level and freed from the global send buffer while a failed subflow continues to try to retransmit that data.

Processing Data-ACKs and Retransmissions: Subflow-level ACKs are processed as in standard TCP. Data-level acknowledgements (D-ACKs) are are also handled at the subflow-level by tcp_dooptions(). As subflows are able to access the mpcb it is possible to check the incoming D-ACK against ds_snd_una and adjust accordingly if the incoming D-ACK progresses the left edge of the transmission window. Subflows do not have access to the global send buffer so freeing of these bytes occurs at the data-level when the scheduler is run. At this time the DSN of the first mapping in the transmitted queue is compared with the current ds_snd_una. If ds_snd_una has progressed passed the mapping then the difference is dropped from the send buffer.

As described in Section 2.1.2 TCP retransmits in response to duplicate ACKs or a retransmission timeout (RTO). This is unchanged at the TCP-level. Duplicate ACKs may be the result of short-term

congestion or a single packet loss due to transmission errors, so we currently do not trigger data-level retransmission for cases of subflow-level fast retransmit. Data-level retransmits occur when a subflow transitions to congestion recovery. Mappings that require retransmission are marked as such, and the scheduler will send these before sending new data.

Subflows will therefore continue to attempt retransmission until the maximum retransmit count is met. Our default behaviour is to terminate a subflow after 3 RTOs. On occasions where a subflow recovers and retransmits locally, the receiver will acknowledge the data at the subflow level and discard the duplicate segments.

6.5 Receiving Data

MPTCP adds a data-level sequence space above the sequence space used in standard TCP. This allows segments received on multiple subflows to be aggregated and ordered before delivery to the application. The processing of received segments is therefore split between subflow-layer and MPTCP-layer.

The task of each subflow at the receiver is to ensure complete and in-order delivery of their data-level mappings to the MPTCP layer. Once segments are in-order at the subflow level an event is queued that invokes the MPTCP layer to complete reception in another thread. The MPTCP layer then aggregates segments from the subflows and puts them into data-sequence order for delivery to the application. Any data-level signals such as the `data-FIN` are interpreted at the MPTCP-layer.

Arriving packets are at first processed as standard TCP segments. Though it is preferable to execute data-level procedures in the MPTCP event-processing task thread (most notably segment aggregation) in some cases it make more sense to handle these in the subflow thread context. This is done when deferring to an MPTCP task thread would be inefficient. For example it is better to update the data-level receive next `ds_rcv_nxt` immediately so that a `data-ACK` can be piggybacked on any TCP-level `ACK`.

An abbreviated call graph showing the path from receiving a packet and sending an acknowledgement to appending to the global receive buffer is shown in Figure 6.5. On the left is the software interrupt thread triggered by packet reception, on the right is the MPTCP task handler run on the `taskqueue_swi`. The following sections discuss parts of this process in more detail.

Processing options: The subflow must first parse and validate MPTCP options in the TCP header. Option processing is handled by the `tcp_dooptions()` function, which has been extended to identify MPTCP options. Each time a segment is received it is passed to `tcp_dooptions()` to be parsed along with a pointer to a `tcpopt` structure to store the result. We have extended `tcpopt` to

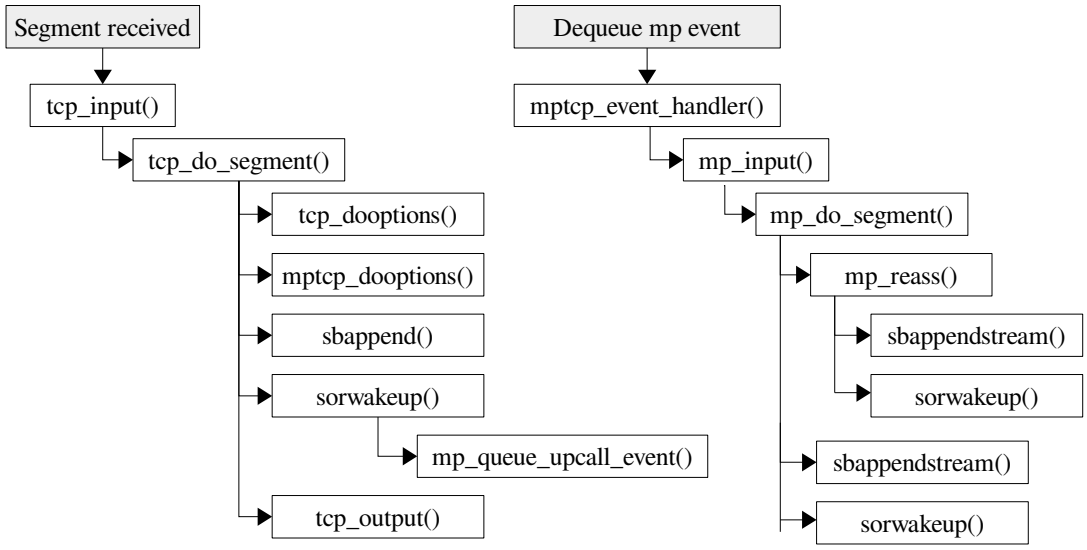


Figure 6.5: Receiving a segment in MPTCP. Processing is divided between two SWI threads - when the subflow receives a packet and for MPTCP-layer aggregation.

support MPTCP options.

MPTCP options can be long, with several containing fields of 64-bits or more. Storing MPTCP option values would increase the structure size significantly. As such only a pointer `*to_mptcp` to the location of the option in the TCP header is added. A flag `TOF_MPTCP` is also defined that is set when an MPTCP option is identified. If the flag is set on returning from `tcp_dooptions()` then `mptcp_dooptions()` is called, the results of which can be inserted into the MPTCP-specific `t_mpconn` structure if necessary. Pseudocode for option processing is shown in Listing 5. Initially the option is detected and a pointer is stored in `tcp_dooptions()`. Then `mptcp_dooptions()` casts a partial header from the pointer to identify the type of MPTCP option. Finally the option handler e.g. `mptcp_do_mpcap_option()` casts the actual option and processes, optionally storing a result in `t_mpconn`.

Options that are not handled immediately in the subflow thread are passed to the MPTCP-layer. The size of the options however prevents values from being written directly the mbuf header. Instead a pointer to the option is stored, which can be cast into an option when required.

Data-level aggregation: For standard TCP the FreeBSD stack maintains a reassembly queue and receive buffer for incoming segments. If a segment arrives that is not the next expected segment, it is placed onto the reassembly queue. When the expected sequence arrives it (and any in-order segments held in the reassembly queue), is appended to the socket receive buffer by `sbappendstream()`.

Listing 5 Identifying and processing MPTCP options.

```
tcp_do_segment()
  tcp_dooptions(*tcphdr, *tcpopt);
  if TOF_MPTCP flag is set
    mptcp_dooptions(tp->t_mpconn, tcpopt->to_mptcp)

/* Handling the MPTCP options seperately */
mptcp_dooptions(storage, ptr_to_option)
  option = (mp_opt_header*) ptr_to_option
   switch (option->option_type)
    case: MP_CAPABLE
      mptcp_do_mpcap_option(storage, ptr_to_option)
    case: MP_DSS
      mptcp_do_dss_option(storage, ptr_to_option)
    /* ... */
    break;
  return;
```

The waiting application (usually a `read()` syscall thread that is sleeping) is notified by calling `sorwakeup()`. If a segment arrives that is in-order and the reassembly list is empty, it is appended to the receive buffer immediately and the application is notified.

Figure 6.6 shows the receive structure for MPTCP. Both the reassembly queue and receive buffer are retained at the subflow level. Use of the reassembly queue is unchanged. Since bytes contiguous at the subflow-level may not be so at the data-level, the receive buffer stores multiple `mbuf` records rather than a single byte-stream, with each record representing a contiguous sequence of bytes at the data level. This means that coalescing is disabled and segments are appended using `sbappend()`, which retains the `pkthdr` of the head `mbuf`. The call to `sorwakeup()` now results in an up-call (`subflow_so_upcall()`) that enqueues a *subflow upcall event* task for the MPTCP layer to process.

At the the MPTCP-layer a global reassembly queue and receive buffer have been added. The first step of data-level reassembly involves iterating each subflow receive buffer and moving segments into the reassembly queue in data-sequence order. Here the pointer to the `DSS` option in the `mbuf` header is used to extract `DSNs`. After all segments are transferred, in-order segments are appended

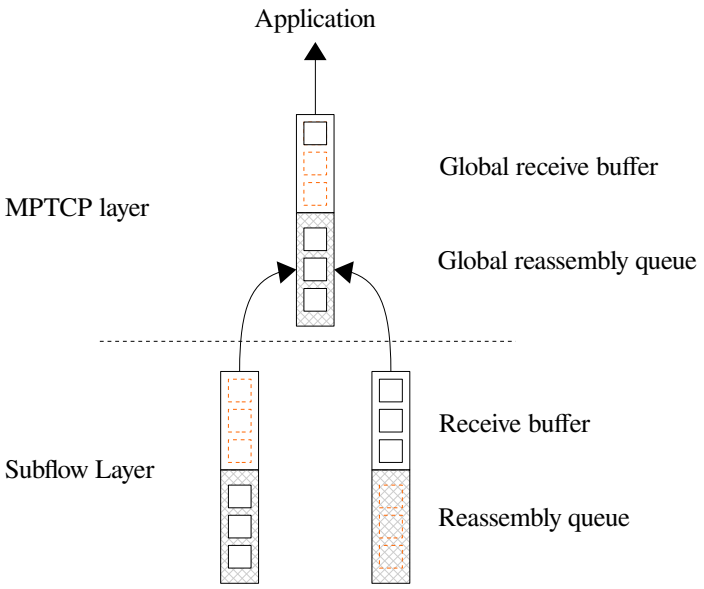


Figure 6.6: Subflow and MPTCP-layer receive structures.

to the global receive buffer and dropped from the reassembly queue before a wakeup is issued for the receiving application.

Scheduling data-ACKs: In the FreeBSD TCP stack an `ACK` is only scheduled after in-order data is appended to the receive buffer. Following this model for MPTCP would mean that a `data-ACK` would only be sent after the data was reassembled at the data-level and written to the global receive buffer. A `data-ACK` would then be explicitly sent by forcing output on a subflow. Earlier versions of the implementation followed this model.

A drawback of this approach is that that the `data-ACK` is delayed when it ideally should be piggybacked with subflow-level `ACK`. `Data-ACKs` are now scheduled when a mapping is received that progresses the receive window (i.e. is the next expected at the data level). The consequence of this is that the acknowledged data-sequence number and received (written to the buffer) sequence must be tracked separately. Therefore receive next is also tracked at the MPTCP-level according to the receive buffer occupancy. In the case where a received segment fills a gap in the data-level reassembly queue a `data-ACK` is explicitly scheduled from the data-level. Scheduling a `data-ACK` does not always mean it is transmitted if *delayed acknowledgment*[14] is enabled.

[14] A mechanism used by TCP to reduce the number of acknowledgements sent by a receiver by delaying `ACK` responses and acknowledging more data per ACK packet.

6.6 Closing a Connection

MPTCP connections must be closed at the subflow and MPTCP level. The connection close sequence is triggered when an application calls `close()` or `shutdown()` to begin disconnecting a socket. Like TCP, MPTCP transitions through a series of closing states. This requires sending a `data-FIN` and receiving a `data-ACK` to complete. As with TCP, all data pending transmission must be sent before the `data-FIN`. The `data-FIN` is carried in a `DSS` option and therefore subflows cannot be closed before the MPTCP connection closes.

Just as the application is responsible for closing the socket, the MPTCP layer is responsible for closing the subflows. The subflow shutdown process is identical to TCP shutdown, therefore no changes are required - the MPTCP layer can simply call the user request routine `tcp_usr_close()` on each subflow socket to initiate shutdown. Subflow sockets free themselves once the closing handshake is complete, meaning the MPTCP structures can be freed once a close has been initiated on all subflows.

Occasionally there is a need to close a subflow in the middle of an ongoing connection. This may be done gracefully via a call to `tcp_usr_close()`, which, once all sent data is acknowledged, will trigger a standard TCP closing handshake. Alternatively the connection can be aborted by sending a `RST`. A subflow may independently send a reset in response to standard TCP events, such as a connection timeout. MPTCP layer may also explicitly abort one or more subflows by calling the `tcp_usr_abort()` routine of that subflow.

A `RST` received on a subflow applies only to that subflow and does not close the MPTCP connection. On receiving a `RST` on a subflow, the MPTCP layer will immediately remove the subflow from the connection and reschedule any unacknowledged data onto an alternate subflow. If no other subflows are available, the connection enters a MPTCP timeout period. If no new connections become available during this period the connection will terminate.

The `MP_FASTCLOSE` option is defined in the specification for MPTCP connection-level aborts. This is not currently supported and the aborting host will simply issue `RSTs` on all of the subflows before closing the socket.

6.7 Conclusion

This chapter presents an overview of a Multipath TCP implementation design for FreeBSD. We introduced the major design elements and discussed the reasons for the approach taken. The overarching

emphasis of the implementation is on ease of maintenance and accessibility for researchers. MPTCP has been deployed for some specific use-cases but there is not broad awareness of the protocol. . Developing an implementation that is easy to comprehend and keep up-to-date can help to encourage further research and development input from the community.

Underpinning this design is the use of the modular TCP stack framework, which allows a significant portion of code to be kept separate from the existing TCP stack. Providing modular scheduling and congestion control makes the stack more accessible for experimentation. Although our design focus is on enabling experimentation, performance is not neglected and where possible efforts have been made to ensure that enabling MPTCP does not encumber the host system.

Chapter 7

Experimental Evaluation

In this chapter we evaluate the functionality of the FreeBSD MPTCP implementation. In Section 7.1 we describe our testbed and the rational behind the experiments. In Section 7.2 we evaluate the implementation across a set of scenarios designed to engage the critical functions of the implementation and demonstrate the pluggable frameworks in use. Lastly we discuss preliminary performance tests used to identify bottlenecks in the architecture.

7.1 Experimental Design

For several years the MPTCP protocol itself has been studied and evaluated across a range of environments - small and large-scale testbeds, in data centres or simply across the Internet. These studies have demonstrated the utility of MPTCP and helped identify areas of improvement for the protocol and provided guidance for implementors. Our goals here are not to evaluate the effectiveness of MPTCP or optimise implementation details. Nor do we analyse the performance of scheduling or congestion control algorithms. Rather we aim to validate the basic functionality of our implementation. Put simply, our central question is: does the implementation work? We attempt to answer by addressing three aspects:

- *Conformance:* Does the implementation cover the basics of the MPTCP specification?
- *Modularity:* Can we load scheduling and congestion control modules to change the dynamics of a connection?
- *Performance:* Does the design create any obvious bottlenecks?

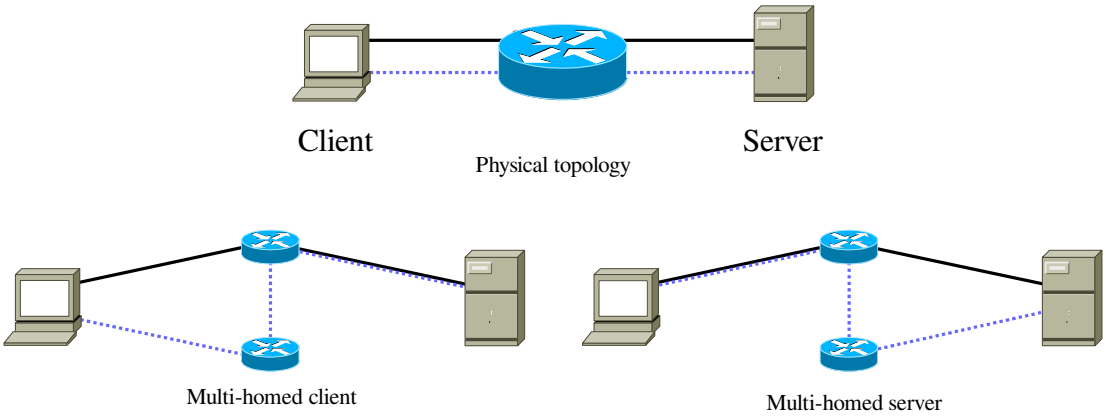


Figure 7.1: Physical and logical representations of the testbed.

Our experiments are thus are small scale and performed in controlled environments. The topologies are easily reproducible and it is expected that early adopters will perform similar tests to gain familiarity with the implementation and common multi-path scenarios.

7.1.1 Testbed Topology

A virtual testbed environment is used for the evaluation. A multipath configuration is used to validate conformance and modularity aspects, as well as the basic performance on low-bandwidth links. The same testbed is used with a single-path, single-router topology to measure performance at higher-bandwidths.

The router, test hosts and network are emulated using VirtualBox [169]. The logical testbed topology is shown in Figure 7.1. It consists of client and server connected via four links, each in its own subnet. Either host may be configured to use a single or multiple addresses. Paths may be configured with or without a shared bottleneck. Network emulation on the router allows path characteristics such as loss, bandwidth and delay to be adjusted.

The host machine is FreeBSD-based, configured with an Intel Xeon E3-1270 3.40GHz processor and 32GB of RAM.The client and server VMs were configured with 512MB of RAM and four emulated Intel Gigabit network adapters. The two router VMs were FreeBSD-based and configured with 128MB of RAM and four emulated Intel Gigabit network adapters. Though the hardware configuration is relatively modest, it is adequate for validating the basic performance and functional aspects of the implementation.

Tools: On the routers, static forwarding rules are added to the FreeBSD system routing table.

Objective	Note
I-D: MP_CAPABLE Flag A set to 1	Checksum enabled
I-D: MP_CAPABLE Flag B ignored	Silently ignore SYNs with the B flag set
I-D: ACK without MP_CAPABLE	Fail if option missing during handshake

Table 7.1: Example conformance tests.

Dummynet [170] is used to set the bandwidth, delay and loss rate for the end-to-end paths. *Iperf* [171] is used to transfer data. Packet traces are presented in *tshark* [172] format.

7.2 Evaluation

In this section we evaluate the functionality of the implementation using a small set of connection scenarios. First we show that the implementation conforms to basic connection establishment behaviours. We then show that the implementation meets the basic requirements for multiple subflows and fault-tolerance, and can function at speeds useful for network experimentation.

7.2.1 Basic Conformance

We first assess whether the implementation meets basic requirements of the protocol. In an IETF Internet Draft [173] Coene details a suite of tests that can be used to evaluate basic compliance of an MPTCP implementation. The tests range from extremely basic (e.g. does the implementation send a `SYN`) to validating an implementation's response to incorrect signaling. Some example behaviours are summarised in Table 7.1.

We look at *ACK without MP_CAPABLE* as an example. Line 3 of the packet sequence in Listing 6 shows the `MP_CAPABLE` option missing from the final `ACK` of a MPTCP handshake. On receiving this packet the server infers that MPTCP should not be used on this path, falls back to regular TCP, and on Line 4 transmits the first data without a `DSS` option. On receiving a data packet without a `DSS` option the client falls back to TCP operation. Our current implementation does not support most cases of fall back, however, for example simply closing the connection in the event that a `DSS` is missing mid-connection.

A successful negotiation of an MPTCP connection and the addition of a subflow is shown in Listing 7. Lines 3-4 show the initial MPTCP handshake. Immediately following this on Line 5 is an `MP_JOIN+SYN` sent from `172.16.4.2` to the server at `172.16.1.2`. The `MP_JOIN` handshake

Listing 6 Fall back to TCP if ACK+MP_CAPABLE is missing

```
1 172.16.3.2 -> 172.16.1.2 MPTCP 86 48507 -> 5001 [SYN] Seq=0 Win=65535
    Len=0 MSS=1460 WS=64 TSval=357172 TSecr=0
2 172.16.1.2 -> 172.16.3.2 MPTCP 86 5001 -> 48507 [SYN, ACK] Seq=0 Ack=1
    Win=65535 Len=0 MSS=1460 WS=64 TSval=2729538658 TSecr=357172
3 172.16.3.2 -> 172.16.1.2 TCP 66 48507 -> 5001 [ACK] Seq=1 Ack=1
    Win=65664 Len=0 TSval=357216 TSecr=2729538658
4 172.16.3.2 -> 172.16.1.2 TCP 90 48507 -> 5001 [PSH, ACK] Seq=1 Ack=1
    Win=65664 Len=24 TSval=357223 TSecr=2729538658
5 172.16.3.2 -> 172.16.1.2 TCP 1514 48507 -> 5001 [ACK] Seq=25 Ack=1
    Win=65664 Len=1448 TSval=357223 TSecr=2729538658
```

Feature to disable	Description
MPTCP checksum	Disabled via `sysctl`
TCP segment offload (TSO)	Disabled via `ethtool`
Generic send offload (GSO)	Disabled via `ethtool`

Table 7.2: Linux-endpoint settings required for interoperability.

completes when the server sends a TCP ACK (Line 9) that acknowledges the MP_JOIN+ACK (Line 8). Following this data can be send on the new subflow (Lines 13-14).

The implementation supports single-subflow MPTCP connections with the UCL Linux implementation. Multi-subflow connections are not possible due to current limitations in the path manager implementation. To achieve interoperability, several configuration changes must applied to the Linux endpoint (summarised in Table 7.2):

- The DSS checksum field is not currently support and must be disabled on the Linux host to prevent a connection reset.
- Features that aggregate multiple segments into larger-than-MSS segments (TSO, GSO), are disabled as they may trigger bugs in the FreeBSD implementation.

Despite present limitations, the ability to perform the MPTCP-handshake and transfer data using MPTCP-DSS signaling with the UCL Linux implementation is a significant step towards the stated goal of interoperability. It can be reasonably expected that, as development work continues, ‘out-of-

Listing 7 Successfully establish a MPTCP connection and additional subflow.

```
1  172.16.3.2 -> 172.16.1.2 MPTCP 86 61767 -> 22 [SYN] Seq=0 Win=65535
   Len=0 MSS=1460 WS=64 TSval=271235 TSecr=0
2  172.16.1.2 -> 172.16.3.2 MPTCP 86 22 -> 61767 [SYN, ACK] Seq=0 Ack=1
   Win=65535 Len=0 MSS=1460 WS=64 TSval=3945774304 TSecr=271235
3  172.16.3.2 -> 172.16.1.2 MPTCP 86 61767 -> 22 [ACK] Seq=1 Ack=1
   Win=65664 Len=0 TSval=271278 TSecr=3945774304
4  172.16.4.2 -> 172.16.1.2 MPTCP 86 27682 -> 22 [SYN] Seq=0 Win=65535
   Len=0 MSS=1460 WS=64 TSval=271283 TSecr=0
5  172.16.1.2 -> 172.16.4.2 MPTCP 90 22 -> 27682 [SYN, ACK] Seq=0 Ack=1
   Win=65535 Len=0 MSS=1460 WS=64 TSval=415670129 TSecr=271283
6  172.16.3.2 -> 172.16.1.2 MPTCP 124 61767 -> 22 [PSH, ACK] Seq=1 Ack=1
   Win=65664 Len=38 TSval=271286 TSecr=3945774304
7  172.16.1.2 -> 172.16.3.2 MPTCP 124 22 -> 61767 [PSH, ACK] Seq=1
   Ack=39 Win=65536 Len=38 TSval=3945774356 TSecr=271286
8  172.16.4.2 -> 172.16.1.2 MPTCP 90 27682 -> 22 [ACK] Seq=1 Ack=1
   Win=65664 Len=0 TSval=271329 TSecr=415670129
9  172.16.1.2 -> 172.16.4.2 TCP 66 [TCP Window Update] 22 -> 27682 [ACK]
   Seq=1 Ack=1 Win=65664 Len=0 TSval=415670172 TSecr=271329
10  172.16.3.2 -> 172.16.1.2 MPTCP 1422 61767 -> 22 [PSH, ACK] Seq=39
   Ack=39 Win=65600 Len=1336 TSval=271366 TSecr=3945774356
11  172.16.1.2 -> 172.16.3.2 MPTCP 1126 22 -> 61767 [PSH, ACK] Seq=39
   Ack=1375 Win=64192 Len=1040 TSval=3945774436 TSecr=271366
12  172.16.3.2 -> 172.16.1.2 MPTCP 134 61767 -> 22 [PSH, ACK] Seq=1375
   Ack=1079 Win=65536 Len=48 TSval=271414 TSecr=3945774436
13 172.16.1.2 -> 172.16.4.2 MPTCP 450 22 -> 27682 [PSH, ACK] Seq=1 Ack=1
   Win=65664 Len=364 TSval=415670261 TSecr=271329
14  172.16.4.2 -> 172.16.1.2 MPTCP 130 27682 -> 22 [PSH, ACK] Seq=1
   Ack=365 Win=65664 Len=44 TSval=271463 TSecr=415670261
```

the-box' interoperability is achievable.

7.2.2 Creating Multiple Subflows

A core requirement of our implementation is data transfer across one or more subflows. We use Iperf to transfer data in the following scenarios:

- Both single-homed. This is equivalent to a standard TCP connection however uses MPTCP signaling and the data-level sequence space.
- Multi-homed client. This is the expected common case for MPTCP connections. A multi-homed client connects to a single-homed server and joins an additional subflow.
- Multi-homed server. A server will not initiate a join in our implementation. This test verifies that the `ADD_ADDR` option is sent and used by the client to join an additional subflow to the connection.

A bandwidth rate-limit of 5Mbps and RTT of 20ms is applied using the router. The router queue length is set to 50 slots, providing a maximum queuing delay of 120ms with 1500 byte packets. Maximum send and receive buffers are set to 128KB for TCP and for individual subflows, allowing the flows to achieve 5Mbps. We also use this path configuration to create a shared bottleneck. The router queue is able to absorb any packet-bursts and does not cause packet loss. The client host always initiates the joining of additional subflows.

Figure 7.2 compares the average throughput obtained for a single subflow against standard TCP when transferring a 20MB file. The throughput is similar, though the MPTCP subflow is consistently lower. There are several known sources of inefficiency that may account for part of the loss in throughput. Firstly, the `DSS` reduces the TCP maximum payload size by 20-bytes for each packet (e.g. from 1448 to 1428).

A second source of inefficiency comes from the creation of MSS-sized maps on each write to the MPTCP-level socket buffer. Bulk data is often written into the socket in multiples of 8KB (e.g. 8KB, 16KB, 32KB). The MSS of an MPTCP segment (1428 bytes in our testbed) does not evenly divide into this, resulting in occasional 'odd' mappings that do not fill an entire MSS. As a result, a small percentage of segments are transmitted that do not fill the available MSS. For example, in trials transferring 20MB, the mean total of packets transferred was 22,784, and on average 193 packets were less than the MSS (predominantly being rounded down to 1124 bytes).

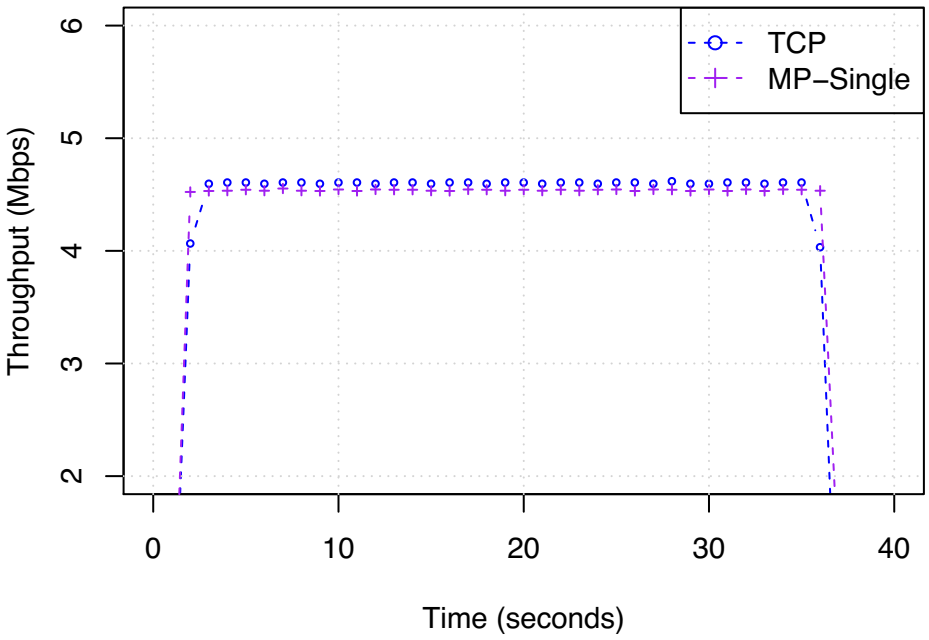


Figure 7.2: Throughput for MPTCP with a single subflow compared with TCP over the same path.

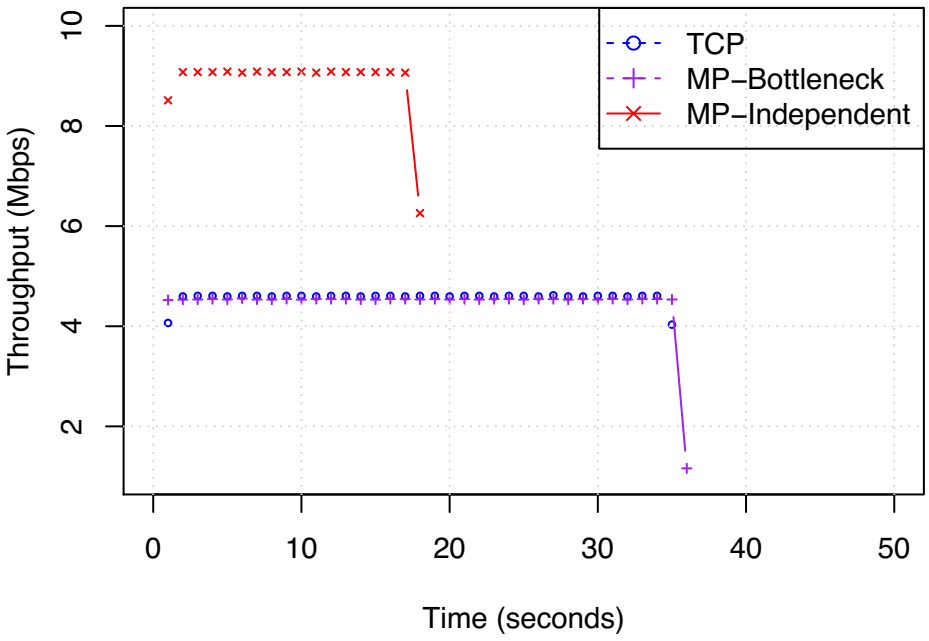
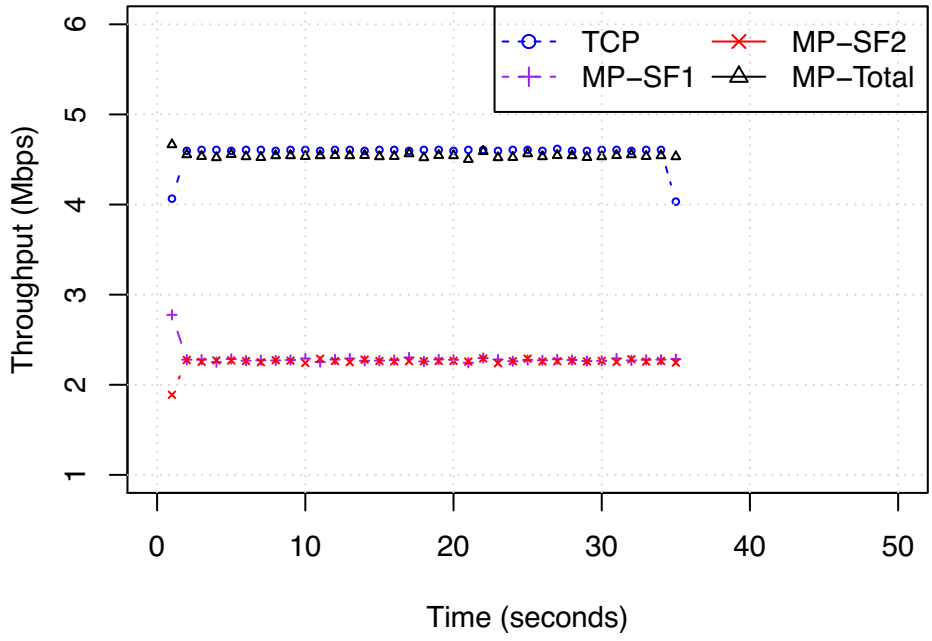
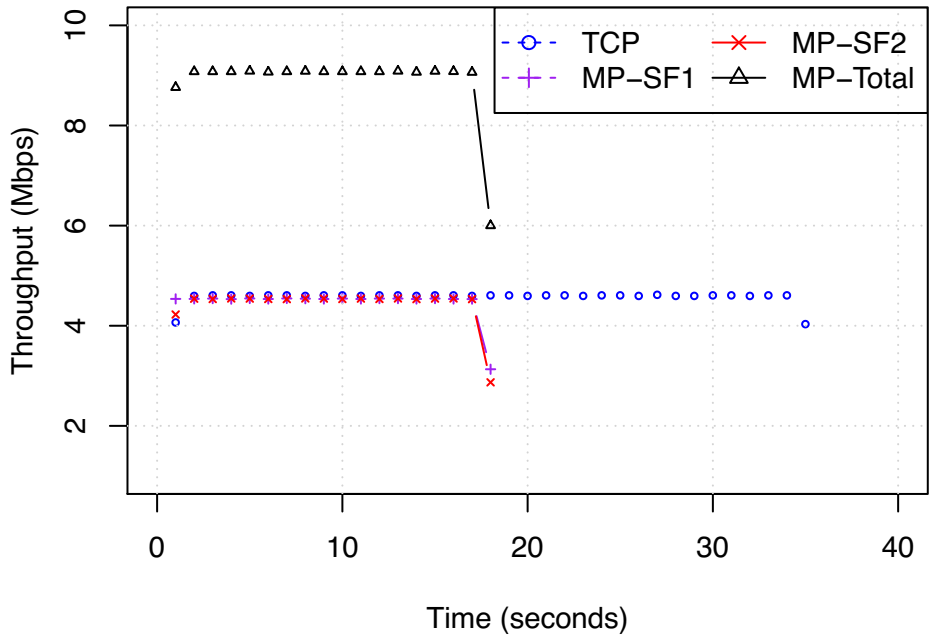


Figure 7.3: Comparing multiple subflows to TCP, with and without shared bottleneck.



(a) Shared bottleneck



(b) No shared bottleneck

Figure 7.4: Comparing per-subflow and total throughput for MPTCP flows with TCP.

Figure 7.3 compares the average throughput obtained when the client is multi-homed, with and without a shared bottleneck link. The *simplerr* RR scheduler is used to stripe data between the subflows in MSS-sized segments.

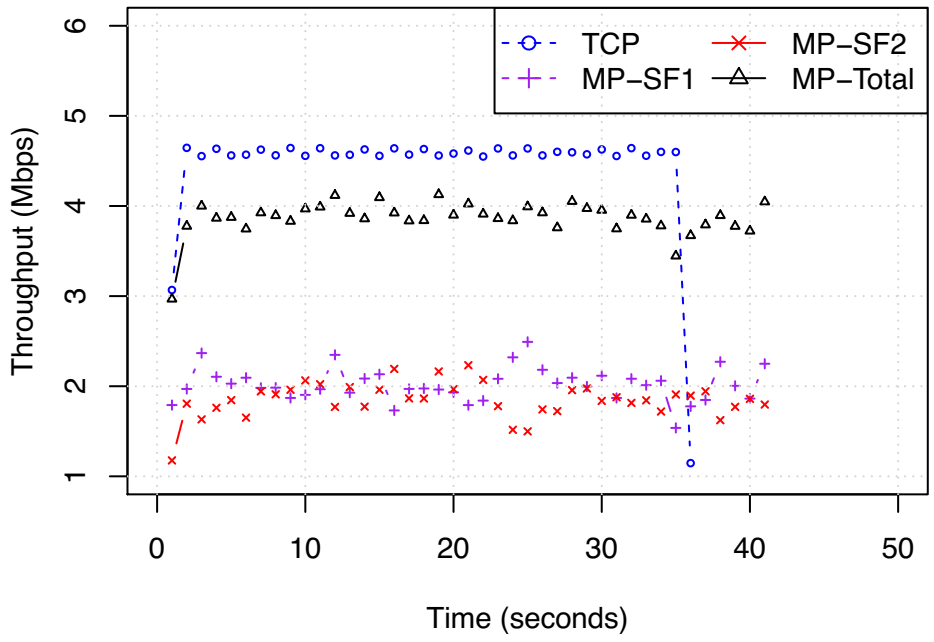
With a shared bottleneck, throughput is again marginally less than that of standard TCP. The previously discussed inefficiencies are present here, but another contributing factor is the interaction of the subflows at the bottleneck, whose independent congestion control mechanisms compete for bottleneck bandwidth. Given two independent 5Mbps paths, the connection is able to achieve approximately double that of the single-path TCP connection and shared-bottleneck MPTCP.

The per-subflow and combined throughputs for each instance are shown in Figure 7.4. We can see that, in the shared-bottleneck case, SF1 has a higher throughput than SF2 in the early stages of the connection. This is due to having several RTTs 'head-start' over SF2, and being allocated a full send-buffer as the connection is opened[1]. Once SF2 is joined to the connection this difference in throughput is reduced as the RR scheduler begins to distribute new data between the subflows equally. After several seconds, throughput is evenly split between the two subflows and remains this way until the connection is terminated. Despite the initial imbalance between the two subflows, the overall throughput consistently approaches the 5Mbps rate limit.

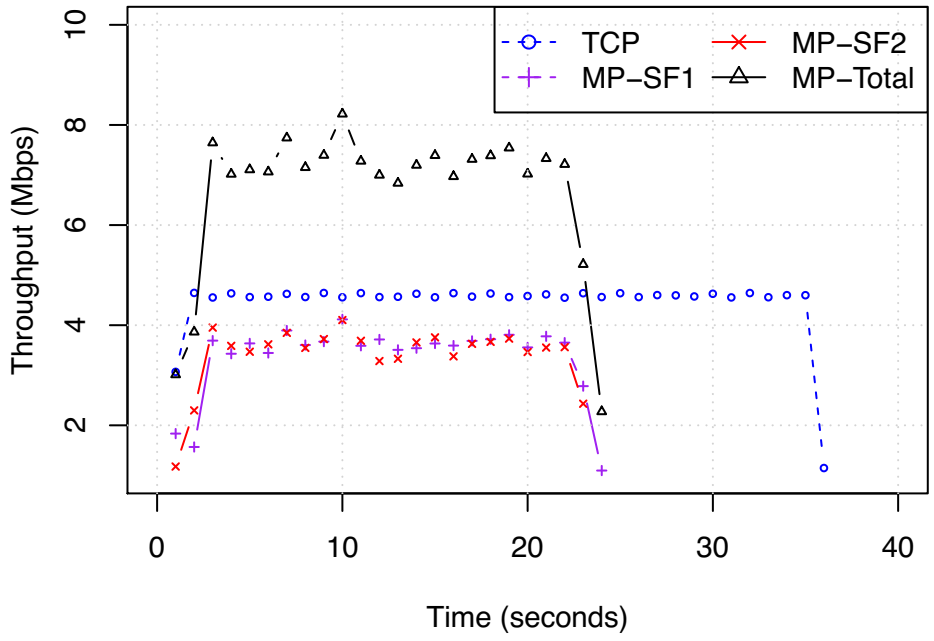
The introduction of an additional subflow clearly presents a change in the dynamic behaviour as compared to TCP. Having shown results with sufficiently sized bottleneck buffers, we offer this cautionary illustration of how undersized bottleneck buffers can significantly distort performance results. Figure 7.5 once more compares the average throughput obtained for multi-subflow connections to TCP. Here the router queue is reduced to 15 slots. This is slightly *undersized* for a single 5Mbps TCP flow and allows the congestion window to grow to the point of inducing packet loss in the router queue. The presence of packet loss is noticeable through the increased variation in achieved throughput. The throughput variance is much higher in the multi-subflow cases, and the average throughput is less than achieved with larger buffers. The reduction in throughput is caused solely by an increase in packet loss. The number of retransmission is given in Table 7.3, and we see that the multi-subflow cases see significantly higher packet-loss than standard-TCP.

In the bottleneck case, the increase in packet-loss is due to the combined input of the two sending subflows exceeding the queuing capacity (recall that the bottleneck queue is sized to match a single 5Mbps TCP flow). It was also observed that packet losses would occur on both subflows simultane-

[1]Recall that SF2 does not exist at connection establishment. It must wait until the application writes more data to the socket to be filled.



(a) Shared bottleneck, undersized router queue



(b) No shared bottleneck, undersized router queue

Figure 7.5: Comparing per-subflow and total throughput for MPTCP flows with TCP where the router queue is undersized.

TCP	MPTCP-Bottleneck	MPTCP-Independent
73	223	194

Table 7.3: Mean count of retransmissions (including fast retransmissions, RTOs) when transferring 20MB.

SF1 retransmits	SF2 retransmits
41	152

Table 7.4: Total-retransmits per-subflow, MPTCP-independent.

ously, due to the tight interleaving of packets by the scheduler and the synchronisation of transmits on application writes. As each subflow maintains a unique congestion control, two subflows continue to grow their transmission windows until the bottleneck queue fills and packet loss occurs.

In the independent-path case, overall packet-loss is reduced in comparison to the bottleneck case, since subflows are not contending for slots on the bottleneck queue. Given the independent paths, it might be expected that the total number of retransmissions would be comparable to TCP. This did not turn out to be the case, and looking at the subflows separately helps to explain the causes. The total number of retransmits for each subflow on an independent-path connection is given in Table 7.4. The number of retransmissions on SF1 are in-line with expectations, and we see that SF2 is the primary source of the additional retransmits. The retransmits on SF2 are the result of bursty behaviour in the initial period after joining the connection.

When the connection is first established, the application performs a large write to the send-buffer of SF1. It takes several RTTs before SF2 is established, and during this time writes from the application are less frequent and smaller. The under-fed SF2 transmits the available data, causing the queue to build on the router without causing loss. When a large write eventually arrives, the send-buffer of SF2 is filled, and the congestion window allows a burst of packets to be sent out back-to-back. This burst causes drop-tail loss at the already partially-filled router queue, and a number of packets must then be retransmitted. This bustiness is present only while the buffer is undersized. After several seconds the send buffer is filled, and the retransmission rate of the two subflows becomes comparable. In this case the losses on SF2 are the result of the combination of scheduling and congestion control algorithms, and the sending application. The congestion window does not evolve in the same way as a solitary, standard TCP-flow might have.

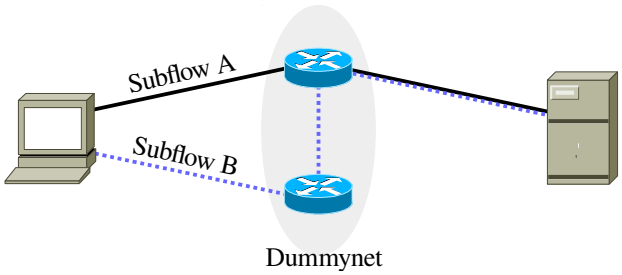


Figure 7.6: Testing fault tolerance. The client is multi-homed and Dummynet and PF provide middlebox emulation.

The underlying observation is that the introduction of additional subflows alters the dynamic behaviour of the connection such that assumptions based on standard TCP-flow behaviour do not hold. This is an important consideration when performing network experiments that involve multi-subflow scenarios.

7.2.3 Retransmissions

A crucial aspect of the implementation is how it reacts to problems along the network path. Common issues are packet loss or a link losing connectivity. Four causes of packet loss are:

- *Rate-control:* Congestion control algorithms probe the network path to maximise bandwidth use. A side-effect of this is packet loss when the size of the congestion window exceeds the path capacity.
- *Random packet loss:* Can be caused by link-level transmission errors. For example wireless links are more susceptible to transmission errors than wired links. Active queue management schemes like RED may also randomly discard packets to forestall congestion.
- *Congestion:* A link in the path does not have adequate capacity or buffer space to forward the aggregate input, causing loss.
- *Link failure:* A link may fail locally (e.g. a mobile device with transient connectivity) or along the path.

We introduce random packet loss and link failure to demonstrate subflow-level and data-level recovery functions, using the topology shown in Figure 7.6.

For MPTCP connections, most losses, such as those caused by TCP congestion control or random packet loss, are handled at the subflow-level by standard TCP mechanisms. Our tests in Section 7.2.2

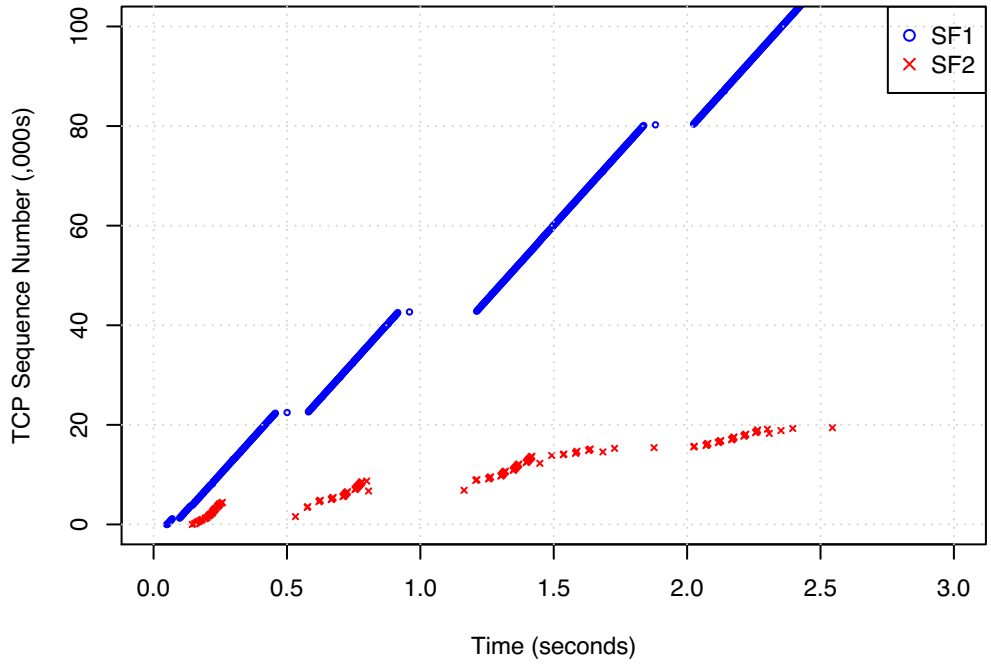


Figure 7.7: Random packet loss causing subflow-level retransmission.

confirm that subflow-level retransmission mechanisms (such as fast retransmit) work in response to path congestion, so here we look at random loss. To ensure that only random loss is a factor, we increase the BDP of both paths (RTT + queuing delay) to exceed the maximum `cwnd` for the subflows[2], preventing losses caused by rate-control. Random packet loss (0.05%) is added to the path of Subflow 2. Figure 7.7 compares the relative TCP-sequence numbers of each subflow. We see the effect of random loss on Subflow 2, which sends data at a much lower rate than Subflow 1. However, even though Subflow 1 does not experience loss, it is still influenced by the losses on Subflow 2, and the overall connection is blocked while Subflow 2 is retransmitting (as the naive scheduler continues to allocate segments to the poor-performing Subflow 2). Loss recovery in this case occurred completely at the subflow layer.

In our topology subflows can only follow a single path (i.e. they cannot be re-routed), so link failure causes a complete loss of that subflow. If a path fails, data that has yet to be acknowledged by `Data-ACK` must be retransmitted on an alternate subflow for the connection to progress. We bring down the path of Subflow 2 for a number of seconds to demonstrate both data-level retransmission

[2]In our case 64KB, as limited by the send and receive buffer sizes.

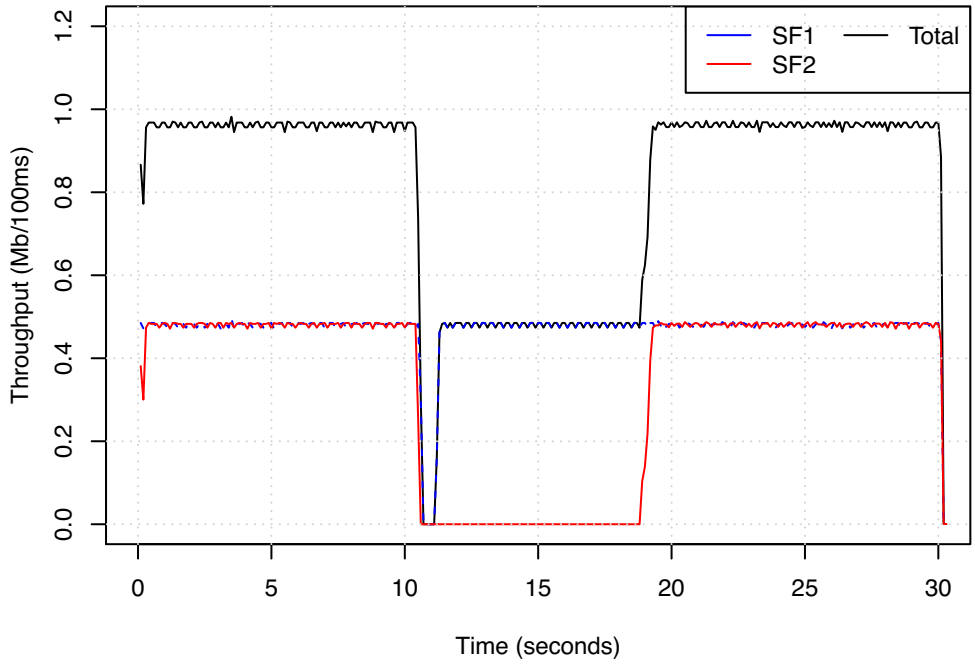


Figure 7.8: Per-subflow and combined throughput during path loss and data-level retransmission.

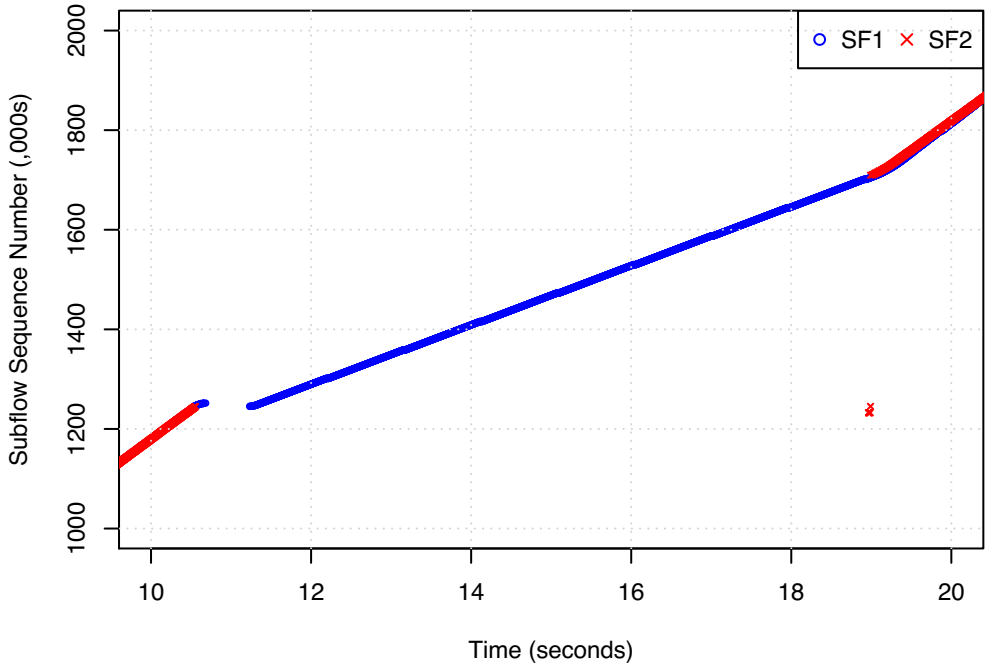


Figure 7.9: Failure on SF2 causes a data-level retransmit. On recovery SF2 is able to send new data.

Listing 8 The `simplerr` module definition.

```
struct sched_algo simplerr_sched_algo = {
   .name = "simplerr",
   .cb_init = simplerr_sched_init,
   .cb_destroy = simplerr_sched_destroy,
   .get_subflow = simplerr_get_subflow,
};
```

and the resumption of transmission when a path regains connectivity. In this example data-level transmission is triggered if a subflow experiences two RTOs[3]. The per-subflow and overall throughput is shown in Figure 7.8. The data-sequence numbers transmitted by each host before, during and after the connection break are shown in Figure 7.9.

In the initial phase of the connection the data-sequence is spread evenly between the subflows. At approximately 11 seconds the connection stalls as a result of the loss of Subflow 2. After two RTOs on Subflow 2 the data is re-scheduled on Subflow 1 and the connection resumes, now transmitting only on Subflow 1. Subflow 2 recovers at approximately 19 seconds, at which time the originally scheduled (but not delivered) data in the send buffer must be transmitted (seen as a small cluster in the lower right corner). Once subflow 2 has recovered, the scheduler is again able to distribute new data between the subflows.

7.2.4 Scheduling and Congestion Control

We have implemented basic modules for scheduling and congestion control using the modular framework. Though limited in practical terms the modules demonstrate the relative ease with which the dynamic characteristics of an MPTCP connection can be changed.

Scheduling: The implementation defaults to RR scheduling if an alternate module is not loaded. Our default scheduler, *`simplerr`*, has been written using the modular scheduling framework. The definition is given in Listing 8. The previous multi-subflow tests in this chapter have used the `simplerr` scheduler. As it's name suggests, `simplerr` is a naive scheduler that allocates data to a subflow output queue in MSS-sized segments. The scheduler is called whenever new data is written to the MPTCP socket by the application, when data is received at the MPTCP-level and when a sub-

[3]Multiple RTOs are a sign that a link has failed or is experiencing heavy congestion. It is however an approach and can lead to a period of window blocking, as other subflows may exhaust their send queues in the meantime.

flow receives a positive TCP-level ACK. As previous research (see Section 4) has shown, a naive RR approach is of limited practical utility, as it introduces head-of-line blocking and jitter. Simplerr is therefore suited only to demonstrate the operation of multiple subflows on an MPTCP connection, and to serve as a module template from which more advanced schedulers can be implemented.

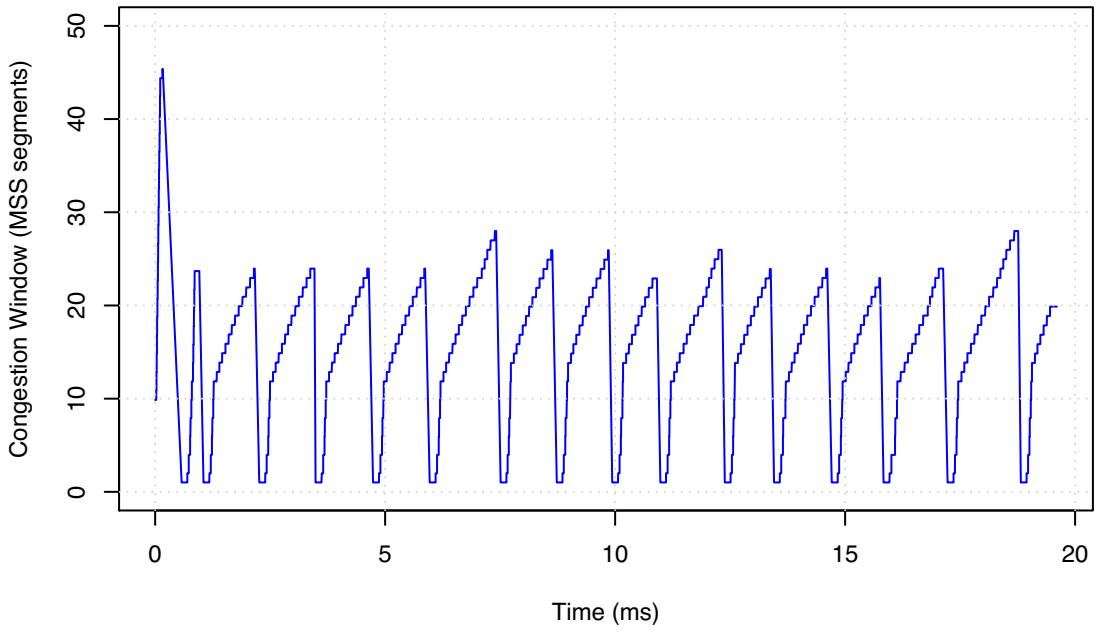
Congestion Control: The implementation defaults to uncoupled per-subflow congestion control, with the system-default TCP-CC applied to each individual subflow. The MPTCP-CC framework can be used to set CC for MPTCP connections. An MPTCP-CC module may specify the use of uncoupled or coupled CC algorithms. The mptcp_uncoupled CC module provides two sysctl variables [4] that are used to set an uncoupled algorithm for two subflows. A more realistic module might select a CC algorithm based on subflow metrics or information, such as the hardware interface type.

Different CC algorithms are unique in how they respond to network feedback, so to demonstrate the mptcp_uncoupled module we compare the cwnd profile of subflows using TCP New Reno and TCP Cubic. We use the same network configuration as per our previous multi-subflow tests, with and without a shared bottleneck[5]. Here the queue depth of the router is reduced to cause drop-tail packet loss and encourage more dynamic congestion window behaviour. Figure 7.10 shows the bottleneck case, while in Figure 7.11 the bottleneck is removed. In both cases we can easily identify the unique cwnd profiles of the algorithms - the linear-growth sawtooth of TCP New Reno and the curved-growth and plateaus of TCP Cubic. Comparing the two scenarios, we see that in the shared bottleneck case Subflow 1 is able to grow its cwnd significantly during slow-start, to more than 40 segments. This fills the queue and by the time Subflow 2 is able to transmit data, the slow-start phase sees limited growth before entering congestion avoidance. When the bottleneck is removed, we see that both subflows are able to grow cwnd to greater than 40 segments during slow-start.

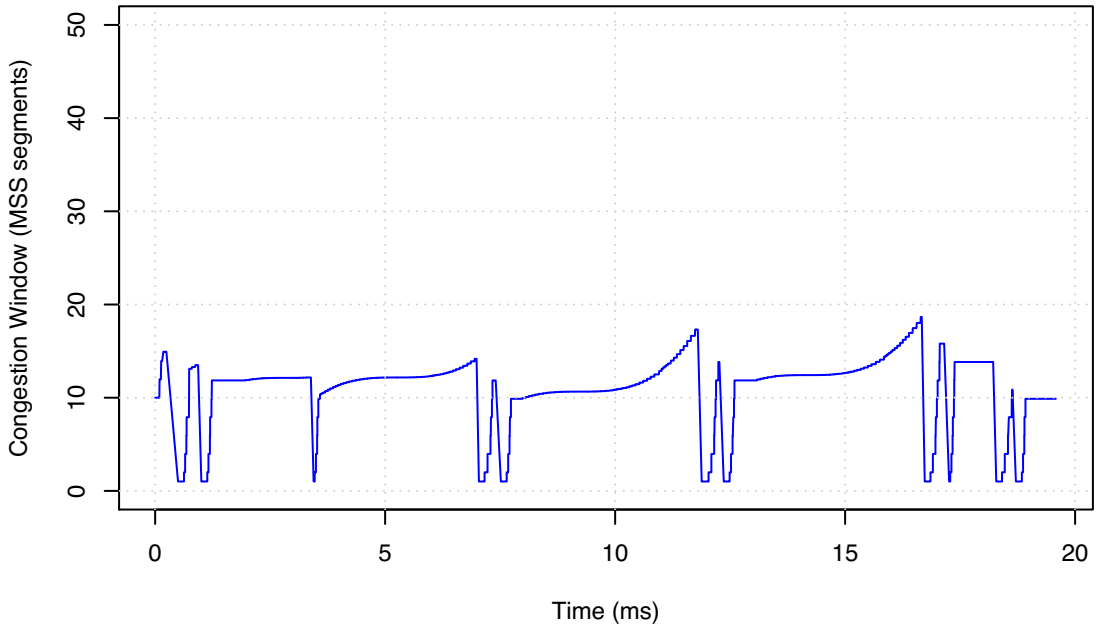
TCP New Reno and TCP Cubic are both loss-based algorithms. Delay-based approaches are suitable for use on paths with excessive buffering (e.g. cellular networks) and it is conceivable researchers may want to explore mixing delay-based and loss-based algorithms for connections that have diverse paths available. We therefore compare TCP New Reno with the delay-based CAIA Delay-Gradient (CDG) [174] algorithm. Again we plot cwnd evolution for each subflow over time. Figure 7.12 shows the shared bottleneck case, while Figure 7.13 is without a shared bottleneck. As with the previous comparison, the algorithms exhibit distinct cwnd profiles, and here the difference between loss-based and delay-based approaches is apparent. In both instances the cwnd profile of CDG shows less of the

[4] net.inet.tcp.mptcp.cc.subflow_<N>_algo

[5] Since the initial subflow has a head start of several RTTs before the second is established, this isn't a comparison as to the effectiveness of the algorithms. It is simply a demonstration of how different approaches could be combined.

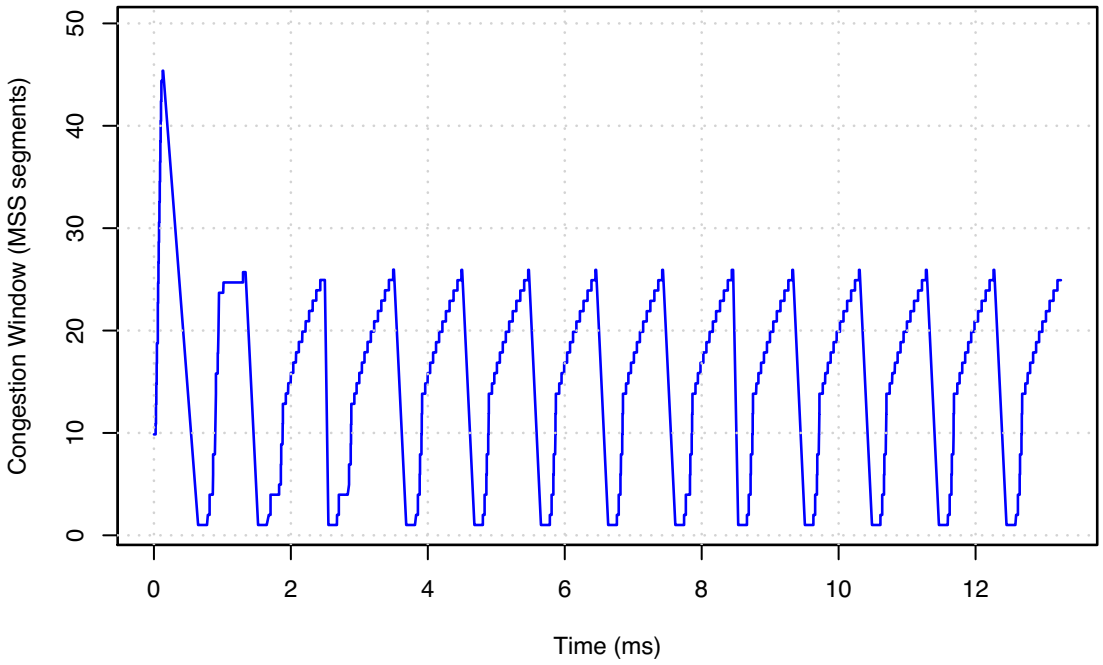


(a) Subflow 1: New Reno

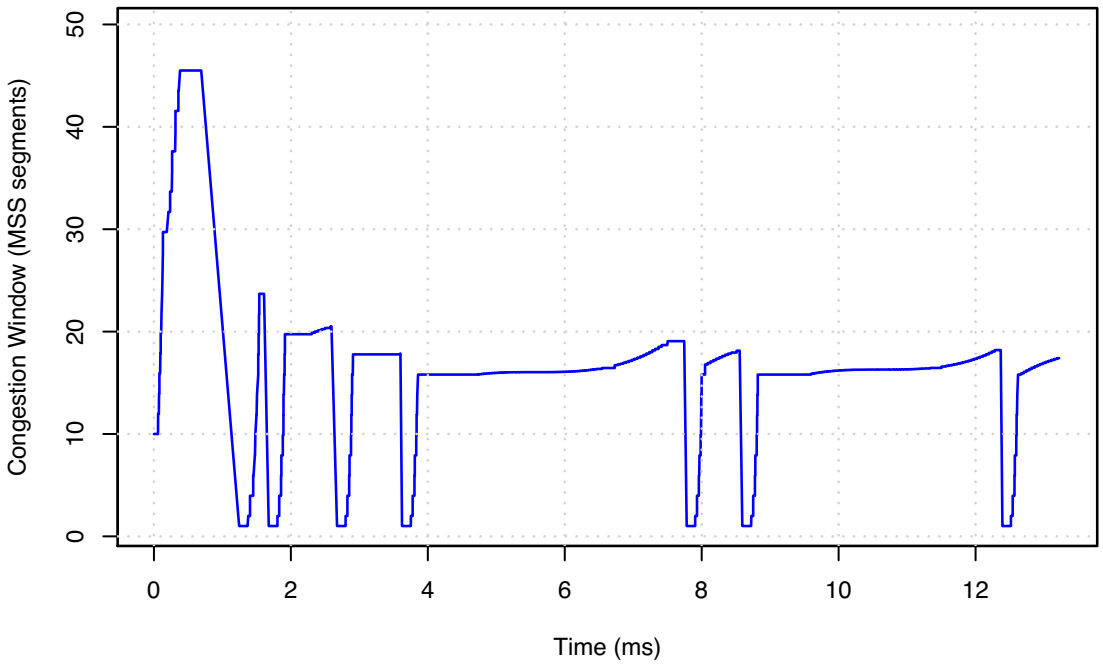


(b) Subflow 2: Cubic

Figure 7.10: Subflow congestion windows for New Reno and Cubic subflows across a shared bottleneck.

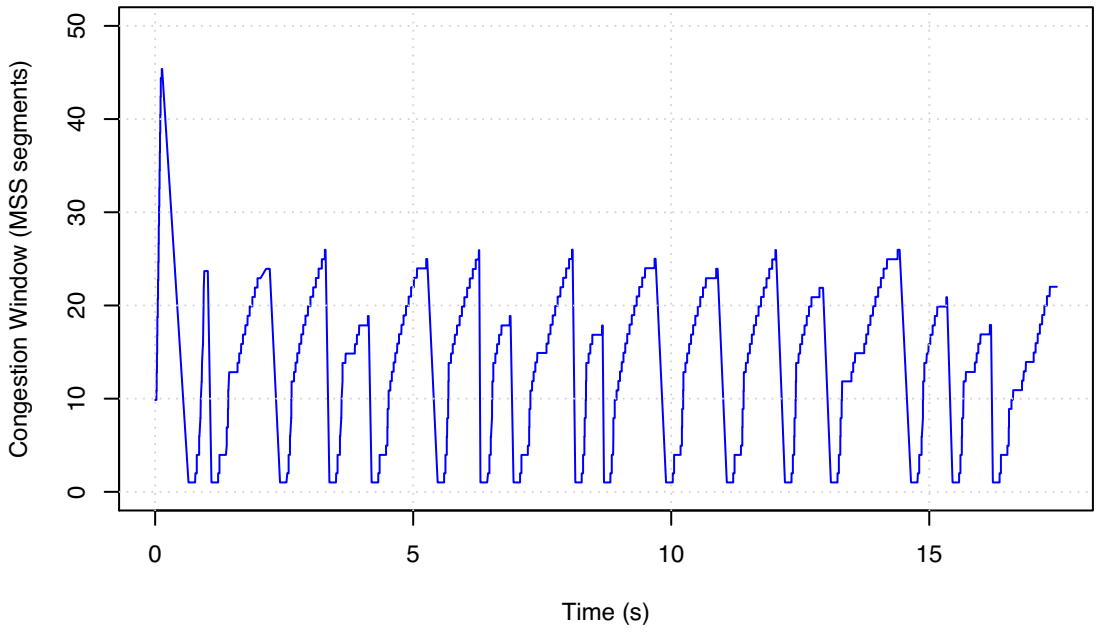


(a) Subflow 1: New Reno

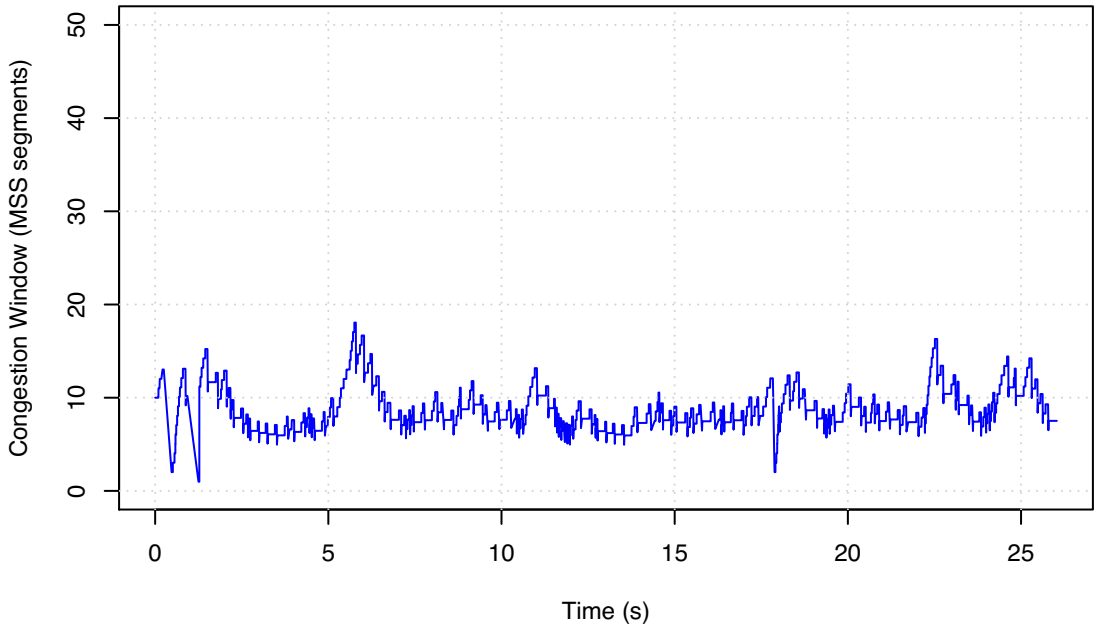


(b) Subflow 2: Cubic

Figure 7.11: Subflow congestion windows without a shared bottleneck.

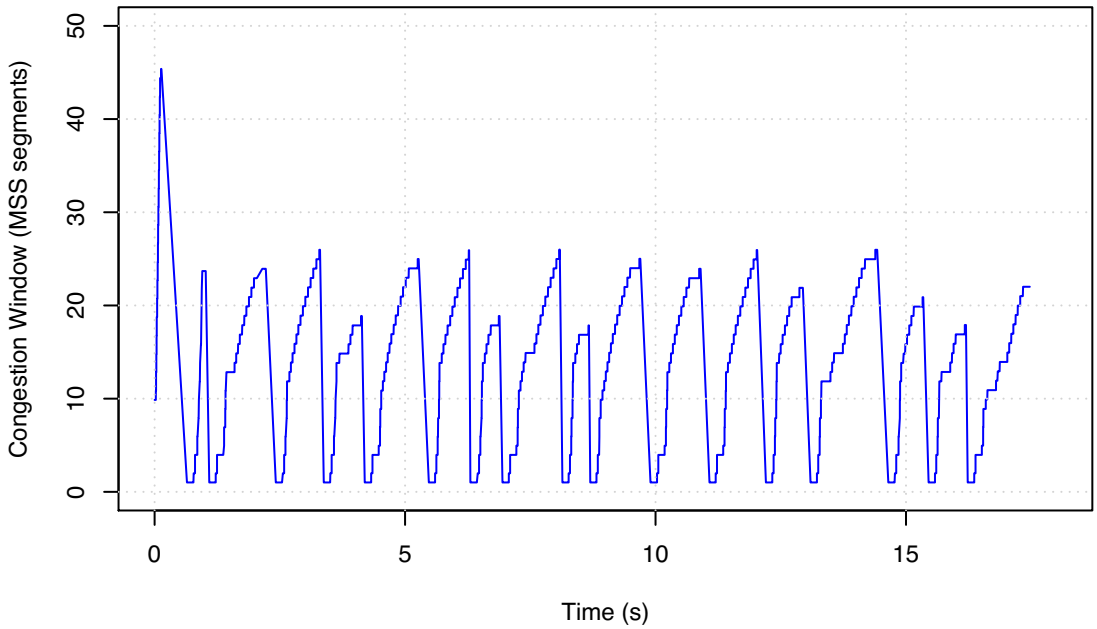


(a) Subflow 1: New Reno

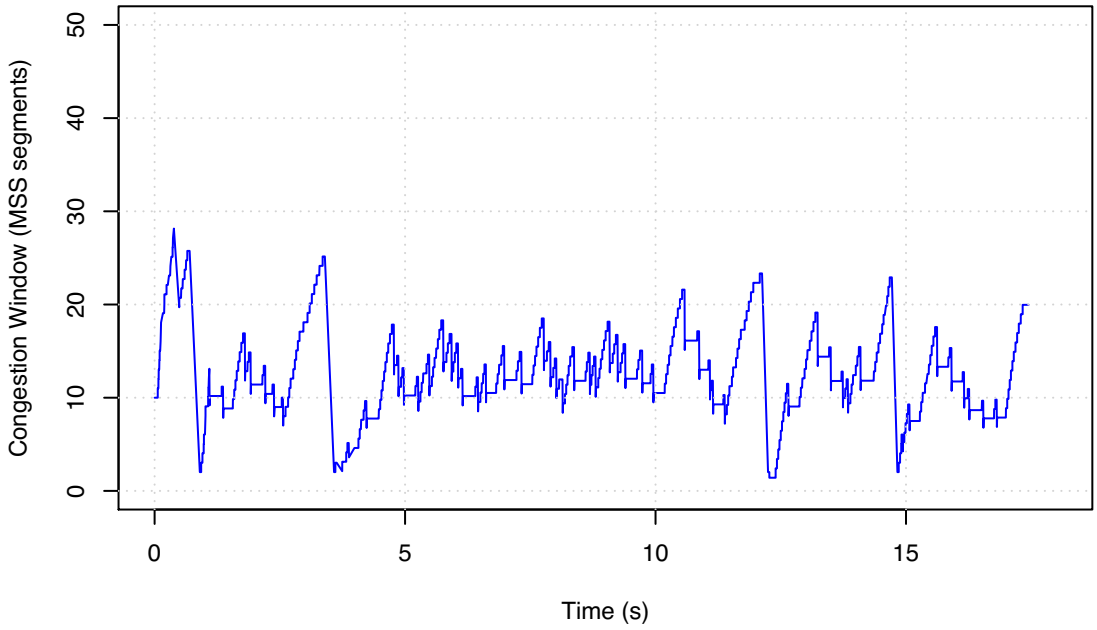


(b) Subflow 2: CAIA Delay Gradient

Figure 7.12: Using delay-based and loss-based congestion control on subflows with a shared bottleneck link.



(a) Subflow 1: New Reno



(b) Subflow 2: CAIA Delay Gradient

Figure 7.13: Using delay-based and loss-based congestion control on subflows without a shared bottleneck.

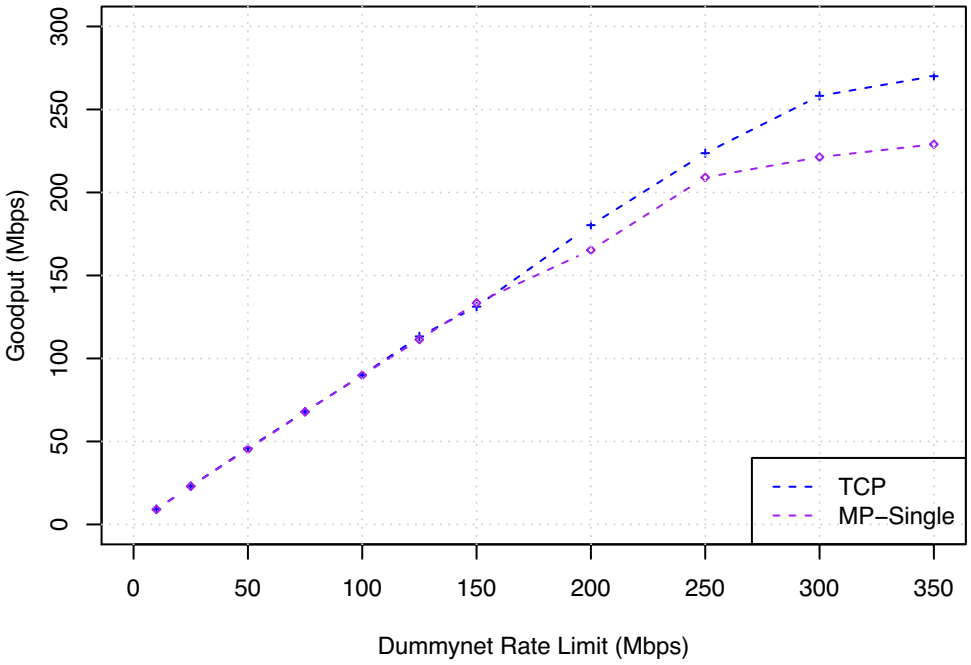


Figure 7.14: Comparing Iperf goodput to TCP as the link bandwidth is increased.

cyclic, loss-inducing probing of the path capacity.

A crucial requirement of our modular framework is the ability to couple congestion functions of the subflows, and much of the traffic steering benefits of MPTCP are derived from combining the congestion functions in this way. The MPTCP-CC framework allows for a coupled approach, though an implementation is left for future work.

7.2.5 Performance

Supporting MPTCP requires significant additions to the network stack. We conduct an introductory assessment of the immediate impact that the MPTCP infrastructure has on network performance. In doing so we can get a sense as to how scalable the implementation will be and if necessary revisit design choices in future work. As the implementation is a work-in-progress, further development work is required before a more extensive performance evaluation can be undertaken. Our initial approach is to measure goodput as the bandwidth is increased.

We measure single-subflow MPTCP goodput as link-bandwidth is increased and compare with standard TCP. There is no hardware offload (TSO, LRO), and SACK is disabled. This is a use-

	TCP-SACK	TCP	MPTCP
Goodput (Mbps)	590	436	360

Table 7.5: Goodput without rate-limiting

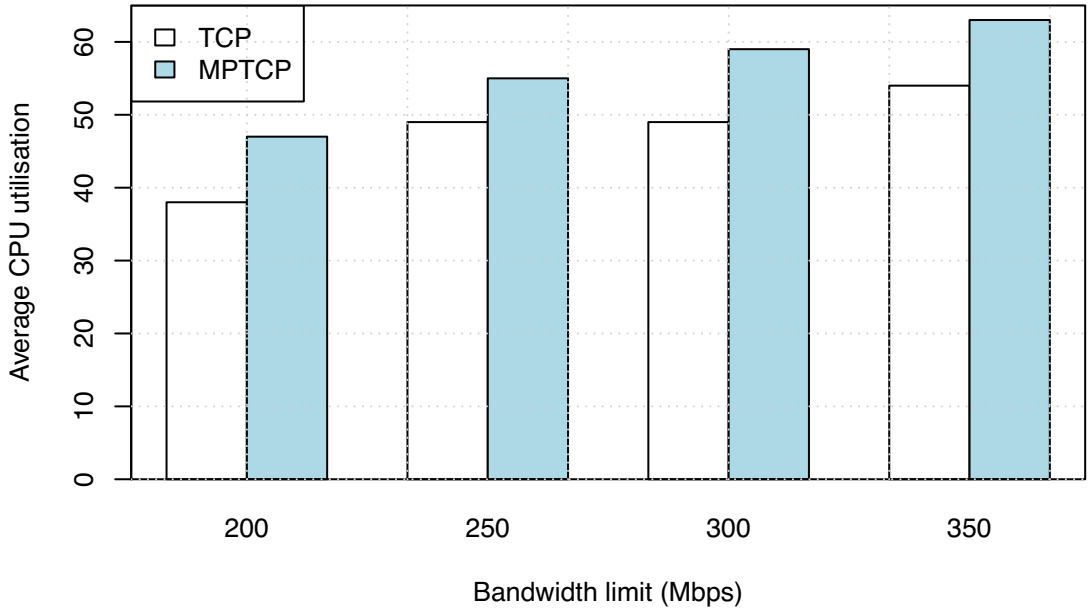


Figure 7.15: Average CPU usage by the kernel as bandwidth increases.

ful comparison, as even with a single subflow MPTCP infrastructure (e.g. multiple receive buffers, taskqueue event processing) must be used, which adds overhead and impacts the dynamics of the underlying TCP subflow (as seen in Section 7.2.2). By using only a single subflow, the effect that scheduling decisions might have on goodput is removed.

Table 7.5 shows the average maximum goodput and CPU use in the testbed, for a single TCP and MPTCP connection without rate-limiting. With a theoretical NIC speed of 1Gbps, the testbed reaches a maximum approaching 600Mbps. Disabling TCP-SACK reduces goodput by approximately 160Mbps. The maximum goodput achieved by MPTCP is approximately 75Mbps less than TCP without SACK. Figure 7.14 compares TCP and MPTCP with Dummynet rate-limiting enabled.

As with the single-subflow case in Section 7.2.2, the implementation achieves goodput similar to standard TCP with link speeds up to 150Mbps. Above 150Mbps, the MPTCP implementation

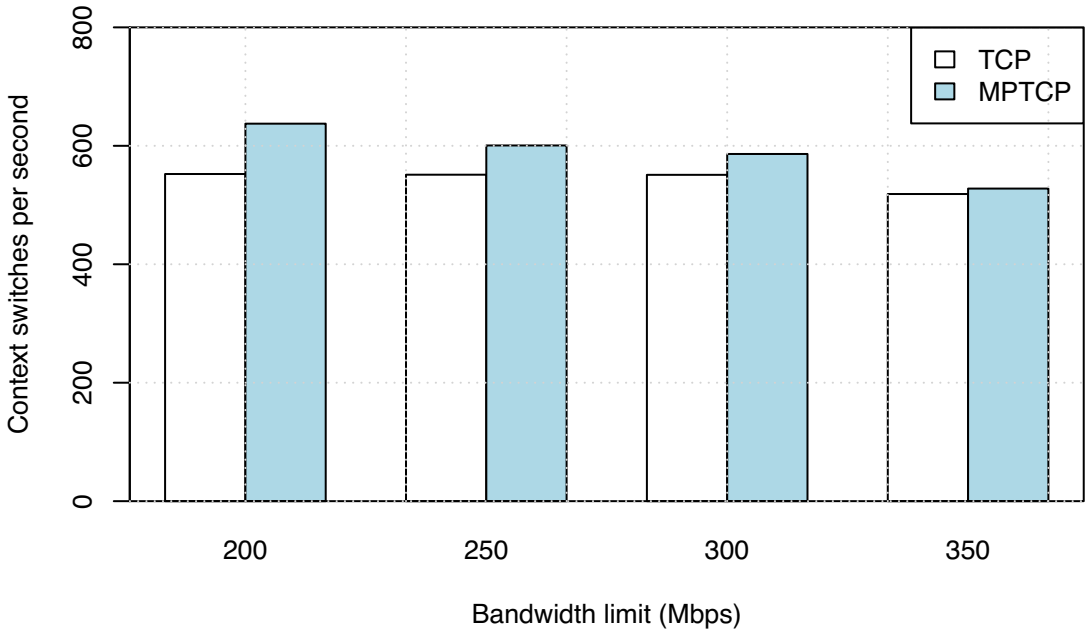


Figure 7.16: Average count of context switches per-second as bandwidth increases.

becomes less efficient at using the available bandwidth. To determine whether this decrease in performance is due to processing overhead, we also measured kernel CPU utilisation and the average number of context switches per second for TCP and MPTCP connections. These are shown for the sender-side in Figure 7.15 and Figure 7.16. There was no significant difference for these measures on the receive side.

We see that, as expected, the deferred dispatch model of processing does increase the number of context switches and CPU time (by at least 10%) required per Mb of goodput. However, in each case there remains a large amount of CPU overhead, meaning that the goodput in this instance is not limited by CPU, but by protocol-processing inefficiencies in the implementation. Aside from disabling SACK, initial analysis showed inadequate send-buffer size (for the subflow socket) to be the most obvious limiting factor.

To fully utilise the capcity of the path, the send buffer must be large enough to sustain the TCP flow-control window. As BDP increases, the send buffer size must be increased to maintain the optimal number of packets in-flight along the path. Accurate and timely data-level buffer sizing has also been identified as a critical part of MPTCP performance tuning [2]. Larger send buffer sizes

are attained in TCP through the use of *auto-tuning* algorithms that allow the buffer-size to be scaled incrementatlly based on an estimation of the path BDP [175].

Our initial experiments involved static allocation of master send and receive buffers. Since the kernel places a limit on how much space can be statically reserved for a buffer, this restricted performance to roughly 150Mbps, as the master send-buffer was not able to adequately feed the subflow send buffer. To allow the master socket send buffer to increase in size, a simple algorithm was implemented to scale the data-level send buffer based on occupancy. Auto buffer-scaling is available in the FreeBSD TCP stack and subflows are thus already able to scale in-line with the master socket buffers. Activating data-level auto-scaling allowed the MPTCP implementation to exceed 150Mbps (as shown earlier in Figure 7.14).

Although the simple buffer-scaling algorithm did improve performance of the MPTCP connection, it was observed that for a given bandwidth the send buffer for the single MPTCP subflow remained undersized compared to that of a standard TCP flow. This indicates that improvement in the scaling algorithm is required. Auto-buffer scaling performance can be improved by employing bandwidth and RTT estimation, and enhancing the current implementation to include BDP estimation will be the first step for future work in this area.

Though the implementation is not yet capable of high-bandwidth connections, we consider the current achievable speeds adequate to support a range of scheduling and CC experiments for lower-bandwidth scenarios (e.g. replicating home-broadband connections). It is expected that the achievable goodput can be improved with further development.

7.3 Conclusion

The tests presented in this chapter show that our MPTCP implementation is functional and can achieve throughputs that are useful for a range of network experiments. The purpose of these tests is to validate that the implementation meets the basic requirements for creating MPTCP connections, to demonstrate the use of scheduling and congestion control modules, and to show that performance is adequate for use in experimentation.

We first showed that the implementation meets the requirements for establishing MPTCP connections, and when unsuccessful falls back to standard TCP. Beyond the negotiation phase we verify that the implementation can operate basic multi-subflow scenarios and can outperform a standard TCP connection when parallel paths are available. We show that subflow-level retransmission mechanisms are preserved and that data-level retransmissions are possible in the event of path failure.

Modular scheduling and CC are key features of the implementation. We show that modules can be created and loaded with modest effort and are able to access the state required to support different scheduling and CC approaches. In these experiments a naive round-robin scheduler is used to distribute data between the subflows. In several instances this causes head-of-line blocking or leads to correlated packet losses between subflows. The use of this scheduler serves to illustrate the need for appropriate scheduling and congestion control for MPTCP connections, and the value of incorporating modular frameworks for experimentation. We show that the implementation is able to set congestion control on a per-subflow basis and that the congestion window growth of each subflow matches the expected behaviour of the specified CC algorithm.

Our initial performance tests show that the implementation is able function across a range of bandwidths. While there is much room for optimisation in future work, the implementation appears efficient enough to not be a bottleneck if used to run network experiments in the near future.

Chapter 8

Conclusion

As the number of multi-homed edge devices grows, there has been a desire to increase the traffic engineering options available to end-hosts. MPTCP is a recent solution that has gained traction due to backwards compatibility with legacy TCP applications and its suitability for Internet deployment. Two MPTCP implementations are already publicly available: the UCL implementation for the Linux kernel and Apple's XNU kernel implementation for OSX and iOS. Of these, only the Linux implementation supports the full suite of MPTCP capabilities such as backwards compatibility with TCP applications, simultaneous data transfer, scheduling and data-level reassembly. A key goal of our implementation is to make MPTCP accessible for FreeBSD-based network research.

In this thesis we have:

- Presented a partially-modular MPTCP architecture for the FreeBSD kernel that includes modular frameworks to support scheduling and congestion control experimentation.
- Developed a functional prototype based on this design and made it available to the public [11].
- Demonstrated the prototype in several basic MPTCP scenarios and showed that the modular frameworks are functional and performance is adequate for experimental use.

This is the first publicly available implementation for FreeBSD and lays much of the groundwork for a fully featured MPTCP stack. Development is ongoing and will continue with input from the FreeBSD community.

8.1 Summary

A survey of existing multi-homing solutions in Chapter 3 supported the view of the transport layer as the logical endpoint for end-to-end multi-path sessions. We show that several post-TCP transport protocols, most notably CMT-SCTP, have already been developed to leverage multi-path enabled hosts. However, deployment of these protocols across the Internet is found to have been hampered by the prevalence of middleboxes that 'normalise' or discard packets from all but the most common protocols.

Chapter 4 discussed the current state of research into scheduling and CC for MPTCP. Sender-side mechanisms are discussed based on their overarching approach - *passive* and *active*, *push* and *pull* for scheduling, with *coupled* and *uncoupled* for congestion control. The purpose of this survey was not to identify the best approach, but rather to understand how approaches differ and how they the can be supported from an implementation standpoint. We identified key requirements of different approaches - such as access to TCP-level statistics or multiple segment buffers - which ultimately informed our design.

In Chapter 6 we presented a MPTCP design suitable for implementation in the FreeBSD kernel. The design includes provision for the primary features of MPTCP. The implementation resulting from this design is open-source and available to the public [11]. After several iterations we arrived at a pragmatic approach that aims to balance ease of modification, performance and maintainability. The core of the implementation combines in-kernel and dynamically loaded code paths. Added to this are modular frameworks for scheduling and congestion control.

We leverage the FreeBSD modular TCP stack framework for part of our implementation. Creating a partially in-kernel and dynamically loaded stack is unusual, though provides some distinct benefits. Principally it allowed us to minimise the amount of code directly added to the in-kernel TCP stack, reducing the maintenance burden. In the longer term it means that the MPTCP can continue to develop in parallel with the TCP stack - integrating new enhancements but remaining distinct and allowing the flexibility for MPTCP-focused changes that would be difficult to apply directly to the TCP stack. Another benefit of using a modular stack approach is that MPTCP code paths are only used for MPTCP connections - TCP connections can be dynamically switched to the standard TCP stack with minimal effort.

In-kernel components consist of MPTCP protocol-hooks, option processing, and support for managing DSN mappings on output. Changes to the kernel were made only when unavoidable (e.g.

socket-facing user request routines), or where which functions are not yet modularised (e.g. the TCP syncache).

We have created modular scheduling and CC frameworks for the MPTCP, allowing such algorithms to be loaded dynamically, and a template module is provided for each of these. The scheduling and CC frameworks are important additions with respect to the goal of enabling further research. Much of the dynamic behaviour of MPTCP is derived through scheduling and congestion control, and by modularising these processes researchers can develop their own schemes without having to change or recompile the kernel. Since modules are decoupled from the core of the implementation, they will require few or minimal changes to remain compatible as the implementation evolves. By using ModCC as a template, the modules present the familiar usage semantics of existing TCP congestion control modules and are configured using the standard `sysctl` interface.

In Chapter 7 we demonstrate the MPTCP implementation in several scenarios. We first show that the implementation can negotiate MPTCP connections under a standard TCP socket and fall back to TCP if the handshake fails. In the event of packet loss, TCP-level loss recovery functions are shown to retransmit the lost segment. In the event of path failure, we show that data-level retransmission is possible, and that a path may recover and rejoin the connection.

Two multi-subflow scenarios are presented that show functional multi-path operation and demonstrate the effect of a shared bottleneck on the subflows. We first show that a multi-addressed client can initiate a connection to a server through a shared bottleneck and then join an additional subflow to the connection. When the server is multi-addressed, we show that the additional address is advertised and that a client issues a join in response. When the bottleneck is removed, we show that MPTCP is able to take advantage of the additional capacity.

Our demonstration scenarios are performed using a naive round-robin scheduler *simplerr* that assigns packets to subflows on a per-segment basis. The scheduler was written using the modular scheduling framework. We demonstrated use of the modular congestion framework by creating a module that sets TCP-New Reno for the first subflow and either TCP-Cubic/CDG for the second subflow. We demonstrate that subflows belonging to the same MPTCP connection can have markedly different congestion window evolution, and that the window profile matches the expected behaviour for the algorithm.

As an introductory performance test, we compare single-path MPTCP against standard TCP over paths with increasing bandwidth limits. We find that when using a single subflow, MPTCP scales in-line with the standard TCP implementation up to 150Mbps.

8.2 Future Work

Beyond improvements to stability and performance, there are a number of key areas to pursue:

- *Further evaluation:* The tests presented in this thesis serve only to demonstrate that our implementation supports the basic MPTCP protocol. The performance in terms of network throughput, CPU and memory consumption have not been extensively tested here and will be revisited as the implementation matures.

- *Further compliance testing:* A more thorough evaluation of fault-tolerance and security issues (e.g. cryptographic hash strength) is mandatory if the implementation is to be deployed on the Internet.

- *Modular* `tcp_output()`*:* Changes to `tcp_output` were made in only a few key points. However, to support these changes a number of MPTCP subroutines must be built as part of the kernel. As `tcp_output` can be replaced by a TCP stack module, a MPTCP-specific version will be created in the future. This will also allow the output path to be tailored specifically for MPTCP.

- *Path management:* In addition to scheduling and congestion control, the path management process poses several interesting questions. The current implementation uses a static path management scheme. A separate path management module would allow more complex path management decisions, for example using link-layer feedback to determine when to initiate a join.

- *Improved buffer management:* The current implementation performs rudimentary buffer scaling. The ability to appropriately resize send and receive buffers is critical if the implementation is to be used in real-world or high-throughput scenarios.

- *Community engagement:* A robust, fully featured release will require ongoing development and engagement from the wider FreeBSD community. Getting '*up-to-speed*' with the requisite knowledge of the MPTCP specification and the FreeBSD kernel is a substantial time investment, meaning there have been no other public MPTCP projects for FreeBSD. With the availability of this implementation, the hurdle for wider community involvement is reduced.

Appendix A

Design Overview of Multipath TCP version 0.3 for FreeBSD-10

Originally publicly released as CAIA Technical Report 130424A on April 24, 2013. Available online at: `http://caia.swin.edu.au/reports/130424A/CAIA-TR-130424A.pdf`

Design Overview of Multipath TCP version 0.3 for FreeBSD-10

Nigel Williams, Lawrence Stewart, Grenville Armitage
Centre for Advanced Internet Architectures, Technical Report 130424A
Swinburne University of Technology
Melbourne, Australia
njwilliams@swin.edu.au, lastewart@swin.edu.au, garmitage@swin.edu.au

***Abstract*—This report introduces FreeBSD-MPTCP v0.3, a modification to the FreeBSD-10 kernel that enables support for the IETF's emerging Multipath TCP (MPTCP) specification. We outline the motivation for (and potential benefits of) using MPTCP, and discuss key architectural elements of our design.**

***Index Terms*—CAIA, TCP, Multipath, Kernel, FreeBSD**

I. Introduction

Traditional TCP has two significant challenges – it can only utilise a single network path between source and destination per session, and (aside from the gradual deployment of explicit congestion notification) congestion control relies primarily on packet loss as a congestion indicator. Traditional TCP sessions must be broken and reestablished when endpoints shift their network connectivity from one interface to another (such as when a mobile device moves from 3G to 802.11, and thus changes its active IP address). Being bound to a single path also precludes multihomed devices from using any additional capacity that might exist over alternate paths.

TCP Extensions for Multipath Operation with Multiple Addresses (RFC6824) [1] is an experimental RFC that allows a host to spread a single TCP connection across multiple network addresses. Multipath TCP (MPTCP) is implemented within the kernel and is designed to be backwards compatible with existing TCP socket APIs. Thus it operates transparently from the perspective of the application layer and works with unmodified TCP applications.

As part of CAIA's NewTCP project [2] we have developed and released a prototype implementation of the MPTCP extensions for FreeBSD-10 [3]. In this report we describe the architecture and design decisions behind our version 0.3 implementation. At the time of writing, a Linux reference implementation is also available at [4].

The report is organised as follows: we briefly outline the origins and goals of MPTCP in Section II. In Section III we detail each of the main architectural changes required to support MPTCP in the FreeBSD 10 kernel. The report concludes with Section IV.

II. Background to Multipath TCP (MPTCP)

The IETF's Multipath TCP (MPTCP) working group[1] is focused on an idea that has emerged in various forms over recent years – namely, that a single transport session as seen by the application layer might be striped or otherwise multiplexed across multiple IP layer paths between the session's two endpoints. An over-arching expectation is that TCP-based applications see the traditional TCP API, but gain benefits when their session transparently utilises multiple, potentially divergent network layer paths. These benefits include being able to stripe data over parallel paths for additional speed (where multiple similar paths exist concurrently), or seamlessly maintaining TCP sessions when an individual path fails or as a mobile device's multiple underlying network interfaces come and go. The parts of an MPTCP session flowing over different network paths are known as *subflows*.

A. Benefits for multihomed devices

Contemporary computing devices such as smartphones, notebooks or servers are often multihomed (multiple network interfaces, potentially using different link layer technologies). MPTCP allows existing TCP-based applications to utilise whichever underlying interface (network path) is available at any given time, seamlessly maintaining transport sessions when endpoints shift their network connectivity from one interface to another.

When multiple interfaces are concurrently available, MPTCP enables the distribution of an application's

[1] http://datatracker.ietf.org/wg/mptcp/charter/

traffic across all or some of the available paths in a manner transparent to the application. Networks can gain traffic engineering benefits as TCP connections are steered via multiple paths (for instance away from congested links) using coupled congestion control [5]. Mobile devices such as smartphones and tablets can be provided with persistent connectivity to network services as they transition between different locales and network access media.

B. SCTP is not quite the same as MPTCP

It is worth noting that SCTP (stream control transmission protocol) [6] also supports multiple endpoints per session, and recent CMT work [7] enables concurrent use of multiple paths. However, SCTP presents an entirely new API to applications, and has difficulty traversing NATs and any middleboxes that expect to see only TCP, UDP or ICMP packets 'in the wild'. MPTCP aims to be more transparent than SCTP to applications and network devices.

C. Previous MPTCP implementation and development

Most early MPTCP work was supported by the EU's Trilogy Project[2], with key groups at University College London (UK)[3] and Université catholique de Louvain in Louvain-la-Neuve (Belgium)[4] publishing code, working group documents and research papers. These two groups are responsible for public implementations of MPTCP under Linux userland[5], the Linux kernel[6] and a simulation environment (htsim)[7]. Some background on the design, rationale and uses of MPTCP can be found in papers such as [8]–[11].

D. Some challenges posed by MPTCP

MPTCP poses a number of challenges.

1) Classic TCP application interface: The API is expected to present the single-session socket of conventional TCP, while underneath the kernel is expected to support the learning and use of multiple IP-layer identities for session endpoints. This creates a non-trivial implementation challenge to retrofit such functionality into existing, stable TCP stacks.

2) Interoperability and deployment: Any new implementation must interoperate with the reference implementation. The reference implementation has not yet had to address interoperation, and as such holes and assumptions remain in the protocol documents. An interoperable MPTCP implementation, given FreeBSD's slightly different network stack paradigm relative to Linux, should assist in IETF standardisation efforts. Also, the creation of a BSD-licensed MPTCP implementation benefits both the research and vendor community.

3) Congestion control (CC): Congestion control (CC) must be coordinated across the subflows making up the MPTCP session, to both effectively utilise the total capacity of heterogeneous paths and ensure a multipath session does not receive *"...more than its fair share at a bottleneck link traversed by more than one of its subflows"* [12]. The WG's current proposal for MPTCP CC remains fundamentally a loss-based algorithm that *"...only applies to the increase phase of the congestion avoidance state specifying how the window inflates upon receiving an ACK. The slow start, fast retransmit, and fast recovery algorithms, as well as the multiplicative decrease of the congestion avoidance state are the same as in standard TCP*" (Section 3, [12]). There appears to be wide scope for exploring how and when CC for individual subflows ought to be tied together or decoupled.

III. Changes to FreeBSD's TCP Stack

Our MPTCP implementation has been developed as a kernel patch[8] against revision 248226 of FreeBSD-10.

A broad view of the changes and additions between revision 248226 and the MPTCP-enabled kernel:

1) Creation of the Multipath Control Block (MPCB) and the repurposing of the existing TCP Control Block (TCPCB) to act as a MPTCP subflow control block.
2) Changes to user requests (called from the socket layer) that handle the allocation, setup and deallocation of control blocks.
3) New data segment reassembly routines and data-structures.
4) Changes to socket send and socket receive buffers to allow concurrent access from multiple subflows and mapping of data.
5) MPTCP option insertion and parsing code for input and output paths.

[2] http://www.trilogy-project.org/
[3] http://nrg.cs.ucl.ac.uk/mptcp/
[4] http://inl.info.ucl.ac.be/mptcp
[5] http://nrg.cs.ucl.ac.uk/mptcp/mptcp_userland_0.1.tar.gz
[6] https://scm.info.ucl.ac.be/trac/mptcp/
[7] http://nrg.cs.ucl.ac.uk/mptcp/htsim_0.1.tar.gz

[8] Implementing MPTCP as a loadable kernel module was considered, but deemed impractical due to the number of changes required.

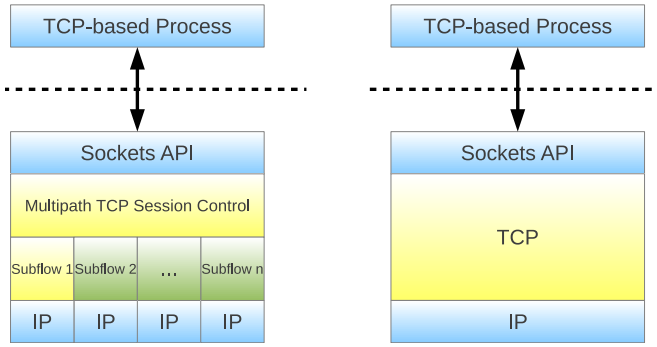


Figure 1. Logical MPTCP stack structure (left) versus traditional TCP (right). User space applications see same socket API.

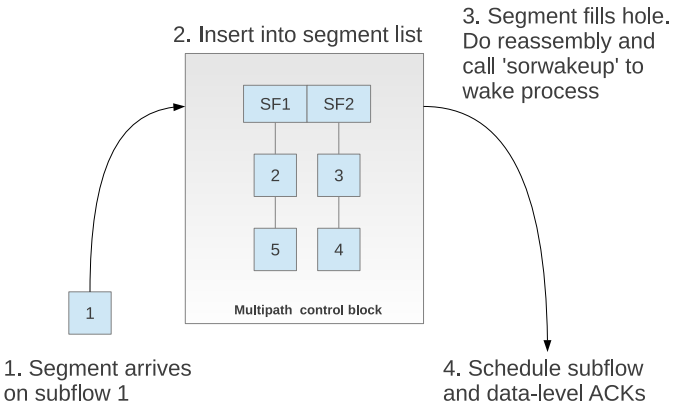


Figure 2. Each subflow maintains a segment receive list. Segments are placed into the list in subflow-sequence order as they arrive (data-level sequence numbers are shown). When a segment arrives in data-sequence order, the lists are locked and data-level re-ordering occurs. The application is then alerted and can read in the in-order data.

6) Locking mechanisms to handle additional concurrency introduced by MPTCP.
7) Various MPTCP support functions (authentication, hashing etc).

The changes are covered in more detail in the following subsections.

A. Protocol Control Blocks

The implementation adds a new control block, the MPTCP control block (MPCB), and repurposes the TCP Control Block (RFC 793 [13]) as a subflow control block. The header file *netinet/mptcp_var.h* has been added to the FreeBSD source tree, and the MPCB structure is defined within.

A MPCB is created each time an application creates a TCP socket. The MPCB maintains all information required for multipath operation and manages the subflows in the connection. It sits logically between the subflow TCP control blocks and the socket layer. This arrangement is compared with traditional TCP in Figure 1.

At creation, each MPCB associated with a socket contains at least one subflow (the *master subflow*, or *subflow 1*). The subflow control block is a modified traditional TCP control block found in *netinet/tcp_var.h*.

Protocol control blocks are initialised and attached to sockets via functions in *netinet/tcp_usrreq.c* (user requests). A call to *tcp_connect()* in *netinet/tcp_usrreq.c* results in a call to *mp_newmpcb()*, which allocates and initialises the MPCB.

A series of functions (*tcp_subflow_**) are implemented in *tcp_usrreq.c* and are used to create and attach any additional *slave subflows* to the MPTCP connection.

B. Segment Reassembly

MPTCP adds a data-level sequence space above the sequence space used in standard TCP. This allows segments received on multiple subflows to be ordered before delivery to the application. Modifications to reassembly are found in *netinet/tcp_reass.c* and in *kern/uipc_socket.c*.

In pre-MPTCP FreeBSD, if a segment arrives that is not the next expected segment (sequence number does not equal RCV.NXT), it is placed into a reassembly queue. Segments are placed into this queue in sequence order until the expected segment arrives. At this point, all in-order segments held in the queue are appended to the socket receive buffer and the process is notified that data can be read in. If a segment arrives that is in-order and the reassembly list is empty, it is appended to the receive buffer immediately.

In our implementation, subflows do not access the socket receive buffer directly, and instead repurpose the traditional reassembly queue for both in-order queuing and out-of-order reassembly. Unknown to subflows, their individual queues form part of a larger multipath-related reassembly data structure, shown in Figure 2.

All incoming segments on a subflow are appended to that subflow's reassembly queue (the *t_segq* member of the TCP control block defined in *netinet/tcp_var.h*) in subflow sequence order. When the head of a subflow's queue is in data sequence order (segment's data level sequence number equals ds_rcv_nxt), then data-level reassembly is triggered (ultimately by a wakeup on the socket which will in turn defer reassembly to the userspace thread context, but due to unresolved bugs we currently trigger from kernel thread context).

Data-level reassembly involves traversing each subflow segment list and appending in-sequence (data-level) segments to the socket receive buffer. This occurs in the

mp_do_reass() function of *netinet/tcp_reass.c*. During this time a write lock is used to exclude subflows from manipulating their reassembly queues.

Subflow and data-level reassembly have been split this way to reduce lock contention between subflows and the multipath layer. It also allows data-reassembly to be deferred to the application's thread context during a read on the socket, rather than performed by a kernel fast-path thread.

At completion of data-level reassembly, a data-level ACK is scheduled on whichever subflow next sends a regular TCP ACK packet.

C. Send and Receive Socket Buffers

In FreeBSD's implementation of standard TCP, segments are sent and received over a single (address,port) tuple, and socket buffers exist exclusively for each TCP session. MPTCP sessions have *1+n* (where n denotes additional addresses) subflows that must access the same send and receive buffers. The following sections describe the changes to the socket buffers and the addition of the *ds_map*. .

1) The ds_map *struct:* The *ds_map* struct is defined in *netinet/tcp_var.h* and used for both send-related and receive-related functions. Send and receive related maps are stored in the subflow control block lists *t_txmaps* (send buffer maps) and *t_rxmaps* (receive buffer maps) respectively.

On the send side, *ds_maps* track accounting information related to DSN maps advertised to the peer, and are used to mediate access between subflows and the send socket buffer. By mediating socket buffer access in this way, lock contention can be avoided when sending data from a *ds_map*. On the receive side, *ds_maps* track accounting information related to received DSN maps and associated payload data from the peer.

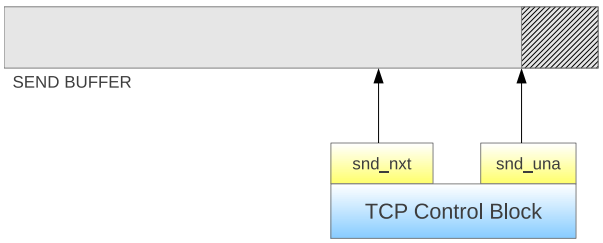


Figure 3. Standard TCP Send Buffer. The lined area represents sent bytes that have been acknowledged by the receiver.

2) Socket Send Buffer: Figure 3 illustrates how in standard TCP, each session has exclusive access to its own send buffer. The variables *snd_nxt* and *snd_una* are used respectively to track which bytes in the send buffer are to be sent next, and which bytes were the last acknowledged by the receiver.

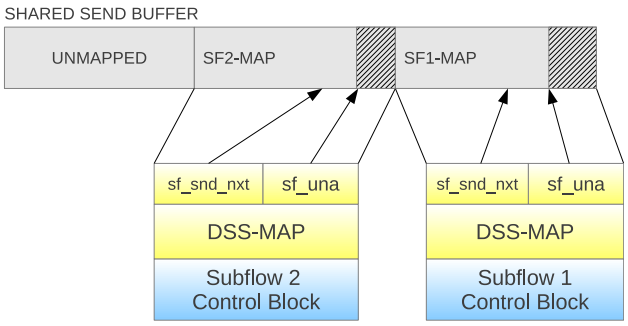


Figure 4. A MPTCP send buffer contains bytes that must be mapped to multiple TCP-subflows. Each subflow is allocated one or more *ds_maps* (DSS-MAP) that define these mappings.

Figure 4 illustrates how in the multipath kernel, data from the sending application is still stored in a single send socket buffer. However access to this buffer is moderated by the packet scheduler in *mp_get_map(),* implemented in *netinet/mptcp_subr.c* (see Section III-D)

The packet scheduler is run when a subflow attempts to send data via *tcp_output()* without owning a *ds_map* that references unsent data.

When invoked, the scheduler must decide whether the subflow should be allocated any data. If granted, allocations are returned as a *ds_map* that contains an offset into the send buffer and the length of data to be sent. Otherwise, a NULL map is returned, and the send buffer appears 'empty' to the subflow. The *ds_map* effectively acts as a unique socket buffer from the perspective of the subflow (i.e. subflows are not aware of what other subflows are sending). The scheduler is not invoked again until the allocated map has been completely sent.

This scheme allows subflows to make forward progress with variable overheads that depend on how frequently the scheduler is invoked i.e. larger maps reduce overheads.

As a result of sharing the underlying send socket buffer via *ds_maps* to avoid data copies, releasing bytes becomes more complex. Firstly, data-level ACKs rather than subflow-level ACKs mark the multipath-level stream bytes which have safely arrived, and therefore control the advancement of ds_snd_una. Secondly, *ds_maps* can potentially overlap any portion of their socket buffer mapping with each other (e.g. data-level retransmit), and therefore the underlying socket buffer bytes (encapsulated in chained mbufs) can only be dropped when both ds_snd_una has acknowledged them

and all maps which reference the bytes have been deleted.

To potentially defer the dropping of bytes from the socket buffer without adversely impacting application throughput requires that socket buffer occupancy be accounted for logically rather than actually. To this end, the socket buffer variable *sb_cc* of an MPTCP socket send buffer refers to the logical number of bytes held in the buffer without data-level acknowledgment, and a new variable *sb_actual* has been introduced to track the actual number of bytes in the buffer.

3) Socket Receive Buffer: In pre-MPTCP FreeBSD, in-order segments were copied directly into the receive buffer, at which time the process was alerted that data was available to read. The remaining space in the receive buffer was used to advertise a receive window to the sender.

As described in Section III-B, each subflow now holds all received segments in a segment list, even if they are in subflow sequence order. The segment lists are then linked by their list heads to create a larger data-level reassembly data structure. When a segment arrives that is in data sequence order, data-level reassembly is triggered and segments are copied into the receive buffer. As the size of the reassembly list is effectively unbounded, we currently advertise a maximum receive window (*TCP_MAXWIN* ∗ *scaling factor*) on all subflows.

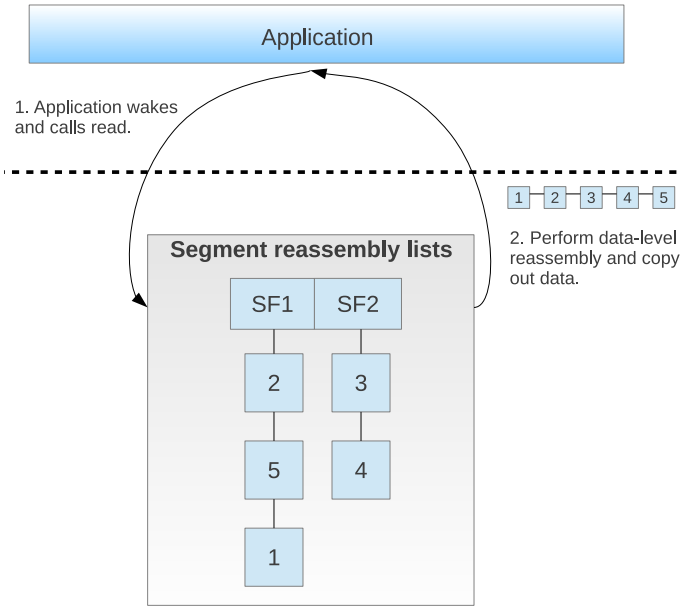


Figure 5. A future release will integrate the multipath reassembly structure into the socket receive buffer. Segments will be read directly from the multi-subflow aware buffer as data-level reassembly occurs.

We plan to integrate the multipath reassembly structure into the socket receive buffer in a future release. Coupled together with deferred reassembly, an application's thread context would be responsible for performing data-level reassembly on the multi-subflow aware buffer after being woken up by a subflow that received the next expected data-level segment (see Figure 5).

D. Packet Scheduler

The packet scheduler is responsible for determining which subflows are able to send data from the socket send buffer, and how much data they can send. A basic packet scheduler is implemented in the v0.3 patch, and can be found in the *mp_get_map()* function of *netinet/mptcp_subr.c*.

The current algorithm checks the number of unmapped bytes (bytes not yet allocated to an existing *ds_map* struct) in the buffer and the total occupancy of the buffer. If the buffer is not full, the application is free to write more data to the socket so we assign the current contents of the buffer to the requesting subflow on the basis more data will be written soon. If the buffer is full (*sb_cc* is equal to the buffer's maximum capacity *sb_hiwat*), the application is stalled waiting for buffer space to become available and subflows are competing for the unmapped data in the buffer. In this case, we allocate an amount of data equal to the unmapped bytes divided by the number of active subflows, with a floor value set to the MSS of the requesting subflow. This provides some protection against a subflow being starved of data to transmit.

The scheduler returns an appropriate *ds_map* struct, and since the length allocated can exceed oneMSS, this mapping forms the basis of a multi-packet MPTCP DSS-map.

The packet scheduler will be modularised and extended with congestion control hooks in future updates, providing scope for more complex scheduling of maps.

IV. Conclusions and Future Work

This report introduced FreeBSD-MPTCP v0.3, a modification of the FreeBSD kernel enabling Multipath TCP [1] support. We outlined the motivation behind and potential benefits of using multipath TCP, and discussed key architectural elements of our design.

We expect to update and improve our MPTCP implementation in the future, and documentation will be updated as this occurs. We also plan on releasing a detailed design document that will provide more in-depth detail about the implementation. Code profiling and analysis of on-wire performance are also planned.

Our aim is to use this implementation as a basis for further research into MPTCP congestion control, as noted in Section II-D3.

Acknowledgements

This project has been made possible in part by a gift from the Cisco University Research Program Fund, a corporate advised fund of Silicon Valley Community Foundation.

References

[1] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 6824, Internet Engineering Task Force, 12 January 2013. [Online]. Available: http://tools.ietf.org/html/rfc6824

[2] G. Armitage and L. Stewart. (2013) Newtcp project website. [Online]. Available: http://caia.swin.edu.au/urp/newtcp/

[3] G. Armitage and N. Williams. (2013) Multipath tcp project website. [Online]. Available: http://caia.swin.edu.au/urp/newtcp/mptcp/

[4] O. Bonaventure. (2013) Multipath tcp linux kernel implementation. [Online]. Available: http://multipath-tcp.org/pmwiki.php

[5] D. Wischik, C. Raiciu, A. Greenhalgh and M. Handley, "Design, Implementation and Evaluation of Congestion Control for Multipath TCP," in *USENIX Symposium of Networked Systems Design and Implementation (NSDI'11)*, Boston, MA, 2012.

[6] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, V. Paxson, "Stream Control Transmission Protocol," RFC 2960, Internet Engineering Task Force, October 2000. [Online]. Available: http://tools.ietf.org/html/rfc2960

[7] P. Amer, M. Becke, T. Dreibholz, N. Ekiz, J. Iyengar, P. Natarajan, R. Stewart, M. Tuexen, "Load sharing for the stream control transmission protocol (SCTP)," Internet Draft, Internet Engineering Task Force, September 2012. [Online]. Available: http://tools.ietf.org/html/html/draft-tuexen-tsvwg-sctp-multipath-05

[8] A. Ford, C. Raiciu, M. Handley, S. Barré, and J.Iyengar, "Architectural Guidelines for Multipath TCP Development," RFC 6182, Internet Engineering Task Force, March 2011. [Online]. Available: http://tools.ietf.org/html/rfc6182

[9] C. Raiciu, C. Paasch, S. Barré, A. Ford, M. Honda, F. Duchène, O. Bonaventure and M. Handley, "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP," in *USENIX Symposium of Networked Systems Design and Implementation (NSDI'12)*, San Jose, California, 2012.

[10] S. Barré, C. Paasch, and O. Bonaventure, "Multipath tcp: From theory to practice," in *IFIP Networking, Valencia*, May 2011.

[11] C. Raiciu, S. Barré, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," in *SIGCOMM 2011, Toronto, Canada*, August 2011.

[12] C. Raiciu, M. Handley, and D. Wischik, "Coupled congestion control for multipath transport protocols," RFC 6356, Internet Engineering Task Force, October 2011. [Online]. Available: http://tools.ietf.org/html/rfc6356

[13] J. Postel, "Transmission Control Protocol," RFC 793, Internet Engineering Task Force, September 1981. [Online]. Available: http://tools.ietf.org/html/rfc793

Appendix B

Design Overview of Multipath TCP version 0.4 for FreeBSD-11

Originally publicly released as CAIA Technical Report 140822A on August 22, 2014. Available online at `http://caia.swin.edu.au/reports/140822A/CAIA-TR-140822A.pdf`

Design Overview of Multipath TCP version 0.4 for FreeBSD-11

Nigel Williams, Lawrence Stewart, Grenville Armitage
Centre for Advanced Internet Architectures, Technical Report 140822A
Swinburne University of Technology
Melbourne, Australia
njwilliams@swin.edu.au, lastewart@swin.edu.au, garmitage@swin.edu.au

***Abstract*—This report introduces FreeBSD-MPTCP v0.4, a modification to the FreeBSD-11 kernel that enables support for the IETF's emerging Multipath TCP (MPTCP) specification. We outline the motivation for (and potential benefits of) using MPTCP, and discuss key architectural elements of our design.**

***Index Terms*—CAIA, TCP, Multipath, Kernel, FreeBSD**

I. Introduction

Traditional TCP has two significant challenges – it can only utilise a single network path between source and destination per session, and (aside from the gradual deployment of explicit congestion notification) congestion control relies primarily on packet loss as a congestion indicator. Traditional TCP sessions must be broken and reestablished when endpoints shift their network connectivity from one interface to another (such as when a mobile device moves from 3G to 802.11, and thus changes its active IP address). Being bound to a single path also precludes multihomed devices from using any additional capacity that might exist over alternate paths.

TCP Extensions for Multipath Operation with Multiple Addresses (RFC6824) [1] is an experimental RFC that allows a host to spread a single TCP connection across multiple network addresses. Multipath TCP (MPTCP) is implemented within the kernel and is designed to be backwards compatible with existing TCP socket APIs. Thus it operates transparently from the perspective of the application layer and works with unmodified TCP applications.

As part of CAIA's NewTCP project [2] we have developed and released a prototype implementation of the MPTCP extensions for FreeBSD-11 [3]. In this report we describe the architecture and design decisions behind our version 0.4 implementation. At the time of writing, a Linux reference implementation is also available at [4].

The report is organised as follows: we briefly outline the origins and goals of MPTCP in Section II. In Section III we detail each of the main architectural changes required to support MPTCP in the FreeBSD 11 kernel. The report concludes with Section IV.

II. Background to Multipath TCP (MPTCP)

The IETF's Multipath TCP (MPTCP) working group[1] is focused on an idea that has emerged in various forms over recent years – namely, that a single transport session as seen by the application layer might be striped or otherwise multiplexed across multiple IP layer paths between the session's two endpoints. An over-arching expectation is that TCP-based applications see the traditional TCP API, but gain benefits when their session transparently utilises multiple, potentially divergent network layer paths. These benefits include being able to stripe data over parallel paths for additional speed (where multiple similar paths exist concurrently), or seamlessly maintaining TCP sessions when an individual path fails or as a mobile device's multiple underlying network interfaces come and go. The parts of an MPTCP session flowing over different network paths are known as *subflows*.

A. Benefits for multihomed devices

Contemporary computing devices such as smartphones, notebooks or servers are often multihomed (multiple network interfaces, potentially using different link layer technologies). MPTCP allows existing TCP-based applications to utilise whichever underlying interface (network path) is available at any given time, seamlessly maintaining transport sessions when endpoints shift their network connectivity from one interface to another.

When multiple interfaces are concurrently available, MPTCP enables the distribution of an application's

[1] http://datatracker.ietf.org/wg/mptcp/charter/

traffic across all or some of the available paths in a manner transparent to the application. Networks can gain traffic engineering benefits as TCP connections are steered via multiple paths (for instance away from congested links) using coupled congestion control [5]. Mobile devices such as smartphones and tablets can be provided with persistent connectivity to network services as they transition between different locales and network access media.

B. SCTP is not quite the same as MPTCP

It is worth noting that SCTP (stream control transmission protocol) [6] also supports multiple endpoints per session, and recent CMT work [7] enables concurrent use of multiple paths. However, SCTP presents an entirely new API to applications, and has difficulty traversing NATs and any middleboxes that expect to see only TCP, UDP or ICMP packets 'in the wild'. MPTCP aims to be more transparent than SCTP to applications and network devices.

C. Previous MPTCP implementation and development

Most early MPTCP work was supported by the EU's Trilogy Project[2], with key groups at University College London (UK)[3] and Université catholique de Louvain in Louvain-la-Neuve (Belgium)[4] publishing code, working group documents and research papers. These two groups are responsible for public implementations of MPTCP under Linux userland[5], the Linux kernel[6] and a simulation environment (htsim)[7]. Some background on the design, rationale and uses of MPTCP can be found in papers such as [8]–[11].

D. Some challenges posed by MPTCP

MPTCP poses a number of challenges.

1) Classic TCP application interface: The API is expected to present the single-session socket of conventional TCP, while underneath the kernel is expected to support the learning and use of multiple IP-layer identities for session endpoints. This creates a non-trivial implementation challenge to retrofit such functionality into existing, stable TCP stacks.

2) Interoperability and deployment: Any new implementation must interoperate with the reference implementation. The reference implementation has not yet had to address interoperation, and as such holes and assumptions remain in the protocol documents. An interoperable MPTCP implementation, given FreeBSD's slightly different network stack paradigm relative to Linux, should assist in IETF standardisation efforts. Also, the creation of a BSD-licensed MPTCP implementation benefits both the research and vendor community.

3) Congestion control (CC): Congestion control (CC) must be coordinated across the subflows making up the MPTCP session, to both effectively utilise the total capacity of heterogeneous paths and ensure a multipath session does not receive *"...more than its fair share at a bottleneck link traversed by more than one of its subflows"* [12]. The WG's current proposal for MPTCP CC remains fundamentally a loss-based algorithm that *"...only applies to the increase phase of the congestion avoidance state specifying how the window inflates upon receiving an ACK. The slow start, fast retransmit, and fast recovery algorithms, as well as the multiplicative decrease of the congestion avoidance state are the same as in standard TCP*" (Section 3, [12]). There appears to be wide scope for exploring how and when CC for individual subflows ought to be tied together or decoupled.

III. Changes to FreeBSD's TCP Stack

Our MPTCP implementation has been developed as a kernel patch[8] against revision 265307 of FreeBSD-11. Table I provides a summary of files modified or added to the FreeBSD-11 kernel.

A broad view of the changes and additions between revision 265307 and the MPTCP-enabled kernel:

1) Creation of the Multipath Control Block (MPCB) and the re-purposing of the existing TCP Control Block (TCPCB) to act as a MPTCP subflow control block.
2) Changes to user requests (called from the socket layer) that handle the allocation, setup and deallocation of control blocks.
3) New data segment reassembly routines and data-structures.
4) Changes to socket send and socket receive buffers to allow concurrent access from multiple subflows and mapping of data.

[2] http://www.trilogy-project.org/
[3] http://nrg.cs.ucl.ac.uk/mptcp/
[4] http://inl.info.ucl.ac.be/mptcp
[5] http://nrg.cs.ucl.ac.uk/mptcp/mptcp_userland_0.1.tar.gz
[6] https://scm.info.ucl.ac.be/trac/mptcp/
[7] http://nrg.cs.ucl.ac.uk/mptcp/htsim_0.1.tar.gz

[8] Implementing MPTCP as a loadable kernel module was considered, but deemed impractical due to the number of changes required.

File	Status
sys/netinet/tcp_var.h	Modified
sys/netinet/tcp_subr.c	Modified
sys/netinet/tcp_input.c	Modified
sys/netinet/tcp_output.c	Modified
sys/netinet/tcp_timer.c	Modified
sys/netinet/tcp_reass.c	Modified
sys/netinet/tcp_syncache.c	Modified
sys/netinet/tcp_usrreq.c	Modified
sys/netinet/mptcp_var.h	Added
sys/netinet/mptcp_subr.c	Added
sys/kern/uipc_sockbuf.c	Modified
sys/sys/sockbuf.h	Modified
sys/sys/socket.h	Modified
sys/sys/socketvar.h	Modified

Table I
KERNEL FILES MODIFIED OR ADDED AS PART OF MPTCP IMPLEMENTATION

5) MPTCP option insertion and parsing code for input and output paths.
6) Locking mechanisms to handle additional concurrency introduced by MPTCP.
7) Various MPTCP support functions (authentication, hashing etc).

The changes are covered in more detail in the following subsections. For detail on the overall structure and operation of the FreeBSD TCP/IP stack, see [13].

A. Protocol Control Blocks

The implementation adds a new control block, the MPTCP control block (MPCB), and re-purposes the TCP Control Block (RFC 793 [14]) as a subflow control block. The header file `netinet/mptcp_var.h` has been added to the FreeBSD source tree, and the MPCB structure is defined within.

A MPCB is created each time an application creates a TCP socket. The MPCB maintains all information required for multipath operation and manages the subflows in the connection. This also includes variables for data-level accounting and session tokens. It sits logically between the subflow TCP control blocks and the socket layer. This arrangement is compared with traditional TCP in Figure 1.

At creation, each MPCB associated with a socket contains at least one subflow (the *master,* or *default subflow*). The subflow control block is a modified traditional TCP control block found in `netinet/tcp_var.h`. Modifications to the control block include the addition of subflow flags, which are used to propagate subflow state to the MPCB (E.g. during packet scheduling).

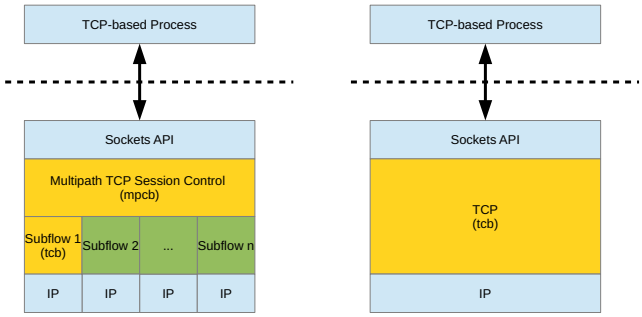


Figure 1. Logical MPTCP stack structure (left) versus traditional TCP (right). User space applications see same socket API.

Protocol control blocks are initialised and attached to sockets via functions in `netinet/tcp_usrreq.c` (user requests). A call to `tcp_connect()` in `netinet/tcp_usrreq.c` results in a call to `mp_newmpcb()`, which allocates and initialises the MPCB.

A series of functions (`tcp_subflow_*`) are implemented in `tcp_usrreq.c` and are used to create and attach any additional subflows to the MPTCP connection.

B. Asynchronous Task Handlers

Listing 1 Asynchronous tasks: Provide deferred execution for several MPTCP session-related tasks.

```
struct task join_task; /* For enqueuing
    aysnc joins in swi */
struct task data_task; /* For enqueuing
    aysnc subflow sched wakeup */
struct task pcb_create_task; /* For
    enqueueing async sf inp creation */
struct task rexmit_task; /* For enqueuing
    data-level rexmits */
```

When processing a segment, traditional TCP typically follows one of only a few paths through the TCP stack. For example, an incoming packet triggers a hardware interrupt, which causes an interrupt thread to be scheduled that, when executed, handles processing of the packet (including transport-layer processing, generating a response to the incoming packet).

Code executed in this path should be directly relevant to processing the current packet (parsing options, updating sequence numbers, etc). Operations such as copying out data to a process are deferred to other threads.

Maintaining a multipath session requires performing several new operations that may be triggered by incom-

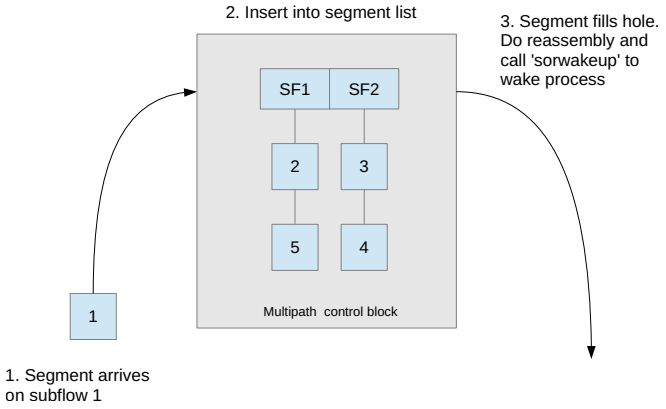


Figure 2. Each subflow maintains a segment receive list. Segments are placed into the list in subflow-sequence order as they arrive (data-level sequence numbers are shown). When a segment arrives in data-sequence order, the lists are locked and data-level re-ordering occurs. The application is then alerted and can read in the in-order data.

ing or outgoing packets. Some of these operations are not immediately related to the current packet therefore can be executed asynchronously. We have thus defined several new handlers for these tasks (Listing 1) which are attached to a software interrupt thread using *taskqueue*[9].

Each of the task variables has an associated handler in `netinet/mptcp_subr.c`, and provide the following functionality:

Join task (`mp_join_task_handler`): Attempt to join addresses the remote host has advertised.

Data task (`mp_datascheduler_task_handler`): Part of packet scheduling. Call output on subflows that are waiting to transmit.

PCB Create Task (`mp_sf_alloc_task_handler`): Allocate PCBs for subflows on an address we are about to advertise.

Retransmit Task (`mp_rexmit_task_handler`): Initiate data-level re-injection of segments after a subflow has failed to deliver data.

The *data task* and *retransmit task* are discussed further in Section III-F and Section III-G respectively.

C. Segment Reassembly

MPTCP adds a data-level sequence space above the sequence space used in standard TCP. This allows segments received on multiple subflows to be ordered before delivery to the application. Modifications to reassembly are found in `netinet/tcp_reass.c` and in `kern/uipc_socket.c`.

[9]http://www.freebsd.org/cgi/man.cgi?query=taskqueue

In pre-MPTCP FreeBSD, if a segment arrives that is not the next expected segment (sequence number does not equal receive next, `tcp_rcv_nxt`), it is placed into a reassembly queue. Segments are placed into this queue in sequence order until the expected segment arrives. At this point, all in-order segments held in the queue are appended to the socket receive buffer and the process is notified that data can be read in. If a segment arrives that is in-order and the reassembly list is empty, it is appended to the receive buffer immediately.

In our implementation, subflows do not access the socket receive buffer directly, and instead re-purpose the traditional reassembly queue for both in-order queuing and out-of-order reassembly. Unknown to subflows, their individual queues form part of a larger multipath-related reassembly data structure, shown in Figure 2.

All incoming segments on a subflow are appended to that subflow's reassembly queue (the `t_segq` member of the TCP control block defined in `netinet/tcp_var.h`) in subflow sequence order. When the head of a subflow's queue is in data sequence order (segment's data level sequence number is the data-level recieve next, `ds_rcv_nxt`), then data-level reassembly is triggered. In the current implementation, data-level reassembly is triggered from a kernel thread context. A future optimisation will see reassembly deferred to a userspace thread context (specifically that of the reading process).

Data-level reassembly involves traversing each subflow segment list and appending in-sequence (data-level) segments to the socket receive buffer. This occurs in the `mp_do_reass()` function of `netinet/tcp_reass.c`. During this time a write lock is used to exclude subflows from manipulating their reassembly queues.

Subflow and data-level reassembly have been split this way to reduce lock contention between subflows and the multipath layer. It also allows data-reassembly to be deferred to the application's thread context during a read on the socket, rather than performed by a kernel fast-path thread.

At completion of data-level reassembly, a data-level ACK is scheduled on whichever subflow next sends a regular TCP ACK packet.

D. Send and Receive Socket Buffers

In FreeBSD's implementation of standard TCP, segments are sent and received over a single (address,port) tuple, and socket buffers exist exclusively for each TCP session. MPTCP sessions have *1+n* (where n denotes

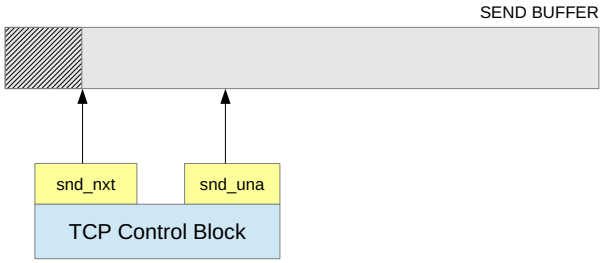


Figure 3. Standard TCP Send Buffer. The lined area represents sent bytes that have been acknowledged by the receiver.

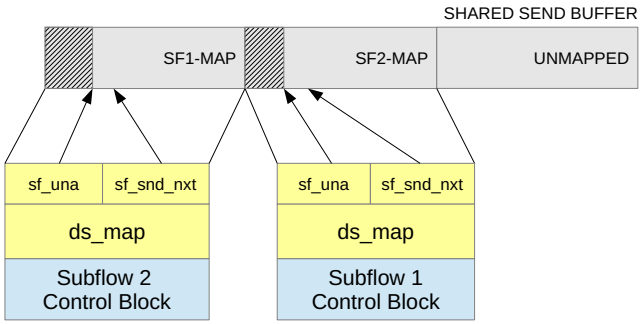


Figure 4. A MPTCP send buffer contains bytes that must be mapped to multiple TCP-subflows. Each subflow is allocated one or more *ds_maps* (DSS-MAP) that define these mappings.

additional addresses) subflows that must access the same send and receive buffers. The following sections describe the changes to the socket buffers and the addition of the `ds_map`.

1) The ds_map *Struct:* The `ds_map` struct (shown in Listing 2), is defined in `netinet/tcp_var.h`, and is used for both send-related and receive-related functions. Maps are stored in the subflow control block lists `t_txmaps` (send buffer maps) and `t_rxmaps` (received maps) respectively. A data-level list, `mp_rxtmitmaps`, is used to queue ds_maps that require retransmission after a data-level timeout. The struct itself contains variables for tracking sequence numbers, memory locations and status. It also includes several list entries (e.g `mp_ds_map_next`) as an instantiated map may belong to different (potentially multiple) lists, depending on the purpose.

On the send side, `ds_maps` track accounting information (bytes sent, acked) related to DSN maps advertised to the peer, and are used to access data in the socket send buffer (via for example `ds_map_offset`, `mbuf_offset`). By mediating socket buffer access through ds_maps in this way, rather than accessing the send buffer directly, lock contention can be reduced when sending data using multiple subflows. On the receive side, *`ds_maps`* are created via incoming DSS options and maintain mappings between subflow and sequence spaces. .

2) Socket Send Buffer: Figure 3 illustrates how in standard TCP, each session has exclusive access to its own send buffer. The variables `snd_nxt` and `snd_una` are used respectively to track which bytes in the send buffer are to be sent next, and which bytes were the last acknowledged by the receiver.

Figure 4 illustrates how in the multipath kernel, data from the sending application is still stored in a single send socket buffer. However access to this buffer is moderated by the packet scheduler in `mp_get_map()`, implemented in `netinet/mptcp_subr.c` (see Section III-F)

The packet scheduler is run when a subflow attempts to send data via `tcp_output()` without owning a `ds_map` that references unsent data. When invoked, the scheduler must decide whether the subflow should be allocated any data. If granted, allocations are returned as a *`ds_map`* that contains an offset into the send buffer and the length of data to be sent. Otherwise, a NULL map is returned, and the send buffer appears 'empty' to the subflow. The `ds_map` effectively acts as a unique socket buffer from the perspective of the subflow (i.e. subflows are not aware of what other subflows are sending). The scheduler is not invoked again until the allocated map has been completely sent.

This scheme allows subflows to make forward progress with variable overheads that depend on how frequently the scheduler is invoked i.e. larger maps reduce overheads.

As a result of sharing the underlying send socket buffer via `ds_maps` to avoid data copies, releasing acknowledged bytes becomes more complex. Firstly, data-level ACKs rather than subflow-level ACKs mark the multipath-level stream bytes which have safely arrived, and therefore control the advancement of ds_snd_una. Secondly, `ds_maps` can potentially overlap any portion of their socket buffer mapping with each other (e.g. data-level retransmit), and therefore the underlying socket buffer bytes (encapsulated in chained mbufs) can only be dropped when acknowledged at the data level and all maps which reference the bytes have been deleted.

To potentially defer the dropping of bytes from the socket buffer without adversely impacting application

Listing 2 ds_map struct

```
struct ds_map {
   TAILQ_ENTRY(ds_map) sf_ds_map_next;
   TAILQ_ENTRY(ds_map) mp_ds_map_next;
   TAILQ_ENTRY(ds_map) mp_dup_map_next;
   TAILQ_ENTRY(ds_map) rxmit_map_next;
   uint64_t ds_map_start; /* starting DSN of mapping */
   uint32_t ds_map_len; /* length of data sequence mapping */
   uint32_t ds_map_offset; /* bytes sent from mapping */
   tcp_seq sf_seq_start; /* starting tcp seq num of mapping */
   uint64_t map_una;  /* bytes sent but unacknowledged in map */
   uint16_t ds_map_csum; /* csum of dss psuedo-header & mapping data */
   struct mbuf* mbuf_start; /* mbuf in which this mappings starts */
   u_int mbuf_offset; /* offset into mbuf where data starts */
   uint16_t flags;    /* status flags */
};
...
/* Status flags for ds_maps */
#define MAPF_IS_SENT 0x0001 /* Sent all data from map */
#define MAPF_IS_ACKED 0x0002 /* All data in map is acknowledged */
#define MAPF_IS_DUP 0x0004 /* Duplicate, already acked at ds-level */
#define MAPF_IS_REXMIT 0x0008 /* Is a rexmit of a previously sent map */
```

throughput requires that socket buffer occupancy be accounted for logically rather than actually. To this end, the socket buffer variable `sb_cc` of an MPTCP socket send buffer refers to the logical number of bytes held in the buffer without data-level acknowledgment, and a new variable `sb_actual` has been introduced to track the actual number of bytes in the buffer.

3) Socket Receive Buffer: In pre-MPTCP FreeBSD, in-order segments were copied directly into the receive buffer, at which time the process was alerted that data was available to read. The remaining space in the receive buffer was used to advertise a receive window to the sender.

As described in Section III-C, each subflow now holds all received segments in a segment list, even if they are in subflow sequence order. The segment lists are then linked by their list heads to create a larger data-level reassembly data structure. When a segment arrives that is in data sequence order, data-level reassembly is triggered and segments are copied into the receive buffer.

We plan to integrate the multipath reassembly structure into the socket receive buffer in a future release. Coupled together with deferred reassembly, an application's thread context would be responsible for performing data-level reassembly on the multi-subflow aware buffer after being woken up by a subflow that received the next expected data-level segment (see Figure 5).

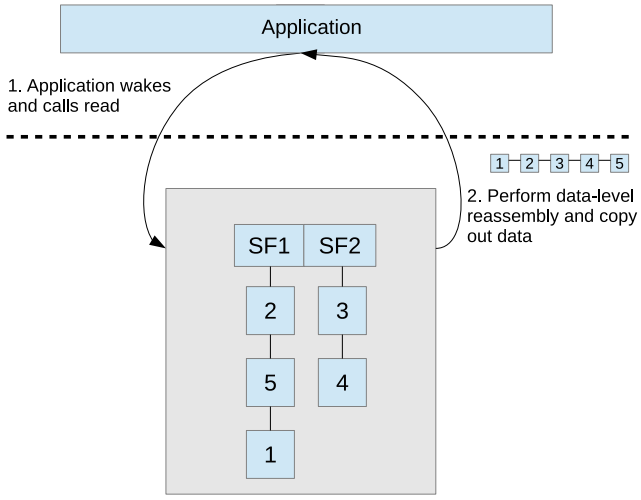


Figure 5. A future release will integrate the multipath reassembly structure into the socket receive buffer. Segments will be read directly from the multi-subflow aware buffer as data-level reassembly occurs.

E. Receiving DSS Maps and Acknowledgments

As mentioned in Section III-D1, the `ds_map` struct is used within the send and receive paths as well as packet scheduling. The struct allows the receiver to track incom-

ing data-maps, and the sender to track acknowledgement of data at subflow- and data- levels. The following subsections detail the primary uses of `ds_maps` in the send and receive paths.

1) Receiving data mappings: New *ds_maps* are created when a packet that contains a MPTCP DSS (Data-Sequence Signal) option that specifies a DSN-map (Data-Sequence Number) is received. Maps are stored within the subflow-level list `t_rxdsmaps` and are used to derive the DSN of an incoming TCP segment (in cases where a mapping spans multiple segments, the DSN will not be included with the transmitted packet). The processing of the DSS option (Figure 6), is summarised as follows:

1) If an incoming DSN-map is found during option parsing, it is compared to an existing list of mappings in `t_rxdmaps`. While looking for a matching map, any fully-acknowledged maps are discarded.
2) If the incoming data is found to be covered by an existing ds_map entry, the incoming DSN-map is disregarded and the existing map is selected. If the mapping represents new data, a new `ds_map` struct is allocated and inserted into the received map list.
3) The returned map - either newly allocated or existing - is used to calculate the appropriate DSN for the segment. The DSN is then "tagged" (see below) onto the mbuf header of the incoming segment.

The *mbuf_tags*[10] framework is used to attach DSN metadata to the incoming segment. Tags are attached to the mbuf header of the incoming packet, and can hold additional metadata (e.g. VLAN tags, firewall filter tags). A structure, `dsn_tag` (Listing 3) is defined in `netinet/mptcp_var.h` to hold the mbuf tag and the 64-bit DSN.

A dsn_tag is created for each packet, regardless of whether a MPTCP connection is active. For standard TCP connections this means the TCP sequence number of the packet is placed into the dsn_tag. Listing 4 shows use of the tags for active MPTCP connections.

Once a DSN has been associated with a segment, standard input processing continues. The DSN is eventually read during segment reassembly.

2) ACK processing and DS_Maps: The MPTCP layer separates subflow-level sequence space and the socket send buffers. As the same data may be mapped to multiple subflows, data cannot be freed from the send buffer until all references to it have been removed. A single ds_map is stored in both subflow-level and data-level transmit lists, and must be acknowledged at both levels before the data can be cleared from the send buffer.

Although subflow-level acknowledgment does not immediately result in the freeing of send buffer data, the data is considered 'delivered' from the perspective of the subflow. Subflow-level processing of ACKs is shown in Figure 7.

On receiving an ACK, the amount of data acknowledged is calculated and the list of transmitted maps, `t_txdmaps`, is traversed. Maps covered by the acknowledgement are marked as being 'acked' and are dropped from the transmitted maps list. At this point

Listing 3 dsn_tag struct: This structure is used to attach a calculated DSN to an incoming packet.

```
/* mbuf tag defines */
#define PACKET_TAG_DSN 10
#define PACKET_COOKIE_MPTCP 34216894
#define DSN_LEN 8

struct dsn_tag {
   struct m_tag tag;
   uint64_t  dsn;
};
```

Listing 4 Prepending a dsn_tag to a received TCP packet. The tag is used later during reassembly to order packets from multiple subflows. Unrelated code ommitted for brevity.

```
/* Initialise the mtag and containing
    dsn_tag struct */
struct dsn_tag *dtag = (struct dsn_tag *)
   m_tag_alloc(PACKET_COOKIE_MPTCP,
      PACKET_TAG_DSN, DSN_LEN, M_NOWAIT);
struct m_tag *mtag = &dtag->tag;
...
/* update mbuf tag with current data seq
    num */
dtag->dsn = map->ds_map_start +
    (th->th_seq - map->sf_seq_start);
...
/* And prepend the mtag to mbuf, to be
    checked in reass */
m_tag_prepend(m, mtag);
```

[10] http://www.freebsd.org/cgi/man.cgi?query=mbuf_tags

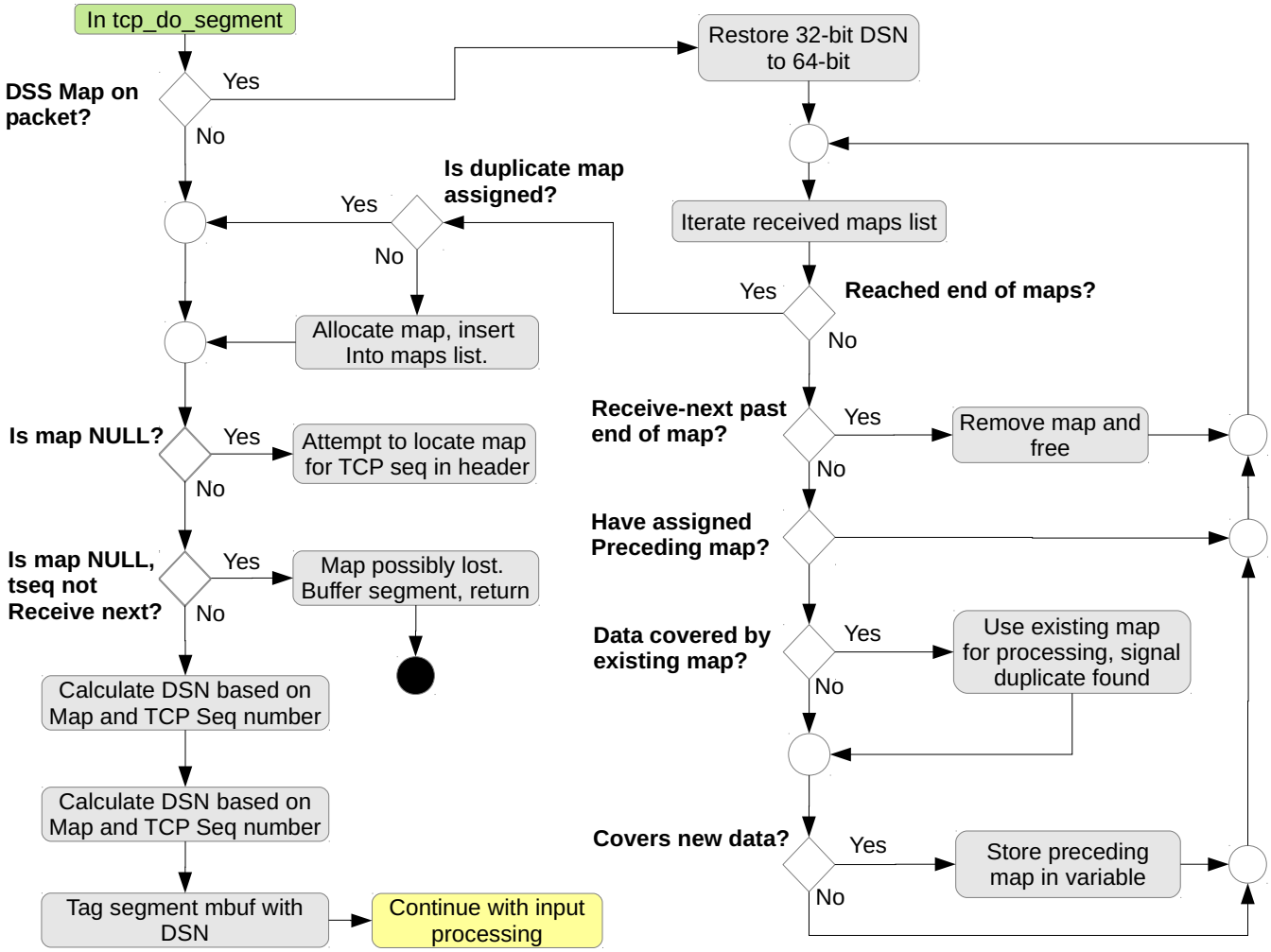


Figure 6. Receiver processing of DSN Maps. A list of ds_maps is used to track incoming packets and tag the mbuf with an appropriate DSN (mapping subflow-level to data-level).

a reference to dropped maps still exists within the data-level transmit list.

If any maps were completed, the `mp_deferred_drop()` function is called (detailed in Section III-E3 below). At this point the data has been successfully delivered, from the perspective of the subflow. It is the MPTCP layers responsibility to facilitate retransmission of data if it is not ultimately acknowledged at the data-level. Data-level acknowledgements (DACKs) are also processed at this time, if present.

3) Deferred drop from send buffer: The function `mp_deferred_drop()` in `netinet/mptcp_subr.c` handles the final accounting of sent data and allows acknowledged data to be dropped from the send buffer. The 'deferred' aspect refers to the fact that the time at which segments are acknowledged is no longer (necessarily) the time at which that data is freed from the send buffer. The process is shown in Figure 8, and broadly described below:

1) Iterate through transmitted maps and store a reference to maps that have been fully acknowledged. The loop is terminated at the end of the list, or if a map is encountered that overlaps the acknowledged region or shares an mbuf with another map that has not yet been acknowledged.
2) If there are bytes to be dropped, the corresponding maps are freed and the bytes are dropped from the socket send buffer. The process is woken up at this time to write new data. If there are no bytes to drop, all outstanding data has been acknowledged

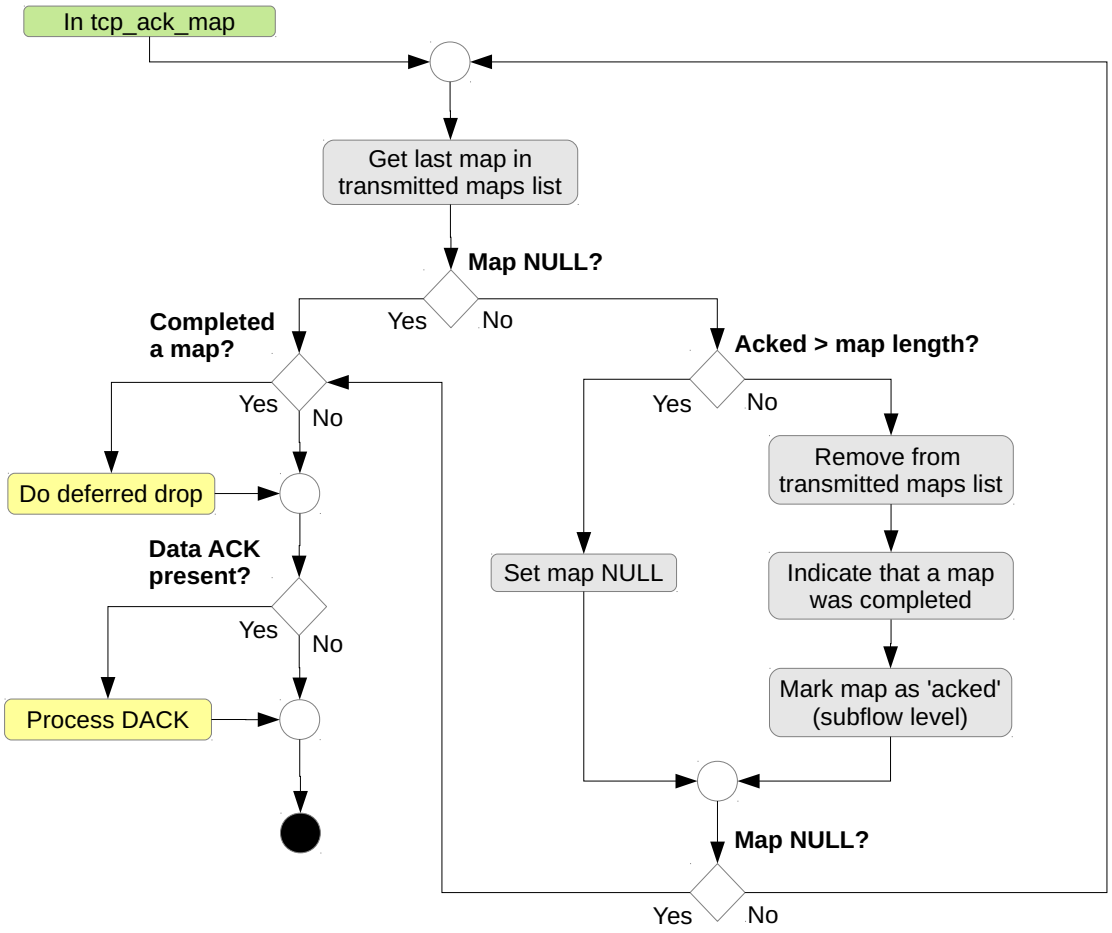


Figure 7. Transmitted maps must be acknowledged at the subflow- and data-levels. However, once acknowledged at the subflow level, the subflow considers the data as being 'delivered'.

and the send buffer is empty, the process is woken so that it may write new data.

F. Packet Scheduling

The packet scheduler is responsible for determining which subflows are able to send data from the socket send buffer, and how much data they can send. A basic packet scheduler is implemented in the v0.4 patch, and can be found within the `mp_find_dsmap()` function of `netinet/mptcp_subr.c` and `tcp_usr_send()` in `netinet/tcp_usrreq.c`. The current scheduler implementation controls two common pathways through which data segments can be requested for output - calls to `tcp_usr_send()` from the socket, and direct calls to `tcp_output()` from within the TCP stack (for example from `tcp_input()` on receipt of an ACK). The packet scheduler will be modularised in future updates, providing scope for more complex scheduling schemes.

Figure 9 shows these data transmission pathways, and the points at which scheduling decisions are made. To control which subflows are able to send data at a particular time the scheduler uses two subflow flags: `SFF_DATA_WAIT` and `SFF_SEND_WAIT`.

1) SFF_SEND_WAIT: On calls to `tcp_usr_send()`, the list of active subflows is traversed. The first subflow with `SFF_SEND_WAIT` set is selected as the subflow to send data on. The flag is cleared before calling `tcp_output()`.
2) SFF_DATA_WAIT: If a subflow is not allocated a map during a call to `tcp_output()`, the `SFF_DATA_WAIT` flag is set. An asynchronous task, `mp_datascheduler_task_handler` is enqueued when the number of subflows with this flag set is greater than zero. When run, the task will call `tcp_output()` with the waiting subflow.

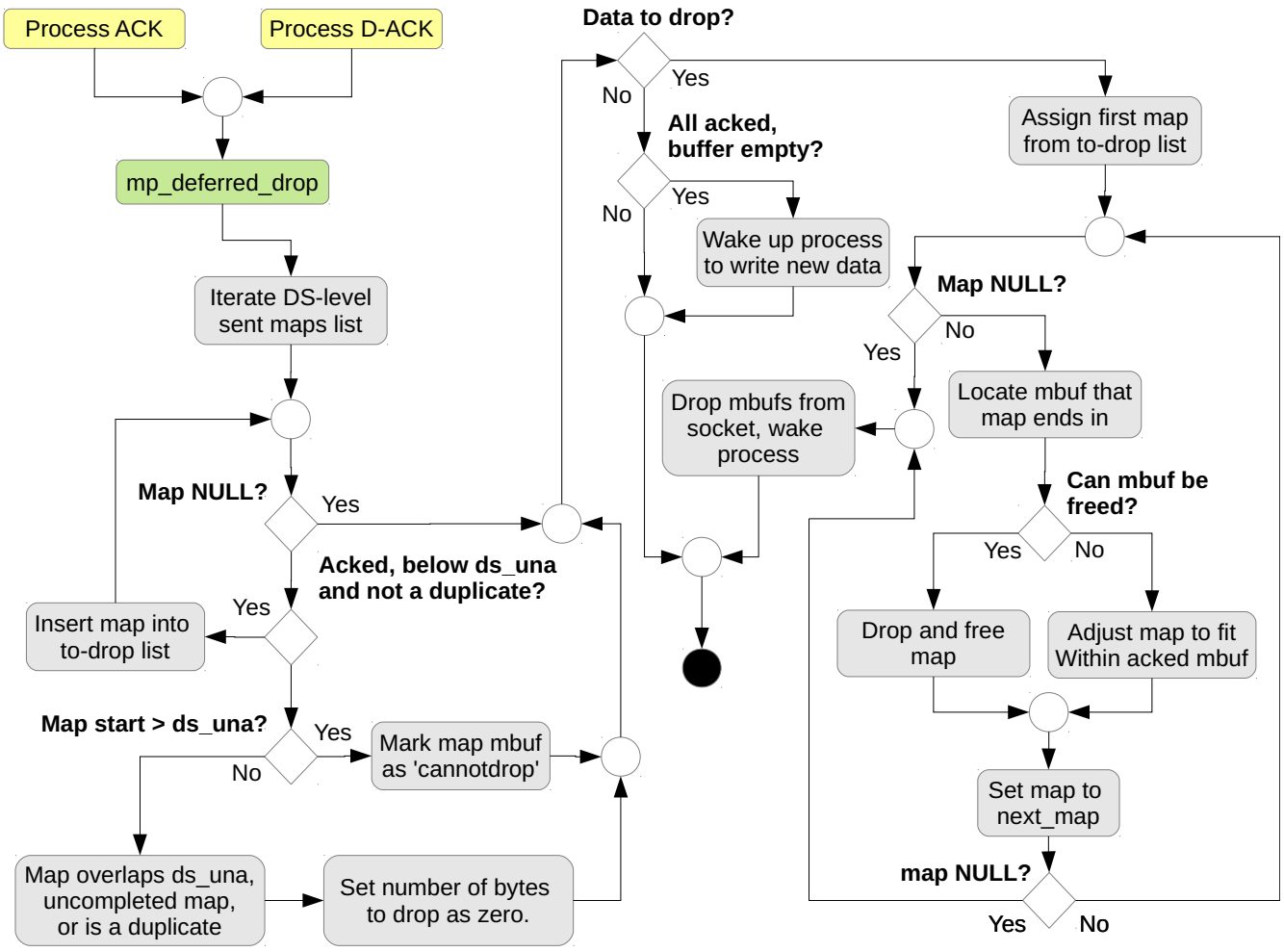


Figure 8. Deferred removal of data from the send buffer. Data bytes are dropped from the send buffer only when acknowledged at the data-level. It is considered deferred as the bytes are not necessarily dropped when acknowledged at the subflow level.

Subflow selection via SEND_WAIT: Figure 10 illustrates the use of the `SFF_SEND_WAIT` flag. When a process wants to send new data, it may use the `sosend()` Socket I/O function, which results in a call to the `tcp_usr_send()` function in `netinet/tcp_usrreq.c`. On entering `tcp_usr_send()` the default subflow protocol block ('master subflow') is assigned.

At this point the list of subflows (if greater than one) is traversed, checking subflow flags for `SFF_SEND_WAIT`. If not set, the flag is set before iterating to the next subflow. If set, the assigned subflow is switched, the loop terminated, and the flag is cleared before calling `tcp_output()`. If no subflows are found to be waiting for data, the 'master subflow' is used for transmission.

Subflow selection via DATA_WAIT: The `SFF_DATA_WAIT` flag is used in conjunction with an asynchronous task to divide ds_map allocation between the active subflows (Figure 11). When in `tcp_output()`, a subflow will call `find_dsmap()` to obtain a mapping of the send buffer. The process of allocating a map is shown in Figure 12. The current implementation restricts map sizes to 1420 bytes (limiting each map to cover one packet). In cases where no map was returned, the subflow flag is marked `SFF_DATA_WAIT`, and the count of 'waiting' subflows is increased. If a map was returned, then the `SFF_DATA_WAIT` flag is cleared (if set) and the 'waiting' count is decremented.

As map sizes are currently limited to a single-packet size, it is likely that on return from `mp_get_map()`

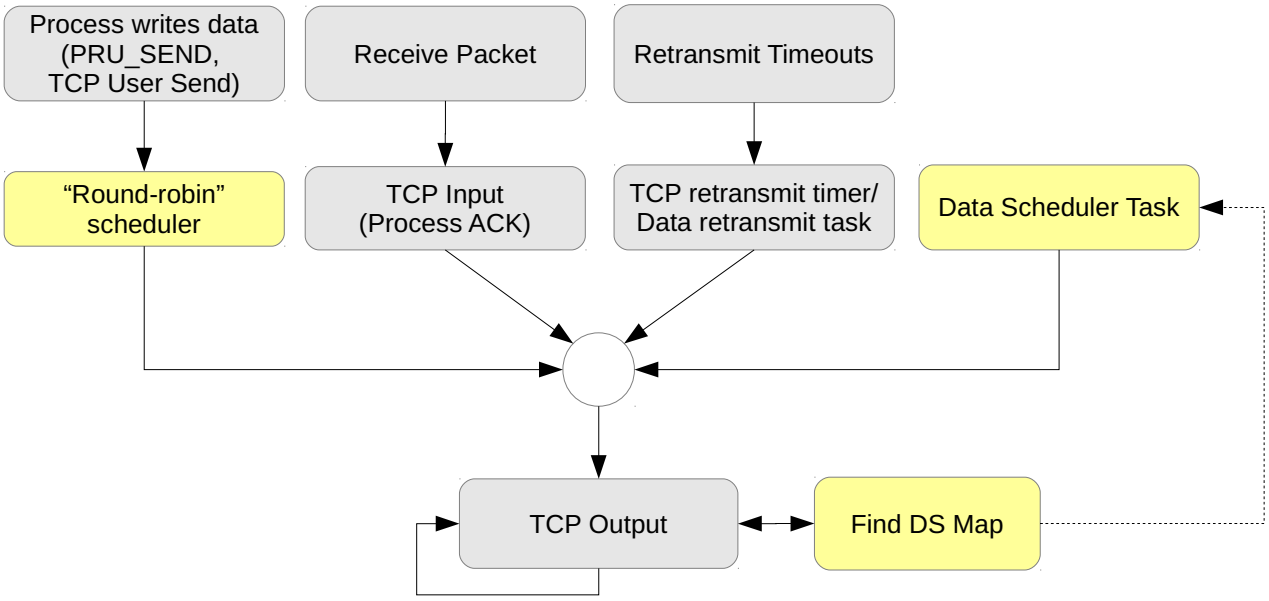


Figure 9. Common pathways to *tcp_output()*, the function through which data segments are sent. Packet scheduling components are shown in orange. Possible entry paths are via the socket (PRU_SEND), on receipt of an ACK or through a retransmit timer. The data scheduler task asynchronously calls into *tcp_output()* when there are subflows waiting to send data. Find DS Map is allocates *ds_maps* to a subflow, and can enqueue the data scheduler task.

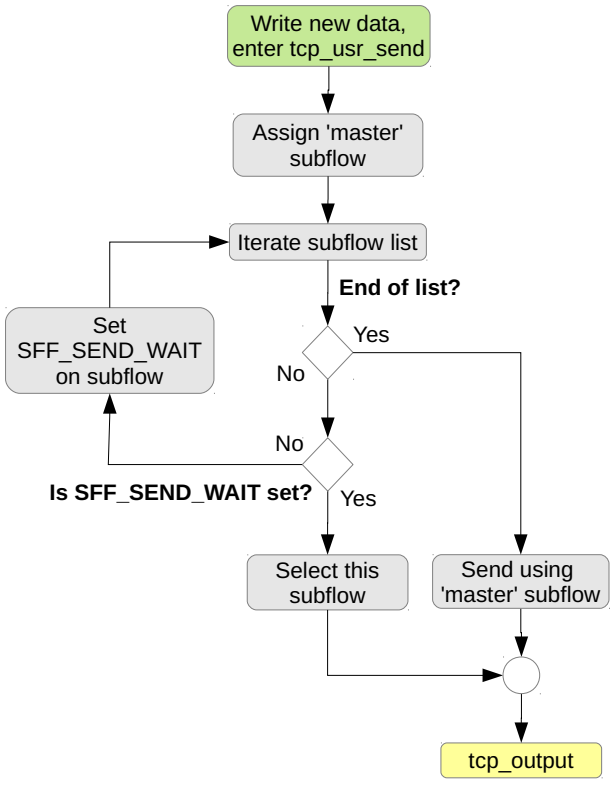


Figure 10. Round-robin scheduling. When a process writes new data to be sent, the scheduler selects a subflow on which to send data.

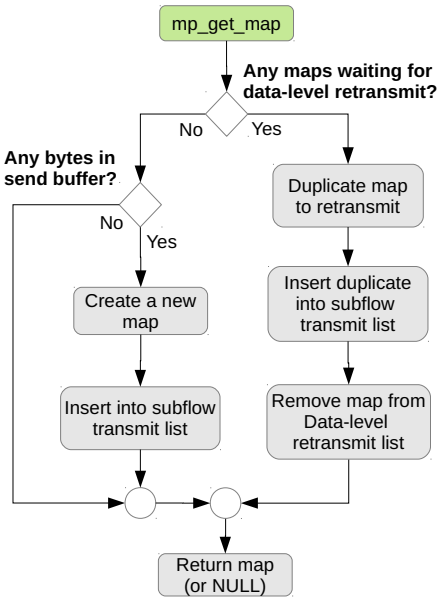


Figure 12. Allocating a ds_map in *mp_get_map()*. First check for maps that require retransmission. Otherwise, if unsent bytes are in the send buffer, a new map is allocated, inserted into the transmission list and returned.

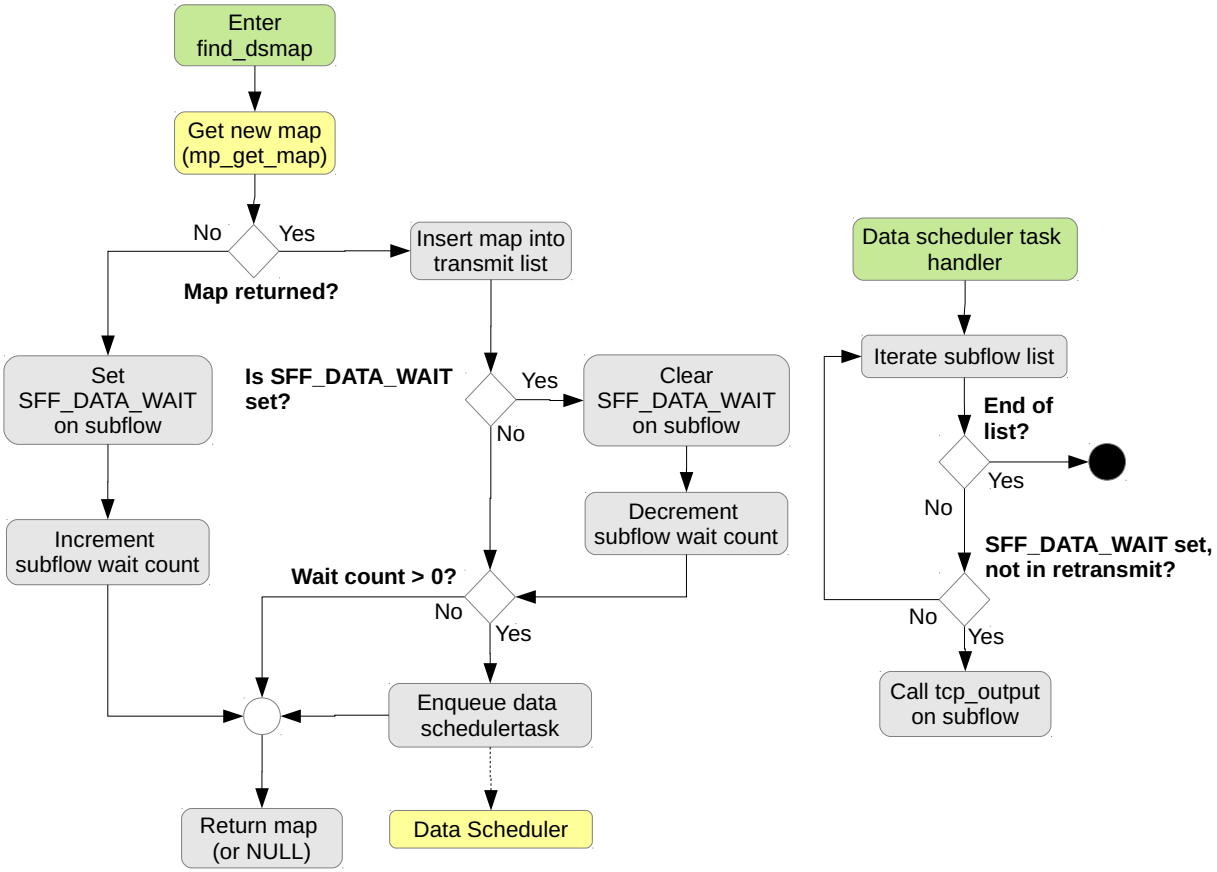


Figure 11. DATA_WAIT subflow selection. Enqueing the data scheduler (left) and the data scheduler (right). Rather than send data segments back-to-back on the same subflow, the scheduler spreads data across the available subflows.

unmapped data remains in the send buffer. Therefore a check is made for any 'waiting' subflows that might be used to send data, in which case a data scheduler asynchronous task is enqueued. When executed, the data scheduler task will call `tcp_output()` on the first subflow with `SFF_DATA_WAIT` set.

G. Data-level retransmission

Data-level retransmission of segments has been included in the v0.4 patch (Figure 13). The current implementation triggers data-level retransmissions based on a count of subflow-level retransmits. In future updates the retransmission strategy will be modularised.

The chart on the left of Figure 13 shows the steps leading to data-level retransmit. Each subflow maintains a retransmission timer that is started on packet transmission and stopped on acknowledgement. If left to expire (called a retransmit timeout, or RTO), the function `tcp_timer_rexmt()` in `netinet/tcp_timer.c` is called, and the subflow will attempt to retransmit from the last bytes acknowledged. The length of the timeout is based in part on the RTT of the path. A count is kept each time an RTO occurs, up to `TCP_MAXRXTSHIFT` (defined as 12 in FreeBSD), at which point the connection is dropped with a RST.

We define a threshold of: `TCP_MAXRXTSHIFT / 4` (or 12/4, giving 3 timeouts) as the point at which data-level retransmission will occur. A check has been placed into `tcp_timer_rexmt()` that tests whether the count of RTOs has met this threshold. If met, a reference to each ds_map that has not been acknowledged at the data-level is placed into `mp_rxtmitmaps` (a list of maps that require data-level re-injection). Finally, an asynchronous task is enqueued (Figure 13, right) that, when executed, locates the first subflow that is not in subflow-level retransmit and calls `tcp_output()`. The packet scheduler will ensure that ds_maps in `mp_rxtmitmaps` are sent before any existing `ds_maps` in the subflow transmit list (*`t_txdsmaps`*).

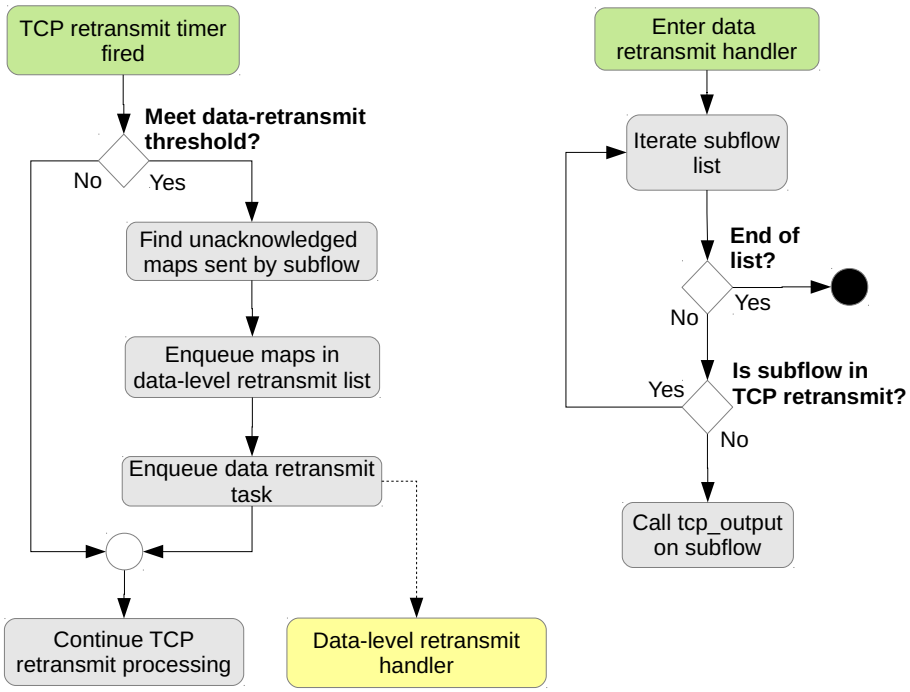


Figure 13. Data-level retransmit logic (left) and task handler (right). Retransmission is keyed off the count of TCP (subflow-level) retransmit timeouts.

It should be noted that subflows retain standard TCP retransmit behaviour independent of data-level retransmits. Subflows will therefore continue to attempt retransmission until the maximum retransmit count is met. On occasions where a subflow recovers from retransmit timeout after data-level retransmission, the receiver will acknowledge the data at the subflow level and discard the duplicate segments.

H. Multipath Session Management

The current implementation contains basic mechanisms for joining subflows and subflow/connection termination, detailed below. Path management will be expanded and modulularised in future updates.

1) Adding subflows: An address can be manually specified for use in a connection between a multi-homed host and single-homed host. This is done using the sysctl[11] utility. Added addresses are available to all MPTCP connections on the host, and will be advertised by all MPTCP connections that reach the established stage.

Subflow joining behaviour is static, and a host will attempt to send an MP_JOIN to any addresses that are received via the ADD_ADDR option[12]. The asynchronous tasks `mp_join_task_handler` and `mp_sf_alloc_task_handler` currently provide this functionality. Both will be integrated with a Path Manager module in a future release.

2) Removing Subflows/Connection Close: The implementation supports removal of subflows from an active MPTCP connection only via TCP reset (RST) due to excessive retransmit timeouts. In these cases, a subflow that has failed will proceed through the standard TCP timeout procedure (as implemented in FreeBSD-11) before closing. Any remaining active subflows will continue to send and receive data. There is currently no other means by which to actively terminate a single subflow on a connection.

On application close of the socket all subflows are shut down simultaneously. The last subflow to be closed will cause the MPCB to be discarded. Subflows on the same host are able to take separate paths (active close, passive close) through the TCP shutdown states.

3) Session Termination: Not documented in this report are modifications to the TCP shutdown code paths.

[11] sysctl net.inet.tcp.mptcp.mp_addresses

[12] One caveat exists: If a host is the active opener (client) in the connection and has already advertised an address, it will not attempt to join any addresses that it receives via advertisement.

Currently the code has been extended in-place with additional checks to ensure that socket is not marked as closed while at least one subflow is still active. These modifications should however be considered temporary and will be replaced with a cleaner solution in a future update.

IV. Conclusions and Future Work

This report describes FreeBSD-MPTCP v0.4, a modification of the FreeBSD kernel enabling Multipath TCP [1] support. We outlined the motivation behind and potential benefits of using Multipath TCP, and discussed key architectural elements of our design.

We expect to update and improve our MPTCP implementation in the future, and documentation will be updated as this occurs. We also plan on releasing a detailed design document that will provide more in-depth detail about the implementation. Code profiling and analysis of on-wire performance are also planned.

Our aim is to use this implementation as a basis for further research into MPTCP congestion control, as noted in Section II-D3.

Acknowledgements

This project has been made possible in part by a gift from the Cisco University Research Program Fund, a corporate advised fund of Silicon Valley Community Foundation.

References

[1] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 6824, Internet Engineering Task Force, 12 January 2013. [Online]. Available: http://tools.ietf.org/html/rfc6824

[2] G. Armitage and L. Stewart. (2013) Newtcp project website. [Online]. Available: http://caia.swin.edu.au/urp/newtcp/

[3] G. Armitage and N. Williams. (2013) Multipath tcp project website. [Online]. Available: http://caia.swin.edu.au/urp/newtcp/mptcp/

[4] O. Bonaventure. (2013) Multipath tcp linux kernel implementation. [Online]. Available: http://multipath-tcp.org/pmwiki.php

[5] D. Wischik, C. Raiciu, A. Greenhalgh and M. Handley, "Design, Implementation and Evaluation of Congestion Control for Multipath TCP," in *USENIX Symposium of Networked Systems Design and Implementation (NSDI'11)*, Boston, MA, 2012.

[6] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, V. Paxson, "Stream Control Transmission Protocol," RFC 2960, Internet Engineering Task Force, October 2000. [Online]. Available: http://tools.ietf.org/html/rfc2960

[7] P. Amer, M. Becke, T. Dreibholz, N. Ekiz, J. Iyengar, P. Natarajan, R. Stewart, M. Tuexen, "Load sharing for the stream control transmission protocol (SCTP)," Internet Draft, Internet Engineering Task Force, September 2012. [Online]. Available: http://tools.ietf.org/html/html/draft-tuexen-tsvwg-sctp-multipath-05

[8] A. Ford, C. Raiciu, M. Handley, S. Barré, and J.Iyengar, "Architectural Guidelines for Multipath TCP Development," RFC 6182, Internet Engineering Task Force, March 2011. [Online]. Available: http://tools.ietf.org/html/rfc6182

[9] C. Raiciu, C. Paasch, S. Barré, A. Ford, M. Honda, F. Duchène, O. Bonaventure and M. Handley, "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP," in *USENIX Symposium of Networked Systems Design and Implementation (NSDI'12)*, San Jose, California, 2012.

[10] S. Barré, C. Paasch, and O. Bonaventure, "Multipath tcp: From theory to practice," in *IFIP Networking, Valencia*, May 2011.

[11] C. Raiciu, S. Barré, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," in *SIGCOMM 2011, Toronto, Canada*, August 2011.

[12] C. Raiciu, M. Handley, and D. Wischik, "Coupled congestion control for multipath transport protocols," RFC 6356, Internet Engineering Task Force, October 2011. [Online]. Available: http://tools.ietf.org/html/rfc6356

[13] G. Wright, W. Stevens, *TCP/IP Illustrated, Volume 2, The Implementation*. Addison Wesley, 2004.

[14] J. Postel, "Transmission Control Protocol," RFC 793, Internet Engineering Task Force, September 1981. [Online]. Available: http://tools.ietf.org/html/rfc793

References

[1] J. Postel, "Rfc 793: Transmission control protocol, september 1981," *Status: Standard*, vol. 88, 2003. (document), 1, 2.1.2, 2.1.2

[2] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 6824, Internet Engineering Task Force, Jan. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6824.txt (document), 2.1.2, 3.3.3, 4.2, 4.2.1, 5, 5.3, 7.2.5

[3] "Internet domain survey, january, 2016," https://ftp.isc.org/www/survey/reports/current/, accessed: 2016-10-23. 1

[4] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," Tech. Rep., 2011. 1

[5] V. Cerf, Y. Dalal, and C. Sunshine, "Specification of internet transmission control program," Tech. Rep., 1974. 1

[6] J. R. Iyengar, P. D. Amer, and R. Stewart, "Concurrent multipath transfer using sctp multihoming over independent end-to-end paths," *IEEE/ACM Trans. Netw.*, vol. 14, no. 5, pp. 951–964, Oct. 2006. [Online]. Available: http://dx.doi.org/10.1109/TNET.2006.882843 1, 3.3.3, 4.1.1, 4.1.3

[7] "The Point-to-Point Protocol (PPP)," RFC 1661, Internet Engineering Task Force, Jul. 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1661.txt 1, 3.3.1

[8] O. Mehani, R. Holz, S. Ferlin, and R. Boreli, "An early look at multipath tcp deployment in the wild," in *Proceedings of the 6th International Workshop on Hot Topics in Planet-Scale Measurement*. ACM, 2015, pp. 7–12. 1, 3.4, 4.3

[9] O. Bonaventure, “Commercial usage of multipath tcp,” http://blog.multipath-tcp.org/blog/html/2015/12/25/commercial_usage_of_multipath_tcp.html, UniversitÃ© catholique de Louvain, accessed: 2016-10-20. 1

[10] C. Raiciu, M. Handley, and D. Wischik, “Coupled Congestion Control for Multipath Transport Protocols,” RFC 6356, Internet Engineering Task Force, Oct. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6356.txt 1, 4.2, 4.2, 4.2.1, 4.2.2

[11] N. Williams, “Source repository: caia-mptcp-freebsd,” https://bitbucket.org/nw-swin/caia-mptcp-freebsd, accessed: 2016-10-04. 1, 8, 8.1

[12] “Gnu general public license,” https://www.gnu.org/licenses/gpl-3.0.en.html, accessed: 2016-10-04. 1

[13] “Requirements for Internet hosts - Communication Layers,” RFC 1122, Internet Engineering Task Force, Oct. 1989. [Online]. Available: http://www.ietf.org/rfc/rfc1122.txt 2.1, 2.1.1

[14] S. Floyd and M. Allman, “Comments on the Usefullness of Simple Best-Effort Traffic,” RFC 5290, Internet Engineering Task Force, Jul. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5290.txt 2.1, 2.1.2

[15] “Internet Engineering Task Force,” http://www.ietf.org, Internet Engineering Task Force, Mar. 2016. [Online]. Available: http://www.ietf.org 2.1.1

[16] R. Fielding and J. Reschke, “Hypertext transfer protocol (http/1.1): Message syntax and routing,” 2014. 2.1.1

[17] P. V. Mockapetris, “Domain names-concepts and facilities,” 1987. 2.1.1

[18] M. Zhang, M. Dusi, W. John, and C. Chen, “Analysis of udp traffic usage on internet backbone links,” in *Proceedings of the 2009 Ninth Annual International Symposium on Applications and the Internet*, ser. SAINT ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 280–281. [Online]. Available: http://dx.doi.org/10.1109/SAINT.2009.65 2.1.1, 2.2.2

[19] “Internet Protocol,” RFC 791, Internet Engineering Task Force, Sep. 1981. [Online]. Available: http://www.ietf.org/rfc/rfc791.txt 2.1.1, 2.2.3

[20] "Cisco visual networking index: Forecast and methodology, 2014–2019," http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.pdf, Tech. Rep., may 2015. 2.1.2

[21] "Service Name and Transport Protocol Port Number Registry," http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml, Internet Assigned Numbers Authority, Mar. 2016. [Online]. Available: http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml 2.1.2

[22] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," RFC 5681, Internet Engineering Task Force, Sep. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5681.txt 2.1.2, 4.2.2

[23] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "Timely: Rtt-based congestion control for the datacenter," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 537–550, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2829988.2787510 2.1.2

[24] L. S. Brakmo and L. L. Peterson, "Tcp vegas: End to end congestion avoidance on a global internet," *IEEE Journal on selected Areas in communications*, vol. 13, pp. 1465–1480, 1995. 2.1.2

[25] K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," RFC 3168, Internet Engineering Task Force, Sep. 2001. [Online]. Available: http://www.ietf.org/rfc/rfc3168.txt 2.1.2

[26] B. Trammell, M. Kühlewind, D. Boppart, I. Learmonth, G. Fairhurst, and R. Scheffenegger, "Enabling internet-wide deployment of explicit congestion notification," in *International Conference on Passive and Active Network Measurement*. Springer, 2015, pp. 193–205. 2.1.2

[27] D. Borman, B. Braden, and V. Jacobson, "TCP Extensions for High Performance," RFC 7323, Internet Engineering Task Force, Sep. 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7323.txt 2.1.2

[28] ——, "TCP Selective Acknowledgment Options," RFC 2018, Internet Engineering Task Force, Oct. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2018.txt 2.1.2

[29] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend tcp?" in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '11. New York, NY, USA: ACM, 2011, pp. 181–194. [Online]. Available: http://doi.acm.org/10.1145/2068816.2068834 2.1.3, 5.1

[30] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and Issues," RFC 3234, Internet Engineering Task Force, 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3234.txt 2.1.3

[31] J. Sherry and S. Ratnasamy, "A survey of enterprise middlebox deployments," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-24, Feb 2012. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-24.html 2.1.3

[32] B. Mukherjee, L. T. Heberlein, and K. N. Levitt, "Network intrusion detection," *IEEE Network*, 1994. 2.1.3

[33] M. Handley, V. Paxson, and C. Kreibich, "Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. Berkeley, CA, USA: USENIX Association, 2001, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267612.1267621 2.1.3

[34] R. Fonseca, G. M. Porter, R. H. Katz, S. Shenker, and I. Stoica, "Ip options are not an option," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2005-24, Dec 2005. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-24.html 2.1.3

[35] J. Iyengar, I. Swett, R. Hamilton, and A. Wilk, "QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2," Internet Engineering Task Force, Internet-Draft draft-tsvwg-quic-protocol-02, Jan. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02 2.1.3

[36] M. Tuexin and R. Stewart, "UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication," RFC 6951, Internet Engineering Task Force, May 2013. [Online]. Available: https://tools.ietf.org/rfc/rfc6951.txt 2.1.3, 3.3.3

[37] W. John, M. Dusi, and K. C. Claffy, "Estimating routing symmetry on single links by passive flow measurements," in *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, ser. IWCMC '10. New York, NY, USA: ACM, 2010, pp. 473–478. [Online]. Available: http://doi.acm.org/10.1145/1815396.1815506 2.2.1, 4.1.2

[38] E. Stephan, "IP Performance Metrics (IPPM) Metrics Registry," RFC 4148, Internet Engineering Task Force, Aug. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4148.txt 2.2.1

[39] "Metrics for the Evaluation of Congestion Control Mechanisms," RFC 5166, Internet Engineering Task Force, Mar. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5166.txt 2.2.1, 2.2.2

[40] M. Allman and V. Paxson, "On estimating end-to-end network path properties," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 263–274, Aug. 1999. [Online]. Available: http://doi.acm.org/10.1145/316194.316230 2.2.1, 2.2.2

[41] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu, "Characterizing residential broadband networks," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '07. New York, NY, USA: ACM, 2007, pp. 43–56. [Online]. Available: http://doi.acm.org/10.1145/1298306.1298313 2.2.1, 2.2.2, 2.2.3

[42] R. Caceres and L. Iftode, "Improving the performance of reliable transport protocols in mobile computing environments," *IEEE J.Sel. A. Commun.*, vol. 13, no. 5, pp. 850–857, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1109/49.391749 2.2.1

[43] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz, "Improving tcp/ip performance over wireless networks," in *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '95. New York, NY, USA: ACM, 1995, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/215530.215544 2.2.1

[44] G. Hasslinger, J. Mende, R. Geib, T. Beckhaus, and F. Hartleb, "Measurement and characteristics of aggregated traffic in broadband access networks," in *Proceedings of the 20th International Teletraffic Conference on Managing Traffic Performance in Converged Networks*, ser. ITC20'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 998–1010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1769187.1769293 2.2.1

[45] G. Maier, A. Feldmann, V. Paxson, and M. Allman, "On dominant characteristics of residential broadband internet traffic," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '09. New York, NY, USA: ACM, 2009, pp. 90–102. [Online]. Available: http://doi.acm.org/10.1145/1644893.1644904 2.2.1

[46] N. Brownlee and K. C. Claffy, "Understanding internet traffic streams: Dragonflies and tortoises," *Comm. Mag.*, vol. 40, no. 10, pp. 110–117, Oct. 2002. [Online]. Available: http://dx.doi.org/10.1109/MCOM.2002.1039865 2.2.2

[47] K.-c. Lan and J. Heidemann, "A measurement study of correlations of internet flow characteristics," *Comput. Netw.*, vol. 50, no. 1, pp. 46–62, Jan. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2005.02.008 2.2.2

[48] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 323–336, Aug. 2002. [Online]. Available: http://doi.acm.org/10.1145/964725.633056 2.2.2

[49] B. Braden, D. Clark, and J. Crowcroft, "Recommendations on Queue Management and Congestion Avoidance in the Internet," RFC 2309, Internet Engineering Task Force, Oct. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2309.txt 2.2.2, 2.2.3

[50] J. Esteban, S. A. Benno, A. Beck, Y. Guo, V. Hilt, and I. Rimac, "Interactions between http adaptive streaming and tcp," in *Proceedings of the 22Nd International Workshop on Network and Operating System Support for Digital Audio and Video*, ser. NOSSDAV '12. New York, NY, USA: ACM, 2012, pp. 21–26. [Online]. Available: http://doi.acm.org/10.1145/2229087.2229094 2.2.2

[51] R. van Brandenburg, O. van Deventer, F. L. Faucheur, and K. Leung, "Models for HTTP-Adaptive-Streaming-Aware Content Distribution Network Interconnection (CDNI)," RFC 6983, Internet Engineering Task Force, Jul. 2013. [Online]. Available: https://tools.ietf.org/rfc/rfc6983.txt 2.2.2

[52] Sandvine, "Global internet phenomena report 2h 2014," https://www.internetsociety.org/globalinternetreport/, Tech. Rep., 2014. 2.2.2

[53] R. Morris, "Tcp behavior with many flows." in *ICNP*. IEEE Computer Society, 1997, pp. 205–211. 2.2.2

[54] J. R. Iyengar, O. L. Caro, and P. D. Amer, "Dealing with short tcp flows: A survey of mice in elephant shoes," 2007. 2.2.2

[55] C. Holman, J. But, and P. Branch, "The effect of round trip time on competing TCP flows," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 120405B, 05 April 2012. [Online]. Available: http://caia.swin.edu.au/reports/120405B/CAIA-TR-120405B.pdf 2.2.2

[56] K. Kurata, G. Hasegawa, and M. Murata, "Fairness comparisons between tcp reno and tcp vegas for future deployment of tcp vegas," in *Proceedings of INET 2000*, 2000. 2.2.2

[57] J. Mo, R. J. La, V. Anantharam, and J. Walrand, "Analysis and comparison of tcp reno and vegas," in *In Proceedings of IEEE Infocom*, 1999, pp. 1556–1563. 2.2.2

[58] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: Illuminating the edge network," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 246–259. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879173 2.2.2

[59] M. Claypool, R. E. Kinicki, M. Li, J. Nichols, and H. Wu, "Inferring queue sizes in access networks by active measurement," in *Passive and Active Network Measurement, 5th International Workshop, PAM 2004, Antibes Juan-les-Pins, France, April 19-20, 2004, Proceedings*, 2004, pp. 227–236. 2.2.2

[60] K. Nichols and V. Jacobson, "Controlling queue delay," *Queue*, vol. 10, no. 5, pp. 20:20–20:34, May 2012. [Online]. Available: http://doi.acm.org/10.1145/2208917.2209336 2.2.2

[61] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet," *Queue*, vol. 9, no. 11, pp. 40:40–40:54, Nov. 2011. [Online]. Available: http://doi.acm.org/10.1145/2063166.2071893 2.2.2

[62] C. Villamizar and C. Song, "High performance tcp in ansnet," *SIGCOMM Comput. Commun. Rev.*, vol. 24, no. 5, pp. 45–60, Oct. 1994. [Online]. Available: http://doi.acm.org/10.1145/205511.205520 2.2.2

[63] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 281–292, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1030194.1015499 2.2.2

[64] C. D. A. Dhamdhere, H. Jiang, "Tcp performance under aggregate fair queueing," vol. 2. IEEE, 2005, pp. 1072–1083. 2.2.2

[65] M. Enachescu, A. Goel, T. Roughgarden, Y. Ganjali, and N. Mckeown, "Routers with very small buffers," in *in IEEE Infocom*, 2006. 2.2.2

[66] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon, "Experimental study of router buffer sizing," in *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '08. New York, NY, USA: ACM, 2008, pp. 197–210. [Online]. Available: http://doi.acm.org/10.1145/1452520.1452545 2.2.2

[67] J. Moy, "OSPF Version 2," RFC 2328, Internet Engineering Task Force, Oct. 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2328.txt 2.2.3

[68] B. Fortz, J. Rexford, and M. Thorup, "Traffic engineering with traditional ip routing protocols," *IEEE Communications Magazine*, vol. 40, pp. 118–124, 2002. 2.2.3

[69] R. Nagle, "Traffic Engineering (TE) Extensions to OSPF Version 2," RFC 3630, Internet Engineering Task Force, Sep. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3630.txt 2.2.3

[70] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," RFC 2992, Internet Engineering Task Force, Nov. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2992.txt 2.2.3

[71] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," RFC 3031, Internet Engineering Task Force, Jan. 2001. [Online]. Available: http://www.ietf.org/rfc/rfc3031.txt 2.2.3

[72] "LDP Specification," RFC 5036, Internet Engineering Task Force, Oct. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5036.txt 2.2.3

[73] "Deprecation of Type 0 Routing Headers in IPv6," RFC 5059, Internet Engineering Task Force, Dec. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5036.txt 2.2.3

[74] R. H. S. Deering, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, Internet Engineering Task Force, Dec. 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2460.txt 2.2.3

[75] F. Gont, "Security Assessment of the Internet Protocol Version 4," RFC 6274, Internet Engineering Task Force, Jul. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6274.txt 2.2.3

[76] A. E. P. Biondi, "Ipv6 routing header security," in *CanSecWest Security Conference 2007*. Vancouver, BC, Canada: IEEE Computer Society, april 2007. 2.2.3

[77] J. Hui, J. Vasseur, D. Culler, and V. Manral, "An ipv6 routing header for source routes with the routing protocol for low-power and lossy networks (rpl)," Internet Engineering Task Force, 2012. 2.2.3

[78] B. Raghavan and A. C. Snoeren, "A system for authenticated policy-compliant routing," in *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '04. New York, NY, USA: ACM, 2004, pp. 167–178. [Online]. Available: http://doi.acm.org/10.1145/1015467.1015487 2.2.3

[79] R. Nagle, "On Packet Switches With Infinite Storage," RFC 970, Internet Engineering Task Force, Dec. 1985. [Online]. Available: http://www.ietf.org/rfc/rfc970.txt 2.2.3

[80] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, "The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm," Internet-Draft, Internet Engineering Task Force, Mar. 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-aqm-fq-codel-06 2.2.3

[81] V. J. K. Nichols, "Controlled Delay Active Queue Management," Internet-Draft, Internet Engineering Task Force, Mar. 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-aqm-codel-03 2.2.3

[82] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," RFC 2475, Internet Engineering Task Force, Dec. 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2475.txt 2.2.3

[83] B. Davie, A. Charny, J. Bennett, K. Benson, J. L. Boudec, W. Courtney, S. Davari, V. Firoiu, and D. Stiliadis, "An Expedited Forwarding PHB (Per-Hop Behavior)," RFC 3246, Internet

Engineering Task Force, Mar. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3246.txt 2.2.3

[84] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski, "Assured Forwarding PHB Group," RFC 2597, Internet Engineering Task Force, Jun. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2597.txt 2.2.3

[85] "Configuration Guidelines for DiffServ Service Classes," RFC 4594, Internet Engineering Task Force, Aug. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4594.txt 2.2.3

[86] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the performance of tcp pacing," in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 2000, pp. 1157–1165. 2.2.3

[87] D. Wei, P. Cao, S. Low, and C. EAS, "Tcp pacing revisited," in *Proceedings of IEEE INFOCOM*, 2006. 2.2.3

[88] "Internet society global internet report 2015," https://www.internetsociety.org/globalinternetreport/, Tech. Rep., 2015. 2.3

[89] J. Abley, K. Lindqvist, E. Davies, B. Black, and V. Gill, "IPv4 Multihoming Practices and Limitations," RFC 4116, Internet Engineering Task Force, Jul. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4116.txt 2.3, 3.3.2

[90] "About wi-fi assist," https://support.apple.com/en-au/HT205296, accessed: 2016-05-05. 3

[91] D. Wischik, M. Handley, and M. B. Braun, "The resource pooling principle," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 47–52, Sep. 2008. [Online]. Available: http://doi.acm.org/10.1145/1452335.1452342 3.1, 4.2

[92] A. Snoeren, "Adaptive inverse multiplexing for wide-area wireless networks," oct 1999. 3.2

[93] C. B. Traw and J. M. Smith, "Striping within the network subsystem," *Netwrk. Mag. of Global Internetwkg.*, vol. 9, no. 4, pp. 22–32, Jul. 1995. [Online]. Available: http://dx.doi.org/10.1109/65.397041 3.2

[94] J. R. Iyengar, P. D. Amer, and R. Stewart, "Performance implications of a bounded receive buffer in concurrent multipath transfer," *Comput. Commun.*, vol. 30, no. 4, pp. 818–829, Feb. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.comcom.2006.10.011 3.2, 4.1.2

[95] A. Thompson, "Link Aggregation and Failover," FreeBSD Foundation. [Online]. Available: https://www.freebsd.org/doc/handbook/network-aggregation.html 3.3.1

[96] W. T. Thomas Davis, C. T. Constantine Gavrilov, J. V. Janice Girouard, and M. Williams, "Linux Ethernet Bonding Driver HOWTO," Linux Kernel Organization, 2005. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/bonding.txt 3.3.1

[97] K. Sklower, B. Lloyd, G. McGregor, D. Carr, and T. Coradetti, "The PPP Multilink Protocol (MP)," RFC 1990, Internet Engineering Task Force, Aug. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1990.txt 3.3.1

[98] C. Perkins, "IP Mobility Support for IPv4, Revised," RFC 5944, Internet Engineering Task Force, Nov. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5944.txt 3.3.2

[99] E. Nordmark and M. Bagnulo, "Shim6: Level 3 Multihoming Shim Protocol for IPv6," RFC 5533, Internet Engineering Task Force, Jun. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5533.txt 3.3.2

[100] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure, "Experimental evaluation of multipath tcp schedulers," in *Proceedings of the 2014 ACM SIGCOMM Workshop on Capacity Sharing Workshop*, ser. CSWS '14. New York, NY, USA: ACM, 2014, pp. 27–32. [Online]. Available: http://doi.acm.org/10.1145/2630088.2631977 3.3.3, 4.1.1, 4.1.2, 4.1.3

[101] L. Magalhaes and R. Kravets, "Transport level mechanisms for bandwidth aggregation on mobile hosts," in *Proceedings of the Ninth International Conference on Network Protocols*, ser. ICNP '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 165–. [Online]. Available: http://dl.acm.org/citation.cfm?id=876907.881588 3.3.3

[102] J. Crowcroft and P. Oechslin, "Differentiated end-to-end internet services using a weighted proportional fair sharing tcp," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 3, pp. 53–69, Jul. 1998. [Online]. Available: http://doi.acm.org/10.1145/293927.293930 3.3.3

[103] H.-Y. Hsieh and R. Sivakumar, "A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts," in *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '02. New York, NY, USA: ACM, 2002, pp. 83–94. [Online]. Available: http://doi.acm.org/10.1145/570645.570656 3.3.3

[104] K. Rojviboonchai, T. Osuga, and H. Aida, "R-m/tcp: Protocol for reliable multi-path transport over the internet," in *Proceedings of the 19th International Conference on Advanced Information Networking and Applications - Volume 1*, ser. AINA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 801–806. [Online]. Available: http://dx.doi.org/10.1109/AINA.2005.289 3.3.3

[105] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang, "A transport layer approach for improving end-to-end performance and robustness using redundant paths," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '04. Berkeley, CA, USA: USENIX Association, 2004, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247415.1247423 3.3.3

[106] "Stream Control Transmission Protocol," RFC 4960, Internet Engineering Task Force, Sep. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4960.txt 3.3.3

[107] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control," RFC 2581, Internet Engineering Task Force, Nov. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2581.txt 3.3.3

[108] R. Stewart, M. Tuexen, and I. Ruengeler, "Stream Control Transmission Protocol (SCTP) Network Address Translation Support," Internet Draft, Internet Engineering Task Force, Jul. 2009. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tsvwg-natsupp-08 3.3.3

[109] "ios: Multipath tcp support in ios 7," https://support.apple.com/en-au/HT201373, accessed: 2016-06-08. 3.4, 4.1.3

[110] S. Barre, "Implementation and assessment of modern host-based multipath solutions," Ph.D. dissertation, Universite catholique de Louvain, 2011. 4.1, 6

[111] A. Singh, C. Goerg, A. Timm-Giel, M. Scharf, and T.-R. Banniza, "Performance comparison of scheduling algorithms for multipath transfer," in *Global Communications Conference (GLOBECOM), 2012 IEEE*. IEEE, 2012, pp. 2653–2658. 4.1

[112] "Understanding csm load balancing algorithms," http://www.cisco.com/c/en/us/support/docs/interfaces-modules/content-switching-module/28580-lb-algorithms.html, accessed: 2016-06-21. 4.1.1

[113] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, "How hard can it be? designing and implementing a deployable multipath tcp," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 29–29. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228338 4.1.1, 4.1.3, 4.1.4, 4.2.2

[114] S. Ferlin-Oliveria, T. Driebholz, and O. Alay, "Tackling the challenge of bufferbloat in multi-path transport over heterogeneous wireless networks," in *Proceedings of the 22nd International Symposium of Quality of Service (IWQoS)*, ser. IWQoS '14. IEEE, 2014, p. 123**128. 4.1.1, 4.1.4

[115] J. R. Iyengar, P. D. Amer, and R. Stewart, "Receive buffer blocking in concurrent multipath transfer," *IEEE Globecom*, 2005. 4.1.1, 4.1.3

[116] I. A. Halepoto, F. Lau, and Z. Niu, "Scheduling over dissimilar paths using cmt-sctp," in *Ubiquitous and Future Networks (ICUFN), 2015 Seventh International Conference on*. IEEE, 2015, pp. 535–540. 4.1.2

[117] J. Hwang and J. Yoo, "Packet scheduling for multipath tcp," in *Ubiquitous and Future Networks (ICUFN), 2015 Seventh International Conference on*. IEEE, 2015, pp. 177–179. 4.1.2

[118] G. Sarwar, R. Boreli, E. Lochin, A. Mifdaoui, and G. Smith, "Mitigating receiver's buffer blocking by delay aware packet scheduling in multipath data transfer," in *Proceedings of the 2013 27th International Conference on Advanced Information Networking and Applications Workshops*, ser. WAINA '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1119–1124. [Online]. Available: http://dx.doi.org/10.1109/WAINA.2013.80 4.1.2

[119] H. A. Kim, B.-h. Oh, and J. Lee, "Improvement of mptcp performance in heterogeneous network using packet scheduling mechanism," in *Communications (APCC), 2012 18th Asia-Pacific Conference on*. IEEE, 2012, pp. 842–847. 4.1.2

[120] T.-A. Le and L. X. Bui, "Forward delay-based packet scheduling algorithm for multipath tcp." 4.1.2

[121] F. Yang, Q. Wang, and P. D. Amer, "Out-of-order transmission for in-order arrival scheduling for multipath tcp," in *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*. IEEE, 2014, pp. 749–752. 4.1.2

[122] T. D. Wallace and A. Shami, "On-demand scheduling for concurrent multipath transfer under delay-based disparity," in *2012 8th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2012. 4.1.2

[123] M. Fiore, C. Casetti, and G. Galante, "Concurrent multipath communication for real-time traffic," *Computer Communications*, vol. 30, no. 17, pp. 3307–3320, 2007. 4.1.2

[124] B.-H. Oh and J. Lee, "Constraint-based proactive scheduling for mptcp in wireless networks," *Computer Networks*, vol. 91, pp. 548–563, 2015. 4.1.2

[125] N. Williams, P. Abeysekera, N. Dyer, H. Vu, and G. Armitage, "Multipath TCP in Vehicular to Infrastructure Communications," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 140828A, 28 August 2014. [Online]. Available: http://caia.swin.edu.au/reports/140828A/CAIA-TR-140828A.pdf 4.1.2

[126] Y.-C. Chen, Y.-s. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley, "A measurement-based study of multipath tcp performance over wireless networks," in *Proceedings of the 2013 conference on Internet measurement conference*. ACM, 2013, pp. 455–468. 4.1.2

[127] O. Bonaventure, Q. D. Coninck, M. Baerts, F. Duchene, and B. Hesmans, "Improving Multipath TCP Backup Subflows," Internet Engineering Task Force, Internet-Draft draft-bonaventure-mptcp-backup-00, Jul. 2015, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-bonaventure-mptcp-backup-00 4.1.3

[128] H. Balakrishnan and S. Seshan, "The Congestion Manager," RFC 3124, Internet Engineering Task Force, Jun. 2001. [Online]. Available: http://www.ietf.org/rfc/rfc3124.txt 4.2

[129] L. Eggert, J. Heidemann, and J. Touch, "Effects of ensemble-tcp," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 1, pp. 15–29, 2000. 4.2

[130] R. Khalili, N. Gast, M. Popovic, and J.-Y. Le Boudec, "Mptcp is not pareto-optimal: performance issues and a possible solution," *Networking, IEEE/ACM Transactions on*, vol. 21, no. 5, pp. 1651–1665, 2013. 4.2, 4.2.2

[131] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 6582, Internet Engineering Task Force, Apr. 2012. [Online]. Available: https://tools.ietf.org/html/rfc6582 4.2.1

[132] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008. 4.2.1

[133] M. Honda, Y. Nishida, L. Eggert, P. Sarolahti, and H. Tokuda, "Multipath congestion control for shared bottleneck," in *Proc. PFLDNeT workshop*, 2009, pp. 19–24. 4.2.1

[134] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, implementation and evaluation of congestion control for multipath tcp," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 99–112. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972468 4.2.1, 4.2.2

[135] H. Adhari, S. Werner, T. Dreibholz, and E. P. Rathgeb, "Ledbat-mp–on the application of" lower-than-best-effort" for concurrent multipath transfer," in *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*. IEEE, 2014, pp. 765–771. 4.2.1

[136] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)," RFC 6817, Internet Engineering Task Force, Dec. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6817.txt 4.2.1

[137] F. Kelly and T. Voice, "Stability of end-to-end algorithms for joint routing and rate control," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 5–12, Apr. 2005. [Online]. Available: http://doi.acm.org/10.1145/1064413.1064415 4.2.2

[138] A. Walid, Q. Peng, J. Hwang, S. Low, "Balanced linked adaptation congestion control algorithm for mptcp," Internet Engineering Task Force, Internet-Draft draft-walid-mptcp-congestion-control-04, Jan. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-walid-mptcp-congestion-control-04 4.2.2

[139] Y.-C. Chen and D. Towsley, "On bufferbloat and delay analysis of multipath tcp in wireless networks," in *Networking Conference, 2014 IFIP*. IEEE, 2014, pp. 1–9. 4.2.2

[140] Q. Peng, A. Walid, J. Hwang, and S. H. Low, "Multipath tcp: Analysis, design, and implementation," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 596–609, Feb. 2016. [Online]. Available: http://dx.doi.org/10.1109/TNET.2014.2379698 4.2.2

[141] Mingwei Xu, Yu Cao, Enhuan Dong, "Delay-based congestion control for mptcp," Internet Engineering Task Force, Internet-Draft draft-xu-mptcp-congestion-control-04, Jan. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-xu-mptcp-congestion-control-04 4.2.2

[142] T.-A. Le, "Improving the performance of multipath congestion control over wireless networks," in *2013 International Conference on Advanced Technologies for Communications (ATC 2013)*. IEEE, 2013, pp. 60–65. 4.2.2

[143] C. P. Fu and S. C. Liew, "Tcp veno: Tcp enhancement for transmission over wireless access networks," *IEEE Journal on selected areas in communications*, vol. 21, no. 2, pp. 216–228, 2003. 4.2.2

[144] M. Li, A. Lukyanenko, S. Tarkoma, and A. Ylä-Jääski, "Mptcp incast in data center networks," *China Communications*, vol. 11, no. 4, pp. 25–37, 2014. 4.2.2

[145] D. Nagle, D. Serenyi, and A. Matthews, "The panasas activescale storage cluster: Delivering scalable high bandwidth storage," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004, p. 53. 4.2.2

[146] 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. 4.2.2

[147] F. Fu, X. Zhou, T. Dreibholz, K. Wang, F. Zhou, and Q. Gan, "Performance comparison of congestion control strategies for multi-path tcp in the nornet testbed." 4.3

[148] B. Hesmans, H. Tran-Viet, R. Sadre, and O. Bonaventure, "A first look at real multipath tcp traffic," in *International Workshop on Traffic Monitoring and Analysis*. Springer, 2015, pp. 233–246. 4.3

[149] "Transmission Control Protocol (TCP) Parameters," http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml, Internet Assigned Numbers Authority, Mar. 2016.

[Online]. Available: http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml 5.1

[150] C. Paasch *et al.*, "Improving multipath tcp," Ph.D. dissertation, Universite catholique de Louvain, 2014. 6

[151] R. Stewart, "FreeBSD-head Revision 292309," https://svnweb.freebsd.org/base?view=revision&revision=292309, FreeBSD Foundation, Dec 2015. [Online]. Available: https://svnweb.freebsd.org/base?view=revision&revision=292309 6

[152] N. Williams, L. Stewart, and G. Armitage, "Design Overview of Multipath TCP version 0.3 for FreeBSD-10," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 130424A, 24 April 2013. [Online]. Available: http://caia.swin.edu.au/reports/130424A/CAIA-TR-130424A.pdf 6, 6.1.3

[153] ——, "Design Overview of Multipath TCP version 0.4 for FreeBSD-11," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 140822A, 22 August 2014. [Online]. Available: http://caia.swin.edu.au/reports/140822A/CAIA-TR-140822A.pdf 6, 6.1.3, 6.3

[154] P. Willmann, R. Scott, and A. Cox, "An evaluation of network stack parallelization strategies in modern operating systems." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 91–96. 6.1.2, 6.2.2

[155] A. Economopoloulos, "An MP-capable network stack for DragonFlyBSD with minimal use of locks," DragonflyBSD, Tech. Rep., 2008. 6.1.2

[156] L. Stewart and J. Healy, "Light-Weight Modular TCP Congestion Control for FreeBSD 7," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 071218A, 18 December 2007. [Online]. Available: http://caia.swin.edu.au/reports/071218A/CAIA-TR-071218A.pdf 6.1.4

[157] S. Barré, C. Paasch, O. Bonaventure *et al.*, "Multipath tcp-guidelines for implementers," *Working Draft, IETF Secretariat, Internet-draft draftbarre-mptcp-impl-00*, 2011. 6.2.1

[158] R. N. Watson, "Introduction to multithreading and multiprocessing in the freebsd smpng network stack," *Proceedings of EuroBSDCon*, 2005. 6.2.1, 6.2.2

[159] D. Rabson, "Taskqueue – asynchronous task execution," https://www.freebsd.org/cgi/man.cgi?query=taskqueue, accessed: 2016-06-12. 6.2.1, 6.2.1

[160] R. Watson, "Freebsd network performance project (netperf)," https://www.freebsd.org/projects/netperf/, accessed: 2016-07-09. 6.2.1

[161] J. Baldwin and R. Watson, "Freebsd network performance project (netperf)," https://www.freebsd.org/doc/en/books/arch-handbook/smp.html, accessed: 2016-07-09. 6.2.1, 6.2.2

[162] T. Hudek, "Hardware dec center: Receive side scaling," https://docs.microsoft.com/en-us/windows-hardware/drivers/network/ndis-receive-side-scaling2, 2017. 6

[163] J. Lemon, "Syncache," https://www.freebsd.org/cgi/man.cgi?query=syncache, accessed: 2016-07-20. 6.3

[164] D. J. Bernstein, "Syn cookies, 1996," *https://cr.yp.to/syncookies.html*, 2016. 6.3

[165] C. Paasch, A. Biswas, and D. Haas, "Making Multipath TCP robust for stateless webservers," Internet Engineering Task Force, Internet-Draft draft-paasch-mptcp-syncookies-02, Oct. 2015, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-paasch-mptcp-syncookies-02 6.3

[166] G. Smirnoff, "mbuf tags – a framework for generic packet attributes," https://www.freebsd.org/cgi/man.cgi?query=sendfile&sektion=2, accessed: 2017-08-01. 6.4

[167] ——, "svn commit: r293439 in head," https://lists.freebsd.org/pipermail/svn-src-head/2016-January/080924.html, FreeBSD Foundation, Jan 2016. [Online]. Available: https://lists.freebsd.org/pipermail/svn-src-head/2016-January/080924.html 6.4

[168] A. Keromytis, "mbuf tags – a framework for generic packet attributes," https://www.freebsd.org/cgi/man.cgi?query=mbuf_tags, accessed: 2016-07-20. 6.4

[169] "Virtualbox," https://www.virtualbox.org/wiki/VirtualBox, accessed: 2017-08-02. 7.1.1

[170] L. Rizzo, "The dummynet project," http://info.iet.unipi.it/~luigi/dummynet/, accessed: 2016-08-15. 7.1.1

[171] "Iperf," https://iperf.fr/, accessed: 2016-10-17. 7.1.1

[172] "Tshark," https://www.wireshark.org/docs/man-pages/tshark.html, accessed: 2016-08-18. 7.1.1

[173] Y. Coene, "Conformance tests for Multipath TCP," Internet-Draft, Internet Engineering Task Force, Jul. 2013. [Online]. Available: https://tools.ietf.org/html/draft-coene-mptcp-conformance-00 7.2.1

[174] D. A. Hayes and G. Armitage, "Revisiting tcp congestion control using delay gradients," in *International Conference on Research in Networking*. Springer, 2011, pp. 328–341. 7.2.4

[175] J. Semke, J. Mahdavi, and M. Mathis, "Automatic tcp buffer tuning," *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4, pp. 315–323, 1998. 7.2.5