"Measuring and extending multipath TCP"

Tran, Viet Hoang

ABSTRACT

TCP has been one of the most important Internet protocols since the early days of this network. The initial version of TCP assumed that (1) each device has a single interface, and (2) its network address is permanent. Today's Internet attached devices have multiple interfaces with dynamic addresses. These deployments do not match anymore the design principles of TCP. By decoupling the transport layer from the underneath IP layer, Multipath TCP brings several key benefits in a variety of use cases. However, this major TCP extension is also significantly more complex than the legacy TCP. Despite growing interests in Multipath TCP, there are still many unknowns about its behaviours and performance in the real world. Moreover, most Multipath TCP implementations are based on existing TCP stacks which are part of operating systems kernels. Therefore, it is challenging to build Multipath TCP stacks that adapt to different network scenarios and user requirements. The purpose of this thesis i...

CITE THIS VERSION

Measuring and Extending Multipath TCP

Viet-Hoang Tran

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

December 2019

INGI, ICTEAM
Louvain School of Engineering (EPL)
Université catholique de Louvain (UCLouvain)

Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Olivier **Bonaventure** (Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Charles **Pecheur** (Chair)	UCLouvain/ICTEAM, Belgium
Pr. Ramin **Sadre**	UCLouvain/ICTEAM, Belgium
Pr. Benoit **Donnet**	Université de Liège, Belgium
Pr. Per **Hurtig**	Karlstad University, Sweden

Measuring and Extending Multipath TCP

by Viet-Hoang Tran

ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was supported by the Marie Curie ITN METRICS project (No. 607728), the FP7 TRILOGY 2 project, the MONROE project (No. 644399) which are funded by the European Commission, by the BESTCOM IAP, by the Walinnov MQUIC project, and by *Action de Recherche Concertée* (ARC budget No. C1.31403.001-F) supported by Belgian Government.

PREAMBLE

The Internet has never ceased to expand since its inception in the 1970s. Now, it plays a crucial role in almost every field and industry. Internet and other dedicated IP networks are an essential part of many critical infrastructures: energy, water supply, heating, agriculture, transportation, banking, etc. Internet is equally important at the personal level. Equipped with multiple network interfaces, mobile devices now are not only a popular commodity but also a key element enabling a myriad of personal and professional Internet-connected services. It is becoming clear that *the Internet is a Critical Infrastructure to our society.*

TCP has been a core protocol in the Internet architecture since its inception in the 1970s. Although TCP has been significantly improved, its basic principles remain largely the same. Some of these principles are not matched with reality anymore, including (1) each device has a single interface, and (2) its network address is permanent. These outdated assumptions prevent TCP from exploiting the path diversity of the underlying infrastructure, or adapting to the address changes.

By decoupling the transport layer from the underneath IP layer, Multipath TCP covers a long-missed resource pooling opportunity from the viewpoint of end-hosts. This brings several key benefits to the internetworking world. However, this major TCP extension is also significantly more complex than the legacy TCP and opens up a much larger design space. Despite growing interests in Multipath TCP, there are still a lot of unknowns about its behaviours and performance in the real world. Moreover, instead of being built from scratch, most MPTCP implementations are based on mature TCP stacks which typically are in kernel space. Therefore, it is challenging to build MPTCP stacks adaptable to different network scenarios, user requirements and objectives.

With the above motivations, the purpose of this thesis is to answer two main research questions:

- How do Multipath TCP and its current implementations behave and perform in the Internet?
- How to customize and extend Multipath TCP implementations to adapt them to current and future use cases?

For the first question, we have conducted two measurement campaigns, one with traditional traffic types and the other is with voice-activated service. For the second question, we have explored the capability of eBPF infrastructure and leveraged it to extend both TCP and Multipath TCP stacks in the Linux kernel. To be concrete, the main contributions of this thesis are the following:

Passive measurements of real MPTCP traffic

In Chapter 3, we present the method and notable results of our passive measurement of MPTCP traffic towards a public server (`https://multipath-tcp.org`) consecutively during five months. The dataset consisted of nearly 200 thousands of MPTCP connections which carried about 113 GBytes of data. To our knowledge, it was the largest public MPTCP dataset at the time. The main insights from this passive measurement include: (1) Multipath TCP correctly passes through a wide range of Internet paths. (2) Current implementations of Multipath TCP try to utilise additional paths as quickly as possible. (3) Multipath TCP could be further improved in terms of traffic overhead and path management.

Active measurements of voice-recognition traffic using MPTCP

Apple Siri is one of the largest applications of Multipath TCP, but neither its source code nor traffic datasets are publicly available for the community. In Chapter 4, we investigate the benefit of using Multipath TCP for voice-recognition traffic. Leveraging the MONROE mobile broadband testbed, we conducted active measurements over multiple European wireless networks. Since changing the kernel of the MONROE nodes is difficult, we adapted the LKL library to use a custom networking stack without changing the underlying Linux kernel. Our measurement results with emulated voice-recognition traffic show that Multipath TCP could help to improve the user-perceived delay in various network conditions. The latency could be further reduced by optimizing the scheduler on the server side and adding regular intermediate responses.

A generic framework to support user-defined TCP options and Multipath TCP options

Since most of matured TCP and MPTCP stacks are implemented in the kernel, it is difficult and time-consuming to extend them. Given the popularity of both Linux server and Android clients, we focus on extending the Linux TCP and MPTCP stacks. In Chapter 5 and Chapter 6, we leverage the eBPF execution environment to allow users to test and deploy new TCP and MPTCP options on the fly without constantly recompiling and rebooting the Linux kernel. This is possible since the eBPF virtual machine provides a safe and secure environment for executing user code in the kernel in a highly efficient way. Using our eBPF-based frameworks, we explore four practical use cases of TCP options and other four use cases of MPTCP options. Two of them are proposed [195, 196] to be standardized within the IETF MPTCP Working Group.

A generic framework to support user-defined Multipath TCP path managers

Path management of Multipath TCP should be controlled by the users or applications. Recently, a netlink-based solution has been merged into the Linux implementation of MPTCP. However, it has a couple of issues: (1) the

cost of switching context between user space and kernel space, and (2) netlink messages may be lost. In Chapter 7, we design and implement an alternative approach using eBPF which is believed to be more deterministic. With this framework, we implement four path managers as the BPF programs that could be loaded dynamically by users or system administrators.

All the kernel code, tools, scripts and collected datasets that are used for this thesis are publicly available at `https://github.com/hoang-tranviet/thesis-resources`. Moreover, these eBPF-based frameworks are lightweight, therefore, it is feasible to upstream them to the mainline Linux or the out-of-tree MPTCP Linux.

PUBLICATIONS

CONFERENCE AND WORKSHOP PUBLICATIONS

B. Hesmans, H. Tran-Viet, R. Sadre, and B. Bonaventure. "A First Look at Real Multipath TCP Traffic." In: *7th International Workshop on Traffic Monitoring and Analysis (TMA)*. 2015.

Viet-Hoang Tran and Olivier Bonaventure. "Beyond socket options: making the Linux TCP stack truly extensible." In: *The IFIP Networking 2019 Conference*. IFIP. IFIP Digital Library, May 2019.

Viet-Hoang Tran, Ramin Sadre, and Olivier Bonaventure. "Measuring and Modeling Multipath TCP." In: *IFIP International Conference on Autonomous Infrastructure, Management and Security*. Springer. 2015, pp. 66–70.

Viet-Hoang Tran, Hajime Tazaki, Quentin De Coninck, and Olivier Bonaventure. "Voice-activated applications and Multipath TCP: A good match?" In: *MNM Workshop (TMA)*. IEEE. 2018, pp. 1–6.

JOURNAL PUBLICATION

Viet-Hoang Tran, Quentin De Coninck, Benjamin Hesmans, Ramin Sadre, and Olivier Bonaventure. "Observing real Multipath TCP traffic." In: *Computer Communications* 94 (2016), pp. 114–122.

IETF CONTRIBUTIONS

Viet-Hoang Tran and Olivier Bonaventure. *Multipath TCP Inactivity Time Option*. Internet-Draft draft-hoang-mptcp-inactivity-time-00. Work in Progress. Internet Engineering Task Force, July 2019. 8 pp.

Viet-Hoang Tran and Olivier Bonaventure. *Multipath TCP Subflow Rate Limit Option*. Internet-Draft draft-hoang-mptcp-sub-rate-limit-00. Work in Progress. Internet Engineering Task Force, July 2019. 6 pp.

MISCELLANEOUS

Viet-Hoang Tran and Olivier Bonaventure. *Poster: Multipath TCP in action: A view from the server side*. TMA 2016. 2016.

Viet-Hoang Tran and Olivier Bonaventure. *Poster: Towards crowd-based MPTCP measurements*. IMC 2017. 2017.

Viet-Hoang Tran and Olivier Bonaventure. *Making the Linux TCP stack more extensible with eBPF*. Netdev 0x13. 2019.

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Professor Olivier Bonaventure, for all his dedicated time and efforts for me and making sure I am in the right direction. Needless to say, he is the best supervisor that I could find (but had never expected).

I would like to thank my thesis committee members (Pr. Charles Pecheur, Pr. Ramin Sadre, Pr. Benoit Donnet, Pr. Per Hurtig) for reviewing the work and giving insightful questions and comments.

The thesis would not be possible without the help from my co-authors. My engagement into the networking research has been heavily inspired by Dr. Son Hong Ngo and Dr. Kien Nguyen. I am also grateful to work with excellent researchers: Benjamin Hesmans, Quentin De Coninck, Hajime Tazaki.

It is my great pleasure to be a part of the excellent team INLab. I would like to thank my great colleagues who I have worked together over the years: Fabien Duchêne, Mathieu Jadin, François Michel, Maxime Piraux, Ramin Sadre, Christoph Paasch, Olivier Tilmans, David Lebrun, Stefano Vissicchio, Marco Canini, Marco Chiesa, Raphael Bauduin, Gregory Detal, Juan Antonio Cordero.

Thank Vanessa, Sophie, Margaux and Chantal. Every time I had a problem, they always helped to solve it quickly.

I also would like to thank my Vietnamese friends in "Lang Louvain". Thank brothers Viet Ho, Trung Bui, Viet-Anh for our great/crazy time together. And thank HuyPham-ThaoLe who did not hesitate to take time to help me.

Thank mom, dad, Phuong and Loan for always being side by side with me, not only in this work but in all steps in my life.

TABLE OF CONTENTS

Part I

BACKGROUND

1 INTRODUCTION

1.1 COMPUTER NETWORKS - A CRITICAL INFRASTRUCTURE

Started as a research project, ARPAnet - the predecessor of the Internet - initially consisted of four nodes located in four U.S. universities. The work of Vint Cerf and Robert Kahn [38] envisioned an architecture to connect different packet-switching networks to form a *network of networks*, or the Internet. The architectural principles of this work have been consolidated into the TCP and IP protocols.

Since then, the Internet has never ceased to expand. Now, it plays a crucial role in almost every field and industry. The Internet and other dedicated IP networks themselves are essential parts of traditional critical infrastructures: energy, water supply, heating, agriculture, transportation, banking, etc.

During the inception of the Internet in the 1970s, its principles were implemented in a single protocol - the original TCP. The need to support unreliable, non-flow-controlled transport services soon led to the detachment of the IP protocol from the original TCP, along with the creation of the UDP protocol. The IP layer provides logical communication, datagram-mode service between nodes, but it does so without guarantees about the reliability, fairness, etc. These tasks are delegated to upper protocols, typically TCP.

1.2 TRANSMISSION CONTROL PROTOCOL

The Transmission Control Protocol (TCP) was originally designed in 1970s for reliable transfer in specific environments. Now it is being used in a wide range of environments, from datacenters having the order of microseconds delay to satellite networks having the delay of multiple seconds. TCP runs on low-powered embedded environments with the bandwidth of only a few of Kbps and it also runs in multi-Gbps networks and beyond.

Sandwiched between the network layer and the application layer, TCP handles several tasks including multiplexing, connection-oriented service, bidirectional transfer, reliable transfer, flow control and congestion control. A protocol supporting these functionalities should be complex, as suggested by its header format in Fig. 1.1. The basic design principles and operations were specified in RFC 793 [161].

1.2.1 *Connection Management*

Each TCP connection is identified by the four tuples: source/destination addresses and the source/destination ports. These port numbers are in the

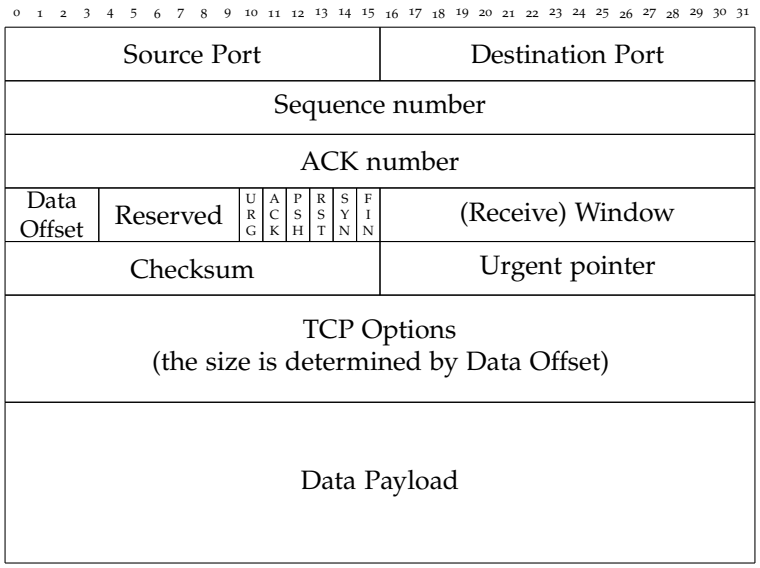


Figure 1.1: TCP Segment Structure Format

first fields of every TCP segment header, which are used for multiplexing traffic with applications.

TCP supports two end hosts exchanging data on both directions, modeled as a single stream on each direction. When the application wants to send data, it passes this byte stream to TCP's *send buffer*. Then, TCP separates it into small chunks that could fit into TCP packets, the size of these chunks are limited by the *maximum segment size* (MSS). Each byte in the stream is mapped to a *sequence number*. Each packet is marked with the sequence number of the first byte of the payload. This sequence number is initialized randomly at the beginning of the connection, and is rotated once it reaches the largest number. Therefore, it requires two sides to negotiate in a *three-way handshake* process to agree on the initial sequence numbers and other parameters used for the transfer, as shown in Fig. 1.2.

TCP hosts also need to close the connections that are not used anymore to save their system resources. Normally, a TCP host gracefully closes a connection by sending a TCP segment with FIN flag to signal that the host has no more data to send, which requires the peer to respond with an acknowledgement. Note that at this point, the connection is only half-closed, the FIN-initiated host cannot send data anymore but it could still receive data from its peer. As shown in Fig. 1.3, the stream of the other direction is closed only after the peer sends its FIN segment and the host sends the acknowledgement for this FIN. The initiator also needs to linger in TIME-WAIT state for as long as twice the Maximum Segment Lifetime (MSL) to avoid the confusion with a new connection incarnated on the same 4-tuple. The state transition of

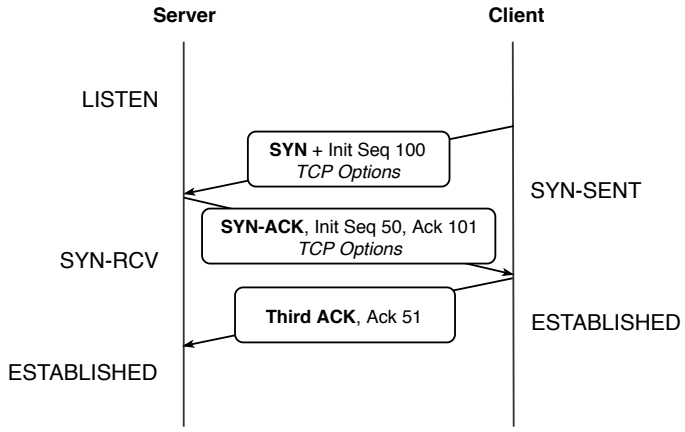


Figure 1.2: TCP connection handshake

the connection teardown process is more complex than that of the connection establishment.

RST is a special TCP flag that can be used to close connections abruptly without handshaking. This may happen when a host is running out of memory and cannot carry a graceful closure. Another scenario is when a TCP host (host A) crashed and rebooted, but its peer (host B) still sends traffic to the host A. Host A needs to tell host B that it does not know about the connection so host A would send a TCP-RST segment. Once receiving this RST segment, host B immediately aborts the connection.

1.2.2 *Reliable Transfer*

To support reliable transfer, TCP must handle packets that are damaged, lost, duplicated, or out-of-order that happened in lower communication layers. TCP receivers use the sequence number of each packet to detect and discard duplicated packets, or to reorder the out-of-order packets. Packet damages are detected by using the checksum field. TCP receivers send ACK packets with the sequence number of last accepted byte plus one. This number is called the *acknowledgment number*. When a TCP packet is considered lost or corrupted, the TCP sender simply retransmits this packet. A TCP sender detects packet losses implicitly in several ways. First, a loss is considered happened when receiving three duplicate ACK packets from the receiver. A threshold of three packets is chosen to avoid the confusion caused by out-of-order arrivals. Second, a stronger indication of loss is when the sender does not receive ACK for the outstanding packet after a timeout duration. This *retransmission timeout* (RTO) is computed based on the measured *round-trip-time* [156].

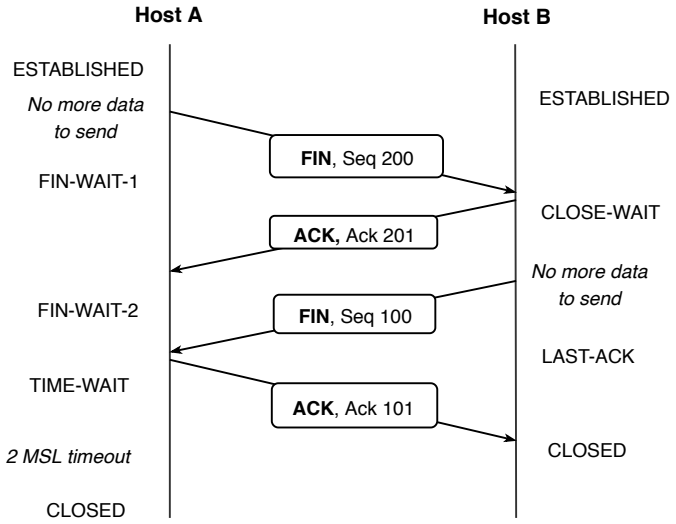


Figure 1.3: TCP connection close

1.2.3 *Flow Control and Congestion Control*

TCP has a built-in *flow control* mechanism. TCP receivers store the arrived packets in their *receive buffer*. This receive buffer could overflow when the host memory is exhausted or the application reads incoming data at a rate slower than the transfer rate. To avoid this problem, TCP receivers advertise their current available receive buffer size in the *window* field. This value is usually adjusted with the *window scale option* to give the actual *receive window*. TCP senders need to make sure that they never send data outside of this window, otherwise the receivers would drop it.

An important functionality of TCP is *congestion control*, which has helped global Internet traffic recovered from *congestion collapse* in 1986 [115]. When the arrival rate is larger than the dispatching rate on a node, its data queue length increases and causes congestion. The congestion problem has many costs. First, it increases the queuing delay of transmitted packets. Second, since routers have limited buffer capacity, incoming packets are dropped, causing the senders to retransmit these lost packets. If the senders do not reduce theirs sending rate, most of the sent packets are just retransmissions. More generally, when the total offered load of all TCP senders onto the network infrastructure is larger than the network capacity, the whole network would cease to serve its clients. This has happened in 1986 [115], leading to the addition of congestion control mechanisms into TCP. Congestion control algorithms have to satisfy multiple goals, which sometimes are mutually conflicting: to detect and avoid congestion, to maximize the goodput and to minimize the latency while maintaining the fairness with other co-existing algorithms.

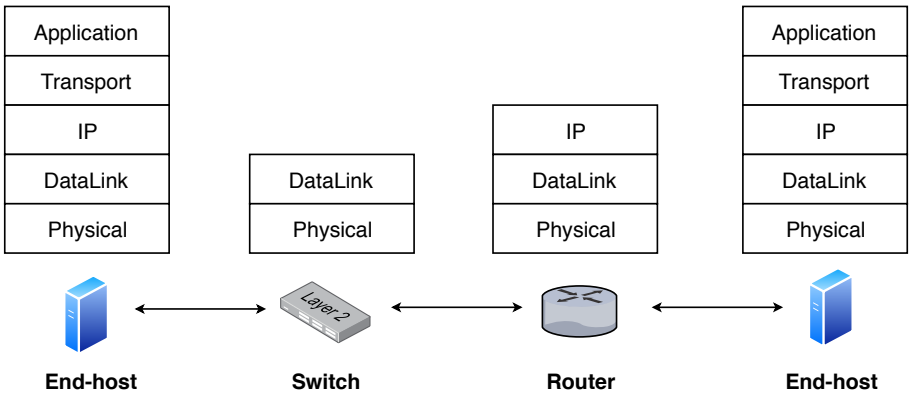


Figure 1.4: Internet path as the original design

Since usually there is no explicit feedback from the network, these algorithms have to infer the congestion from the possible packet loss and the increase of RTT. It is difficult to infer correctly, and false positives would harm the performance. Instead, in-network routers that experience congestions could use Explicit Congestion Notification (ECN) signals [72] to provide ground-truth information about congestions in a timely manner to the end hosts.

1.3 EDGE (AND CORE) EVOLUTION

The initial design of the Internet was based on the end-to-end principle [176]. End hosts fully handle transport functionalities, while intermediate nodes like routers and switches take charge of IP services and are unaware of the transport layer. Therefore, a typical Internet path should be like the one in Figure 1.4. The Internet topology was highly hierarchical with mostly transit links in a typical customer-provider relationship [61].

After more than four decades, the network landscape has changed dramatically. These changes are manyfold:

Network capacity increases quickly. The first remarkable trend through the Internet history is the ever-increasing network capacity, partly thanks to the rapid progress of the router hardware and of transfer medium technologies (copper-wires, fibers, wireless, etc...).

Path diversity increases. In the Internet core, the hierarchical structure has transformed into a flatter topology when the ISPs started peering among them via the Internet eXchange Points (IXPs) [61]. Also, MPLS-based and BGP-based traffic engineering techniques [12, 165] increase path diversity in the core. In the edge, hosts equipped with multiple interfaces are becoming the norm. Notably, mobile devices are popular and become a commodity even in many developing countries.

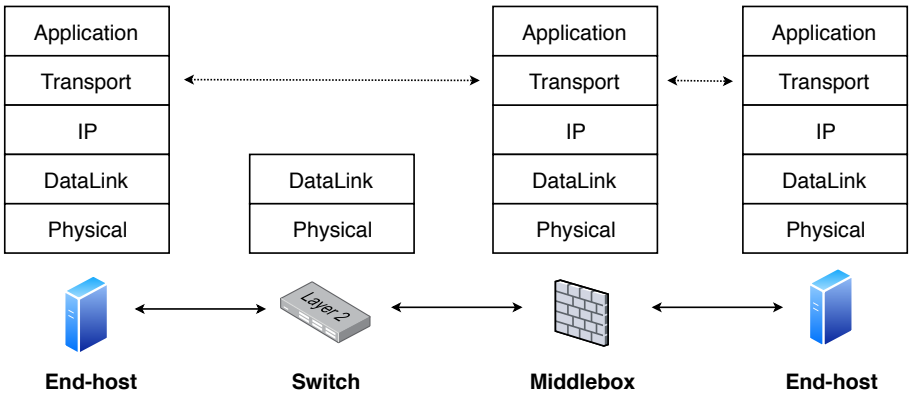


Figure 1.5: Middleboxes are pervasive in today's Internet

Services are more diverse and complex. The increase in network capacity has enabled new service types and models [200]. This trend itself puts more demand on the global network capacity in a positive feedback loop. For example, the flourish of the cloud computing model came from the benefit of *the economy of scale* by pooling a large amount of computing resource and provisioning services quickly to meet users' demands. Another trend is that large content providers want to be as close to their end-users as possible. Many of them directly peer in IXPs [213] or even locate their servers inside ISPs [26, 200].

Middleboxes are pervasive. In the real world, there exist middleboxes which are in-network devices that could analyse and manipulate the transport and even application-level services. Several studies [68, 97, 108, 179] have revealed that the middleboxes are deployed pervasively. They are as common as traditional L2/L3 devices [179]. The network operators use them to achieve mostly specific, short-term benefits. They serve for a wide range of purposes including security, performance, and to deal with IPv4 address depletion problem. End users need to consider that the presence of middleboxes is the norm rather than the exception in today's Internet paths (Figure 1.5). Many of these middleboxes prevent the deployments of new protocols and extensions, hindering the Internet evolutions. More details about middleboxes are described in Section 2.3.

Network layer transitions from IPv4 to IPv6. Due to the IPv4 address space exhaustion, IPv6 has been proposed and standardized [58, 105] since the early 1990s. However, its deployments have really taken off only in the last decade when the IPv4 address depletion is imminent [49]. In this transition phase, dual-stack support is popular in both the end-hosts and the network infrastructure. This also increases the path diversity between end hosts during the transition.

1.4 CONCLUSION

Over the years, TCP protocol and its implementations have significantly evolved. Although today's TCP implementations still use the packet format proposed forty years ago, they include various optimizations [18] to adapt with today's networks. These optimizations and extensions include TCP congestion control schemes [1], TCP timestamp option, large windows extensions, improved round-trip-time estimations and several forms of selective acknowledgements for better retransmission. Despite that, its limitations are becoming imminent in modern times. More importantly, each TCP connection is identified by a 4-tuple. The reason is that TCP was designed when all hosts were stationary and each had a single long-lived network address.

2 MULTIPATH TCP

The previous chapter discussed the basic design of TCP and its limitation given the ongoing Internet evolution. In this chapter, we start with the benefits that resource pooling principle and multipath service can bring to users in Section 2.1. Then, we compare in Section 2.2 various multipath solutions at different layers. Section 2.3 explains how different types of interference caused by middleboxes have blocked the deployent of new transport protocols and stagnated innovations in TCP. Then, we present an overview of the Multipath TCP protocol in Section 2.4. Finally, real-world use cases and best practices of Multipath TCP deployment are dicussed in Section 2.5.

2.1 RESOURCE POOLING

The evolution of the Internet, and the computing industry in general, is directly related to the development of new ways of pooling and sharing resources. D. Wischik et al. [211] have summarized the main benefits of resource pooling as following:

1. increasing resiliency against component failures;
2. better handling of local surges in traffic demand;
3. maximizing global utilization.

There are multiple examples of resource pooling mechanisms in the development of networking and telecommunication. Packet-switched networks replaced circuit-switched networks because the packet-based model allows the flexible sharing of limited physical channels among a large number of users. Cloud providers build large datacenters and virtualize their resources so that they could dynamically provision and scale the service provided to each customer based on their demand. By doing so, cloud providers could serve a much larger number of users than the dedicated-server model and maximize their resource utilization [82]. This is similar to the original motivation of packet-switched networks in their early days: to allow more users sharing the computing power.

As another example, BGP-based traffic engineering [165] and MPLS-based traffic engineering [12] are important mechanisms which network operators use to load balance among multiple network paths and to avoid failures or congestions. However, there was still a missed opportunity for network resource pooling. That is the availability of multiple network paths from *the viewpoint of the end hosts*. This is now more important than ever given that

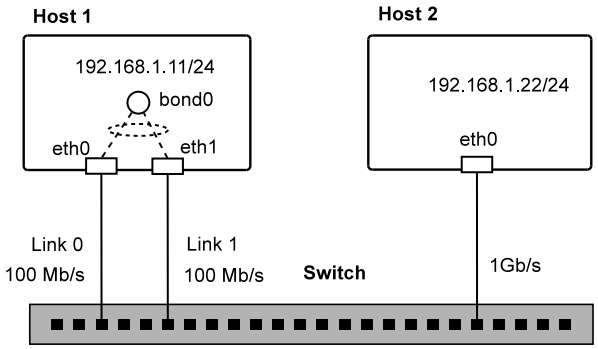


Figure 2.1: Linux bonding

hosts equipped with multiple interfaces (fixed, WiFi, cellular interfaces) and IPv4/IPv6 dual stack have become the norm.

2.2 MULTIPATH APPROACHES

Various multipath solutions have been proposed and implemented at almost every protocol layer. In this section, we analyse typical approaches, in the bottom-up directon from the interface level up to the application level.

2.2.1 *Link-layer and lower layer solutions*

The main multipath techniques at these low-level layers are *interface aggregation* and *link aggregation*. Generally, several physical interfaces are combined into a logical interface. Aggregated interfaces may be configured with the same MAC address or different MAC addresses, but they always share the same IP address. This interface aggregation is supported by popular OS kernels, for example, interface bonding in the Linux kernel or interface teaming in both Windows and Linux. Figure 2.1 illustrates a Linux host bonding two links towards a switch.

There are different modes of *interface aggregation*. If both ends of a link enable *interface aggregation* and they are configured to be in the same mode, it becomes *link aggregation*. This mechanism has been standardized in IEEE 802.3ad [125] and is usually called Link Aggregation Control Protocol (LACP) which is supported by many switches.

The status of each link is monitored continuously, therefore this technique could quickly detect the failure of any link, and if so, the traffic will be diverted to the remaining working links. Since aggregated interfaces share the same IP address, IP and the upper-layer protocols are unaware of the availability of multiple links between two hosts. When a transfer is separated among links with different delays, packets could arrive out of order. For this reason, it is highly recommended that these links are symmetric. However, even if

this is the case, any packet loss would cause a lot of TCP packets to arrive in the wrong order. One TCP-friendly approach is to attach each layer-4 flow to a specific link using a combined hash of MAC, IP and/or transport port pairs. While this technique distributes multiple flows over links and avoids the out-of-order issue, it cannot divide a single elephant flow among these links.

2.2.2 *Network-layer solutions*

Various techniques have been proposed to combine multiple IP address pairs, mostly to allow smooth handovers. Mobile IP [158, 182] is a well-known technique to maintain the continuity of transport connections during client address changes. However, it requires deploying considerable infrastructure to support it. Moreover, traffic has to be redirected through Home Agent and Foreign Agent, causing suboptimal paths and increased latency.

The `shim6` extension [140] for IPv6 is an end-host-only approach that also focuses on handover support. By using the IPv6 extension header to signal between end hosts, it allows associating multiple IPv6 address pairs for smooth handovers [15], while removing the need to deploy extensive in-network infrastructure.

These solutions could support smooth handovers since they hide the network address dynamics to the transport layer. However, each path has different network properties in terms of bandwidth, delay, jitter, loss rate, etc. For reliable transport protocols like TCP, the congestion control, the loss detection and recovery mechanisms are not aware of and may not react properly with the abrupt changes in the network layer when different paths are used.

It is worth to mention about *happy eyeballs* [210] technique, which parallelizes the connection establishment on dual IPv4/IPv6 stack. The purpose of this technique is to shorten the connection establishment time, to select the path with lower latency, and to avoid using broken paths. However, after one connection attempt is selected, the other attempt will be abandoned. Similar mechanisms at the transport layer are proposed such as TAPS [71], for example, allows racing between TCP and UDP-based connection setup.

2.2.3 *Transport-layer solutions*

To be able to manage per-path characteristics, multipath logic should be located at the transport layer or above. Stream Control Transmission Protocol (SCTP) [187] is a notable transport protocol which supports multiple paths by design. However, the original SCTP is only able to use one path at a time. It was later extended to allow using multiple paths concurrently [113]. However, while it gained support from major Operating Systems, its actual deployment is very limited. Two main reasons are (1) applications need to support a new API different from the traditional TCP/UDP sockets, and (2) conservative NAT gateways and other middleboxes block SCTP.

As revealed by measurement literature [60, 179], various types of middleboxes have been deployed pervasively in today's networks. Many of them simply block the protocols and traffic that they do not understand. This practically means that any multipath solution that is targeted for wide deployment has to rely on either TCP or UDP [153]. Middlebox interference is discussed in more details in Section 2.3.

QUIC [123] is a major UDP-based reliable protocol that is promoted and standardized by large Internet companies. QUIC is designed to fix various issues of TCP, including fully encrypted headers to avoid middlebox interference, multiple streams support to reduce head-of-line blocking, connection migration support, better support for zero-RTT handshake, and more. Extensions for QUIC to support multipath are proposed [54, 203] and investigated [110] but not included in the first standardized version of QUIC. However, recent measurements show that UDP protocol is blocked [69, 122] on a non-negligible percentage of Internet paths: more than 4% on port 443 and more than 2% on other ports. UDP rate-limiting policies is another issue that hinders the deployment of QUIC in the public Internet [122].

2.2.4 *Application-layer solutions*

Another interesting approach to work around the ossification problem of the transport layer is to offload the multipath service to the application layer. As proposed by Y. Chen et al. [40], client applications may open several TCP connections in parallel to one or multiple servers to download separate chunks of the same media file. The benefit of this solution is that it leverages both path diversity and source diversity while not requiring kernel modification and is fully compatible with current network infrastructure. On the other hand, this solution requires non-trivial content synchronization among source servers, increased global server load. Moreover, using multi-source does not work with upstreaming or dynamic contents.

A general problem with application-level solutions is that the senders cannot steer traffic quickly among different connections, since they cannot inject data into the head of the connection's send queue. This limits the performance of the approach and its flexibility to adapt to changing network conditions.

2.3 MIDDLEBOX INTERFERENCE

2.3.1 *Landscape*

As mentioned in Section 1.3, the end-to-end principle does not hold anymore in today's Internet, given the presence of the middleboxes. In RFC 2334, a 'middlebox' is defined as "any intermediary box performing functions apart from normal, standard functions of an IP router on the data path between a source host and destination host". Several measurement studies [68, 97, 108, 179] have revealed that the middleboxes are deployed pervasively and

play important roles in various networks. These middleboxes are deployed for multiple purposes, notably to improve the performance and to scale up services, to enforce the security policies, or to deal with the IPv4 addresses depletion problem. Additionally, different types of virtualization technologies have enabled Network Function Virtualization (NFV) which has streamlined the middlebox provision and deployment. It is expected that over the time, more and more types of middleboxes will be deployed in the Internet.

Unfortunately, despite these benefits from the viewpoint of the middlebox operators, at the same time they have undesirable consequences. Middleboxes increase the complexity of network paths significantly without clear visibility from the view of end hosts. Also, this is one of the main causes of the network ossification problem, i.e. deploying new protocols or extensions to popular protocols becomes almost impossible.

Furthermore, due to the complexity of the middleboxes themselves, operators may experience operational issues with the middleboxes more frequently than with the traditional Layer-2 and Layer-3 devices [179], including misconfigurations, scalability, the need to update the software frequently, etc.

2.3.2 *Impacts on TCP*

The impact of middleboxes on both the operation and the evolution of TCP is remarkable given that the TCP headers are not encrypted. In this section, we summarize the main types of middlebox interference on TCP [68, 97, 108].

The simplest interference type is **blocking the connection** when the middlebox observes some unknown TCP options or behaviors. This is normally done by firewalls, intrusion detection/prevention systems (IDS/IPS) to secure the networked assets behind them. For end hosts, the only way to deal with this issue is to re-establish the connection without the new feature disabled after having experienced some timeout.

Security middleboxes may also **remove unknown TCP options**. This behavior only degrades the connection performance by disabling a feature, but still allows the connection to continue. Some middleboxes even block or remove essential TCP features [68] such as the window scale option or Selective ACK option, which severely hamper the performance.

Network Address Translation (NAT) devices are common in access networks to deal with the IPv4 addresses exhaustion problem. They **modify the IP address and port number** field in the packet headers and usually prevent connection attempts from outside.

The sequence number and acknowledgment number are not expected to be modified by the networks. However, since some very old TCP stacks were vulnerable to the TCP sequence prediction attack, some middleboxes tried to protect them by **randomizing the sequence number** of every TCP connection. Although not being relevant anymore, this "security feature" is still common in today's networks [68].

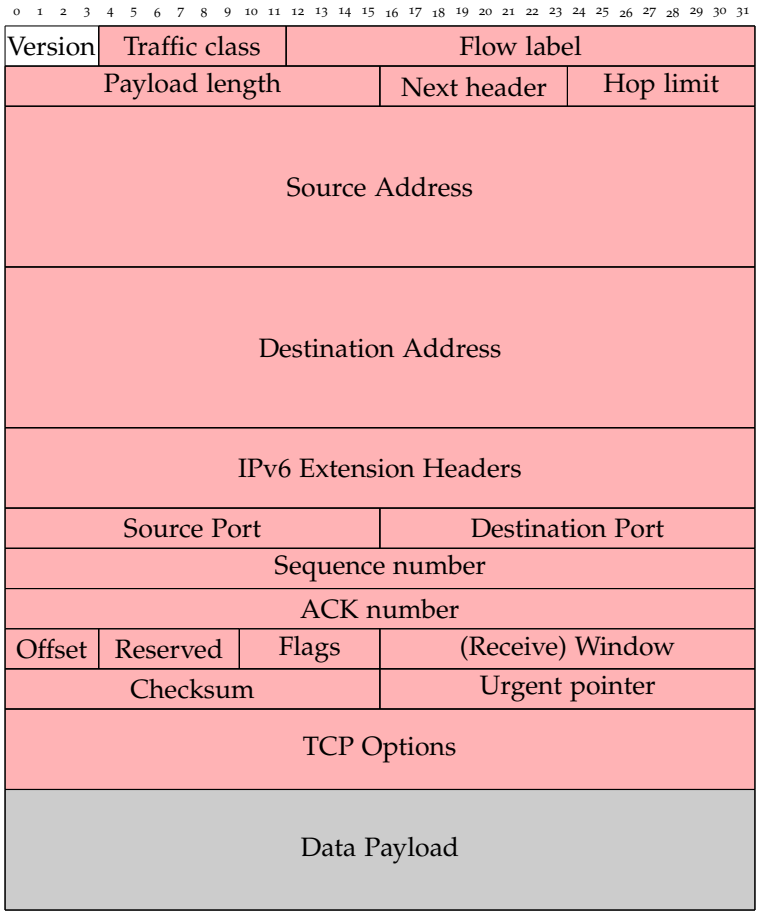


Figure 2.2: Most fields (in red color) of the TCP header and IP header could be modified by middleboxes. Even IPv6 Flow label could be cleared in rare cases by firewalls for security reasons [6]

On end hosts, modern Network Interface Cards (NICs) are increasingly complex. They could handle more and more tasks other than sending and receiving packets. A common feature on modern NICs is TCP Segmentation Offload (TSO) which allows the TCP stacks to process very large TCP segments. NICs would **split large TCP segments** into smaller ones that fit the MTU before actually sending them out. Similarly, on the receiver side, typical NICs support Large Receive Offload (LRO) which **combine multiple segments** into a larger one before sending it to the network stack. These NICs have to compile new IP and TCP headers for newly created TCP segments. From the viewpoint of the TCP stacks, these NICs are one type of middlebox.

For high-latency connections, it may take a long time to reach the right congestion window or to react to packet losses. Performance enhancing proxies (PEPs) typically **terminate** [207] each passed-by TCP connection into two separate connections, actively handshake and acknowledge data to each peer of the connection. PEPs reduce the effective round-trip-time and therefore increase performance.

Web proxies may also support other features [207, 216] such as caching, media compression. Similarly, when clients use active FTP behind a NAT device, the NAT has to change the IP address field in the FTP command. These kinds of **payload-modification** may lead to the change of segment length and TCP sequence numbers.

All these interferences mean that introducing new features to TCP is notoriously difficult. For example, TCP Fast Open [42] is an important feature to avoid the TCP connection handshaking time. However, experimental deployments [146, 190] have shown that middleboxes could block or invalidate the TCP connections, accounting for 20% of total TCP connections. In the worst case, some IDSes even block the clients due to the presence of payload in SYN packets which is wrongly considered as a security anomaly by these IDSes.

2.4 MULTIPATH TCP PROTOCOL

Designing the multipath functionality at the transport layer is the right approach. However, due to the network ossification issue, deploying a new clean-slate transport protocol in the Internet is painfully hard. Instead, Multipath TCP [80, 170] was designed as an extension to regular TCP that is backward compatible with TCP.

2.4.1 *Goals and Requirements*

As mentioned in Section 2.2.3, there were several proposals to support multipath at the transport layer, but none of them was widely deployed in the real world. Therefore, from the beginning, the designers had set the requirement for Multipath TCP protocol to be backward compatible [170] with regular TCP in three aspects:

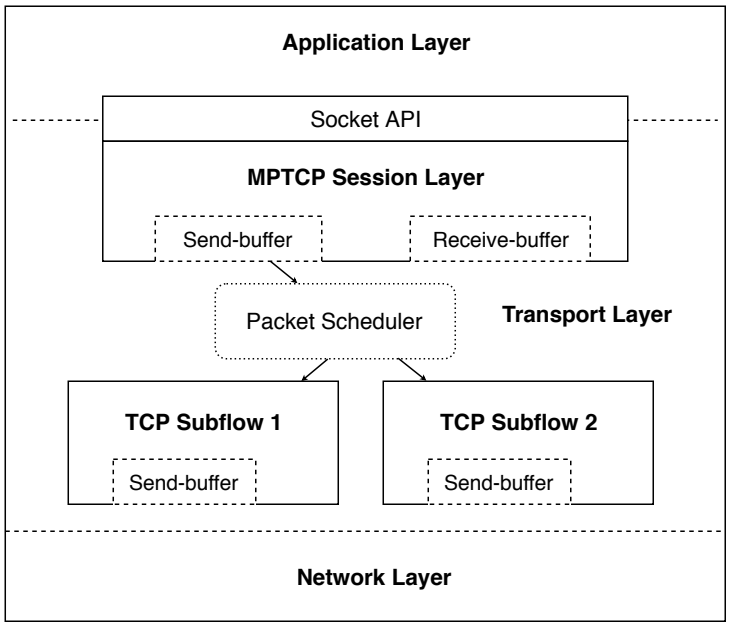


Figure 2.3: MPTCP Protocol Stack

- Application level: Legacy applications designed for regular TCP could use Multipath TCP without any source code modification. To support this, Multipath TCP must use the same socket API as TCP (see Fig. 2.3).
- Network level: Multipath TCP must successfully pass through a majority of middleboxes such as NAT, firewalls, PEPs. It also needs to work with common performance-enhancing mechanisms such as coalescing and segmentation offloading.
- Fallback mechanism: Multipath TCP must be able to fall back to regular TCP when communicating with legacy-TCP peers. Fallback to TCP is also needed when Multipath TCP traffic is blocked, filtered or manipulated by middleboxes.

Moreover, to be a successful protocol, it has to satisfy two important requirements [170]. First, in terms of performance, Multipath TCP must perform at least as well as regular TCP in the same conditions. Second but not less important, it must be feasible to implement Multipath TCP in standard operating systems, without requiring much more system resource consumption.

2.4.2 *Design Overview*

Multipath TCP should enable hosts to exchange the packets that belong to one connection over different interfaces or paths. One simple design of Multipath

TCP could be just taking the outgoing segments from the TCP stack and spreading them over different paths. To gather per-path characteristics, due to the ambiguity of the ACK number, Multipath TCP would then need to track each segment by processing Selective ACK information. However, such a design would not work in the public Internet. Many middleboxes do not allow a hole or a discontinuity of the sequence number and will stall the connection. Multipath TCP needs a more robust design.

It is necessary to establish a separate TCP connection along each path so that traffic could be sent on each path reliably. This also allows the sender to infer and measure per-path characteristics correctly. These TCP connections are called *subflows*. These TCP subflows are combined into a single Multipath TCP connection and they are transparent to the application. Since Multipath TCP must appear as regular TCP to the existing applications, it has to deliver similar services: connection setup and teardown, reliable and in-order data transfer, flow control and congestion control.

The conceptual view of the Multipath TCP protocol stack is depicted in Fig. 2.3. In the remainder of this section, we present a summary of the main protocol operations. More detailed explanations of the protocol can be found in [80, 170]. The second version of Multipath TCP, which will be published soon [81], improves the first version [80] with reliable connection setup and various security enhancements.

All operations of Multipath TCP use TCP options for signaling. All types of MPTCP-specific options share the same option kind number to avoid being blocked selectively by middleboxes. These option types are only differentiated by their subkind field.

2.4.3 *Data plane*

Since Multipath TCP creates a separate subflow per path, it maintains different subflow sequence number (SSN) spaces.

To achieve the reliability and the right order of data delivery at the receiver, MPTCP uses a *data sequence number (DSN)* for tracking the MPTCP-level data stream. This sequence number is 64-bit long and can manage 2^{64} bytes of data without recycling the sequence space. Wrapped sequence numbering is a common issue with regular TCP which uses 32-bit sequence numbers. To allow the receiver to reassemble data stream correctly from different subflows, the sender might simply insert a TCP option containing only the data sequence number of each data chunk. However, typical modern Network Interface Cards (NICs) would split and coalesce TCP segments and copy the same TCP options to each chunk. This makes different TCP segments having the same data sequence numbers and breaks the implicit mapping between DSN and SSN.

Therefore, the alternate solution is to specify explicit mapping in MPTCP *Data Sequence Signal (DSS)* option, which is attached to almost every packet of an MPTCP connection. While this early design worked well in local networks,

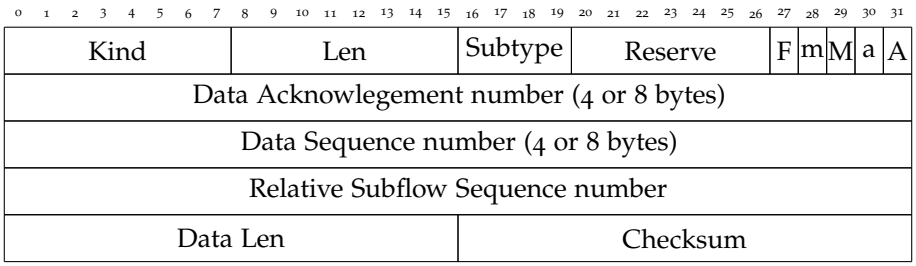


Figure 2.4: DSS Option, mapping between data sequence number and subflow sequence number

the experiments turned out that it was blocked in the public Internet. The reason is that subflow sequence numbers can be modified by on-path middleboxes. This is considered a security feature to protect very old TCP stacks like the one of Windows 95.

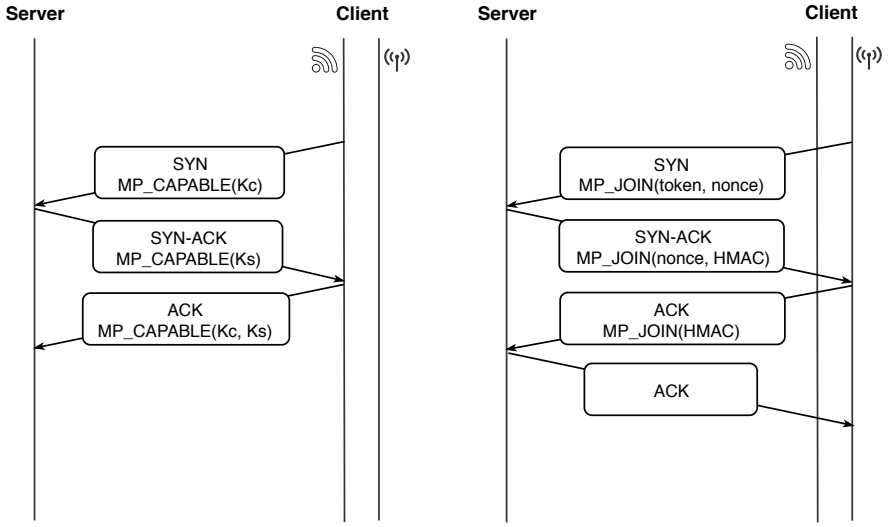
For this reason, instead of using the absolute subflow sequence numbers, the final design of the DSS option stores the relative SSN which can not be changed by any middlebox (otherwise regular TCP traffic would be corrupted). This DSN allows retransmitting or duplicating data on different subflows to either overcome a failure, improve the performance, or reduce the latency. However, it is difficult for the sender to infer which data sequence number has really advanced in case of reordering; therefore, an explicit *data acknowledgement number (Data ACK)* is added to the DSS option. The Data ACK number also provides multipath-level flow control capability. The complete format of the DSS option is shown in Fig. 2.4.

While each subflow has a dedicated send buffer which is separated from the MPTCP-level send buffer, all subflows share the same receive buffer (see Fig. 2.3). This design decision was made to avoid a deadlock scenario [170].

Since application-level middleboxes may change the payload and break the mapping between DSS and subflow sequence number, an MPTCP checksum is added to the DSS option to protect data packets. If a payload manipulation is detected, MPTCP will close the affected subflow. If this is the only subflow of the connection, MPTCP will fall back to regular TCP so that the middlebox could change the payload without breaking the connection.

2.4.4 *Initial Subflow setup*

The MPTCP connection establishment relies on the regular TCP connection handshake. As shown in Fig. 2.5a, MPTCP hosts add the `MPTCP_CAPABLE` option in the three TCP handshaking segments. This option verifies if the remote end supports Multipath TCP and exchanges crypto information (MPTCP key) for authenticating additional subflows later.



(a) Initial subflow (also MPTCP connection)

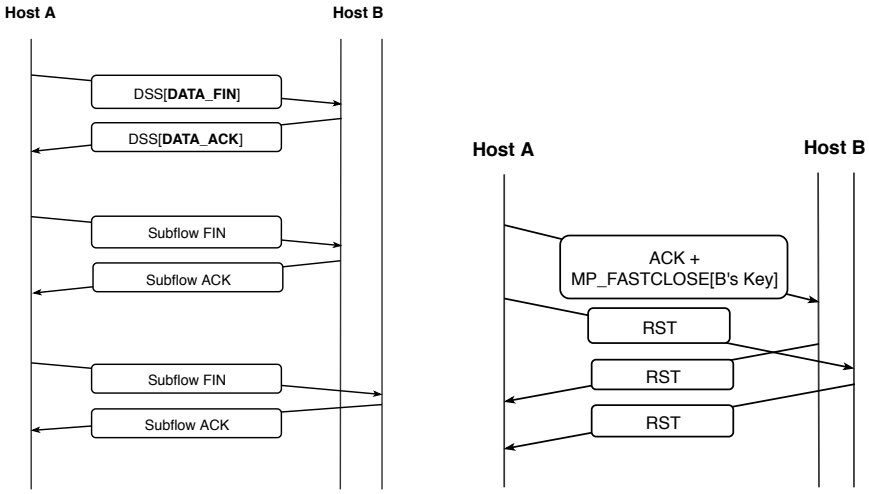
(b) Additional subflow

Figure 2.5: MPTCP connection/subflow establishments

The client may receive a SYN-ACK without an `MPTCP_CAPABLE` option, because either some middlebox removed the option or the server simply does not support MPTCP. In this case, MPTCP was designed to transparently fallback to TCP and continue the session. A rarer but still possible issue is that the MPTCP connection attempt is silently blocked by middleboxes. As stated in the protocol specification, the sender should establish a legacy TCP connection when the timeout expired.

2.4.5 *Additional Subflow setup*

After the initial subflow established and some data has been exchanged, additional subflows could be created to use other available paths. The keys exchanged during the connection setup provide a crypto base for authenticating between end hosts when creating new subflows. Similar to the connection setup, three handshaking segments also carry the `MP_JOIN` options. However, different from the connection setup, the subflow establishment is only done after a four-way handshake, as shown in Fig. 2.5b. The first SYN `MP_JOIN` option carries the 32-bit connection token, which is inferred from the connection key, which the receiver could use to quickly look up the connection. Both sides send a nonce challenge, requiring each other peer to verify itself with the corresponding HMAC. The option also contains the ID of the local IP address and optional flags (not shown in the figure).



(a) Graceful Close (b) Abrupt Close

Figure 2.6: MPTCP Connection Close

In certain situations, a host may want to inform its peer about its available IP addresses. For example, when a client-side NAT prevents a server to directly establish the connection. This can be done by sending an `ADD_ADDR` option which contains both the IP address and its local identification. MPTCP also supports signaling the removal of an address by using the REMOVE_ADDR option which contains the local identification of the IP address.

2.4.6 *Subflow and Connection teardown*

An MPTCP host may close a subflow in certain cases, e.g. due to bad performance or high monetary cost, by initiating the regular 4-way handshake of TCP FIN/ACK packets. It is worth to mention that these FINs only occupy the subflow sequence space but not the data sequence space, so that they do not affect the other subflows. A subflow could also be closed abruptly by TCP RST, for example when a host receives the `REMOVE_ADDR` option and needs to remove all subflows towards the lost IP address.

At the MPTCP connection level, hosts use the `DATA_FIN` flag bit in the DSS option to inform their peers that there is no more data to send on this MPTCP connection, as illustrated in Fig. 2.6a. For simplicity, Fig. 2.6a only presents the MPTCP connection closure in one direction. This happens when the application calls the `close()` system call to close the outward stream. The mechanism is semantically similar to the TCP FIN/ACK handshake for closing a regular TCP connection gracefully. While the `DATA_FIN` signal occupies one bit in the data sequence space, it does not affect the subflow sequence space and is not attached into any TCP segment having a payload. The `DATA_FIN`

can only be DATA_ACKed when all previously sent data has been acknowledged at the connection level.

On the other hand, the subflow-level FIN signal occupies one octet in the subflow sequence space, but it is not included in the data sequence space. A host must not close all subflows (including combining TCP FIN and DATA_FIN in the same segment) if there is outstanding data on other subflows.

Alternatively, MPTCP hosts may terminate a session abruptly by sending an MP_FASTCLOSE option whose semantic is equivalent to TCP RESET but at the MPTCP connection level. MP_FASTCLOSE may be attached to a regular TCP-ACK segment (which is transferred reliably, as shown in Fig. 2.6b) and goes to the FASTCLOSE_WAIT state. The new version of MPTCP (defined in RFC 6824bis [81]) allows the sender to attach the option to the TCP-RST segment (which may be lost) and go directly to CLOSED state without further delays.

In some cases, MPTCP hosts want to quickly close or reject a specific subflow only. Since it is often useful for the peer to know the reason for the connection abrupt close, RFC 6824bis defines a new MP_TCPRST option for hosts to explicitly notify their peers the exact reason.

2.5 MULTIPATH TCP IN PRACTICE

With new capabilities, Multipath TCP enables more use cases and has a much larger decision space than the regular TCP. In Section 2.5.1, we discuss main use cases of using Multipath TCP. Subflow management (in Section 2.5.2) and packet scheduling (in Section 2.5.3) are two new tasks that did not exist with legacy TCP. Multipath TCP also introduces new requirements for the congestion control mechanism, as discussed in Section 2.5.4.

2.5.1 *Main use cases*

Many use cases for Multipath TCP are proposed in the literature and are deployed commercially [22, 23]. Initially, the motivation for Multipath TCP was the emergence of multihomed hosts like smartphones that are equipped with several network interfaces. Such hosts did not exist when TCP was designed. Users of smartphones expect that multiple available interfaces could be used simultaneously to increase the bandwidth [211] or successively to support mobility [143, 169]. In general, Multipath TCP is deployed much faster on the client side than on the server side. Therefore, it is necessary to deploy proxies between MPTCP-capable clients and legacy-TCP servers. There are several commercial deployments that combine fixed broadband network and cellular network as shown in Fig. 2.7, notably by Tessares [119]. In sparsely populated areas, this approach provides high network performance for end users at a much lower cost than deploying broadband technologies such as FTTx, VDSL2, DOCSIS3.0. Similar deployments that combine Wi-Fi and 4G for high-end smartphones have been done by Korean telecom providers [23].

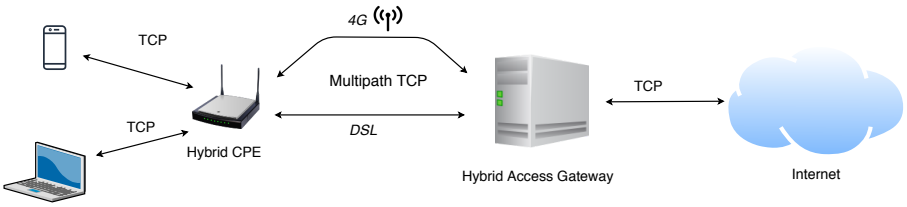


Figure 2.7: Hybrid access networking: One typical use case of Multipath TCP is combining cellular and xDSL paths

One type of this explicit proxy has been generalized for other TCP extensions and is actively standardized within IETF TCPM working group [24].

The second use case is in the datacenters [168] where Multipath TCP improves network utilization by exploiting the path diversity between two hosts when the Equal Cost Multipath (ECMP) feature is enabled.

The third use case is for single-homed but dual-stack hosts that support IPv4 and IPv6. On such hosts, Multipath TCP can use both network layer paths simultaneously for increased performance [131].

In September 2013, Apple enabled Multipath TCP on all of its iOS devices, including smartphones and tablets. Within a few months, hundreds of millions of devices started to use this TCP extension. Siri is the first iOS application using Multipath TCP, which helped significantly improving the availability of its voice recognition service. More recently, Apple Maps and Apple Music are also using Multipath TCP by default to improve responsiveness and throughput [152]. Moreover, all third-party iOS applications now can enable Multipath TCP using provided APIs under one of three modes: handover, aggregation or interactive. It is also included in recent MacOS releases and the upstream project is actively working to add Multipath TCP support into the official Linux kernel [151].

2.5.2 *Subflow management*

When each peer of a connection has more than one interface or IP address, there are multiple path options to establish the subflows for the connection. RFC 6824 specifies the mechanisms to establish and terminate each subflow, but it does not specify the path selection strategies which are intentially left to be decided by the implementers.

The Linux implementation of MPTCP currently supports four subflow management modes (usually called `path managers`). The `default` one actually does not establish any additional subflows neither advertise available addresses. However, it accepts the subflow request from its peer. The `fullmesh` path manager, as its name suggested, creates all possible subflows between two hosts, forming a full-meshed connection. Figure 2.8 illustrates the behavior of this path manager when each host has two interfaces. The third path

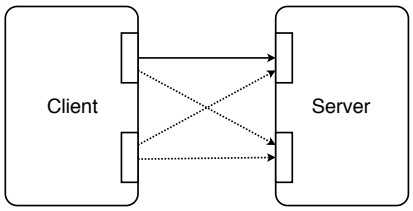


Figure 2.8: Fullmesh path manager tries to establish subflows along all possible paths

manager is `ndiffports` which creates multiple subflows on the same address pair, trying to exploit the path diversity in ECMP-enabled datacenters [168]. The fourth path manager is `binder` [19] which uses Loose Source Routing to combine multiple internet gateways in community networks.

Technically, the subflow establishment is symmetrical, i.e. it could be initiated from either the client or the server. However, all path managers in Linux MPTCP initiate subflows only from the client side. The first reason is that the clients are often behind some NAT devices which block the subflow requests from the server. The second reason is that if both sides initiate the subflows simultaneously then it may end up at least two subflows per path, leading to unnecessary overhead and potential interference.

2.5.3 *Packet Scheduling*

When there are two or more available subflows (or paths), the sender needs to choose on which subflow it should send each outgoing packet. Although it is not strictly a part of the protocol specification, packet scheduling plays a very important role to achieve high performance. The receivers may request the senders to set a subflow to the backup mode or the active mode by sending the MP_PRIO option with relevant flags. The scheduler on the sender side enforces this policy by not selecting the backup subflows when sending data.

The default scheduler algorithm in the reference implementation of MPTCP in Linux is based on subflow-level round-trip-time (`Lowest RTT` [144]). Among the subflows having open sending window, it selects the subflow having the lowest current RTT to send packets.

There are many proposed MPTCP schedulers for different purposes, most of them were implemented for the Linux reference implementation. A comprehensive overview of existing approaches could be found in the ProgMP paper [84]. Pinedo et al. and Frommgen et al. [86] independently proposed and implemented redundant schedulers which duplicate data on multiple subflows to minimize the latency and increase the robustness. The DEMS [92] and RAVEN [126] schedulers selectively duplicate data to reduce the latency for interactive traffic. These schedulers send selective redundant packets on multiple subflows to both probe current status of inactive subflows and to minimize the round-trip-time.

Schedulers may take a diversed range of input parameters to make the decision. The DAPS [177], OTIAS [218] and BLEST [75] schedulers use subflow-level information as input data. The ECF [130] and DEMS [92] and Cross-Layer [51] schedulers rely on application layer information, e.g. data chunk boundary and delivery deadline, to minimize the flow completion time. The MP-DASH solution [94] takes a step further that couples the operation of MPTCP scheduler and DASH layer to optimize HTTP-based video streaming. On the other hand, other works proposed to utilize the lower-layer information to improve the situational awareness of multipath schedulers, e.g. device-driver buffer occupancy [181].

One common problem of Multipath TCP is that the difference of RTT among subflows causes high level out-of-order delivery of packets. This in turns imposes a requirement for large receive buffer. More importantly, it is the reason for the head-of-line (HoL) blocking, i.e. stall when data is still in flight on the slow path and later a large burst when this data is acknowledged at connection level. The Lowest-RTT scheduler [144] mitigates this problem by opportunistic retransmission and penalisation (PR) mechanism, i.e. reinjecting data which was on the slow subflow on to the fast subflow, and halving the congestion window of the slow one. However, this decouples the scheduling and the congestion control mechanism, causing suboptimal behaviors. It does not fully prevent sending data on the slow subflow when it hurts the performance. Meanwhile, when the quality of this subflow gets better, the low congestion window prevents this subflow to be used efficiently. Instead, the BLEST [75] scheduler or MPTCP-LA [139] temporarily disables a slow subflow when it predicts that sending data on this subflow is not effective. Another solution is to schedule low-DSN packets on the fast subflows and high-DSN packets on the slow subflows so that they can arrive in order at the receiver. This has been proposed and implemented in DAPS [177] and OTIAS [218]. Meanwhile, the STMS scheduler [180] adjusts the packet dispatching based on the DATA-ACKed size instead since it is the right feedback for the out-of-order level at the receiver.

2.5.4 *Congestion control*

Multipath TCP could be used with regular TCP congestion controllers which manage outgoing data per subflow independently. However, this was considered to be unfair with regular TCP traffic. Congestion controls for MPTCP need to consider the current states of all subflows to direct traffic through the least congested paths and to maintain fairness with other TCP connections. For that, RFC6356 [167] specifies three criteria that an multipath congestion control must satisfy.

1. "Improve Throughput": each multipath flow should perform at least as well as a single-path flow would on the best path.

2. "Do no Harm": each multipath flow should not occupy more resources shared by its different paths than if it were a single flow using only one of these paths.

3. "Balance Congestion": each multi-path flow should move as much traffic as possible away from the most congested paths.

Researchers proposed various coupled congestion control schemes and implemented them in the Linux as kernel modules. Linked-Increases Algorithm (LIA) [212], which is based on NewReno, is the default congestion control of the Linux kernel implementation of MPTCP. The OLIA [120] and BALIA [157] algorithms are the improved versions of LIA and they focused on friendliness, responsiveness, and window oscillation. Wvegas [33] is an multipath adaptation of the delay-based Vegas congestion control. Most recently, a coupled version [95] of the BBR congestion control was proposed.

2.6 CONCLUSION

This chapter has provided an overview of various multipath approaches and, in specific, the Multipath TCP protocol which is the main research interest of this thesis. The benefits of adding multipath capability to TCP are clear, and actually there were several efforts over the years. However, to the best of our knowledge, none of them are widely deployed. A deployable design of Multipath TCP [170] only comes nearly four decades after the birth of TCP in the 1970s. On one hand, this suggests the network ossification that the pervasive middleboxes have introduced. These middlebox interferences have heavily influenced the design of Multipath TCP. The control plane signaling is heavily based on TCP option and the data plane use a separated data sequence space for multipath level. On the other hand, the development of Multipath TCP protocol is parallel with the development of its implementation the Linux kernel. This has shown that the success of a new protocol depends on not only the protocol specification itself but also the availability of feature-rich implementations.

Part II

THE MEASUREMENTS

As discussed in previous chapters, Multipath TCP was designed to provide more benefits than regular TCP while remaining backward compatible with legacy applications and existing infrastructure. However, despite the growing interests and the number of commercial deployments, there are still many unknowns about the behaviors and performance of Multipath TCP in the real world. Chapters 3 and 4 focus on passively and actively measuring Multipath TCP to improve our understanding of the dynamics of this protocol in the wild.

3 PASSIVE MEASUREMENT: OBSERVING REAL MULTIPATH TCP TRAFFIC

In this chapter, we provide the first detailed analysis of the operation of Multipath TCP in real networks by analyzing long packet traces collected on a Multipath TCP server often used by researchers and implementers.

We first collected a five-month-long packet trace that contains 190,451 Multipath TCP connections originated from more than 7,000 different hosts across the Internet. This was the largest publicly available dataset of Multipath TCP at the time (2015)[1]. To analyze this long trace, we improved the performance and the functionality of open-source `mptcptrace` software [99] and develop custom scripts. Our analysis reveals various important information about the current usage of Multipath TCP. The protocol works well over many Internet paths and we observe very few fallbacks caused by middleboxes. Multipath TCP connections can last thousands of seconds or more and we observe handovers on such connections. From the performance viewpoint, our analysis reveals that Multipath TCP connections are often composed of subflows having very different round-trip-times. We also analyze some inefficiencies of the current Multipath TCP implementations and study in more details the reinjections, i.e. the transmissions of the same data over several paths, and how they affect the performance.

This chapter is organized as follows. We first discuss related work in Section 3.4. We describe the dataset that we have collected in Section 3.1 and our extensions to `mptcptrace` that enabled us to process them efficiently in Section 3.2. In Section 3.3, we analyze how the Multipath TCP connections use their subflows. We discuss in Section 3.5 the main lessons that we learned from this first detailed analysis of Multipath TCP packet traces.

3.1 DATASET

Although Multipath TCP has already been used by Apple iOS devices to support the Siri voice recognition application, our analysis instead focused on the open-source Multipath TCP implementation in the Linux kernel [149]. This is the reference implementation for Multipath TCP, and it is recently used by Apple on the server side to serve all Siri clients around the world. This implementation is distributed from `multipath-tcp.org` and thousands of users have downloaded and installed it on their computers. We use `tcpdump` on the server side to collect all the packets sent and received by `multipath-tcp.org`. At the time of writing, one limitation is that we have a single server only with one interface, but dual IPv4/IPv6 stack is available to clients. In addition

1 This dataset is available at `http://www.multipath-tcp.org/data/COMCOM16`

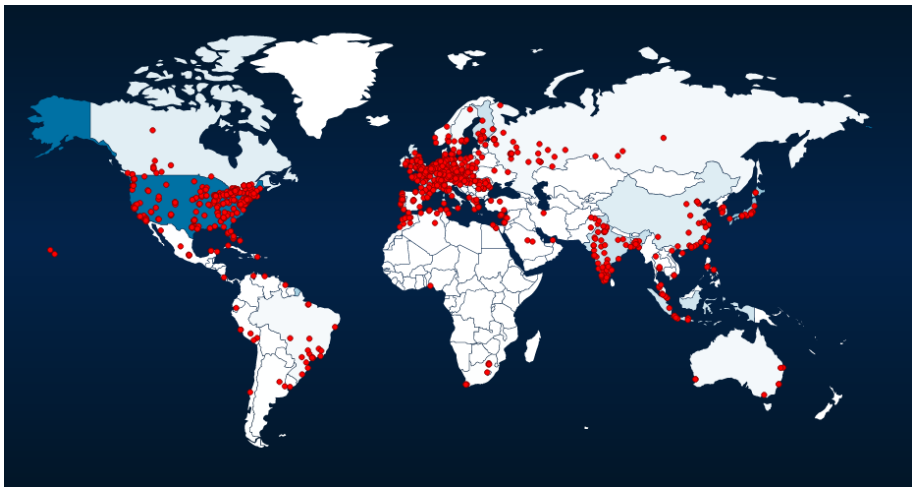


Figure 3.1: Map of MPTCP-enabled clients based on their IP addresses

to providing source code and binaries for the Multipath TCP patch, this host also supports other web servers, an `FTP` server and uses an `iperf3` daemon to enable researchers and users to perform various tests. We extract from the trace the packets that correspond to Multipath TCP connections. They are identified by looking at the utilization of the Multipath TCP options during the three-way handshake. The dataset has been collected during 5 months from November 25th, 2014 to April 29th, 2015. The remaining of this section provides main characteristics and statistics of our dataset.

We observed 10300 IP addresses used by our clients, including 7770 IPv4 addresses. Figure 3.1 shows their locations distributed across many regions. Most of the clients came from Europe, North America, South and East Asia. While the distribution of users could be biased toward MPTCP developers and early adopters, we still observed a very diverse users set based on their Autonomous Systems (AS). According to the IP-to-ASNumber database of Team Cymru [52], these client IP addresses correspond to 588 unique ASes.

We identified 190,451 Multipath TCP connections, which used several destination ports. Most (86.17 %) of the connections used port 80. The remaining connections were on port 21 (FTP, 5.33 %), port 5001 (Iperf, 1.68 %) and other high port numbers linked to passive-mode FTP connections and a private HTTP proxy (port 3128) used for some tests.

The observed connections are diverse in terms of both duration and size. Figure 3.2 plots the durations of the Multipath TCP connections. We observe that most of the Multipath TCP connections last in the range of 1 to 100 seconds. There are only 1.51% of the connections lasting more than 100 seconds.

In terms of connection size, as shown in Fig. 3.3, we could identify some major clusters of connections: a group of exactly 2 Bytes (account for 12% of

Port	MPTCP connections	% Connections	% Traffic volume
20	82	0.04	1.04
21	10158	5.33	0.002
80	164123	86.17	26.80
3128	1215	0.63	0.50
5001	3211	1.68	22.14
Others	11662	6.12	49.52

Table 3.1: The dataset, broken down by TCP server ports

all the connections), large groups around 1 KByte (account for 15%) and 10 KBytes (account for 65%), smaller groups around 100 KBytes and 1 MBytes. The connections containing exactly two bytes are empty connections with one-byte space for DATA_FIN in both directions.

This is expected since today's web browsers usually pre-open connections on the client side to reduce the latency perceived by users. Web traffic and the FTP control channel are responsible for the majority of the small connections which carry around 400 Bytes and 20 KBytes (account for around 80% of all the connections). The remaining large connections are likely FTP data traffic and `iperf` test transfers.

3.2 TRACING MULTIPATH TCP

While many software tools are designed to extract information about TCP connections from a packet trace, such as `tcptrace` [141] or `tstat` [78], these tools do not understand the specificities of Multipath TCP. To analyze our dataset, Hesmans el al. have developed and extended the `mptcptrace` software [99]. The first version of `mptcptrace` [99] was designed to process short packet traces. A summary of this `mptcptrace` tool is presented in the following part.

`mptcptrace` understands the semantics of the Multipath TCP protocol and can extract the keys that are exchanged during the establishment of the initial subflow from the `MP_CAPABLE` option. Thanks to these keys, `mptcptrace` computes the tokens that identify the Multipath TCP connection on both the client and the server. With these keys, `mptcptrace` can link the different subflows that compose each Multipath TCP connection.

The tool recognizes three ways to finish a connection. First, the `DATA_FIN` has been exchanged from both end hosts. Second, one of the two end hosts has sent the `FAST_CLOSE mptcp` option. Third, no activity of the connection has been detected within the last s seconds, which can be set as a parameter. Another option has been added to limit the size of the queue that keeps in memory the mapping between Multipath TCP (or data-level) sequence numbers and the subflow on which it has been sent initially. This queue is

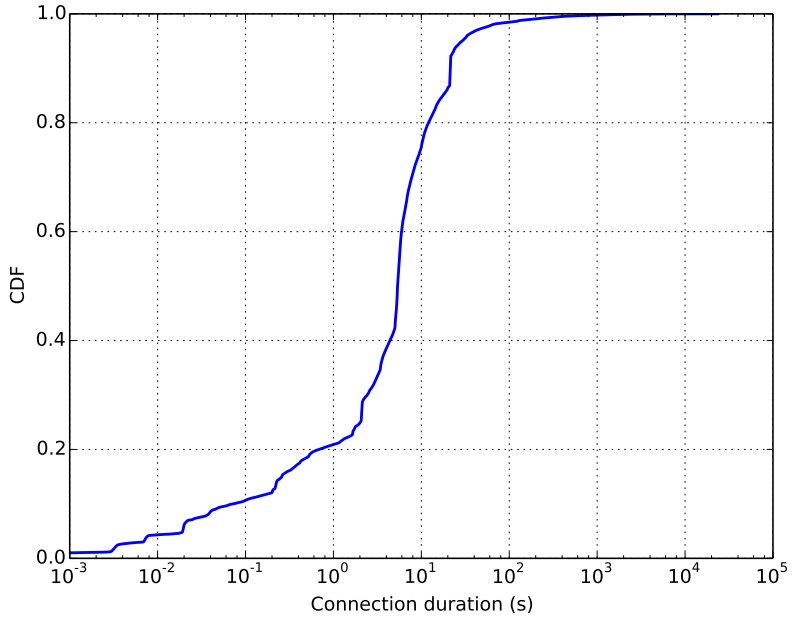


Figure 3.2: Duration of the Multipath TCP connections

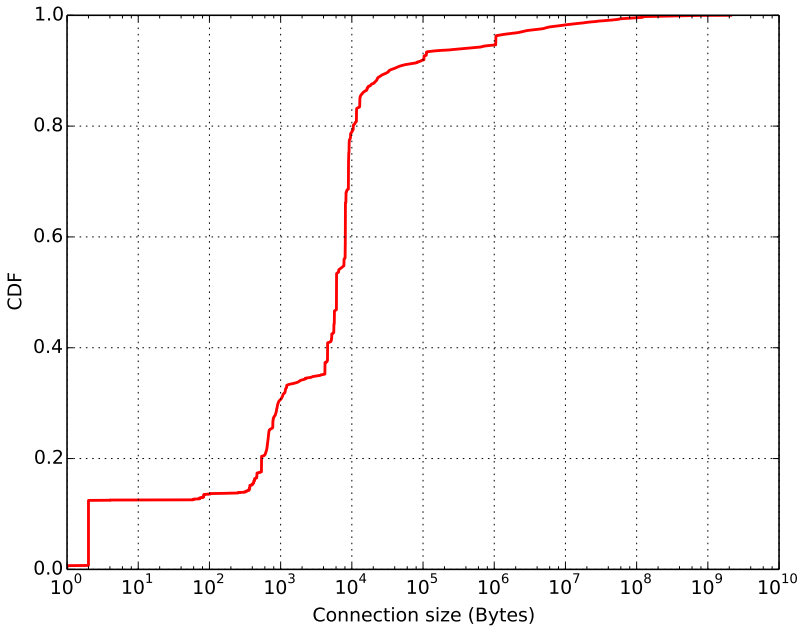


Figure 3.3: Connection size (in Bytes) of Multipath TCP sessions

used to detect reinjections but can become very large if the connection is very long. The internal structure used to store the Multipath TCP connections and subflows has been implemented as hash tables to process long traces faster. New per-file statistics, rather than per-connection statistics (described in [99]) have also been added to ease the analysis. Anomalies detected during the analysis are also reported by `mptcptrace`.

Once the subflows that are associated to one Multipath TCP connection have been grouped together, `mptcptrace` produces detailed plots that provide a graphical representation of the Multipath TCP connection and its subflows. In contrast with `tstat`, `mptcptrace` exploits mainly the contents of the `DSS` option instead of the `SEQ`, `ACK` and other fields of the TCP header. For example, `mptcptrace` can plot the instantaneous throughput at the Multipath TCP level. It can also plot the evolution of the receive window and the amount of data that are in flight. `mptcptrace` also computes global statistics like `tstat` but at the Multipath TCP level. For example, `mptcptrace` computes the duration of each Multipath TCP connection. This is defined to be the delay between the first `SYN` that carries the `MP_CAPABLE` option and the last packet sent over this connection, possibly on another subflow. `mptcptrace` also computes the fractions of the bytes that are sent over each subflow and also detects reinjections (i.e. the transmission of the same data over two or more subflows).

`mptcptrace` is publicly available[2]. In addition, we also implemented a set of Python scripts that post-process the outputs of `tstat`, `tcptrace` and `mptcptrace` for specific analyses. These scripts are also publicly available [3] [4].

3.3 ANALYSIS

This section provides the most important results about both behavior and performance of the protocol, including the interferences caused by middleboxes (Sec. 3.3.1), how fast the hosts establish the additional subflows (Sec. 3.3.2), the round-trip-time pattern of Multipath TCP paths (Sec. 3.3.3), how data is spread over different subflows (Sec. 3.3.4), and the overhead introduced by Multipath TCP compared to regular TCP (Sec. 3.3.5 and 3.3.6).

3.3.1 *Middlebox interference*

Multipath TCP was designed to cope with a wide range of middlebox interferences [80, 101]. If a middlebox interferes too much with the operation of Multipath TCP, the connection should fallback to regular TCP. More concretely, if a middlebox modifies the payload, the mapping between Multipath TCP-level and subflow-level sequence numbers would likely be invalid and corrupt the transferred data. Multipath TCP uses its own checksum to detect any modification of payload, ensuring that the mappings are always correct. If

2 See `https://bitbucket.org/bhesmans/mptcptrace`
3 See `https://github.com/multipath-tcp/mptcp-analysis-scripts`
4 See `https://bitbucket.org/hoang-tranviet/mpmine`

the checksum fails, the receiver will inform the remote peer using the `MP_FAIL` option. Then it will either close this subflow with RST if there are other subflows or it will fall back to regular TCP if this is the only alive subflow.

Among 190,451 Multipath TCP connections, we observe only 125 of them explicitly falling back to regular TCP, which happened with 28 distinct client IP addresses. These include 91 HTTP connections and 34 FTP connections. The FTP interference is expected and due to Application Level Gateways running on NAT boxes which try to "correct" the port number in FTP command to match with forwarded port number. The HTTP interference appeared only on the direction from the server to the client and could have been caused by transparent proxies deployed in cellular and enterprise networks [207]. The rate of fallbacks that we observe is lower than that of earlier work [107]. Note that since we collect data on the server, we cannot detect the removal of the `MP_CAPABLE` option by a middlebox in the SYN. A recent measurement work by Edeline and Donnet [68] in 2019 reports that the `MPTCP_CAPABLE` removal rate is around 0.5% and Multipath TCP is blocked at the rate of one per 500,000 connection attempts.

3.3.2 *Establishment of the subflows*

The number of subflows created by a host depends on various factors including the number of interfaces that it has and its path-manager. As mentioned earlier, the server has a single interface but with dual IPv4/IPv6 stack. Table 3.2 reports the number of subflows per connection (these subflows are not necessarily concurrent). We see that 61.56% of the connections contain only one subflow. This is probably the case in which the client has one interface and does not support dual stack, and the client is configured to not create additional subflows. Another reason for this is that the connection lifetime is too short and there is not enough time to establish a new subflow. Among the connections that have at least two subflows, 49% of them originated from different IP addresses and were likely created by the `full-mesh` path manager[5]. The other 51% of the connections originated from a single address, which normally correspond to the `ndiffports` path manager.

The maximum number of subflows per connection is as large as 68. Notice that this is the number of all subflows during the lifetime, not necessarily active at the same time, so this is not inconsistent with the fact that the Multipath TCP implementation in the Linux kernel supports only 32 concurrent subflows. This unexpected large number was probably due to researchers who performed subflow-rotation tests with the monitored server.

Another interesting point is the delay between the establishment of the connection (i.e. the first subflow) and the establishment of the other subflows.

5 We expect that most of the packets in the dataset were generated by Linux clients. Although other implementations exist, they are not usable for regular traffic (e.g., Apple or Citrix) or not yet mature enough (e.g., FreeBSD). We cannot, however, infer the version of the Multipath TCP implementation used by a client from the packet trace.

Number of subflows	Percentage
1	61.56%
2	23.3%
more than 2	15.14%

Table 3.2: Number of subflows per Multipath TCP connection

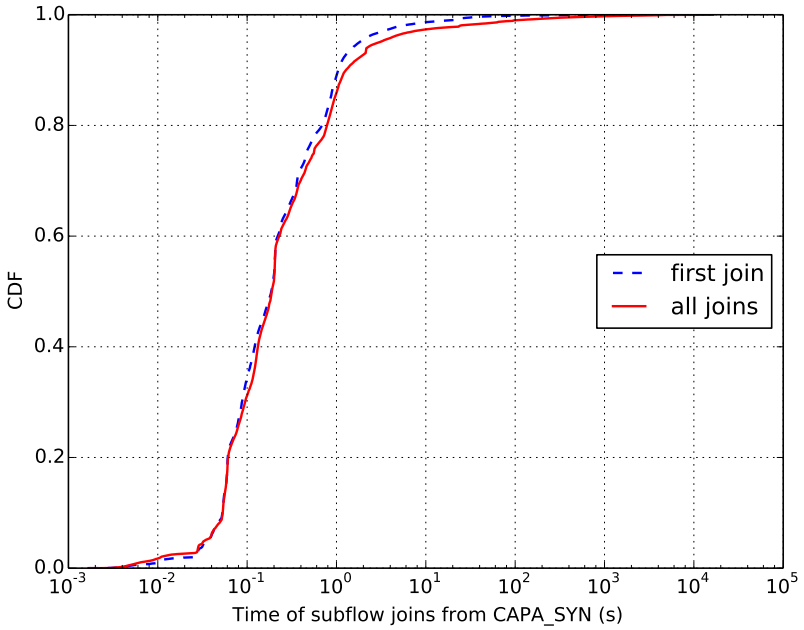


Figure 3.4: Delay from the creation of MPTCP connection to the establishment of an additional subflow

This metric can be used for two purposes: showing how quickly Multipath TCP utilizes additional subflows, and suggesting handovers. The process to establish the subflows is called `join`. With the Linux implementation, path managers try to set up subflows shortly after the creation of the Multipath TCP connection and as soon as a new IP address is learned by the client. To quantify this effect, we plot in Fig. 3.4 the CDF of the delays between the creation of each Multipath TCP connection and all the subflows that are linked to it.

We observe that 86% of the additional subflows are established during the first second of the Multipath TCP connections. Comparing the two lines, we can see the difference with longer duration. After one minute, most of the first joins had been done, but there is still 1.79% of all subflows joined after one

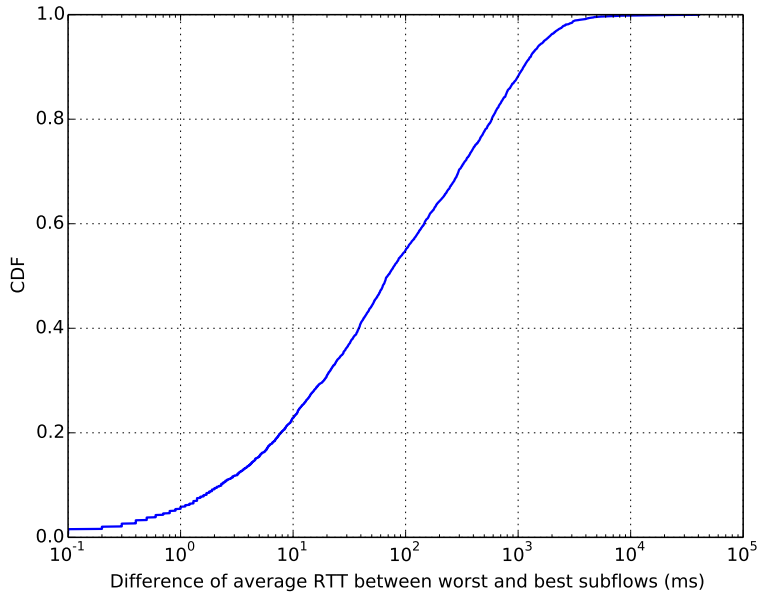


Figure 3.5: Difference of average server-side RTT between the worst subflow and the best subflow of the same MPTCP connection (at least 100 KB)

minute and 0.22% of that after one hour. These joins happening a long time after the connection establishment could be because those connections were experiencing a break-before-make handover or a subflow recreation after a NAT timeout.

3.3.3 *Subflows round-trip-time*

A subflow is established through a handshaking process. Thanks to this exchange, the communicating hosts measure the initial value of the round-trip-time (RTT) for the subflow. For the Linux implementation, the round-trip-time measurement is an important performance metric because the default packet scheduler prefers the subflow having the lowest RTT.

We evaluate the RTTs heterogeneity of Multipath TCP connections. For this, we first extract from the trace the connections that carry at least 100 KBytes. For each connection, we use `tcptrace` to compute the average round-trip-time of all the subflows that it contains, and then extract the minimum and the maximum of these values. Figure 3.5 plots the CDF of the average RTT difference between the fastest and the slowest subflows over all connections. Only 22.6% of the connections have subflows whose RTT difference is within 10 ms.

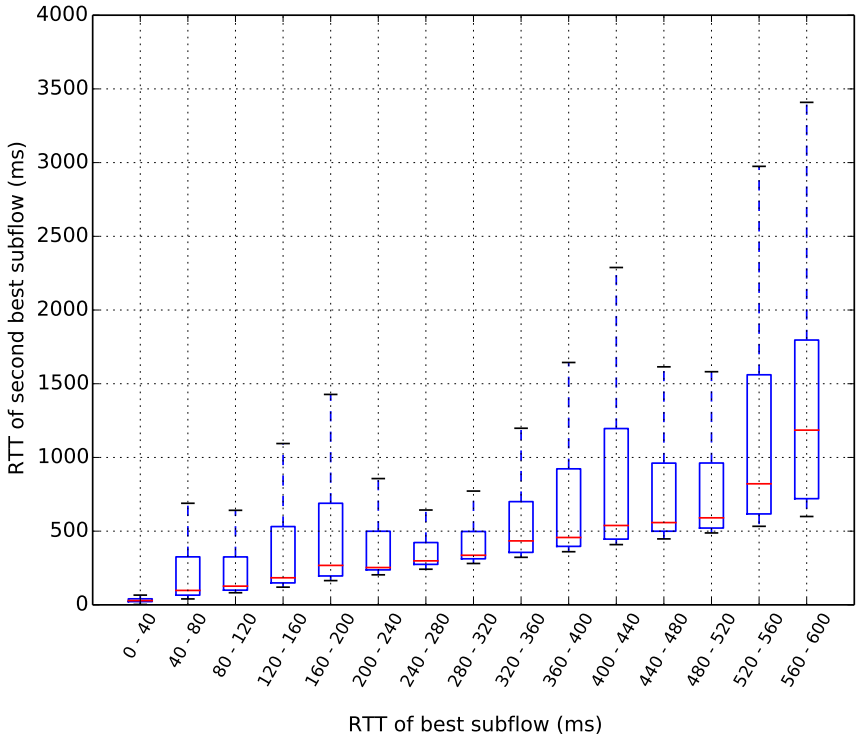


Figure 3.6: The average RTT of the second-best subflow vs. the average RTT of the best subflow of the same MPTCP connection (at least 100 KB)

We observe as many as 12.5% of the considered connections whose RTTs differ by more than one second, which might be caused by bufferbloat. Exceptionally, for some connections, this RTT difference is as high as 22 seconds.

Since the default packet scheduler prefers low RTT subflows, we focus on the two best subflows of each connection. Figure 3.6 shows the boxplot of the average RTT of the second-best subflow in relation with that of the best subflow of the same Multipath TCP connection. Except the first box depicting the connections having best subflow's RTT smaller than 40 ms, the figure shows an observable difference between the RTT of the second-best subflow and the RTT of the best subflow. While the median RTT value of the second-best subflow is not far away from the best one, the tail values may be pretty high.

3.3.4 *Data distribution*

Another important point is the distribution of the data among the available subflows. Observing Multipath TCP connections that contain two or more

active subflows, we see that data can be distributed in very different ways among the subflows. At one extreme, almost all bytes are sent over a single subflow. At the other extreme, packets can be distributed in a round-robin fashion.

A basic question about data distribution is whether the scheduler has bias towards the initial subflow, like the capture effect mentioned in [9]. For this purpose, we extract from the trace all the connections having exactly 2 subflows and compute the percentage of data that has been sent on the initial subflow over the entire connection. Based on the connection size distribution (Fig. 3.3), we examine three different groups of connections: (1) smaller than 50 KBytes, (2) between 50 KBytes and 5 MBytes, and (3) larger than 5 MBytes.

As shown in Fig. 3.7, it turns out that the data distribution of the two-subflow connections suggests some interesting points. Looking at the right part of the graph, there are as many as 29.6% of sub-50KB MPTCP connections which send all traffic over only the initial subflow (the ratio is one). This proportion is smaller for groups of larger connections (13.8% in group 2 and 6.7% in group 3). Moreover, there are around 65.5% of the MPTCP connections larger than 5 MB sending more than half of their traffic over the initial subflow. This bias is even more significant (76.6%) if we consider the MPTCP connections between 50 KBytes and 5 MBytes. Interestingly, only a minority (47.5%) of the connections smaller than 50 KBytes send more traffic over their initial subflow. After checking these connections in detail, we observe that the initial subflow has the RTT worse than that of second subflow in most cases. Usually, the initial subflow carries data during the establishment of the second subflow. The RTT of the initial subflow probably increases over time and is likely higher than that of the second subflow when handshaking without payload. Since the Linux's default scheduler prefers the subflow with smallest RTT, data would be sent through the second subflow. When the connection is very small, the scheduler does not switch back to initial subflow before finishing the transfer.

3.3.5 *Un-used and under-used subflows*

A potential imperfection of Multipath TCP is that subflows can be established without being used to transport data. Creating subflows that are not used consumes resources and energy [103, 129] on smartphones since the interface over which these subflows are established is kept active.

There are three reasons explaining those unused subflows. Firstly, for the small connections, all data exchange can be finished before new subflows are fully established. Secondly, the subflow can be established as a backup subflow. These two reasons explain the fact that 43.15% of the unused additional subflows have better RTT than the active subflow. Thirdly, the difference in round-trip-times between the two subflows can be so large that the subflow with the highest RTT is never selected by the packet scheduler. In fact, 75% of

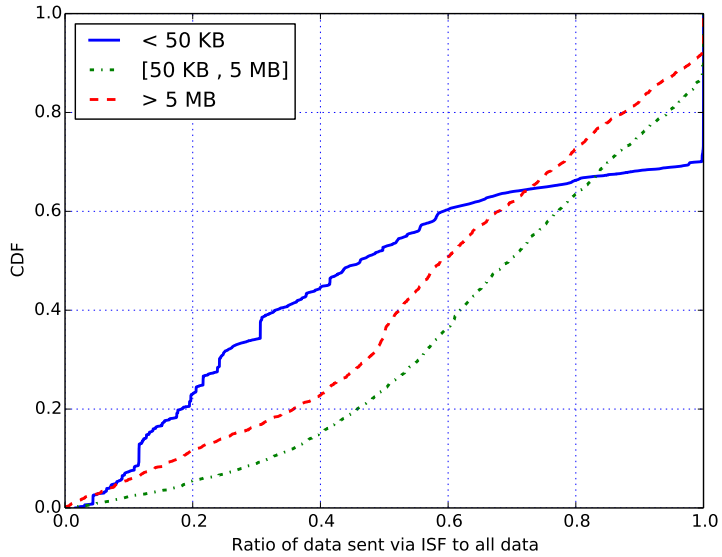


Figure 3.7: Ratio of data sent via initial subflow to all data sent via both subflows.

the connections containing such subflows carry fewer than 1000 bytes, and 95% less than 20 KBytes.

By immediately creating subflows, the `full-mesh` path manager is responsible for some Multipath TCP inefficiencies. An improved path manager [103, 129] would prevent some of these inefficiencies.

3.3.6 *Retransmissions and Reinjections*

A *reinjection* [171], is the retransmission of the same data over two or more subflows. Reinjections can occur for several reasons: *(i)* handover, *(ii)* excessive losses over one subflow or *(iii)* limited windows due to the Opportunistic Retransmission and Penalization (ORP) heuristic proposed in [171] and enhanced in [150]. Similar to TCP retransmissions, a reinjection is a sender's reaction to deal with a performance issue or functional issue.

The Linux implementation of Multipath TCP reinjects data to other subflows only after an RTO happened on the current subflow. This may be too late and may cause a swift transfer stall. A more responsive approach could be reinjecting data when observing a spike of RTT and loss rate on one subflow.

Multipath TCP reinjections are closely linked with regular TCP retransmissions. Since reinjections, by definition, can only occur on connections that contain at least two subflows, we consider only the connections composed of at least two subflows and carrying at least one byte. Figure 3.8 shows the CDF of the reinjections and retransmissions of these connections. The number of

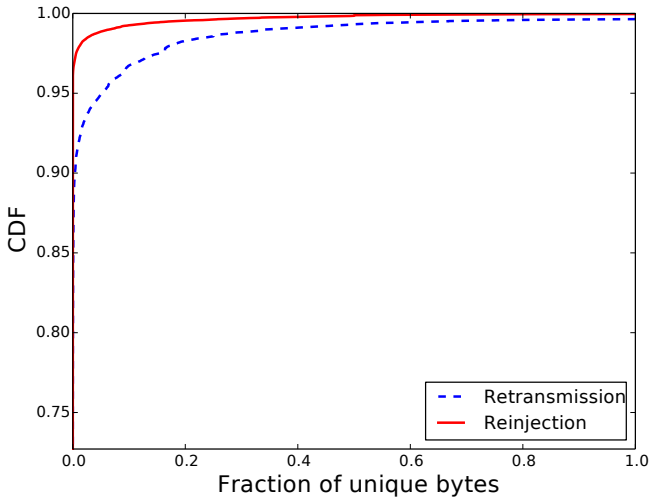


Figure 3.8: The fraction of bytes that are reinjected/retransmitted on the Multipath TCP connections composed of at least two subflows

retransmitted and reinjected bytes are normalized with the number of bytes exchanged over the connection. Since the same data can be sent over several subflows, this fraction can be larger than 1.0 for connections that carry a small amount of data that is retransmitted or reinjected.

We observe that reinjections occur but are less frequent than regular TCP retransmissions. While 25% of the multi-subflow connections do experience retransmissions, only 4.1% of them experience reinjections. 923 MBytes are reinjected and 1323 MBytes are retransmitted, out of a total of 113 GBytes transferred.

To better understand the reinjections and the retransmissions, we plot in Fig. 3.9 the CDF of the normalised times when reinjections and retransmissions occur during each Multipath TCP connection. To produce this plot, we extract from each connection the timestamps of all retransmissions and reinjections and normalize them as a fraction of the duration of the entire Multipath TCP connection.

Moreover, each point corresponds to the timestamp of one retransmitted/reinjected packet. We observe that there are proportionally more retransmissions in the beginning of the connections than close to the end. This is probably because TCP's congestion control algorithm is more aggressive during the slow-start phase than when it enters congestion avoidance mode. For the reinjections, we do not observe in Fig. 3.9 such a strong bias in favor of the beginning of the connections.

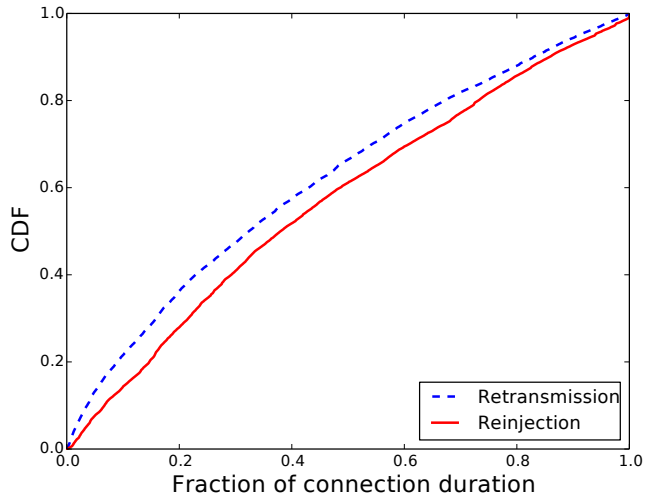


Figure 3.9: Timestamps of reinjections and retransmissions - normalized with the whole connection lifetime

3.4 RELATED WORK

Various researchers have analyzed the performance of Multipath TCP through measurements. Raiciu et al. [169] discuss how Multipath TCP can be used to support mobile devices and provide early measurement results. Pluntke et al. [160] analyze whether Multipath TCP could reduce energy consumption by using several interfaces simultaneously. Paasch et al. [143] propose three modes for the operation of Multipath TCP in wireless networks and describe measurements of handovers. Chen et al. [41] analyze the performance of Multipath TCP in WiFi/cellular networks by using bulk transfer applications running on laptops. Ferlin et al. [74] analyze how Multipath TCP reacts to bufferbloat and propose a mitigation technique. As of this writing, this mitigation technique has not been included in the Linux Multipath TCP implementation. Livadariu et al. explore the performance of Multipath TCP on dual-stack hosts [131] and show that performance over IPv6 and IPv4 paths differ. Ferlin et al. [76] propose a probing technique to detect low performing paths and evaluate it in wireless networks. Lim et al. [128] propose a technique to reduce the energy consumption of smartphones that are using Multipath TCP and evaluate it experimentally. Williams et al. analyze in [209] the performance of Multipath TCP on moving vehicles. Deng et al. [59] compare the performance of single-path TCP over WiFi and LTE networks with the performance of Multipath TCP on multi-homed devices by using active measurements and replaying HTTP traffic observed on mobile applications. Their measurements show that Multipath TCP provides benefits for long flows

but not for short ones. For short flows, they show that the selection of the interface for the initial subflow is important from a performance viewpoint.

Hesmans et al. analyze in [102] a one-week-long subset of the trace that we use in this work. In this chapter, we use a much larger dataset (5 months). This work and its earlier version [102] are the first ones that collect and analyze a large real-world Multipath TCP dataset, revealing some new aspects of Multipath TCP traffic. A similar work to this chapter was done for real smartphone traffic [56] using SOCKS proxies to enable Multipath TCP towards the end devices, while maintaining regular TCP to popular Internet servers. More recently, a large-scale measurement of Multipath TCP on high-speed rails was conducted [127], revealing that Linux Multipath TCP improved the robustness of regular TCP but also had suboptimal throughput in this extreme condition due to very frequent hand-overs.

3.5 CONCLUSION

TCP is probably one of the most studied networking protocols. Hundreds of papers have analyzed its performance and behavior in a wide range of conditions. All this work has led to various improvements that are used in widely deployed implementations [18]. Thanks to the utilization of multiple flows, Multipath TCP brings another dimension to the problem of reliably transmitting data between two hosts. With regular TCP, a host adapts its transmission rate to avoid congestion thanks to its congestion control scheme. With Multipath TCP, when several concurrent subflows are active, the host can spread the data over different paths. Furthermore, the number of subflows that are associated to a connection can vary over time.

Our detailed analysis of a five-month-long packet trace has led to several interesting observations. First, Multipath TCP works correctly over a wide range of Internet paths. This demonstrates the deployability of the protocol extension in the global Internet. Second, the trace reveals two different usages of Multipath TCP. About 86% of the connections are composed of subflows that are created almost immediately.

These connections expect improved performance from the utilization of several subflows. In the remaining 14% of the connections, one or more subflows is created more than one second after the initial subflow. This is likely corresponding to the second use case for Multipath TCP that enables applications to continue to use existing connections after a handover. This use case is very important for mobile devices. Looking at the round-trip-times of the subflows that Multipath TCP uses, our analysis reveals that there can be a huge difference between the fastest and the slowest subflow inside each Multipath TCP connection. This large delay difference must be taken into account by Multipath TCP implementers who propose solutions to improve the performance of the protocol.

We have then analyzed in more details some of the inefficiencies of Multipath TCP. The first identified inefficiency is that the current Multipath TCP

implementation in the Linux kernel often creates subflows that do not carry any data. On a server, this is not a severe problem, but on a smartphone, establishing a subflow on the cellular interface without using it has a cost in terms of energy consumption and radio channel usage.

We have also studied in detail how retransmissions and reinjections occur. Our analysis reveals that reinjections, i.e. retransmitting data over more than one subflow, are common but less frequent than regular retransmissions. We also observed a couple of performance issues due to inefficient receive buffer autotuning and the large RTT difference among active subflows. In general, there are still many aspects of the Linux implementation of MPTCP that could be improved. More recent measurement works [127, 138] have similar observations.

4

ACTIVE MEASUREMENT: MULTIPATH TCP FOR VOICE-ACTIVATED APPLICATIONS

In the previous chapter, we have analysed Multipath TCP traffic on a public domain. However, this dataset contains mostly traditional types of service (web, FTP, echo/discard, proxy, etc.). It is important to know how Multipath TCP performs with newer, modern services. In this chapter, we focus on the usage of Multipath TCP for voice-activated applications.

Voice is progressively becoming a popular way to interact with mobile devices such as smartphones or connected cars. Most of the current deployments depend on cloud services to recognize the user's commands. For this reason, voice-controlled applications have stringent network requirements in terms of delay or availability. On the other hand, many of the devices using such applications are attached to several wireless networks. On iPhones, thanks to Multipath TCP, voice-enabled applications remain available while users move from cellular to WiFi networks or vice versa. This is one of the first Multipath TCP applications which is deployed on a large scale.

In this chapter, we leverage the MONROE platform to analyze the performance of Multipath TCP for voice-activated applications. For this, we port the Multipath TCP Linux kernel code into the Linux Kernel Library so that it can run in user space as a regular application. We extend `iperf3` to emulate voice-activated applications and carry out measurement campaigns. Our measurements show that Multipath TCP brings benefits for users at different levels depending on the network conditions and configurations.

4.1 INTRODUCTION

While voice recognition has a long history of research and development, recent advances in deep learning, the improvement of hardware capability and the pervasiveness of the Internet have enabled a new generation of cloud-based voice recognition platforms, e.g. Apple Siri, Google Speech Recognition, Microsoft Cortana, Amazon Alexa. These platforms have enabled several use cases in various activities of our lives: voice-based search or commands, hands-free infotainment control at home and in connected cars [188], automated customer support [2], voice-based path guides for visually impaired people [17], etc. These systems do not require a high volume of network traffic, but they do present very specific networking requirements: high availability, low latency and energy awareness. These applications typically send voice samples to cloud servers that return the recognised text.

Since 2013, Siri, the voice-recognition and virtual assistant application on iOS, has used Multipath TCP (MPTCP) [80] by default to communicate with

the Apple servers [8]. Given that voice-recognition is one of the largest commercial deployment of Multipath TCP [23], we would like to answer two questions in this chapter: *(1) What are the benefits of using MPTCP for voice-recognition traffic?* and *(2) What are the factors that impact the performance of MPTCP?*

A first possible approach could be conducting passive measurements with real Siri traffic at some vantage points such as on university campus WiFi networks, or on mobile operator gateways. However, this approach is incomplete since Multipath TCP can use different paths and it is difficult to passively collect all the packets sent by a smartphone over both WiFi and cellular networks.

Another approach is to leverage existing mobile measurement platforms to deploy simplified voice-activated applications and conduct active measurements. The challenge now is to have a mature Multipath TCP stack on the mobile nodes used on those platforms. In this chapter, we present a methodology to conduct Multipath TCP measurements on a mobile broadband platform and leverage the Linux Kernel Library [163] to quickly deploy Multipath TCP. We use this methodology to collect sample measurements to demonstrate its benefits by answering the two above questions.

This chapter is organised as follows. Section 4.2 provides some background on the voice-recognition traffic and our observations on recent Siri traffic. Section 4.3 describes the MONROE platform, the challenges to run Multipath TCP on this platform and our approach. The measurement methods and procedures are depicted in detail in Section 4.4. Section 4.5 presents the results of two measurement campaigns and Section 4.6 concludes the chapter.

4.2 VOICE-RECOGNITION TRAFFIC

In this section, we briefly describe the behaviour of voice-recognition applications such as Siri.

Traditionally, unreliable protocols like UDP are used to transport interactive voice traffic to avoid the additional latency induced by reliable transport protocols. However, most of the deployed cloud-based voice-recognition systems use TCP [217] or QUIC[122] - which is on top of UDP but has added reliability - as their transport protocols. This is likely because the responses from servers are typically textual or binary messages, which need to be transferred reliably.

The network traffic generated by Siri has been preliminary analysed in several works [10, 11, 37, 55]. Based on the history of the Siri development and by the reverse engineering effort of previous work [37], it is believed that Siri uses HTTPS as the application protocol layer with a request-response communication pattern. In order to have up-to-date information on the current Siri communication behavior, we have captured Siri traffic on an iPhone running iOS 11.1 (Fig. 4.1). The tests included a series of real-life questions, a current weather query, and a one-off Google search. Below are some of our observations, which are used for traffic emulation in our measurements.

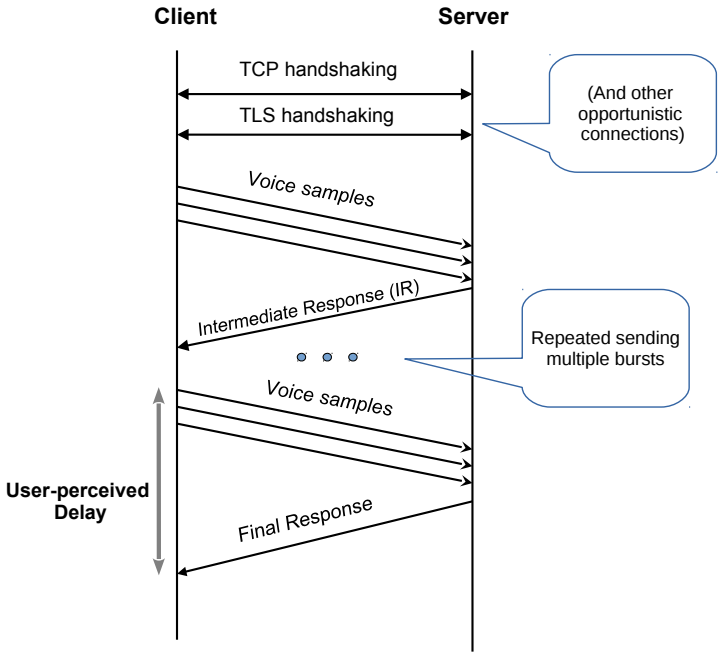


Figure 4.1: The observed Siri traffic pattern

Siri uses TLS 1.2 for encrypting the exchanged data. As an interesting side note, shortly after the beginning of the Siri connection, we also observe that other TCP connections are established towards other servers for other Apple services, e.g. weather forecast, notification update, new system update check. These connections are probably created opportunistically to save the energy consumed by the activation of the network interfaces. Except in the TLS handshaking phase, the client sends data in bursts of several small TCP segments, which likely contain the encoded voice samples. The length of each segment lies in the range of 50-500 Bytes and always has the `PSH` flag set, showing that the `TCP NO_DELAY` socket option has been used. After each burst, the server sends back a small response. It is believed that Siri uses HTTPS as the application protocol for the transaction [37], in which the voice samples are included in HTTP POST messages. Thus, the small responses that we observe are probably the HTTP `100 Continue` informational status messages. When all voice samples have been received and processed, the server sends back its final response. After that, the connection is not closed immediately by the client nor the server, but is maintained persistently for a long time. This process is illustrated in Figure 4.1.

4.3 MPTCP MEASUREMENTS: CHALLENGES AND APPROACH

We use the MONROE platform [3] for our measurements since it supports various multi-homed wireless nodes. To realise our MPTCP measurements on the MONROE platform, we have to overcome a technical challenge. Since the platform only allows experimenters to run their tests inside Docker container [20], we cannot run the experimental MPTCP stack in the Linux kernel directly. To run a Linux MPTCP implementation in user-land, we extend the Linux kernel library to provide Linux MPTCP to the application.

4.3.1 *The MONROE Platform*

MONROE [3] is a large multi-homed mobile-broadband measurement platform. It enables the experimenters to access and control hundreds of APU-based nodes. Each node is equipped with multiple cellular interfaces, or with one WiFi and one cellular interface. These nodes could be stationary or mobile (which are placed on trains, trucks, or buses). While the platform is shared among different experimenters, the access to each node is given exclusively to one experimenter at one time to avoid interferences among concurrent experiments. Detailed descriptions of the platform can be found in several measurement papers [121, 159]. For example, [121] presents simple but extensive download/upload measurements on this platform. In our experiments, we observed - on more than half of the nodes - that the `MP_CAPABLE` option [80] used by Multipath TCP to establish connections was removed from the SYN packet on port 80 towards an external server. However, the MPTCP connections on other port ranges (5201 to 5300) do not suffer from this problem. This suggests that HTTP transparent proxies are used in the cellular networks attached to these nodes, confirming the observation in [121].

The MONROE platform is based on a container technology. The users have to prepare a Docker image to deploy on each MONROE node, and then Docker instances run in user space. However, using Multipath TCP requires kernel upgrades, which is challenging to deploy on the MONROE nodes. We solve this problem by using the Linux kernel library [163] to allow an application to use a custom MPTCP-enabled network stack without requiring any kernel change on the host.

4.3.2 *Linux Kernel Library overview*

The Linux kernel library (LKL) [163] is essentially a library that enables users to use custom Linux kernel code directly inside applications. Unlike a typical build of Linux kernel source tree which produces a bootable image, a build of LKL generates a set of library files which contain the Linux kernel but live in userspace. An application can link the LKL library in order to call alternate system calls (instead of the ones of the host kernel) implemented in LKL.

We leverage this feature to use the Multipath TCP network stack for our test application.

After traversing packet processing in the LKL system calls (completely operated inside userspace), a packet goes through a virtualized device driver composed of Linux kernel code, which is a `virtio` driver implementation, and destined to a `virtio` device of LKL to be transmitted to the outside. LKL supports a series of `virtio` devices: raw socket, tap device, Virtual Distributed Ethernet (VDE) [53], Intel Data Plane Development Kit (DPDK) [112], etc. With those devices, the incoming and outgoing packets processed by the network stack of LKL do not need to pass through the stack of the host operating system.

There are several other options to achieve the customized network stack: User-mode Linux (UML) [62], QEMU [16], or a custom userspace MPTCP implementation. First, UML can also introduce a custom Linux kernel with MPTCP extension executed in userspace. However, an UML instance is usually bundled with a complete Linux runtime which involves various initialization procedures with multiple process invocations such as shell scripts for configurations, DHCP client, etc. This results in a longer boot time than LKL even if the new network stack is only used by a single application. Second, while a complete Linux operating system on top of QEMU with hardware-assisted virtualization is the most transparent solution to utilize a virtualized system on top of an existing operating system, virtualization is not cheap. Furthermore, its overhead may be too much for this solution to run on APU devices. Finally, a custom MPTCP implementation in userspace would have plenty of options to meet requirements of a restricted environment such as the MONROE platform. Such an approach looks tempting but it also drops the maturity of the code quality in terms of both suboptimal performance and functionalities that LKL (UML, or Linux with QEMU) does not have.

4.4 MEASUREMENT DESIGN

The MONROE platform consists of not only stationary nodes which are located in four European countries (Norway, Sweden, Italy and Spain) but also mobile nodes which are set up on trains (Norway), buses (Sweden, Italy), and trucks (Italy). For our experiments, we selected 28 stationary nodes and 40 mobile nodes. Each of them has two cellular interfaces which are connected to different mobile operators. Our server is located in our university campus in Belgium.

Both clients and server run version 0.93 of the Multipath TCP implementation in the Linux kernel [142]. On the client, we merge the source code of LKL and Linux Multipath TCP and build them as a library to the client application [1]. We also use the enhanced socket API [100] to only use the wireless interfaces for creating the subflows. We use the OLIA (Opportunistic Linked-Increases

1 Merged code is available at `https://github.com/hoang-tranviet/mptcp/commits/lkl_4.13-mptcp_v0.93_API`

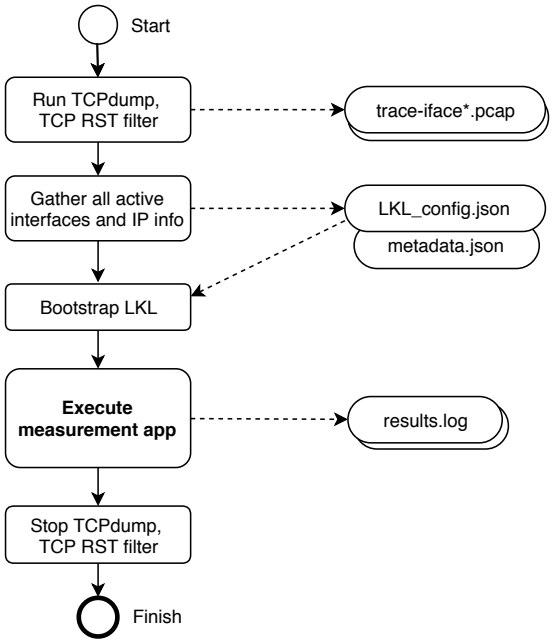


Figure 4.2: General Experiment Procedure with LKL

Algorithm) congestion control [120] which is a coupled one and is proven to be stable. For packet scheduling, we use the default *Lowest-RTT* scheduler.

4.4.1 *Measurement Procedure*

Figure 4.2 depicts our general procedure for the experiments with LKL on clients. While this procedure was originally designed for voice-activated traffic measurements and for the MONROE project, it can be adapted to run on other measurement platforms and with other custom Linux network stacks. At the beginning of the experiment, we run `tcpdump` to capture packets sent by both clients and server. The current LKL implementation uses the `virtio` device and communicates with the outside world through raw sockets. After bootstrapping the LKL, the main measurement application can run. It stores its results in log files. The next subsection elaborates this stage with our simulated voice-activated application (Fig. 4.3).

4.4.2 *Measurements with Voice-activated Applications*

Since the implementation of popular voice-recognition systems like Siri, Alexa or Google Assistant are closed-source and their traffic is encrypted, we use simulated traffic for our measurements. For this purpose, we modified the

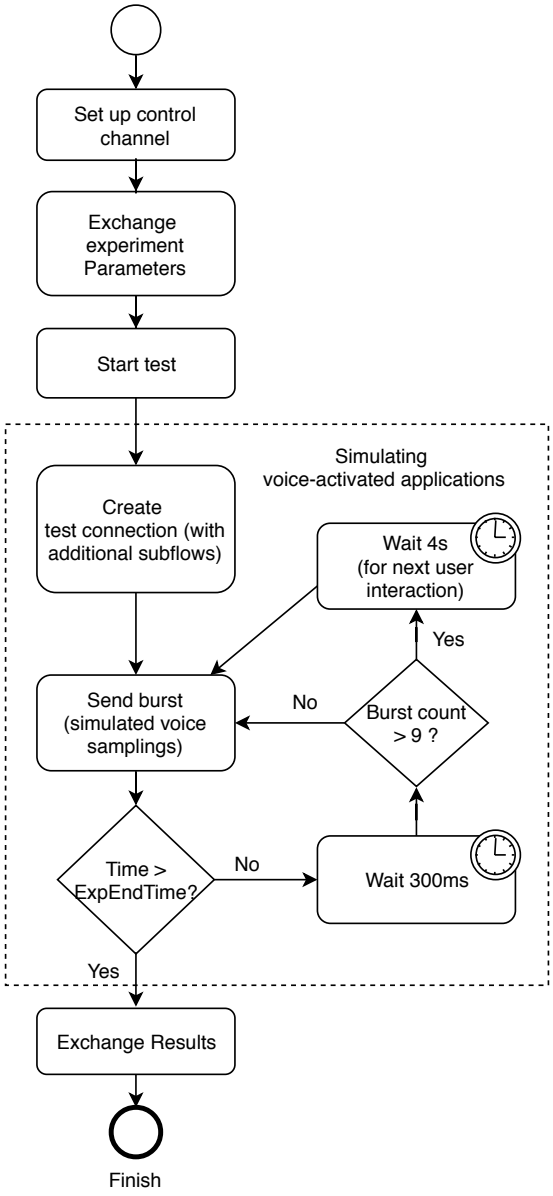


Figure 4.3: Execution Flow of Measurement on client (simulating voice-activated applications)

popular `iperf3` measurement software [63] on both clients and server [2]. The original `iperf3` software uses a separate control channel between the clients and the server. We modified this channel to exchange the experiment parameters at the beginning of each test as well as the results after the data transfer. Given that latency is an important factor in our measurements, we enabled the `TCP NO_DELAY` socket option on the clients and the server. We also configured the Linux stacks with TCP Small Queue and Tail Loss Probe enabled.

The emulated traffic is based on our observations on Siri traffic pattern, as presented in Section 4.2. As shown in Fig. 4.3, after a client connects to a server, it starts sending voice data in a series of bursts. Each request consists of 9 bursts and each burst spans 10 TCP segments (ranging from 50 to 500 Bytes) on average. For every burst, the server may respond with a small reply corresponding to the HTTP `100 Continue` (Intermediate-Response). The inter-burst time is set to 300 msec. Once the server has received all request data, it immediately sends back a 750 Bytes response to the client. Our version of `iperf3` uses the enhanced socket API [100] to have more control on the creation of subflows. For users, the important metric is the request-response delay, which is defined as the delay between the transmission of the last burst and the arrival of the first response packet. A similar metric has been used in previous work [10, 11].

4.5 SAMPLE MEASUREMENT RESULTS

In this section we present some results of two measurement campaigns [3]. The first one compares the performance of TCP and MPTCP. The second campaign evaluates the performance of different configurations of MPTCP.

4.5.1 *MPTCP versus TCP*

Initially, we ran the client application on 28 different *stationary* nodes towards our server. Each node performs five transactions with the server within 60 seconds (which gives five request-response delay samples), and the entire experiment is repeated three times. For this experiment, Figure 4.4a shows that Multipath TCP provides much better Request-Response Delays than regular TCP.

Then, we ran the measurement with similar configuration but this time on 40 *mobile* nodes. As shown in Fig. 4.4b, it is clear that the average delay in this case is much higher than that of the stationary nodes, with much longer tails that are not fully shown here. This is understandable given that the mobility of nodes likely reduces the connection quality, significantly increasing packet losses and delays. Notice that for this reason, we plotted this figure with a different range (0 to 2 seconds) than the previous one (0 to 1 second). We can see in this case that the difference between the two protocols is not always

2 Source code is available at `https://github.com/hoang-tranviet/iperf-siri`

3 Measurement scripts and collected data is available at `https://zenodo.org/record/1290469`

significant. Multipath TCP delivers similar performance as TCP when the delay is good. However, when delay is bad, Multipath TCP with the default configuration could clearly reduce the tail of delay.

Digging into details, we also collected the *Received signal strength indication (RSSI)* of the default cellular interface on each node, as shown in Fig. 4.5 and Table 4.1. The high correlation between the delay and the signal strength in the case of TCP shows that the signal quality on the last mile has a significant impact on the overall perceived delay. For MPTCP, there is no clear correlation between the user-level delay and the signal strength. This is likely the result of the Lowest-RTT scheduler decision, which tends to switch the outgoing packets from the first subflow to the second subflow when the RTT on the first subflow increases (as explained in Section 2.4.2).

4.5.2 *Different MPTCP server configurations*

In this campaign, we use two different schedulers on the server: *Default scheduler* and *Server scheduler*. As mentioned earlier in Section 2.4.2, the *Default scheduler* of the Linux implementation prefers the subflow with the lowest round-trip-time. Meanwhile, the *Server scheduler* [55] was designed for servers that serve mobile devices. This scheduler always prefers to transmit data over the last subflow on which it received data or a new data acknowledgement.

On the server side, the default (*Lowest-RTT*) scheduler may take undesired decisions in some cases. For example, consider a client that has both WiFi and cellular interfaces and initially sends data through the WiFi path. If the WiFi connectivity fails, e.g. because the user moves, then the server still sends traffic through this path since it does not know about the failed WiFi connection on the client side. The *server scheduler* [55] tries to avoid this problem by choosing the most recently used subflow on which the server has just received client data. To be concrete, it remembers the timestamp of the latest original packet received on each subflow. A packet is considered original if it contains new data based on its Data Sequence Number, or if it contains new Data-ACK that advances the left edge of data-level send window. More information about this scheduler can be found in [55].

For any scheduler, the up-to-date information about each subflow should play an important role in improving data transfer performance. As described in Fig. 4.1, the Siri server sends an intermediate response (`100 Continue`) after each received burst. To reveal the performance impact of these responses, we run the experiment with two different behaviors: servers send back these intermediate responses (default behavior) or do not send them back (the tests with "No-IR" annotation in the figures).

In all situations, there is no clear difference of delay in the first 70 percentiles. This represents the situations in which the default path is always the best path, so there is no impact of using different MPTCP configurations. The main differences are in the last 30% percentiles. In the case of mobile nodes (Fig. 4.6b), the server scheduler gives better performance for mobile nodes by

Client	Stationary nodes	Mobile nodes
TCP	-0.34	-0.20
MPTCP Default	0.03	-0.11
MPTCP Server	0.07	-0.10
MPTCP Default No-IR	0.01	0.11
MPTCP Server No-IR	0.01	-0.10

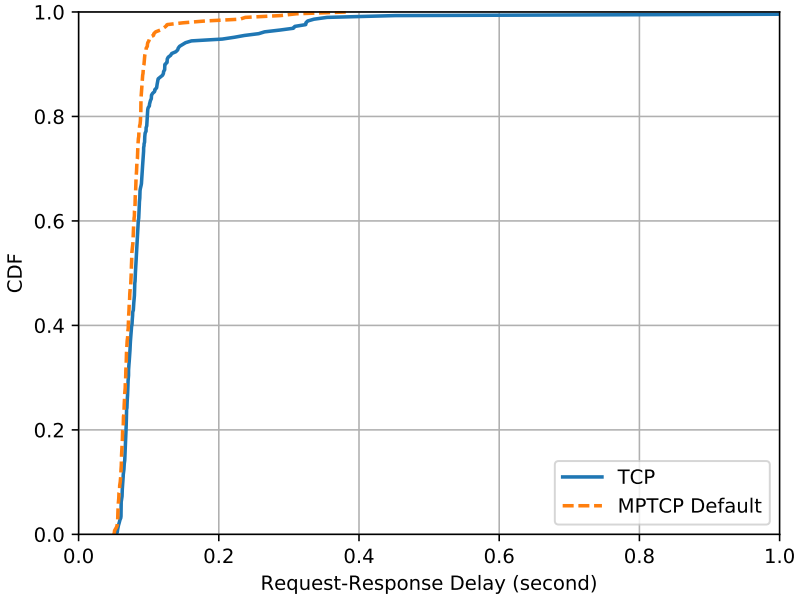
Table 4.1: Correlation between Request-Response Delay and the RSSI of the default interface

keeping track of the most recent working subflow. However, for stationary nodes (Fig. 4.6a), the server scheduler gives nearly identical results. This is expected given that the connectivity status of each client interface is generally unchanged.

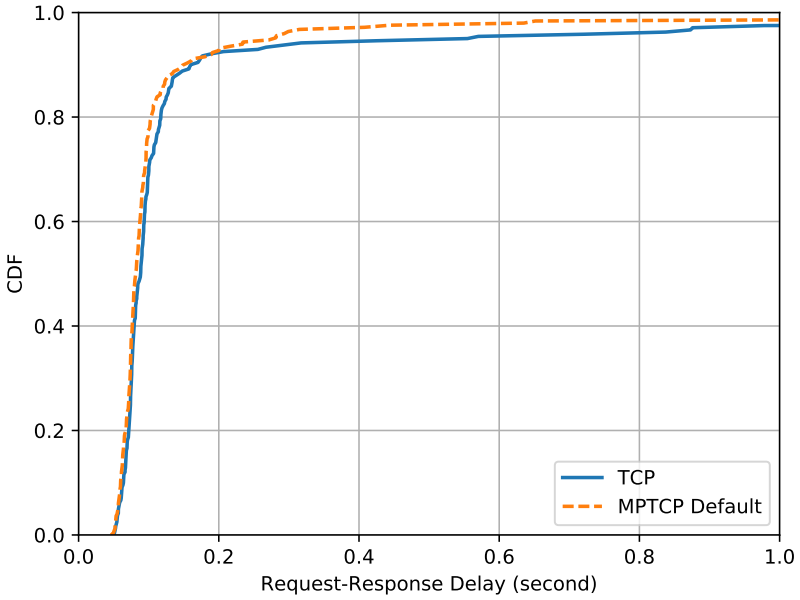
Additionally, when the server does not send intermediate responses, the delay increases significantly for both schedulers, showing the importance of up-to-date RTT information on the server side. While the *server scheduler* relies passively on the incoming traffic to select the subflow, the *intermediate responses* play the role of the active probing on the current status of subflows. A more effective and generic solution is to let the scheduler conduct these active probes regularly on unused subflow, independently from the application layer. This approach has been proposed in several recent works [85, 126], which reduce the tail delay significantly.

4.6 CONCLUSION

Voice-activated applications are more and more popular and will likely play a more important role in the future. In this chapter, we have extended `iperf3` for modeling the network behaviour of such applications. We have then proposed and implemented a measurement methodology that leverages the LKL library to use a specific networking stack without changing the underlying Linux kernel. This is key to be able to test new protocols on platforms such as MONROE or Planetlab. We demonstrate the benefits of this approach on the MONROE platform. Our measurement results show that Multipath TCP with a proper configuration could help improving the user-perceived delay in various network conditions. Moreover, to achieve good performance with Multipath TCP, it is important to improve the senders' awareness about the current state of each subflow.

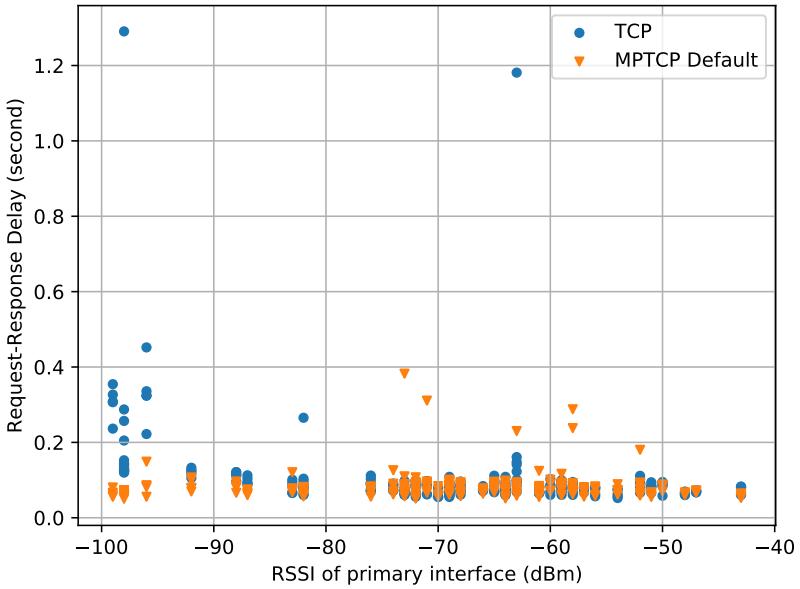


(a) Stationary nodes

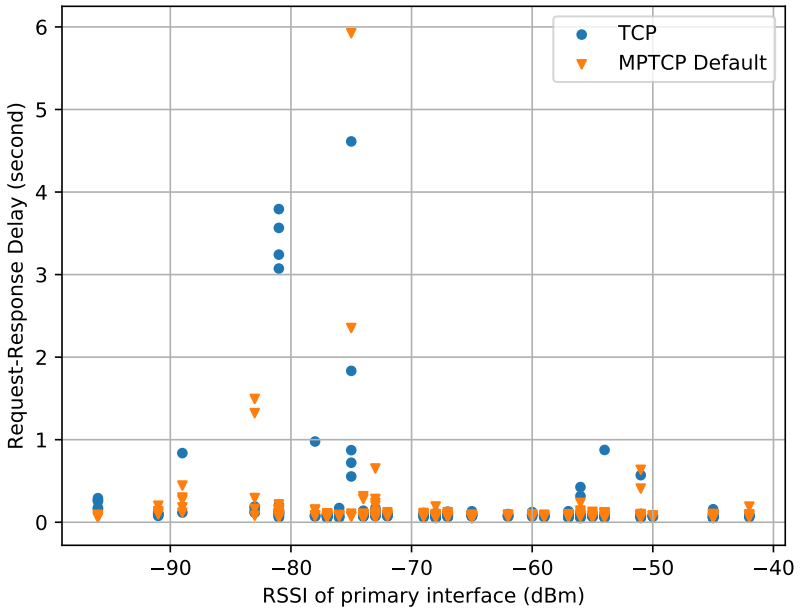


(b) Mobile nodes

Figure 4.4: Request-Response Delay: MPTCP vs. TCP

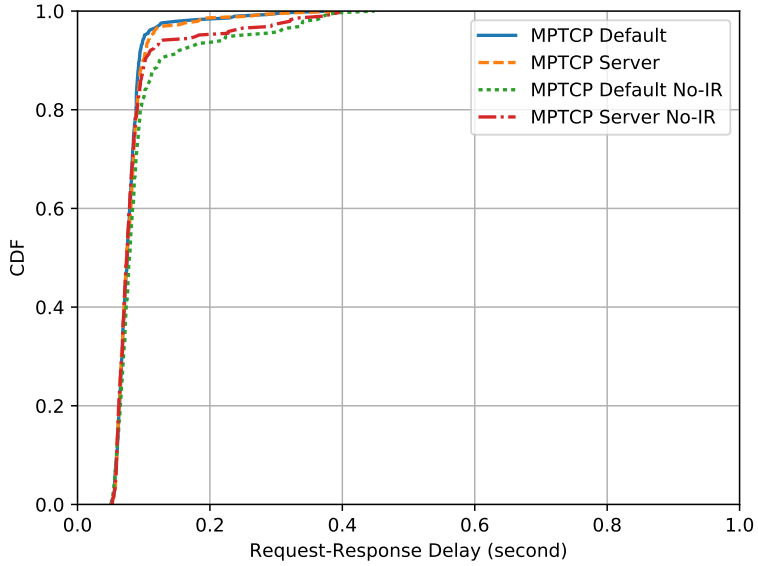


(a) Stationary nodes

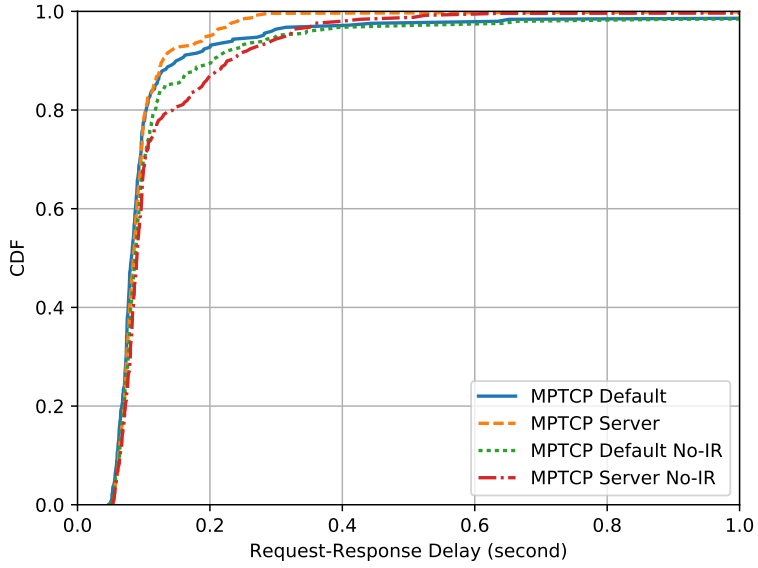


(b) Mobile nodes

Figure 4.5: Request-Response Delay vs. Signal Strength



(a) Stationary nodes



(b) Mobile nodes

Figure 4.6: Request-Response Delay: various MPTCP configurations

Part III

REINVENTING THE LINUX STACK

The previous part has shown that current Multipath TCP implementations could be improved in several aspects, and should be customized for realistic scenarios. Unfortunately, Multipath TCP stacks typically reside in the kernel space, which makes them difficult to be extended or customized. In this thesis, we focus on the Linux networking stack due to the popularity of Linux servers and Android clients. While this thesis is about Multipath TCP, we realized that the difficulty of extending Multipath TCP stack is a generic problem. Therefore, in Chapter 5 we start by extending regular Linux TCP stack. Then, we apply a similar approach to Multipath TCP in Chapters 6 and 7.

5 EXTENDING LINUX TCP USING EBPF

The Transmission Control Protocol (TCP) remains one of the most important protocols in today's Internet. It was designed to be extensible for various use cases. A client can propose to use an extension over a given TCP connection by sending an option that identifies this extension during the three-way handshake, while a few other options such as User Timeout Option [70] can be sent directly without negotiation. That's the theory that all networking students learn in networking textbooks. In practice, deploying a TCP extension is much more difficult as the maintainers of client stacks often wait until servers implement a given extension and server maintainers look at clients in the same manner. It often takes several years to actually deploy an option at a large scale. In this chapter, we focus on the Linux TCP stack since it is one of the most widely used TCP stacks, given its utilisation on many servers and Android devices.

Our goal is to support experimenting and deploying new TCP options in a quick, simple, and efficient way. This includes inserting new TCP options at the sender side and parsing them at the receiver side. The implementation and the interface should be simple, generic, and introduce as few changes to the kernel code as possible.

5.1 INTRODUCTION

The designers of TCP did not expect that it would be used by billions of devices, but they did foresee the importance of designing an extensible protocol. TCP's extensibility depends on two important factors: (i) the extensibility of the protocol and (ii) the extensibility of its implementations.

To be extensible, the TCP protocol supports TCP options that can be placed in the TCP header. A TCP connection starts with a three-way handshake during which the client proposes a set of extensions as TCP options placed in the SYN packet and the server replies with its supported options. The accepted TCP options can then be attached to the other packets exchanged over this connection. Various TCP extensions have been proposed during the last decades: TCP Timestamp and large windows [114], Selective Acknowledgements [132], TCP Fast Open [42], Multipath TCP [80] and so on. However, deploying a new TCP option takes time. It needs to be defined, accepted by the IETF and then implemented by major TCP stacks. Measurements show that Selective Acknowledgements took more than a decade to be widely deployed [87] and the Timestamp option is still not enabled by the Microsoft stacks [109]. More recently, middlebox interference became an important concern [68, 107] which ossifies the Internet infrastructure.

The second, and often forgotten, factor is the extensibility of the TCP implementations. For many years, the Unix 4.x BSD stack has served as the reference TCP implementation [214]. When Van Jacobson wrote his seminal paper on congestion avoidance and control [115], his work had a large impact because his code was quickly integrated inside this reference implementation. Today, this stack is less popular than the Linux TCP stack that is used by a large fraction of Internet servers and all Android smartphones. This Linux stack has been extended to support TCP Fast Open [166], Multipath TCP [171] and many other TCP extensions. The TCP stack in Linux 1.0 in 1994 contained 3k lines. It grew to 18k lines in version 2.6 (2010). Today's TCP implementation spans more than 80k lines of C code in the Linux kernel. Most of the recent additions to the Linux TCP stack have been driven by the needs of large content providers.

The Linux TCP stack is highly optimised for the most common use cases, but it has very limited ability to *adapt* to a changing environment of network conditions, workloads or user requirements. It can be tuned through a myriad of `sysctl` parameters [1]. These parameters allow to tune many TCP aspects e.g. delayed ACK timeout, ACKing strategy, congestion control scheme. More importantly, the `sysctl` interface only allows changing system-wide or per-network-namespace behaviors, but it does not support per-connection policies. Some of these parameters and others are exposed as socket options[2] or via socket syscalls that can be set by applications on a per-connection basis. However, it is difficult and hacky, though not impossible, for the system administrators to use these socket-level interfaces.

As will be explained in Section 5.2, some researchers have proposed techniques to extend the Linux TCP stack, but these approaches do not allow to read or write new TCP options. We consider that supporting new TCP options is a crucial part of a truly extensible framework for TCP. In short, the main contributions of this chapter are as follows:

1. We propose and implement a light eBPF-based framework that enables users to easily add support for new TCP options in the Linux TCP stack
2. We propose four use cases that leverage our framework to adapt the stack to various scenarios or user requirements.

The remainder of this chapter is organized as follows: Section 5.2 discusses the related work and presents the objectives of this work. We present our methodology and implementation of our TCP option framework in Section 5.4. Several use cases for new TCP options are presented in Section 5.5. Finally, Section 5.6 discusses the insights and future work.

1 See `https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt`
2 See `http://man7.org/linux/man-pages/man7/socket.7.html`

5.2 STATE OF THE ART

Transport protocols such as TCP can be implemented inside the operating systems' kernel [214] or as a library inside the application. The main motivation of kernel stacks is that a single stack can support all applications, ensure that they do not interfere and achieve high performance [48]. A drawback of in-kernel implementations is that they are more difficult to extend than user space ones. On the other hand, user space implementations are more flexible, but they are often less mature than the in-kernel ones. Recent advances have enabled user space implementations to reach higher performance [117].

5.2.1 *In-kernel approaches*

Several researchers have proposed solutions to simplify the extension of in-kernel implementations. STP [154] was an ambitious effort to allow end hosts to load untrusted code from remote peers to upgrade their transport protocols. The idea of loading user code into a sandbox in the kernel is similar to the utilisation of the eBPF virtual machine in today's Linux kernel.

The idea of exposing and allowing applications to set internal state variables of TCP connections was proposed earlier [91, 133]. This permits the control plane of TCP congestion control to be moved from kernel to userland [91, 136]. This also enables adding new non-intrusive features [91], as long as they do not change the wire format or the internal state of TCP. In terms of performance, this approach requires costly switching back and forth between userspace and kernelspace for both reading and writing parameters.

5.2.2 *Userland approaches*

Besides kernel stacks, there are complete user-space TCP stacks [66, 67, 117]. Their nature makes them be easier to be modified by application developers than the in-kernel TCP stacks. However, they often lack many crucial features (e.g. PMTU discovery) or the rich ecosystem of supportive facilities (notably but not limited to `iptables, namespacing, cgroup`) and debugging utilities. New transport protocols such as QUIC [123] were designed with user space implementations in mind. Several QUIC implementations are being actively developed[3]. The QUIC protocol was designed to be easier to extend than the TCP protocol and its encrypted packets should prevent most types of middlebox interference. However, a portion of networks currently block (4.4%) [123] or rate-limit UDP traffic.

The Linux Kernel Library (LKL) [164] (which was mentioned in Section 4.3.2) is a compromise between in-kernel and user space implementations since it wraps a custom Linux network stack into a user library, allowing each application to use a different Linux network stack. This approach allows

3 See `https://github.com/quicwg/base-drafts/wiki/Implementations`.

applications to use new features (e.g. TCP Fast Open, MPTCP) even if updating the host kernel is not possible or not desirable. However, it currently induces some memory overhead and the dynamicity of the network stack has not yet been considered.

5.2.3 *Current Linux kernel facilities*

The Linux kernel includes several facilities which can be used to extend its TCP implementation. First, the Linux TCP stack provides the socket option interface to observe or change the state of the underlying TCP connections. For example `TCP_INFO` socket option returns many state variables. Some mobile applications use it frequently [178]. Second, `Netlink` [175] establishes channels between kernel space and user space. It has been used to support user-level control plane for MPTCP path manager [104]. However, this approach requires the addition of a lot of code into both kernel and userspace, causing both memory and processing overhead.

A low-level way to change the kernel execution path is to use `kprobes`' capability [118] of changing the register set and instruction pointer. It allows capturing some information when a specific kernel function is executed. However, this approach is highly fragile and prone to error, which could lead to serious consequences such as kernel panics or kernel data leaks.

Another approach is to implement each extension as a loadable kernel module. For example, multiple congestion controllers are implemented as kernel modules that are loaded dynamically into the Linux TCP stack. The active congestion controller can be selected through a `sysctl` or configured on a per connection basis once loaded. While we could implement other features as kernel modules, loading user code directly into kernel without restrictions is very dangerous in general.

A comprehensive approach is to build a custom sandbox which allows userspace to load custom code into the kernel and change the behavior of stack, similar to STP [154]. However, there was no follow-up work since the date of STP. Recently, extended BPF (eBPF) has emerged as a safe and efficient way to add programmability into the mainstream Linux kernel. The next section gives a brief summary of this eBPF infrastructure.

5.3 EBPF EXECUTION ENVIRONMENT

The Classic BPF (cBPF) virtual machine is a part of the Linux kernel for more than two decades. It has been mainly used to write filters to capture packets, which is the core part of famous tools such as `tcpdump` and `Wireshark`. Since the BPF filter runs in the kernel space, it may capture a large amount of traffic and consume too much system resources. The cBPF virtual machine allows users to specify a subset of the traffic flows (e.g. only capture TCP traffic towards a specific IP address on port 80), and to optionally capture the packet headers instead of the full packets.

Recently, this virtual machine has been thoroughly extended and renamed the extended BPF (eBPF). It supports several use cases such as sandboxing system calls (`seccomp`), tracing kernel events [89], implement hyperupcalls [7]. Several networking use cases already leverage eBPF. For example, XDP uses it for fast packet processing [106], IPv6 Segment Routing uses it to support network programming [215] and it improves the extensibility of Open vSwitch [201]. Motivated by its success in the Linux kernel, there are ongoing works to port eBPF to userspace [162] or to the FreeBSD kernel [96].

5.3.1 *eBPF Virtual Machine*

Classic BPF provides a small number of 32-bit RISC instructions. To take advantage of modern hardware, the eBPF virtual machine was designed to closely resemble modern CPU architectures. The most important changes include using 64-bit registers and increasing the number of user-accessible registers from two to ten. Nine of them are general-purpose read-write registers, and one is a read-only stack pointer. Table 5.1 explains the role of these eBPF registers. The eBPF Virtual Machine also includes a program counter and a 512-byte stack. Additionally, eBPF simplifies the just-in-time (JIT) compilation, gaining much better performance.

Each eBPF bytecode, called eBPF program, runs inside this virtual machine. One important remark is that the execution of eBPF programs is *event-driven*. Before the Linux 5.1, the size of these eBPF programs was limited at 4096 instructions to make sure its execution could be terminated quickly and avoid the kernel lock-up.

There are multiple eBPF program types. Each program type cam interact with only one or a subset of kernel subsystems. Each eBPF program needs to be associated with a single BPF context object during its execution. Register R1 always stores the pointer to this context object. For example, a BPF program can directly operate on either a socket context or an `skb` packet context, but not both.

5.3.2 *eBPF maps*

eBPF maps are efficient key-value data storage structures. They are the main storage for BPF programs. They could be used for sharing data between userland and an in-kernel BPF program, or among BPF programs of different types. eBPF maps are typically created from userspace and are handled by a file descriptor. All maps are defined by a set of four values: a type, a maximum number of elements, a value size in bytes, and a key size in bytes. The key is used to store and retrieve data value. There are two kinds of eBPF maps: the generic ones that store any arbitrary kind of data, while the non-generic ones are used to store data of a specific type. For example, `BPF_MAP_TYPE_PROG_ARRAY` is used to store a list of BPF programs, while `BPF_MAP_TYPE_SOCKMAP` can only

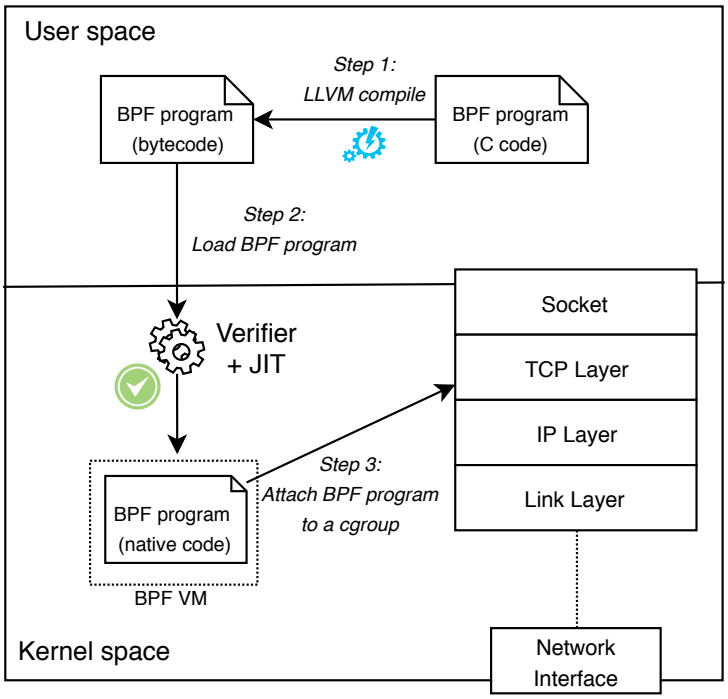


Figure 5.1: A user application compiles and injects eBPF program into kernel

Register	Purpose
R0	stores return values
R1-R5	stores function call arguments
R6-R9	is preserved on helper function call
R10	points to the per-program stack

Table 5.1: eBPF Registers and their current purposes

store the socket information. Maps of both kinds can be accessed by generic functions: `bpf_map_lookup_elem()` and `bpf_map_update_elem()`.

5.3.3 *eBPF helper functions*

Each eBPF program is sandboxed. By default, it can only access the context object passed to it. To read and change other in-kernel objects, it has to call eBPF helper functions. Each helper function is defined with a function signature, similar to the system calls, allowing the verifier to perform type-check to make sure the access is safe and secure and allowing these functions to be JIT compiled efficiently. The eBPF authors decided that all BPF helper functions are part of the core kernel and cannot be extended via kernel modules. This is to encourage eBPF developers to merge their internal-used helper functions into the mainstream Linux. There are many helper functions in the latest Linux kernel and this number is quickly increasing.

5.3.4 *In-kernel verifier*

Each eBPF bytecode needs to be verified before being injected inside the kernel. This job is carried by an in-kernel verifier, ensuring that the eBPF bytecode cannot harm the running kernel. This task is more critical when a non-privileged user tries to load an eBPF program. This is possible since some program types do not require the `CAP_SYS_ADMIN` privilege.

The verifier performs multiple checks before actually loading an eBPF program into the kernel. First, it needs to make sure that the program execution will terminate quickly to avoid kernel lock up. This is done by building a control flow graph (CFG) of the program and doing a depth-first search. Then, the verifier simulates the execution of the eBPF program, making sure that the states of registers and stack are always valid, and the memory accesses are bounded. Finally, each program type could only use a subset of map type and is restricted to call a subset of helper functions.

5.4 METHODOLOGY

As explained earlier, the standard method to extend TCP is to define a new TCP option. In the early days, researchers introduced new TCP options and registered them with the IANA. Then, the IETF took control of most of the evolution of the TCP stack and most recent TCP extensions have been discussed within the IETF. Today, researchers willing to deploy a new TCP option cannot anymore simply register their new option within IANA. The IETF has defined a format for experimental TCP options [189], which we could leverage in our work to minimize the possibility of middlebox interference when using new TCP extensions.

From an implementation viewpoint, a TCP extension can be added to the Linux kernel as a set of patches. This approach has been used by many researchers (see e.g. [166, 171]). However, users are forced to recompile their kernels with those patches to support the proposed extension. This severely limits their deployment.

A better approach is to leverage as much as possible the eBPF execution environment that we mentioned above. Thanks to eBPF, any application can inject code inside the underlying TCP stack to modify its behaviour (as shown in Fig. 5.1). For example, an interactive application running on a smartphone could inject a retransmission technique that is optimised for short packets while a datacenter server could inject another congestion control scheme. This injection could be done directly by the network application or by a system daemon in userspace. Before loaded, the eBPF code needs to be passed through a static verifier to make sure it is both secure and fast. The eBPF code can be executed in an efficient way thanks to the JIT compiling support.

As explained in the previous section, several use cases have been developed for eBPF in the Linux kernel. These address various components of the Linux kernel and many focus on performance monitoring. In 2017, Lawrence Brakmo proposed the TCP-BPF framework [30] which is specifically built for the TCP stack and provides basic support to extend the TCP stack. We leverage TCP-BPF as a starting point for our work.

5.4.1 *An overview of TCP-BPF*

Before the introduction of TCP-BPF, there were generic built-in eBPF helper functions in the Linux kernel for directly reading from and writing onto packet data buffer, notably `bpf_skb_load_bytes` and `bpf_skb_store_bytes`. While this approach may fit with lower layer operations, e.g. replacing source/destination IP addresses, changing ToS value, it is extremely difficult to control the TCP layer because the BPF programs are not aware of the TCP connection, nor have any information about the dynamic and internal states of TCP connections.

TCP-BPF has been gradually added [28, 29, 32] into mainstream kernel in versions 4.13 through 4.15. It was mainly designed to help network administra-

Hook	Calls a BPF program when
BPF_SOCK_OPS_CONNECT_CB	An active connection is initialised
BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB	An active connection is established
BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB	A passive connection is established
BPF_SOCK_OPS_STATE_CB	TCP changes state
BPF_SOCK_OPS_RTO_CB	Retransmission timeout happened
BPF_SOCK_OPS_RETRANS_CB	A packet is retransmitted
BPF_SOCK_OPS_TIMEOUT_INIT	To set per-connection SYN-RTO
BPF_SOCK_OPS_RWND_INIT	To set per-connection initial rwind

Table 5.2: Notable TCP-BPF hooks (available in mainline Linux 4.17)

tors to tune the TCP configurations of servers in datacenters at the connection level. The main objective of TCP-BPF was to optimize the TCP parameters in a programmable manner. For example, TCP-BPF would configure the stack to use small buffers and a small SYN retransmission timer for a container that includes applications running inside a given datacenter. However, a different eBPF code would be used for applications that perform bulk transfers between datacenters.

TCP-BPF [30] adds several callbacks (also called hooks by the authors) to call BPF programs at different stages of a TCP connection, as shown in Table 5.2. In this thesis, we use the terms callback and hook interchangeably. There are two main types of callbacks. The first type is the callback in the slow path of each connection: e.g. when the client calls `connect()` or when the server calls `listen()` or when the connection is fully established. These callbacks are always enabled. On the contrary, callbacks of the second type are only enabled once they have been requested by a BPF program to limit the overhead when they are unused. These include callbacks triggered when the RTO fires, when a packet is retransmitted, or when the TCP connection state changes. TCP-BPF allows BPF programs to read and write to many fields of data structures (the `tcp_sock`) maintained by the TCP stack via a mirror structure `bpf_sock_ops`. It also provides indirect access to other internal TCP variables via two helper functions `bpf_getsockopt()` and `bpf_setsockopt()`. All these hooks call the BPF programs by using the same helper function `tcp_call_bpf()`. Since a BPF program can be called from different places in the kernel, the hooks are also associated with an argument (`op`) to indicate the callback type to let the BPF program know the current context in the kernel. Additionally, some TCP-BPF callbacks also affect the connection parameters through their return values, e.g. `BPF_SOCK_OPS_RWND_INIT` to set the initial receive window for the connection.

Since TCP-BPF was implemented by Facebook engineers to work in data center environment, it requires `cgroup` version 2 to manage various system resources such as CPU or memory for their containers. For this reason, it is necessary to attach the BPF program to the same `cgroup-v2` of the user

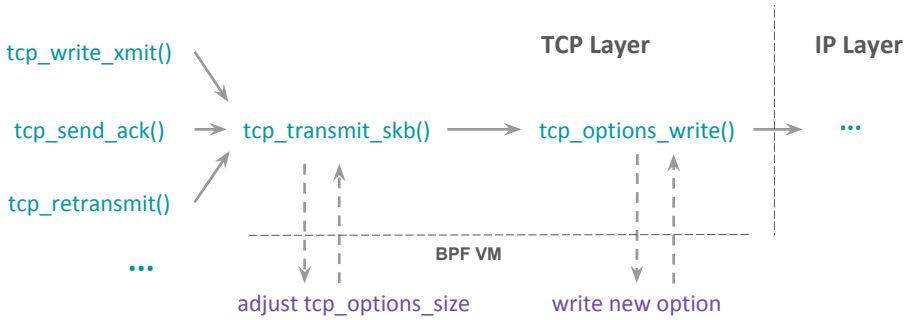


Figure 5.2: Insert TCP options to outgoing packets

application. However, this is not a permanent requirement, rather it should be considered as an implementation caveat which can be changed later.

5.4.2 *Supporting user-defined TCP options*

5.4.2.1 *At the sender: Inserting a new TCP option*

As an illustration of how it is possible to use eBPF programs to extend the Linux TCP stack, we first describe the changes that are required to support a new TCP option. Table 5.3 summarizes the new hooks added by our framework and their meaning.

Let us first analyse the sender side. When sending packets, the function `tcp_transmit_skb()` creates the TCP header and the required TCP options. TCP options are written in two steps: (i) the stack computes the size of all provisioned TCP options and (ii) it writes the TCP options in `tcp_options_write()`. Therefore, to insert a new TCP option we add two separate hooks into above places, as illustrated in Fig. 5.2.

We first add into the function `tcp_transmit_skb()` a hook which calls the TCP-BPF program to adjust the provisioned size of all TCP options (`tcp_options_size`). We also verify that it does not exceed 40 bytes - the maximum size of the TCP option header section. Then, at the end of `tcp_options_write()`, a second hook calls a BPF program which passes the new option data to the kernel. The kernel is then responsible for writing the new option data at the current option pointer. To avoid the overhead on the TCP fast path, these hooks are only activated when the BPF program sets the appropriate flag (per connection in struct `tcp_sock`, as explained below).

There is still one thing the framework has to take care of. Since the TCP stack calculates the current MSS at multiple places, the composed packets may be too large and could be fragmented on the wire. We update the `tcp_current_mss()` function to take the length of to-be-added option into consideration. This is performed by a hook with the same `op` type as the above

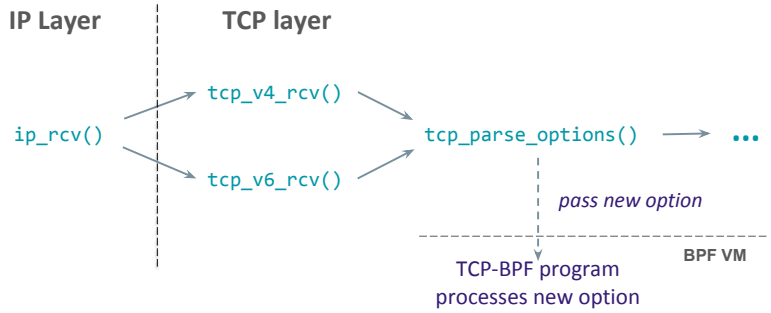


Figure 5.3: Pass unknown TCP options of incoming packets to BPF program

Hook	In kernel function	Passed arguments	Purpose
BPF_TCP_OPTIONS _SIZE_CALC	tcp_transmit_skb tcp_current_mss	Size of all TCP options	BPF prog adjusts the size of all options
BPF_TCP_OPTIONS_WRITE	tcp_options_write	-	actually inserts option
BPF_TCP_PARSE_OPTIONS	tcp_parse_options	option kind, size, data	BPF prog parses option

Table 5.3: New BPF hooks added by TCP option framework

hook (which adjusts `tcp_options_size`) that is added to `tcp_current_mss()` and thus is completely transparent to the BPF programs.

5.4.2.2 *At the receiver: Parsing unknown TCP options*

On the receiver side, the extension is simpler. Linux TCP parses the options of incoming TCP packets in `tcp_parse_options()`, in which all new options which are unknown to the stack are ignored. At the end of this function, we added a hook to pass these unknown options to the BPF program, as shown in Fig. 5.3. This hook, once activated, passed the option data along with option kind and length to the BPF program. The hook can also pass several new options of the same TCP packet to one or more BPF programs. The BPF program reads the option and applies a relevant change to the TCP socket, e.g. by setting socket values via `bpf_sock_ops` or `bpf_setsockopt()`.

5.4.3 *How to select the desired packets for inserting a new option?*

The first question is how to select the relevant connections. A user daemon can specify the cgroup that the targeted connections are associated with, before loading the BPF program. At runtime, the BPF program can check the 4-tuple of IP addresses and ports to only take care of the interesting connections. These operations are already supported by the vanilla kernel so no kernel change is required.

	Kernel changes	BPF program
TCP Option framework	75	-
Use case: TCP User Timeout	16	76
Use case: Congestion Control	0	92
Use case: Initial Window	0	76
Use case: Delayed ACK	94	77

Table 5.4: Lines of code (LoC) of the framework and each use case

The second question is how to insert new options in the desired packets only. To mark when the program wants to actually insert new options, we add a new flag. TCP-BPF already uses a flag array (`bpf_sock_ops_cb_flags`) in the `tcp_sock` struct for enabling and disabling the hooks at different phases of a TCP connection. We extend this flag array with our flag to minimize the amount of changes. The BPF program can set the flag at one hook (e.g. when the connection is fully established) to enable option writing onto all following `skb`s of the same TCP connection, and unset the flag at another hook (e.g. when the RTO fires) to disable option writing from this point.

5.4.4 *Code changes*

By building on top of TCP-BPF, we can implement our framework with modest changes to the kernel (75 LoCs). The TCP-option-insertion support requires around 60 LoCs, while the TCP-option-parsing support requires only 15 LoCs since it is much simpler as explained above. Table 5.4 lists the size of our framework and each use case with regards to the number of lines of code (LoC) changed in the kernel.

We added a minor kernel change to support getting and setting internal TCP user timeout value directly in eBPF program, while current kernel has already supported setting and getting Congestion Control algorithm or Initial Window. The implementation to support configurable TCP Delayed ACK, which is essentially based on an unmerged RFC patch [88] proposed by Ben Greear and Daniel Baluta, is reasonably larger.

5.4.5 *Performance Overhead*

Linux TCP is a high-performance stack. Any proposed extension should take the performance impact into consideration. To evaluate the performance impact of our BPF extensions, we run an iPerf3 [63] test between two dedicate machines over a 10 Gbps link. Each machine is equipped with an Intel Xeon X3440 2.53 GHz CPU and 16 GB RAM. Our framework is implemented in Linux kernel version 4.17-rc5. We use different TCP-BPF programs that

are called to manipulate *each* transmitted packet. We consider four different experiments.

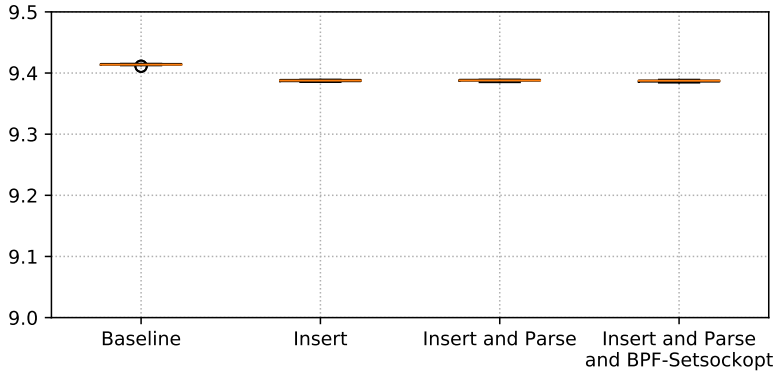
1. Baseline, no BPF program is loaded
2. A BPF program inserts a new TCP option on the sender
3. A BPF program on the sender (to insert a new option) and one on the receiver (to parse this new option)
4. A BPF program on the sender that inserts a new option while the receiver parses this option and then calls both `bpf_setsockopt()` and `bpf_getsockopt()`

Each measurement lasts 40 seconds and each scenario is repeated 20 times. Figure 5.4 shows the benchmark results reported by iPerf3 for each situation. The average throughput is reduced from 9.41 Gbps in the baseline case to 9.38 Gbps in all three BPF-enabled scenarios, mostly because our newly inserted TCP option has increased the TCP header size. Meanwhile, there is no statistically meaningful difference of round-trip-time among all cases (all around 410 microseconds) therefore we do not present them here. The CPU utilisation overhead is the most noticeable one which is about 10% in the worst case, as shown in Figure 5.4b and Figure 5.4c.

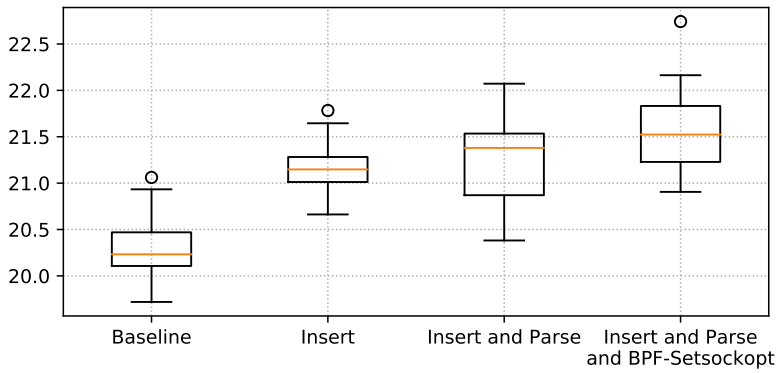
To push the TCP stack to the limit, we conducted another extreme benchmark with the iPerf3 client and server on the same host machine. This benchmark tries to send as much data as possible via the loopback interface to saturate the TCP stack, which is an extreme but unrealistic scenario. As shown in Figure 5.5, the average throughput obtained with baseline tests is 30.1 Gbps (about 2.5 Mpps) and the average RTT is 27.1 usecs. Using a BPF program that inserts a new TCP option introduces a throughput reduction of about 12.7% and a delay increment of 14.8% (4 usecs). Using a BPF program that parses a new TCP option reduces further the throughput by 3.8% and increases the delay by 4.5%. Calling operations such as `bpf_getsockopt` or `bpf_getsockopt` does not have a noticeable impact. These results suggest that most of the overhead of the framework comes from the call-backs to the BPF program, not from the execution of the BPF program itself.

5.5 USE CASES

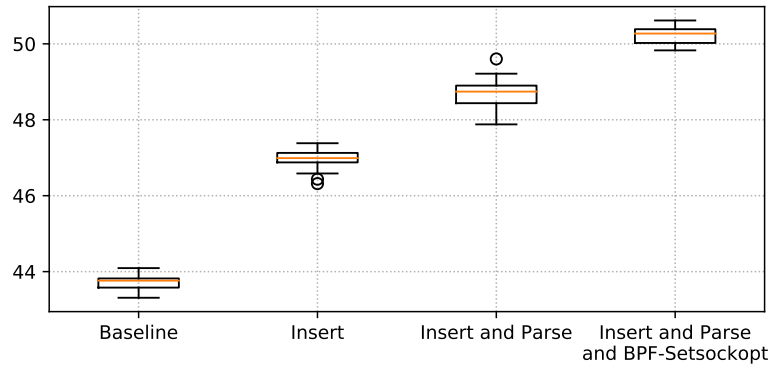
In this section, we demonstrate a variety of use cases how it is possible to leverage BPF programs to extend the Linux TCP stack. We start in Section 5.5.1 with the TCP User Timeout Option [70] that has not been implemented in the Linux TCP stack. We then propose and implement in Section 5.5.2 a TCP option that enables a client to suggest the congestion control scheme to be used by a server. We then propose a TCP option to set the initial congestion window in Section 5.5.3 and finally how eBPF code can be used to tune the acknowledgement strategy in Section 5.5.4.



(a) Average Throughput (Gbps)

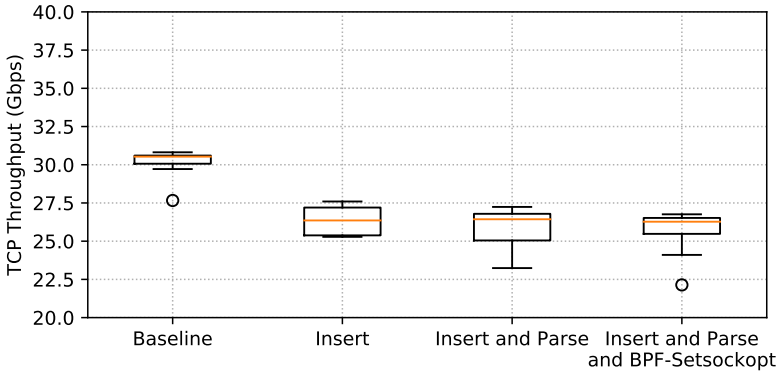


(b) Sender's CPU Usage (%)

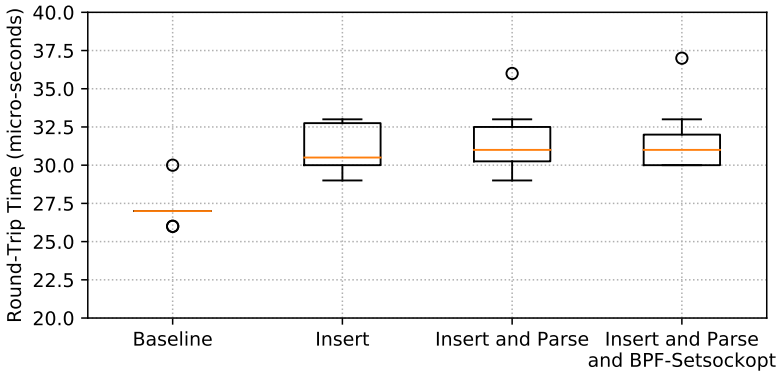


(c) Receiver's CPU Usage (%)

Figure 5.4: Benchmarking results: iPerf3 test over a 10Gbps link



(a) Average Throughput (Gbps)



(b) Average Round-trip Time (usecs)

Figure 5.5: iPerf3 stress test over the loopback interface

5.5.1 *TCP User Timeout Option*

The TCP User Timeout (UTO) option [70] was proposed to allow a host to inform its peer of the maximum time that data could remain unacknowledged before forcing the termination of the associated connection. There are several use cases for this option. First, an application that wants to survive transient failures would select an UTO value larger than the default. The second situation is contrary: many interactive applications on smartphones equipped with Wi-Fi and cellular interfaces could use a short UTO (e.g. one second) to quickly detect connectivity problems and switch to the other network interface. As the third use case, a busy server can also announce a small User Timeout value to let clients know that it may not keep the connections experiencing intermittent unavailability.

The UTO option [70] carries the suggested timeout value. It is sent unreliably, typically inside a TCP ACK. In contrast with most TCP extensions, the utilisation of this option is not negotiated during the three-way handshake. It is simply used once the connection has been established. Linux allows applications to set the maximum value of the retransmission timers through the `TCP_USER_TIMEOUT` socket options. However, it does not announce the UTO as a TCP option. In Linux, when the UTO timer fires, the kernel signals a timeout error to the user application and changes the connection state to `TCP_CLOSE`. However, it is the responsibility of the application to terminate the connection with TCP RST.

On the client side, we implement the UTO option support with a BPF program (76 lines of C code) using our option-writing hooks described in the previous section. On the server side, when it receives a UTO option from the peer, the kernel stack passes the option to a BPF program that parses the option and sets the local socket timer value by leveraging the `bpf_setsockopt()` helper function. We also extend the `bpf_getsockopt()` helper function to query the current User Timeout value of the connection.

5.5.2 *TCP Congestion Control Option*

The Linux TCP stack supports a dozen of pluggable congestion control modules [50]. Depending on its configuration, a Linux host may directly support two to three TCP congestion control schemes, e.g. NewReno [4], CUBIC [93], or Vegas [27] or BBR [35]. Content Distribution Networks (CDN) often tune their congestion control scheme to better serve their customers [36]. However, a given CDN supports a variety of customers and a congestion control scheme that works well to serve a user connected through an optical fiber might not work well for a user connected over a slow ADSL link. Some CDNs tune their TCP stack on a per-prefix basis, but there are many situations where the client that downloads information from a server has much better knowledge on the performance of its access network than the server. For example, a smartphone can easily collect statistics about the amount of reordering and the delay variations that it has observed recently. Based on this information, it could suggest a specific congestion control scheme to be used by a given server.

In our implementation, each supported TCP congestion control scheme is identified by an integer. The mappings between the TCP congestion control schemes and their identifiers could be distributed together with the Linux kernel.

Our BPF programs on both the client and the server store the list of congestion control algorithms in an array map. This map contains algorithm IDs as the keys and the string names as the corresponding values. When the server receives the congestion control option, the BPF program extracts the identifier and looks it up in the map to retrieve the name of the requested algorithm. It then changes the congestion control scheme applied to this connection using the `bpf_setsockopt()` helper function.

To illustrate the utilisation of this congestion control option, we set up the emulation environment similar[4] to Mininet [124]. We set up separate network namespaces for client and server, a Linux bridge in-between, and using Traffic Control (TC) with HTB qdisc to set link bandwidth to 8 Mbps and 40 ms delay per direction. Our emulated client downloads the same large file using the `curl` software. We use our BPF program to insert in the third ACK packet the TCP congestion control option to request the utilisation of a specific congestion control scheme by the server.

We consider NewReno [4], CUBIC [93], Vegas [27] and BBR [35] in our experiments. These four congestion control algorithms correctly use the 8 Mbps link, but they differ in the amount of bufferbloat that they cause. Figure 5.6 plots the round-trip-times measured by the server for each congestion control scheme. We repeated the tests multiple times, but they produced nearly identical graphs. Vegas and BBR, the delay-based algorithms, have the lowest Round-trip times (RTT) which are close to the two-way link delays. While Cubic escaped the slow-start phase early, it does not prevent the RTT from increasing. Among all, NewReno performs worse in terms of delay.

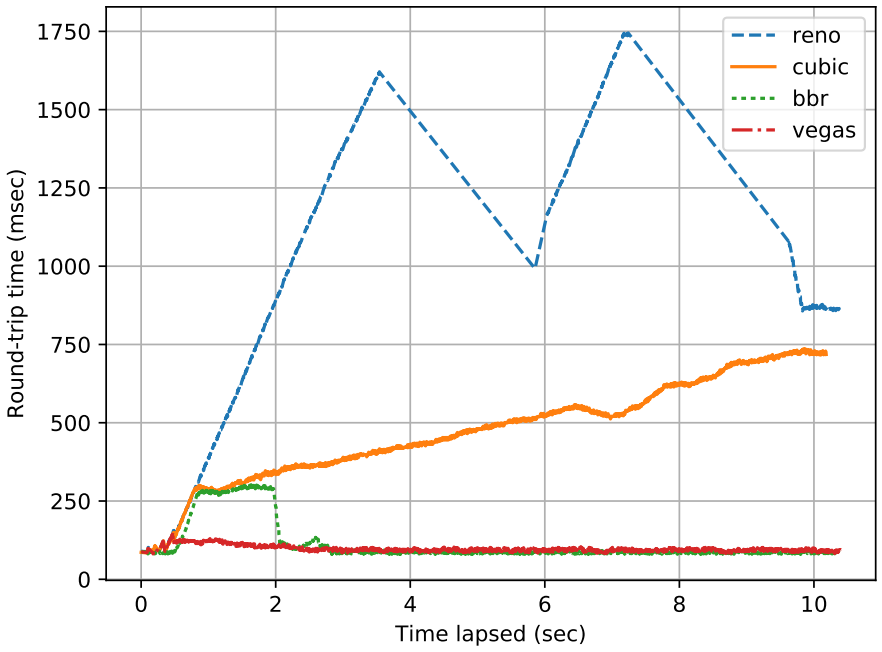


Figure 5.6: Congestion Control Option test: RTT on the server (8 Mbps bandwidth, 40 ms link delay)

4 We do not use Mininet directly but use directly built-in facilities in Linux (`netns`, `tc`,...) because Mininet uses `cgroup v1` while `cgroup v2` is currently required by `tcp-bpf` framework.

In this example, we used the congestion control option to exchange the identifier of the congestion control scheme that the peer should use. The same option could also be extended to provide some parameters of the congestion control scheme. For example, Google QUIC [123] uses a variant of CUBIC that is more aggressive than the standard one. This was motivated by the fact that a QUIC session is equivalent to several HTTP/1.1 sessions since it supports streams. The same applies to HTTP/2 running over TCP.

5.5.3 *Option to Request Initial Congestion Window*

While the congestion control algorithm has a significant impact on the performance of long flows, the selection of the initial congestion window (IW) decisively affects the flow completion time for short flows. This clearly applies to web traffic. The standard IW value has increased over the years from 2 MSS to 4 MSS [5] and later 10 MSS[47, 64] to keep up with typical network speeds without harming the robustness of the whole system. However, a fixed value cannot adapt to various network conditions. On long fat networks, the sender usually takes a lot of time to reach the congestion avoidance state. But the same IW value may be too large in highly congested networks.

Recent large-scale measurements [173, 174] show that while most web servers use the default values of their TCP stacks, CDN operators usually apply much larger values of IW [174]. These measurements also suggest that some CDNs customize their IW configuration based on the network and/or content type.

Brakmo suggested [30] to heuristically select the IW based on the IP prefix using TCP-BPF, with a simple example [31]. We extend this approach by defining a new TCP option that lets a client specify its desired IW value. In many deployments, the receivers have more information about the impact of the IW than the senders by observing packet losses at the beginning of connections. However, this opens up the possibility that the malicious peers may use this option to leverage DoS attacks. To deal with this class of attacks, we use two mitigations. First, we restrict that this option can be sent only in the SYN-ACK or third ACK of the three-way handshake, but not in the first SYN packet. This also helps implementing the server side more easily since the Linux TCP initializes the full socket only after the completion of the 3-way handshake. Second, the sender needs to verify the peer is from a trusted IP prefix before setting the requested IW value. This client IP verification could be done directly in the BPF program. The BPF program can also combine client requests with local policies, e.g. take the content type into account when selecting proper IW for the connection.

To demonstrate the impact of tuning the initial congestion window with web traffic, we use the methodology proposed by Wang et al. [206] with the `epload` software [205]. This enables us to emulate real web contents and gather web page download times.

We set up a similar testbed to the previous use case in Section 5.5.2. The path between client and server was configured with 40 Mbps of bandwidth and 40 msec of delay per direction. The server uses `nginx` to serve the mirrored web contents of top Alexa 170 websites list. On the client side, we ran the `epload` tool that analyses the dependency graph of web objects, which were recorded with the Chrome browser console, and replays fetching web resources. Every test with each website is repeated three times.

Figure 5.7 shows the relative Page Load Time (PLT) results for each IW value, which is the difference of the Page Load Time between the tests with tuned IW value and the tests with the default IW value (10 MSS) for each website. For about 70% of websites, the increase of IW yields better Page Load Time. Looking at the top of the figure, we could see that a few of sites suffered from a higher value of IW, notably when IW is 40. The reason is, since the complex pages comprise hundreds or thousands of web objects, large IW may cause the link to be saturated and congested, therefore the PLT is increased. With high network capacity in the experiment, we did not observe much congestion; however, the results could change if the network resource is more limited. Therefore, these results do not suggest that increasing IW always produces better performance, but show how flexible the Linux TCP stack can be. One drawback of controlling the IW is that the optimal value is dependent on the round-trip-time which is usually not yet available to the client before the handshaking. Our solution could be easily be modified to let clients requesting their estimated available bandwidth, instead of requesting the IW, to the servers. In fact, as early as 2007, the RFC4782 [79] has proposed an IP option for signaling the optimal initial sending rate. However, the IP options have more chance to be filtered than the TCP options.

5.5.4 *Tuning the acknowledgement strategy*

As a reliable protocol, TCP crucially relies on the ACK packets to detect losses and control the data transfer. Sending ACKs too frequently may impose too much overhead in wireless networks or on fat pipes. On heavily loaded servers, the ACK processing may consume as much as 20% of the CPU cycles [39]. On the other hand, sending too few ACKs could probably harm the performance of traditional congestion controls like Reno/Cubic: slow down the increase of congestion window in the slow-start phase, trigger bursty transmissions, overestimate RTT and RTO, or prevent Fast/Early Retransmit recoveries from real losses.

For these reasons, the IETF in RFC2525 (section 2.13) [155] recommended a trade-off: do not delay ACK for more than 500 ms and immediately send ACK for every second packet. Linux follows this recommendation and has hard-coded the minimum and maximum values of the delayed ACK timeout at 40 ms and 200 ms.

However, such fixed values cannot adapt to connections which have very different delay, bandwidth and loss characteristics. They may be too large for

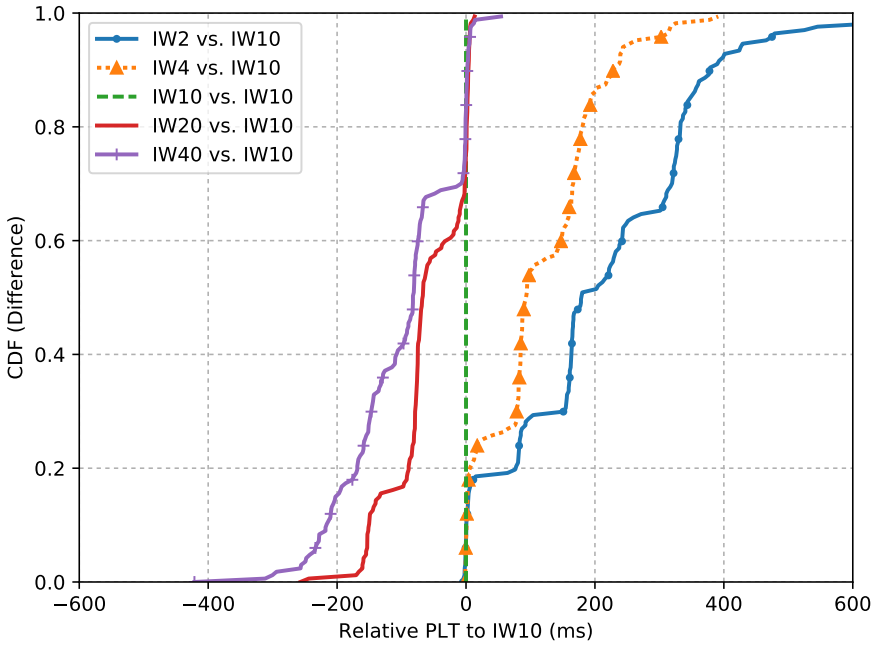


Figure 5.7: Initial Window Option test: Page Load Time relatively to IW=10 (40 Mbps bandwidth, 40 ms link delay)

local connections, but too small for inter-continental connections. The only customization supported by Linux is to disable the delayed ACK mechanism for each route [204]. However, there is no way for a sender to know the acknowledgement strategy used by its peer.

In low-latency environments, the delayed acknowledgement timer causes too many spurious retransmission timeouts, harming the performance. The measured RTTs are inflated by the delayed ACK timeout. The RTO calculation is based on sRTT, so RTO may also be over-estimated by delayed ACKs. There are two separate reasons for this: (1) the default delayed ACK timeout is set too high, and (2) the sender has no information about the delayed ACK behavior on the receiver. For example, in datacenters, the typical RTT is in the order of a few milliseconds, so the estimated RTO is likely dominated by the delayed ACK timeout which is 40 ms at minimum in Linux. While Linux tried to guess delayed ACK to exclude from RTT sampling, there is no reliable way to do this.

Meanwhile, modern networking stacks have adapted to the stretch ACK technique. First, popular networking stacks support pacing, which helps to avoid the bursty transmission issue, a side-effect of the interaction between the stretched ACKing and the classical congestion controls. Second, the congestion control implementations were adapted to increase the congestion window

properly with stretch ACKs [34, 43]. Furthermore, the Recent ACK (RACK) [45] (subsumed Tail-Loss Probe (TLP) [65]) mechanism which is being standardized and deployed in Linux and Windows [14]. This allows TCP senders to quickly detect losses based on a per-packet timer instead of using duplicated ACKs, reducing the impact of stretch ACK.

Google proposed a TCP Option [208] to negotiate a custom delayed ACK timeout during the three-way handshake. However, as discussed during the IETF99 TCPM WG meeting[111], there are several issues with this proposal: (1) it is an absolute value, which must be defined before the establishment of the connection, so it cannot adapt to different environments. Even a well-thought heuristic cannot match all network conditions. (2) A malicious middlebox on the path could inject weird values to drive the hosts into abnormal states. (3) The negotiation uses the SYN and SYN-ACK packets, which may have not enough TCP option space.

We define a similar TCP Option, but with different semantics. Our option contains two fields: (*i*) the delayed ACK value as a fraction of the minimum RTT and (*ii*) the amount of unacknowledged data (in units of MSS) that should trigger an immediate ACK. To allow the sender to properly adjust its congestion window during the slow-start, out-of-order receive or retransmission phases, we still keep the original Linux acknowledgement strategy during these phases.

eBPF helps us to change the strategy or parameters dynamically based on the current situation, for example, a client on a crowded wireless network or a server that is sending heavily.

5.6 DISCUSSION

TCP was designed to be extensible by using TCP options. However, the last decades have shown that it remains very difficult to extend TCP by defining such a new option. While the IETF has reserved a set of option types for experimental options [189] to avoid the middlebox interference, TCP implementations such as the Linux TCP stack are monolithic and difficult to extend. In this chapter, we have leveraged the eBPF virtual machine in the Linux kernel to demonstrate that it becomes possible to incrementally extend the Linux TCP stack. Our work has shown that, with little changes to the kernel code, it is possible to leverage eBPF programs to quickly implement a range of new TCP features. The main drawback of this method remains the limitation of the TCP option space, which cannot be larger than 40 bytes. On the other hand, it should be considered as a first step to make the Linux TCP stack truly extensible. The results described in this chapter open different directions for future work.

A first direction is to actually use eBPF to extend TCP in real deployments. On the public Internet, adding new TCP options remains difficult given the prevalence of middleboxes [107]. However, TCP is also widely used inside enterprise networks, datacenters and in controlled environments where there

is no middlebox interference. It is also used between proxies such as Hybrid Access Networks [23] or between edge servers and core servers of CDNs. Furthermore, there is anecdotal evidence that large content providers use a tuned version of the Linux TCP stack that has diverged from the mainline Linux kernel over the years. This implies that either they frequently need to backport new features of the Linux kernel or do not use these improvements in their stack. Using eBPF would enable them to both completely tune their Linux TCP stack and still benefit from the community improvements.

A second and more interesting direction in the long term would be to make the Linux TCP stack completely modular. It currently contains a wide range of hard-coded heuristics and optimisations such as congestion control, retransmission techniques, loss detection heuristics, automatic buffer tuning. All these heuristics could be implemented as eBPF programs to enable applications to replace or tune them based on their requirements.

Finally, this approach could be applied to extend other protocols that support an optional field e.g. IP option fields or UDP option fields. However, they are also susceptible to the middlebox interference. Therefore, the implementers and the users need to take the middlebox issue into consideration.

6 EXTENDING LINUX MPTCP WITH USER-DEFINED OPTIONS

6.1 INTRODUCTION

In the previous chapter, we have presented several use cases of extending the regular TCP stack with user-defined TCP options. For Multipath TCP, the ability to use multiple paths enables even more use cases to extend the protocol/stack than regular TCP. Moreover, Multipath TCP has richer semantics and a much larger decision space than regular TCP. For example, path management and packet scheduling are two new tasks that did not exist with legacy TCP.

The MPTCP Linux kernel implementation [149], which is based on the regular Linux TCP stack, is considered the reference implementation for the Multipath TCP protocol and the most complete implementation of RFC6824. It is being used in various commercial deployments, typically the hybrid access solutions [119] and WiFi/Cellular aggregation on high-end smartphones [23]. Recently, Apple uses it in the server side to run their services including Apple Siri, Apple Maps and Apple Music. Usually, for each deployment, the developers have to modify the MPTCP Linux kernel implementation to suit their use cases. Implementing, testing, tweaking, forward-porting the out-of-tree kernel patches are tedious and error-prone. It is desirable to minimize the code changes to the kernel and to dynamically customize the MPTCP stack without constantly rebuilding the kernel.

Similar to the TCP option framework in the previous chapter, we implement a generic eBPF framework to support user-defined MPTCP options. The methodology and implementation details are presented in Section 6.3. Then we illustrate the usage of user-defined MPTCP options with four different use cases in Section 6.4. In the next chapter, we apply the same approach to support user-defined path managers. But first of all, we present an overview of the Multipath TCP implementation in the Linux kernel.

6.2 MULTIPATH TCP IMPLEMENTATION IN THE LINUX KERNEL

In this section, we present how the specifications of Multipath TCP (which were briefly presented in Section 2.4) is implemented in the Linux kernel. More details of this implementation can be found in Chapters 5 and 6 of Christoph Paasch's thesis [145]. However, many details of the implementation have been changed since then, notably on the server side.

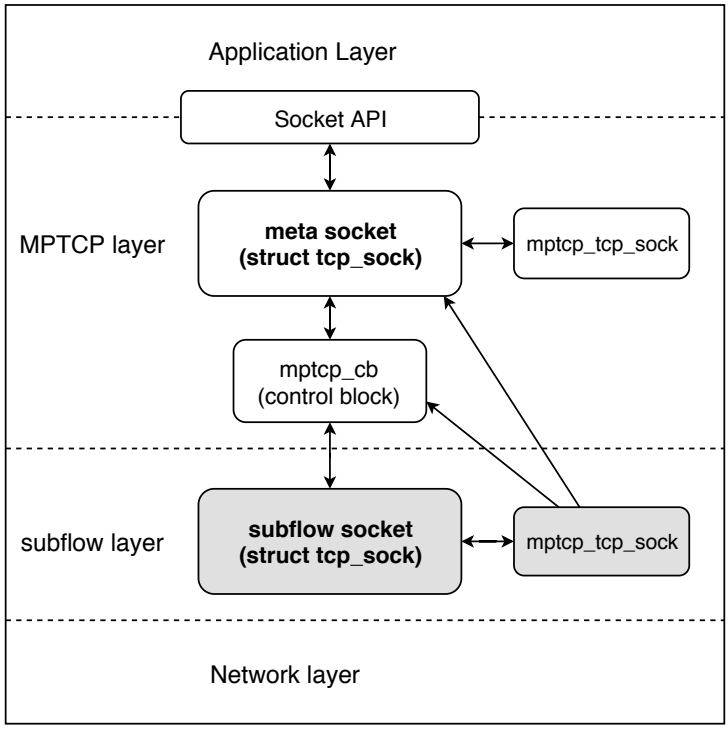


Figure 6.1: Main data structures of the Multipath TCP implementation in the Linux kernel

6.2.1 *Data structures*

Following the architectural design of the Multipath TCP protocol, its implementation in Linux is also logically divided into two sublayers (Figure 6.1). At the subflow layer, each subflow is handled by a subflow socket which interacts directly with the IP layer. These subflow sockets are transparent to the applications and only controlled by the kernel. In the MPTCP layer, subflows of an Multipath TCP connection are aggregated in a *meta socket* that represents the state-machine of the MPTCP connection. The meta socket serves the application via the regular socket API. The implementers reused the socket structure `tcp_sock` of legacy TCP to represent both subflow sockets and meta sockets. Additionally, for both types of sockets, each `tcp_sock` instance is associated with a dedicated structure `mptcp_tcp_sock` which contains MPTCP-specific information of the socket. The reason for this design is to limit the amount of code changes to the vanilla TCP stack, while avoiding the increased memory footprint of generic structure `tcp_sock` used by regular TCP connections.

6.2.2 *Connection setup*

From the implementation viewpoint, the connection setup is the process of creating the data structures and states which represent an MPTCP connection and its subflows. A straightforward implementation may initiate all MPTCP-specific structures at the beginning of the connection. However, since regular TCP is still the de-factor standard and Multipath TCP is the exception but not the rule, the above design would harm the performance of all TCP traffic. Therefore, the implementers decided that the host initializes most MPTCP-specific structures only *after* determining that its peer also supports Multipath TCP. To be concrete, this happens once the client receives a SYN-ACK packet with an `MP_CAPABLE` option or when the server receives a third ACK packet with an `MP_CAPABLE` option. In both cases, the kernel calls the function `mptcp_create_master_sk()` which creates the master subflow socket (`tcp_sock`), the multipath control block (`mptcp_cb`), `mptcp_tcp_sock` and links them with the meta socket. The function call chain is shown in Figure 6.2.

All meta sockets (representing MPTCP connections) on the host are tracked by a hashtable which uses the connection tokens as the keys for looking up. The hashtable structure also allows host to quickly verify the uniqueness of newly created tokens.

6.2.3 *Subflow setup*

After the initial subflow established and some data has been exchanged, hosts are allowed to create additional subflows. As mentioned in Section 2.5.2, additional subflows are initiated from the client side only. The path manager makes this decision based on the user objectives and various parameters, typically the type and status of local network interfaces. Unlike the master

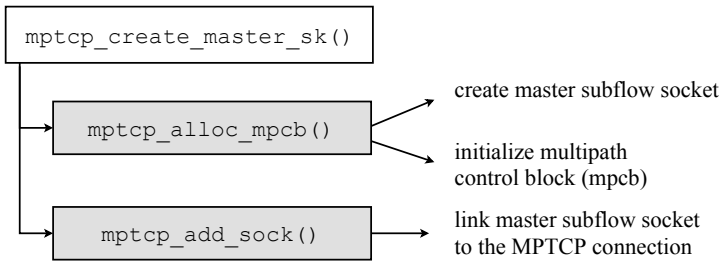


Figure 6.2: On both the client and server, MPTCP control block and master subflow socket is created only after 3-way handshake finished

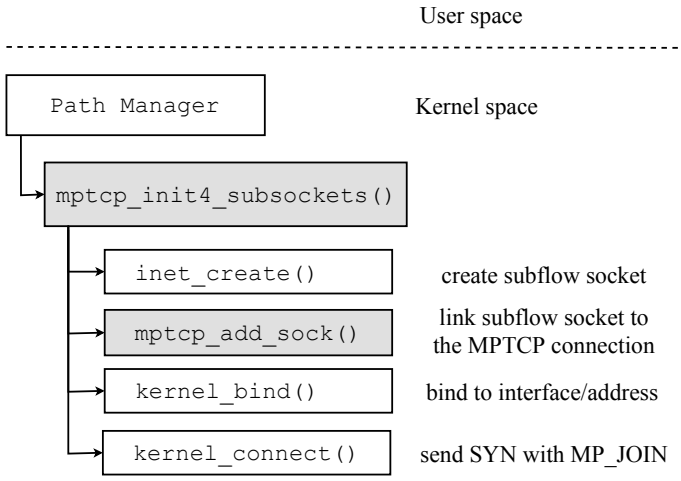


Figure 6.3: On the client side, a new subflow socket is created and added to the MPTCP connection before the 4-way JOIN handshake

sockets, the subflow sockets are added to the MPTCP connection by the `mptcp_add_sock()` function *before* the handshake, as shown in Figure 6.3. The reason is that, additional subflows do not fall back to regular TCP if the negotiation fails. For example, if the SYN-ACK responding from the server does not contain an `MP_JOIN` option or has the `MP_JOIN` option but with the wrong HMAC then the client would send a TCP RST and immediately close the subflow.

On the server side, the receiver needs to check if the incoming SYN packet corresponds to any existing MPTCP connection. If it is true, the server creates the light-weight *request socket* as usual and links it to the MPTCP connection. Later, when the third ACK packet arrives, the function `mptcp_check_req_child()` checks the `MP_JOIN` option and verifies the HMAC. If they are valid then a full subflow socket is created to replace the *request socket*

and is added to the MPTCP connection by the function `mptcp_add_sock()` (Figure 6.4).

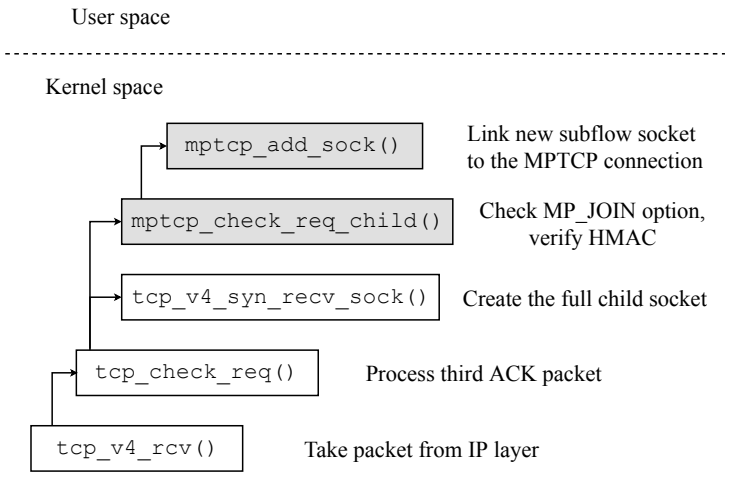


Figure 6.4: On the server side, a full subflow socket is created and added to the MPTCP connection after the third JOIN ACK packet arrives

6.2.4 *Data transfer*

Data sending process starts with the application passing some amount of data to the kernel stack by one socket API system call (e.g. `send()`, `sendmsg()`, `sendmmsg()`). For the regular Linux TCP stack, this data is splitted into separate segments stored in the `skb` structures. This `skb` also contains a control block structure (`tcp_skb_cb`) which stores the sequence numbers, acknowledging status of this segment and other information. For Multipath TCP, there are two sequence number spaces handled by two separate layers. First, the continuous stream of data is passed from the application to the meta socket layer. Then, it is also splitted into small segments which are managed by the `skb` structures and are queued in the meta send-queue. The `tcp_skb_cb` control block now stores the data sequence number. These segments are pushed to the different subflows in the `mptcp_write_xmit()` function. At this step, the packet scheduler takes charge of selecting the best subflow for data dispatching. Then, the `mptcp_skb_entail()` function pushes each segment into a subflow send-queue and switches from the DSN to the subflow sequence numbers. Later, packets are actually sent by the `__tcp_push_pending_frames()` function.

On the receiving path, the packet processing is cleanly separated between the subflow layer and the Multipath TCP layer. The receiver handles incoming packets on individual subflows just like regular TCP connections. Out-of-order packets are stored in the subflow out-of-order queue. When an incoming segment arrives in order, it (and out-of-order packets whose sequence numbers

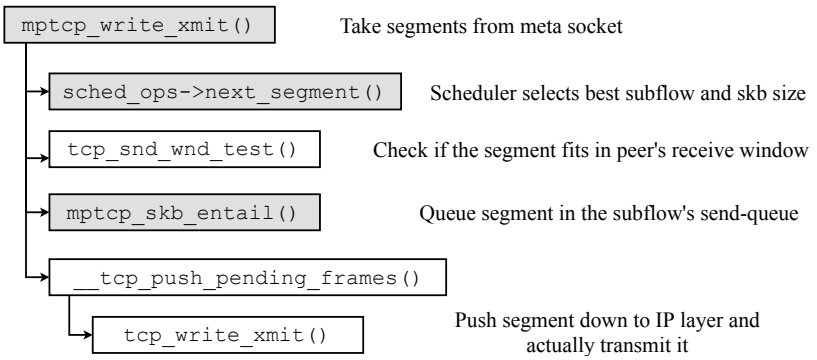


Figure 6.5: MPTCP sending path

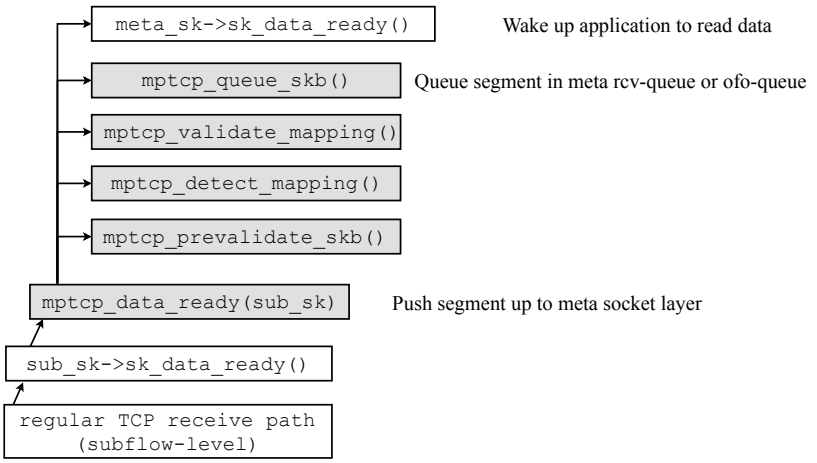


Figure 6.6: MPTCP receiving path

next to it) is passed to the function `mptcp_data_ready()` to carry various checks. If the sender's NIC has splitted outgoing segments or the receiver's NIC has merged incoming segments, the DSN mapping will be broken. The MPTCP stack must undo this effect by rebuilding the mapping. Then, the function `mptcp_queue_skb()` passes these segments to the meta layer, either in the receive queue or the out-of-order queue, depending on whether their DSNs are in order or not. Finally, when valid and in-order data is ready for the application to read, the `sock_def_readable()` function wakes up the application just like regular TCP.

6.2.5 *Connection teardown*

A subflow is shut down in a fashion similar to a regular TCP connection, involving 4-way FIN negotiation. The closure of the regular TCP connection

includes executing `tcp_close()` in the `process context`, since it is initiated by the application. However, the closure of a subflow is triggered by the kernel which is usually in the `interrupt context`. Therefore, the kernel has to schedule the task into a `workqueue` so that `tcp_close()` of the subflow could be run in the `process context`. On the other hand, the multipath connection is closed by the DATA-FIN option exchange as discussed in Section 2.4.6. When the application issues a `close()` syscall, the `mptcp_close()` function is executed to close the meta socket. However, the meta socket has to remain available until all subflows are closed. After the subflow socket transitions to the `TCP_CLOSED` state, the subflow's data structures are destroyed in the `mptcp_sock_destruct()` function. Similarly, when the meta socket transitions to the `TCP_CLOSED` state and all subflows are closed, all data structures belonging to the connection are destroyed in the same function above.

6.3 METHODOLOGY

To support user-defined MPTCP options, we need to extend TCP-BPF to be aware of Multipath TCP. We add to TCP-BPF the ability of tracking MPTCP connections, subflows, and passing MPTCP-specific information. The MPTCP-connection-level event tracking is not discussed here because our four use cases do not use it. Instead, it will be elaborated in Section 7.2.1 since event tracking is important for the path management operation. Section 6.3.1 describes how we track the MPTCP subflows and Section 6.3.2 discusses how to access MPTCP-specific information from BPF programs. The implementation details of the MPTCP options handling are presented in Section 6.3.3 and 6.3.4.

6.3.1 *Tracking MPTCP subflows*

To track the subflows in Linux, one might think of reusing the TCP-BPF callbacks which are already available in the recent Linux kernel. However, these callbacks do not pass the subflow ID information to the BPF programs, so it is hard for the BPF programs to make any MPTCP-wise decision. For this reason, we need to add new callbacks with MPTCP-specific information at the right places. Since these callbacks are located in the MPTCP-specific code path, they are not called against the regular TCP traffic and, therefore, have zero overhead in this case. Currently, we focus on two events: a newly created subflow socket is linked with the MPTCP connection, and the subflow is established. In the future, we may also want to track when a subflow is switched to backup mode or closed by the kernel.

On both the server and the client sides, when a new subflow socket is created, the Linux stack initializes its meta-level information and links the subflow socket with the MPTCP connection in the function `mptcp_add_sock()`. On the server side, this happens when the third ACK arrived as mentioned in Section 6.2, marking that the subflow has been established. By adding a hook

in this `mptcp_add_sock()` function we can start tracking the subflow with the `sock` structure, along with accompanying MPTCP-level metadata.

On the client side, the master subflow is established when an `MP_CAPABLE SYN-ACK` arrives, while a joined subflow is established when an `MP_JOIN SYN-ACK` arrives. In both cases, the same `mptcp_rcv_synsent_state_process()` function is called to finish the subflow establishment. Therefore, we only need to add one hook (`MPTCP_SYNACK_RCV`) to track the establishment of both master subflow and joined subflows. The subflow ID is passed as an argument to the BPF program, so that the BPF program knows it is the master subflow or a joined subflow.

Callbacks	**Events**	**Passed arguments**
`MPTCP_ADD_SOCK`	A subflow socket is added, and the subflow is estab. (server side)	subflow ID
`MPTCP_SYNACK_RCV`	A subflow is estab. (client side)	subflow ID, dev type

Table 6.1: New TCP-BPF callbacks for tracking MPTCP subflows

6.3.2 *Accessing MPTCP metadata*

One issue is that the TCP-BPF callbacks only support at most three arguments accompanying each call. This limits the amount of MPTCP metadata that could be passed through these calls. For this reason, we extended the object context of the TCP-BPF programs (the `bpf_sock_ops` structure) to keep track of common metadata per MPTCP session as shown in Listing 6.1. This brings more MPTCP metadata to BPF programs and also simplifies new MPTCP callbacks. Moreover, this approach provides better performance since the extended fields of the `bpf_sock_ops` structure are mirrored directly from that of `tcp_sock` structure without any data copy operation.

```
struct bpf_sock_ops {
    ...
    /* fields below are mapped directly from tcp_sock */
+   __u32 mptcp_flags;
+   __u32 mptcp_loc_token;
+   __u32 mptcp_rem_token;
+   __u64 mptcp_loc_key;
+   __u64 mptcp_rem_key;
};
```

Listing 6.1: Context object is extended with MPTCP metadata, allowing TCP-BPF programs to directly access

6.3.3 *At the Sender: Inserting New MPTCP Options*

To actually insert new MPTCP options, we may reuse our facilities for TCP (in Section 5.4.2). However, for a cleaner design, we add an MPTCP-specific hook in the function `mptcp_options_write()` instead of `tcp_options_write()`, as in Figure 6.7. This also simplifies the BPF programs since they only need to handle the sub-option part, not the whole MPTCP option. Since these hooks are on the fast path, the overheads of calling these hooks are non-negligible even when the BPF program does nothing. Therefore, we disable these hooks by default and only enable them when needed. We store the `option_write` flag in the per-connection structure `bpf_sock_ops_cb_flags`. This flag array is used by TCP-BPF to control its expensive callbacks. If we pass the TCP subflow sock as the main BPF context, the BPF program will control the option-writing action per subflow basis. Furthermore, to make sure these conditional checks in the fast path do not cause the CPU branch mispredictions, the C-macro `likely` and `unlikely` are used to instruct the compiler to generate optimized assembly code.

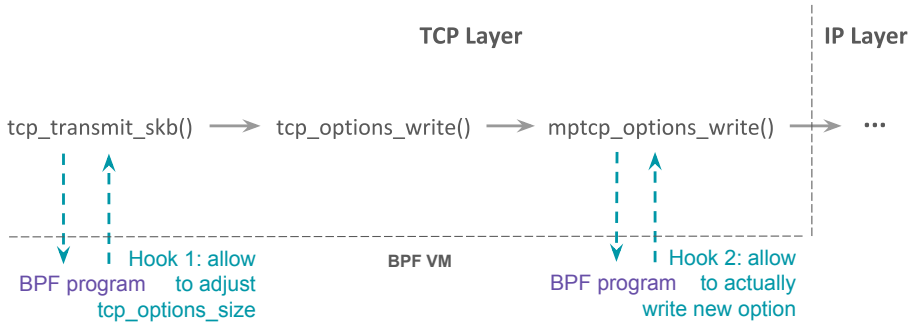


Figure 6.7: New hooks to insert user-defined MPTCP options

6.3.4 *At the Receiver: Parsing New Options*

To allow BPF programs to parse new MPTCP options, we cannot reuse our TCP parsing hook (which is mentioned in 5.4.2) since it only processes regular TCP options. Instead, the Linux reference implementation handles all MPTCP options in the function `mptcp_parse_options()`. In this function, we add a new TCP-BPF callback when encountering an unknown MPTCP option. The option subtype, the length and the option data are passed to the BPF program via three arguments of the callback function (Figure 6.8). Similar to the option-insertion callback above, this option-parsing callback is disabled by default and is activated on demand by the BPF program.

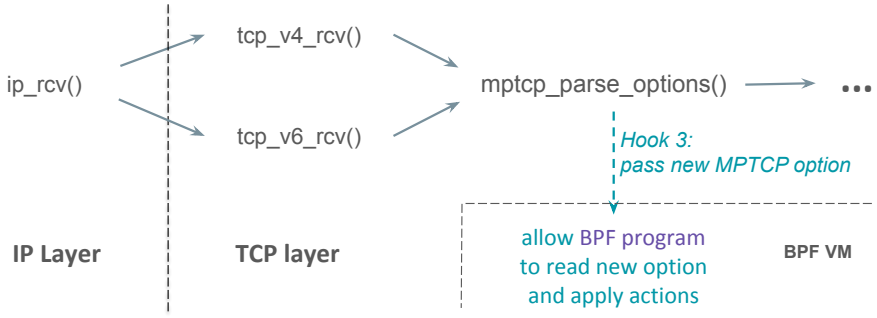


Figure 6.8: New hook to parse user-defined MPTCP options

6.4 USE CASES

In this part, we leverage the MPTCP option framework to implement four use cases and collect the experiment results. The first option allows a client to request the server to limit the sending rate on a subflow (Section 6.4.1). The second option enables a host to request its peer to select a packet scheduler for the connection (Section 6.4.2). The third one is a delay-threshold option for MPTCP hosts to specify that the peer should use the backup subflow when the delay is above a threshold (Section 6.4.3). Finally, a host may request its peer to set a desired timeout to remove the MPTCP connection state after the last subflow closed (Section 6.4.4).

6.4.1 *Subflow Rate-Limit Option*

Motivation: Most of the mobile clients do not have an unlimited cellular data subscription. Even if this is the case, mobile network operators may still silently throttle the bandwidth of those customers who have used up a large amount of cellular data. A good LTE or 5G connection running at full speed in a few hours could consume the entire monthly budget cellular quota of many users. A common scenario is that the mobile users want to limit the monetary cost of using cellular networks or to avoid running out of the mobile data quota. Clients could limit its upstream traffic, however, most of the traffic are downstreamed. For these applications, rate-limit should be controlled on the server side. For this, the clients would desire the servers to limit the maximum throughput on the cellular network subflow. This is more important when the mobile clients are roaming abroad where the monetary cost for cellular data is usually very high. The rate-control mechanism could be exchanged and enforced at the application layer, however, applying the mechanism at the transport layer would be more generic and could be done automatically by the system itself.

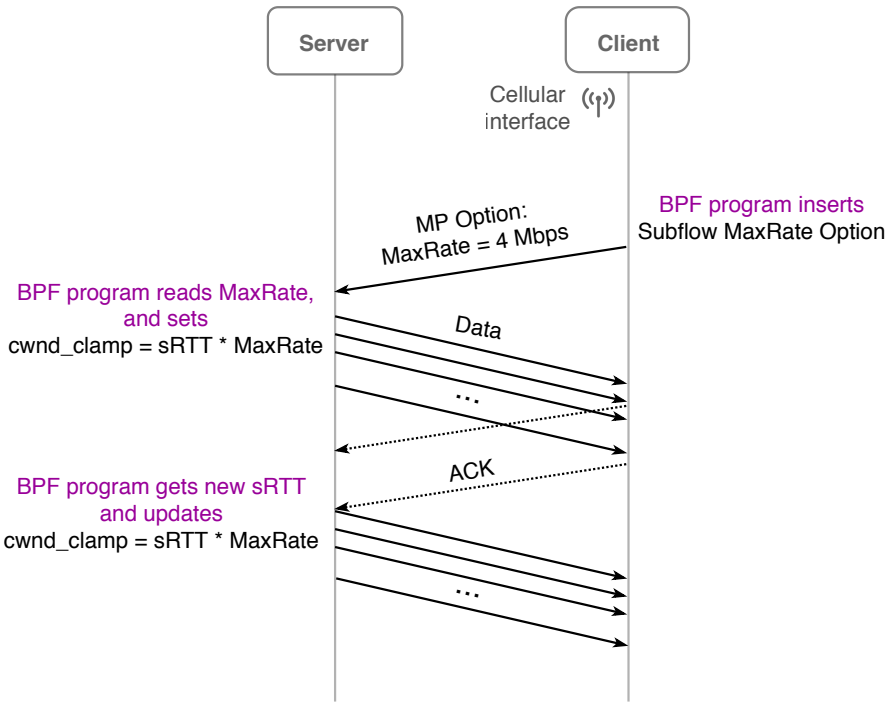


Figure 6.9: Usage of an MPTCP option to signal the maximum rate of the cellular subflow

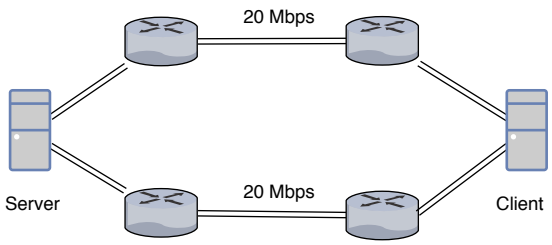


Figure 6.10: Emulated topology for the MPTCP options experiments

As discussed on the `multipathtcp` IETF mailing list [148], this rate-control mechanism can also be used when a client wants to tell a sender to close a subflow gracefully by requesting a zero transfer rate. Though the client may send a TCP-RST on this subflow instead, in-flight data would be lost and must be reinjected over other subflows. Another solution is to send an `MP_PRIO` option to the sender to put the cellular subflow into backup mode, but this request could be overridden by the sender's local policy.

After using a certain amount of cellular data, for example 80% of the monthly quota, the client could request to reduce the usage of the cellular paths. The detailed solution is described as below.

At first, on the client side, the BPF program needs to select which subflow to send the MPTCP option to cap the maximum throughput. For that, the kernel stack needs to pass the type of interface (e.g. WiFi or Cellular) to the BPF program. We can retrieve the device type (`net_device.type`) from the `sock` structure when a new subflow socket is linked to an MPTCP session in the kernel function `mptcp_add_sock()`. However, this will not work, since at this point the subflow join SYN has not even been sent (as shown in Figure 6.3), and the device type is not yet determined. Our solution is to get the device type when the SYN-ACK packet arrives (extracted from the `sk_buff` packet structure). We pass this information via the hook `MPTCP_SYNACK_RCV` as mentioned in subsection 6.3.1.

If the BPF program sees that this subflow is on the cellular interface, it activates the `option_write` flag on this subflow. Afterwards, for each outgoing packet of the marked subflow, the BPF program inserts an MPTCP option. This new MPTCP option signals the server to cap the transfer rate to a desired value on this subflow.

When the server receives this TCP packet, the BPF program parses the MPTCP option. Combined with the current smoothed RTT and MSS values collected from the socket structure `tcp_sock`, it calculates the maximum congestion window (`cwnd_clamp`) needed to apply on this subflow.

As a side note, an alternative technique to apply the rate-limiting policy on a subflow is to rely on the TCP pacing feature. Linux networking stack supports two TCP pacing mechanisms. The first one, qdisc-based TCP pacing, works very much the same way as ours, but it relies on the fair-queuing (FQ)

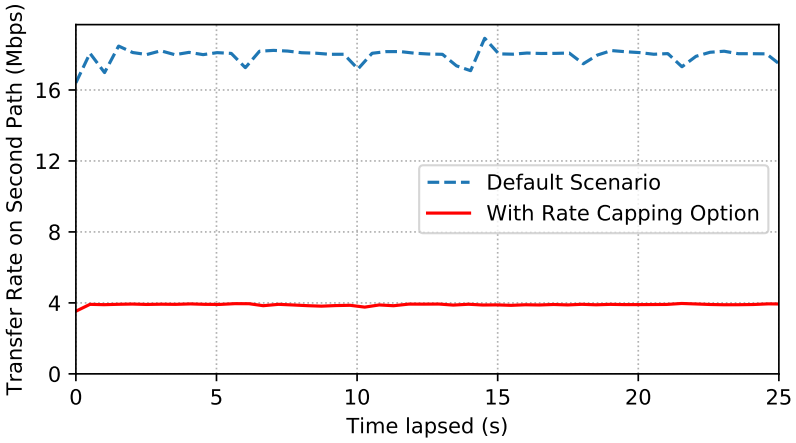


Figure 6.11: Throughput on the second path before and after capping at 4 Mbps

scheduler in the `tc qdisc` layer. The second one is the internal TCP pacing which has lower precision than the qdisc-based one, and only works with BPF out-of-the-box since Linux 5.1 [44]. On the other hand, on newer kernel versions, our server-side BPF program could be easily adapted to use the TCP pacing directly for rate limiting.

To illustrate the usage of this option, we set up a simple emulated experiment using built-in facilities in Linux (`netem`, `tc`, network namespaces, etc.) similar to the ones in Section 5.5. The topology consists of two hosts that communicate through two paths as shown in Figure 6.10. The bandwidth capacity of each path is 20 Mbps and the one-way delay is 20 ms. Figure 6.11 shows the transfer rate on the second path before and after using this capping option. We observe that our throughput capping mechanism based on adjusting `cwnd_clamp` works well enough for this use case.

6.4.2 *Scheduler-Request Option*

Motivation: It is well known that the selection of the packet scheduler has a significant impact on the performance of the MPTCP protocol [75, 84]. But normally receivers cannot control it. Most of the web traffic is in the downstream direction. It is desirable that the clients could request the scheduler algorithm on the server side. E.g. if the client wants to reduce the bufferbloat issue, an optimized scheduler like BLEST [75] could be used. Or if the client wants to reduce the latency as much as possible, even at the cost of more redundant traffic, it could request the redundant scheduler [86].

At the moment, the Linux MPTCP stack supports multiple packet schedulers, but it only allows the users to select the scheduler before the connection

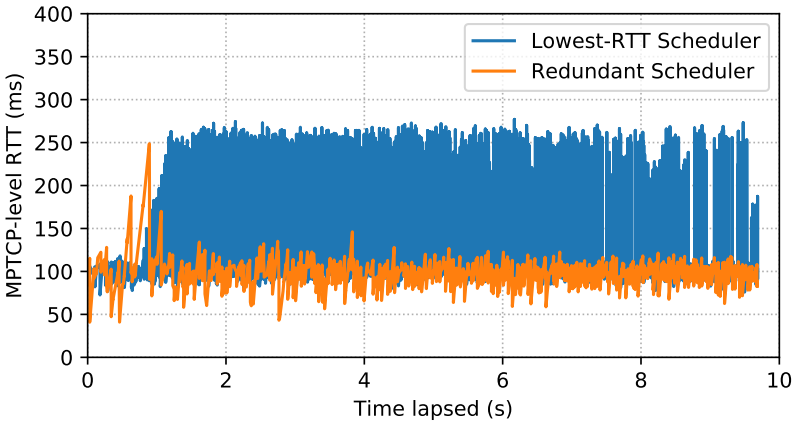


Figure 6.12: MPTCP-level Round-trip Time: the redundant scheduler achieves lower latency than the default one

is established. To implement this feature, we have tweaked the MPTCP stack so that the sender could replace its scheduler on the fly, even after the connection has been established. The exact signaling mechanism is similar to the way we requested the TCP congestion control algorithm in Section 5.5.2. Due to the limited TCP option space, the MPTCP option does not carry the name string of the scheduler. Instead, we assume that the server pre-shares to the client the list of its available schedulers. Then, the client specifies in the MPTCP option the ID number of the desired scheduler.

To verify the idea, we use a test scenario similar to the subflow-rate-limit option use case above. Two hosts are connected through two symmetric paths as in Figure 6.10. Each path also has the capacity of 20 Mbps but the delay varies in the range of 20 ms to 150 ms. A bulk transfer is conducted in the direction from the server to the client. We experimented with two scenarios: (1) the server always uses the default scheduler, and (2) the server changes the scheduler to the redundant one as per requested by the client. Figure 6.12 shows the MPTCP-level RTT in these two cases, since this metric represents most of the latency observed by the application. The redundant scheduler delivers better latency than the Lowest-RTT scheduler for most of the time. The exception is the spike at the beginning of the connection in the case of redundant scheduler. This RTT increment is due to the host sending bursts in the slow start phase and filling the buffers on both paths. The RTT increases quicker with redundant scheduler because it sends more traffic per path than the default scheduler.

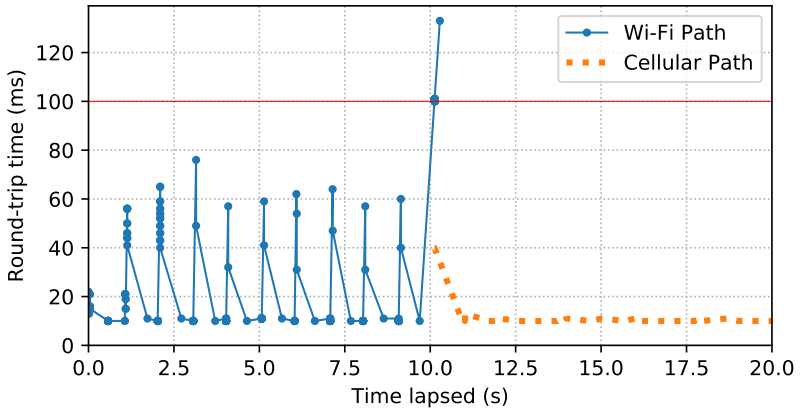


Figure 6.13: The server switches to the Cellular path after the experiencing high latency on the WiFi path

6.4.3 *Delay-Threshold Option for Thin Streams*

Motivation: For many mobile applications, e.g. voice recognition applications, it is important to maintain a low latency for the network connection. Recent cellular technologies e.g. 4G and 5G provide a lower latency than the Wi-Fi in many cases. However, mobile users do not want to consume cellular traffic as long as the delay on the Wi-Fi path is still good enough. The basic idea is that an additional subflow is created on the cellular path but it should not be used unless neccessary. Several flavours of this use case have been discussed on the IETF mailing list by Paasch et al. [147]. While the rate-limiting option which we mentioned above 6.4.1 focused on the heavy streams, this use case is mostly about the thin streams. The detailed mechanism is explained as below.

At the beginning of the MPTCP session, the client creates an additional subflow on the cellular path and sets it to back-up mode. The client signals the server to put the additional subflow in inactive state by setting the backup flag in the MP_JOIN option in the SYN packet. We allow the BPF program to do this by using the `bpf_setsockopt()` helper function. As per protocol definition, the receiver seeing this flag sets the joined subflows into the backup mode.

Then, on the master subflow, the client sends an MPTCP option which includes an RTT value in milliseconds. This is the maximum delay threshold that the server should keep the RTT below. Only when the RTT on the master subflow surpasses this theshold, the server would start sending data over the second subflow. Since the server needs to keep track of the delay threshold per MPTCP connection, we store this value in the `mptcp_cb` structure, which was designed to store metadata of each MPTCP connection.

For illustration, we set up an experiment that emulates a mobile client connecting to the server via two paths: Wi-Fi and cellular. The topology remains the same as in Figure 6.10 but the one-way delay is 5 ms. At the beginning, the client sets up two subflows on both paths, but only uses the Wi-Fi path, as shown in Fig. 6.13. Then when the network condition on the Wi-Fi becomes worse (RTT surpasses the 100 ms threshold), the server switches to the cellular path to satisfy the low latency requirement.

6.4.4 *MPTCP Inactivity Timeout Option*

Motivation: Hosts need to maintain the state of Multipath TCP connections for some time after all established subflows has been closed, as mentioned in RFC 6824 [80]:

> "If all subflows have been closed with a FIN exchange, but no `DATA_FIN` has been received and acknowledged, the MPTCP connection is treated as closed only after a timeout. [...] This permits "break-before-make" scenarios where connectivity is lost on all subflows before a new one can be re-established."

However, the document does not specify how long an implementation should maintain this state. Therefore, hosts may have their own timing-out policy for inactive Multipath TCP sessions. In practice, the current Linux kernel implementation keeps these inactivity MPTCP sessions forever. It leaves to the applications the responsibility to check and terminate these sessions. However, it is difficult for the system administrators to control the lifetime of these sessions. On the other hand, it does support configuring the keepalive timer at the meta level. Once it is enabled, the host sends keepalive packets regularly when the session is idle to keep at least one subflow alive. Nonetheless, this mechanism does not control how an MPTCP session without subflow should be kept locally. For this reason, we implement the MPTCP session inactivity timeout (ITO) support in the Linux kernel. The ITO timer is scheduled when the last subflow is removed from the MPTCP session (in the function `mptcp_del_sock()`) and is cleared when a new subflow is added (in the function `mptcp_add_sock()`). Since adding a new kernel timer would impose a lot of overhead, we instead reuse the existing keepalive timer facility to handle the ITO. Users or BPF programs could control the ITO via a new socket option (`SOCK_KILL_ON_IDLE`). After the idle timeout fired, the stack would close the MPTCP session and report the timeout error to the user.

In several cases, it is necessary to communicate the inactivity timeout value. A host that wants to extend the lifetime of a connection through the transient failures may request its peer to apply a high session timeout value. On the other hand, by reducing this value, a highly-loaded server can quickly terminate currently unused MPTCP connections. It can send this reduced value to signal its peers that the connections will be closed shortly. For regular TCP, extending the connection lifetime by increasing the inactivity timeout on

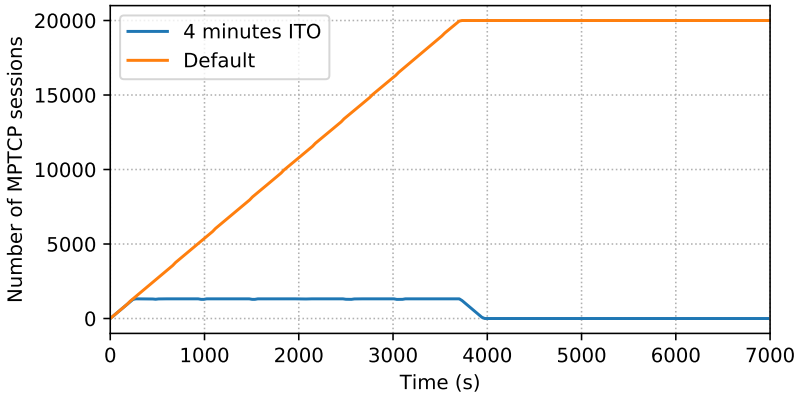
both ends is usually not enough since it is common that the NAT/middleboxes on the path could explicitly or silently terminate the connection. For Multipath TCP, this is a much lesser problem since the closure of all subflows does not terminate the multipath-level session. With ITO support, the TCP keepalive mechanism may be not necessary for Multipath TCP.

For evaluation, we set up a local testbed in which the client and the server are two dedicate machines connected via intermediate routers. The server machine is equipped with 8 GB of memory and a 4-core Intel Xeon X3440 2.53 GHz CPU. In our experiment, `curl` tool on the client initiates an MPTCP session to download a file which is served by `nginx` server. A BPF program on the client sets an ITO value of 4 minutes on the session locally and inserts an MPTCP option with this ITO value into an outgoing TCP segment. Another BPF program on the server parses this MPTCP option and applies this ITO value on its side. Then, we use the `tcpkill` program to terminate the subflow on both sides with the `TCP_RST` packets but the MPTCP session still persists. The client creates 20000 instances of such sessions in one hour. Figure 6.14 shows the server resource usage comparison between this case and the default case where no inactivity timeout is set. The number of established MPTCP connections is collected by an extended version [46] of `netstat` utility which understands MPTCP. The memory usage is reported by the `free` utility in the `procps-ng` suite and the system file `/proc/meminfo`. Kernel memory usage is inferred by substracting the total memory usage from the user one (`AnonPages` field in `/proc/meminfo`). We can observe a clear difference between the two in the both number of existing MPTCP sessions and the memory usage after the experiments running for 4 minutes. For the ITO test case, this is the moment that the inactivity timers of the first MPTCP sessions start to expire. From this point, newly-created sessions and expired sessions are coming and leaving at the same rate, therefore, the number of sessions and the memory usage are stable. Meanwhile, in the default case, inactive sessions never expire, so that the resource usage keeps increasing on the server.

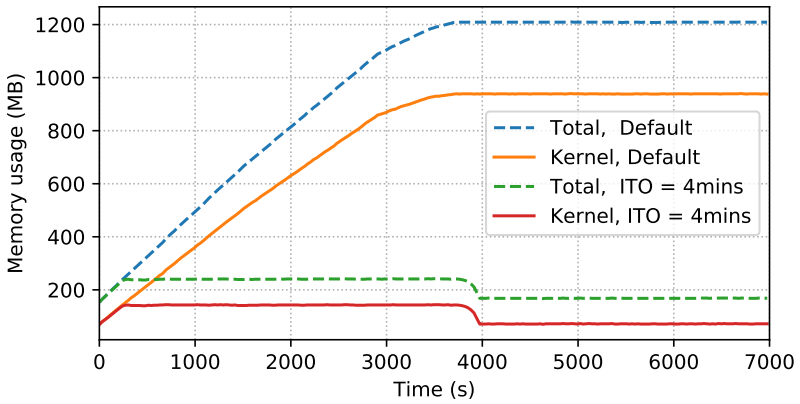
6.5 DISCUSSION

In this chapter, we have implemented an MPTCP option framework which allows extending the Linux MPTCP stack with new features to match various use cases, as shown in Section 6.4. We have written two IETF draft proposals to standardize the SRL option [196] and the ITO option [195]. Additionally, this framework could also be used to implement One-Way-Delay (OWD) option proposed at IETF 101 [219]. Supporting this OWD option is an important step [184] to optimize MPTCP performance issues such as packet scheduling, reinjection mechanism, or shared bandwidth detection.

For the use cases in Section 6.4, we implemented each MPTCP option using a different option subtype. However, the issue is that current version of Multipath TCP supports only 16 different option subtypes at maximum, and 12 of them have already been assigned in RFC 6824bis. To avoid the subtype



(a) Number of established MPTCP sessions on the server



(b) Memory usage on the server

Figure 6.14: Inactivity Timeout option helps reducing significantly the resource usage on the server

	Kernel changes	**BPF program**
MPTCP Option Framework	~100	0
Use case: Suflow Rate Capping	8	~65
Use case: Scheduler Request	~70	~95
Use case: Delay Threshold	~80	~150
Use case: Inactivity Timeout	~130	~150

Table 6.2: Lines of code (excluding comments) of MPTCP Option Framework and each Use Case

usage exhaustion and conflicts, one solution is to leverage the experimental Multipath TCP option [21] which supports up to 65536 different MPTCP option types.

One current limitation of our framework is that it now only supports MPTCP options of four bytes. This is because we pass the option data directly through the TCP-BPF callbacks. Instead, we can implement dedicated helper functions to write and parse the option and keep the temporary option data in the socket context. This approach will both remove the above restriction of the option size and allow better validity checking of new options.

Another challenge comes from the current limitations of the eBPF infrastructure in the Linux kernel. For the Linux version 4.17 that we have used, eBPF programs are limited by several technical constraints which are imposed to guarantee the performance and responsiveness of the kernel. BPF programs cannot contain more than 4096 instructions, BPF functions cannot have more than 5 arguments, loops are not allowed, global-scope data is supported, and there are no built-in queue and stack structures. These restrictions make it difficult to implement complex features, forcing the utilization of workarounds such as using multiple BPF programs that are linked together by BPF tail calls. Most of them are not architectural or design flaws but temporary caveats. For example, a map-based implementation of queues and stacks [202] was added in Linux kernel since 4.20. After several efforts [73], BPF subsystem maintainers have recently implemented constrained loops support [185], supported global data [25] and extended the program size limit to one million instructions [186]. While eBPF was designed to work over different platforms, it may be an issue to deploy the same eBPF program on different Linux kernel versions which do not support some map types or helper functions needed by the eBPF program. There are several implemented methods to make eBPF programs to be compile-once run-everywhere, e.g. allowing userspace to query the list of supported features [134], using BTF type information and layout [135], or embedding kernel header within the running kernel image [77].

7 SUPPORTING USER-DEFINED MPTCP PATH MANAGERS

7.1 MOTIVATION

Two important tasks of a Multipath TCP stack are managing the subflows (i.e. path management) and choosing which subflow(s) for sending each packet (i.e. packet scheduling). The current MPTCP implementation in the Linux kernel was designed following a modular approach. These functionalities are implemented as kernel modules, allowing different algorithms to be used. However, there are several issues with this approach. First, this task should be controlled by applications which have a wide range of requirements, or by the system administrators who know the characteristics of the network environment and want to enforce specific policies. User applications can select one among the available algorithms in the kernel, but cannot deploy a new one. Second, system administrators could implement new algorithms as kernel modules and load them into the kernel, but this is dangerous because it lacks of the protections against memory corruption, invalid memory accesses, deadlocks, race conditions and others.

Several efforts have proposed to give users more control over Linux MPTCP. Frömmgen et al. [84] proposed an eBPF-based programming model for users to write packet schedulers in high-level languages and then the custom in-kernel verifier validates and compiles them to eBPF native code running inside the kernel. Hesmans et al. [100] extend the socket option interface to allow MPTCP-aware applications to query subflow status information, to directly create and remove each subflow.

The path managers decide which subflows should be created or removed, and which addresses are announced based on the current situation and the user requirements. The enhanced socket API [100] allows MPTCP-aware applications to directly manage the subflows. However, it does not allow building a generic path manager that could be used by multiple applications and it is difficult for the system administrators to control it. A `netlink`-based framework has been introduced to support generic path managers with the control plane in user space [104]. It has been recently merged in the `mptcp-trunk` branch [13]. Using the `netlink` communication channel is a natural approach that provides a clean separation between control plane and data plane. However, it is not without issues. It introduces overhead due to context switches between user and kernel space as well as due to `netlink` channel handling. But the most important issue is that the netlink channel is unreliable. Under high load, `netlink` messages may be lost. Additionally, this approach requires separate facilities to support various but maybe necessary features, most notably getting/setting subflow socket options (e.g. access subflow-level info) and TCP

state change notification. Additionally, it is difficult to enforce the policy to accept or refuse the establishment of a subflow.

For these reasons, we have investigated and implemented an alternative approach based on eBPF. The motivation for this approach includes:

- Performance: Once a BPF program is loaded into the kernel, it could avoid switching between user space and kernel space for every operation like a netlink-based approach. We can also avoid the overhead due to sending and receiving netlink messages. The performance of eBPF is one major reason for its quick adoption in the Linux community.
- The BPF approach does not rely on message passing so it does not suffer from the message loss issue.
- TCP-BPF has built-in support for TCP state tracking.
- TCP-BPF has built-in support to read and change many values of the TCP socket.
- It is straightforward to enforce accepting/refusal policies on the subflow establishment.

This approach also supports multiple path managers running in parallel, one per `cgroup`. This may be necessary, for example, when MPTCP proxies want to use different path managers for upstream and downstream traffic. Another use case is in the container-based virtualized environments, in which each container uses a custom path manager. While the netlink-based solution may support different path managers isolated by network namespaces, the eBPF approach relies on `cgroups` which are more flexible than the network namespaces.

However, it is expected that this approach would have its own limitations. First, eBPF programs are restricted by current eBPF limits. For example, until Linux 5.2 each BPF program cannot have more than 4096 instructions, loops were not supported until Linux 5.3. Second, since BPF programs can be called from different contexts, the locking mechanism is probably trickier than userspace solutions like the netlink one.

7.2 EBPF-BASED FRAMEWORK FOR PATH MANAGERS

We have implemented a prototype of a generic path-manager framework based on eBPF. The next three subsections explain its basic design: how to track events, how to store addresses and subflows, how to send signals to the remote peer and how to open a new subflow.

7.2.1 *Tracking events*

In order to give decisions, the path managers must know when and which MPTCP-related events happen, as well as the associated information. It is

theoretically possible to perform these operations using BPF programs of the BPF_PROG_TYPE_KPROBE type which dynamically inserts kprobe without requiring any kernel code change. However, we also need to carry actions on the connection (e.g. create or delete a subflow) which should requires TCP-BPF programs of the BPF_PROG_TYPE_SOCK_OPS type. This means that we need two BPF programs of two different types to fulfill the task. Connecting these two BPF programs and synchronizing shared data would be complicated if not ugly.

For this reason, we add new TCP-BPF callbacks to track important events for the path managers (Table 7.1). Since these callbacks are inserted at the same places as the netlink-based Path Manager solution does [13], we do not present these locations in the table. At the moment, to track the subflow-level events we reuse the available TCP-BPF hooks for regular TCP stack (as shown in Table 5.2), and new subflow-specific hooks (as described in Section 6.3.1). The way we pass MPTCP-specific metadata to the BPF programs has been mentioned in Section 6.3.2.

Callbacks	**Events**	Passed arguments
BPF_MPTCP_NEW_SESSION	A new MPTCP session is created	-
BPF_MPTCP_FULLY_ESTABLISHED	An MPTCP session is established	master_sk flag
BPF_MPTCP_CLOSE_SESSION	An MPTCP session is closed (including fallback to legacy TCP)	-
BPF_MPTCP_ADDR_SIGNAL	Call PM to send an ADD_ADDR or RM_ADDR option	-
BPF_MPTCP_ADD_RADDR	A remote IP address is added	IP, port, address ID
BPF_MPTCP_REM_RADDR	A remote IP address is removed	address ID

Table 7.1: New TCP-BPF callbacks for generic PM framework

7.2.2 *Storing local addresses and remote addresses*

The path managers must know the local addresses, as well as remote addresses and subflows for established MPTCP sessions. We use BPF maps - the standard BPF way - to store this information. Local addresses are retrieved and loaded to a BPF map when the BPF program is loaded. This is done by the same user daemon which loads and attaches BPF programs, because this daemon has enough privileges and cgroup context information.

7.2.3 *Sending the MPTCP ADD_ADDR and RM_ADDR option*

An interesting feature of the MPTCP protocol is that it allows a host to signal its peer about its local IP addresses on which it would like to accept additional subflows. In principle, it could be done with our BPF-based option framework mentioned in the previous chapter. However, the ADD_ADDR option size

is variable and may be larger than 16 bytes - the maximum data structure size supported by current TCP-BPF. Meanwhile, there are equivalent facilities which already implemented and optimized for the similar jobs in the MPTCP Linux kernel. Therefore, we created an helper function to reuse these facilities, which considerably simplifies the BPF-based path manager. This helper function (`bpf_mptcp_addr_signal()`) is called when the kernel stack prepares the MPTCP options.

7.2.4 *Opening a subflow*

Since opening subflows is an action that changes the state of the kernel stack, BPF programs can not directly create a subflow. We follow the eBPF common practice by implementing it via a new helper function (`bpf_open_subflow()`). This helper function takes five arguments as input:

- BPF socket context (`bpf_sock_ops`): The helper function uses this context structure to retrieve both subflow-level and mptcp-level information.
- The pointers to source `sockaddr` and destination `sockaddr` of new subflows to be created: Each `sockaddr` includes the IP address and the port number. When any field in the 4-tuple is absent, the function `bpf_open_subflow()` uses the existing or kernel-assigned values when creating the subflow.
- The associated lengths of the above `sockaddrs`, as required by eBPF when passing memory regions.

However, there is one subtle issue here: BPF programs can be called from different *context*s. If we are in the *user context*, we can immediately open a new subflow. However, the helper function is usually called in *softirq context*, in this case we cannot open subflows directly. The reason is that this task requires allocating kernel memory and may sleep which is not possible in the *softirq context*. Our solution is to delegate the actual subflow creation to a `workqueue`. First, we create a custom global `workqueue` when the meta socket is initialized. Every time this helper function is called, it schedules a `work` into this `workqueue` to delegate the actual task in the future. Since we cannot add our custom parameters into the `work` structure itself, we need to embed the `work` and four tuples in a wrapping structure `bpf_pm_priv` to keep track of the subflow request. We store the list of all subflow requests per MPTCP session, which are implemented as a linked list of the structures `bpf_pm_priv` and linked to the multipath control block (the `mptcp_cb` structure). Then, when the kernel scheduler wakes up the worker thread, the work handler actually opens the requested subflow by calling function `mptcp_init4_subsockets()` (or function `mptcp_init6_subsockets()` for IPv6 subflow).

7.3 USE CASES

To illustrate the usage of this path manager framework, we have implemented four path managers as BPF programs (Table 7.2). Two first path managers are *ndiffports* (Section 7.3.1) and *fullmesh* ((Section 7.3.2)), whose kernel-module versions have been included in the Linux MPTCP implementation. The third path manager is designed to recreate a subflow on the same 4-tuple when needed (Section 7.3.3) and the fourth path manager can delay the additional subflows to avoid the overhead of unused subflows (Section 7.3.4).

7.3.1 *ndiffports path manager*

```
SEC("sockops")
int bpf_ndiffport(struct bpf_sock_ops *skops)
{
        int rv = -1;
        skops->reply = rv;

        if (skops->op ==  BPF_MPTCP_FULLY_ESTABLISHED) {
                /* if this is not master sk, skip it */
                if (!skops->args[1])
                        return 0;

                /* when passing (NULL, 0):
                 * existing addresses is used to set up new subflows.
                 * Call twice to open two new subflows */
                rv = bpf_open_subflow(skops,  NULL, 0,  NULL, 0);
                rv = bpf_open_subflow(skops,  NULL, 0,  NULL, 0);
        }
        skops->reply = rv;
        return 1;
}
```

Listing 7.1: ndiffports path manager as a BPF program

ndiffports path manager creates multiple subflows towards the same source and destination IP addresses, only differing in their source port numbers. It is designed to exploit the path diversity to avoid the bottlenecks in ECMP-enabled datacenters [168]. Due to its simplicity, we implemented *ndiffports* program using only around 20 LoCs, as shown in Listing 7.1. We start creating subflows when the MPTCP session is fully established. Notice that for an MPTCP session, this state can be triggered several times, not only on the master subflow, but also on the additional subflows. To reduce the overhead of calling `bpf_open_subflow()` multiple times, it is desired to also pass the number of new subflows as an argument to this function and call it only once.

However, current eBPF infrastructure has the upper limit of five arguments for each helper function, therefore, no available argument left to pass this information to the helper function.

7.3.2 *fullmesh path manager*

The second one, *fullmesh* PM, is more complex since it tries to establish a full mesh of subflows using all IP addresses between the two hosts. It is neccessary to store the local and remote addresses. The local addresses are loaded into an array map (`local_addr_map`) by a user daemon. They are global to all connections. Meanwhile, `add_addr_map` stores the remote addresses per connection, with the MPTCP tokens as keys. The map value is a data structure that contains remote IP addresses and their corresponding address IDs. The remote address list is updated every time the host receives an `ADD_ADDR` or `REM_ADDR` from the remote peer. Upon the MPTCP session closing event, the remote address list of that session is removed from `add_addr_map`. Due to its complexity, we implemented the *fullmesh* program in more than 200 LoCs (Table 7.2).

7.3.3 *Subflow-Refreshing path manager*

There are certain cases where the clients want to recreate a current subflow, due to either performance or security reasons. Carrier-Grade NATs (CGNs) are commonly used by the ISPs to deal with the exhaustion of their public IPv4 address pools [172]. Usually, the idle connections would be terminated by CGNs after some timeout duration, e.g. to reduce the memory usage of CGN equipment. The recommended minimal timeout value is two hours for TCP [90]. However, the timeout in practice could be as low as 30 seconds - the default value on Juniper equipments [137]. This short timeout could disrupt many services. For example, Internet banking applications may need to re-authenticate and to restart the transactions. In these cases, Multipath TCP could help avoiding session interruption by recreating a new subflow when the current subflow is closed due to receiving TCP RST or timeout.

Another scenario is when the network operators deploy transparent middleboxes for traffic shaping. For example, they could apply the throughput throttling on connections which last more than a certain duration, causing performance degradation of long connections. To deal with this issue, clients may create a new subflow to replace the under-performing one.

To illustrate our approach, we use the TCP-BPF framework to monitor when a subflow is closed, but the MPTCP-level session is still alive. The usual reasons for this transition typically are (1) the host has just received a TCP RST or (2) experienced a TCP connection timeout. Since the TCP-BPF framework already supports monitoring TCP states, no kernel change is needed. Once the BPF program observes this event, it can open a brand new subflow. To emulate the CGN timeout events, we use the `tcpkill` program (in the `dsniff`

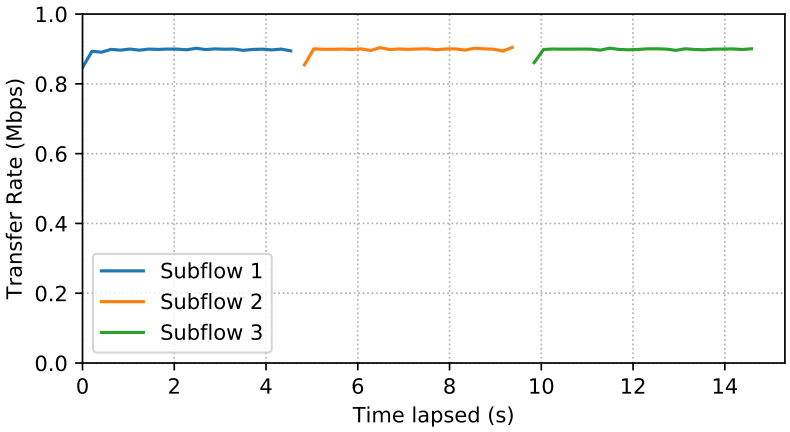


Figure 7.1: Subflow-Refreshing path manager opens a new subflow if the current subflow is abruptly closed

networking tool suite [183]) to regularly inject TCP RST on existing TCP connections. As shown in Section 7.1, every time the current subflow is closed, the client quickly creates a new subflow to replace the old one.

7.3.4 *Subflow-Delaying path manager*

Measurements in Section 3.3.5 have indicated that MPTCP hosts create additional subflows, but they do not transfer any data when the connections are small. The creation of these subflows is useless, but consumes system resources and cellular energy. Therefore, it is often desirable to create additional subflows only for large enough flows. A similar use case has been mentioned in [100], though it uses the HTTP header `Content-Length` for signaling.

To support this use case, it is necessary to regularly check the amount of received data in the BPF program. We can add a new TCP-BPF hook to call the BPF program for every incoming TCP packet, however, it would be very expensive. To reduce the cost of switching context on the fast path, instead we conduct this check only when the user applications make a syscall to get received data. We insert one BPF hook in the `tcp_recvmsg()` function, which is called along with all flavors of receiving syscalls: `read()`, `recvmsg()`, `recvmmsg()` and `recvfrom()`. To further avoid the overhead of this check, we disable this hook by default and only enable it when necessary. With this solution, the BPF program is called in the *user context*. Therefore, we do not need to defer the subflow opening task to a workqueue, instead a new subflow can be established immediately.

To change the threshold of connection size, we can replace this BPF program by another one, or simply make it available via BPF map so that the user space could set it on demand. The path manager could rely on other metrics to

decide opening new subflows, e.g. the duration since the connection started, or when the target bandwidth is not reached after some duration.

	Kernel (LoCs)	**BPF program** (LoCs)
Generic PM Framework	~300	0
ndiffports PM	0	20
fullmesh PM	0	~200
Subflow-delaying PM	5	45
Subflow-refreshing PM	0	50

Table 7.2: The implementation size (Lines of Code) of generic eBPF PM framework and each path manager

7.4 DISCUSSION

Our framework prototype allows custom path management mechanisms to be deployed or replaced on the fly. At the moment, however, several features have not been implemented in this prototype.

First, we need to handle the events when a local IP address status changes. Since these events are global, we can use a userspace daemon to track the status changes of local IP addresses. Once detecting the IP address status changes, the daemon will update the map of local addresses and trigger all relevant TCP-BPF programs. However, there is one technical caveat in this step. Since the current TCP-BPF implementation is based on `cgroup-v2`, it is necessary to send the same address-change events to each BPF program in each `cgroup-v2`. For the TCP-BPF program type, we need to pass the appropriate `sock` struct which contains the cgroup information. A possible but ugly solution is to create and store a dummy socket per `cgroup` when we start loading path-manager BPF program, then use these dummy sockets to trigger the corresponding BPF programs.

Second, the framework currently does not support the subflow removal operation. This feature could be implemented as a helper function, in a similar way to the function `bpf_open_subflow()`. In fact, for the common case when receiving a `REMOVE_ADDR` option, current Linux MPTCP implementation has already closed automatically the impacted subflows in the kernel. On the other hand, this feature may be needed in other cases. One use case is to close bad subflows e.g. those causing repeated retransmissions and reinjections. Another use case is to close the cellular subflows when cellular traffic quota is being reached. This may require to store the list of active subflows (e.g. in a sockmap) or to query the subflow list on demand (e.g. by using the `MPTCP_INFO` socket option). The implementation of this function may be similar to the `bpf_open_subflow()` function.

Third, only IPv4 is supported in the function `bpf_open_subflow()` so far. In the mainline Linux, dual-stack support has been implemented in the helper function `bpf_bind()` which is used to customize the `bind()` syscall. The dual-stack handling in the function `bpf_open_subflow()` could be implemented in a similar way.

We expect that there will be new use cases of Multipath TCP in the future. For example, 5G technology supports different operating modes, e.g. multiple radio links that may be exposed via single or multiple IP addresses per client. This creates an opportunity but also set new challenges [83] for Multipath TCP to exploit last-mile path diversity in 5G networks. An eBPF approach may allow deploying sophisticated yet flexible subflow management strategies.

8 CONCLUSION

Multipath TCP is more than simply a TCP extension. By decoupling TCP from IP and enabling resource pooling, it brings several benefits. It allows seamless handover, aggregates the bandwidth of multiple paths, increases resiliency, can help to reduce latency and so on. However, Multipath TCP is significantly more complex than regular TCP. It is important to understand how it currently works in practice. In Part II, we gathered new understandings of Multipath TCP by analysing its traffic in both fixed and wireless networks.

In Chapter 3, we revealed general insights of Multipath TCP traffic from passive measurements on the server side. Multipath TCP works correctly through a large portion of Internet paths, with a small number of connections falling back to legacy TCP. Looking at the data exchanges, there could be a large difference of RTTs among the concurrent subflows, which is a challenge for the packet schedulers. We also showed that the path managers need to be improved to avoid creating subflows unnecessarily.

In Chapter 4, we focused on a typical use case of Multipath TCP: cloud-based voice recognition service. We leveraged the MONROE platform to run experiments with simulated traffic over multiple wireless interfaces. Our results show that Multipath TCP brings benefits to this service, but the performance varied with different configurations of Multipath TCP. The sender should have up-to-date information about the current state of each subflow.

Our experiences with the Linux kernel implementation of Multipath TCP have shown that it is difficult to customize the stack. This is a generic problem for in-kernel stacks, notably the Linux vanilla TCP stack which is highly stable and optimized but hard to extend. In Part III, we explored the possibility of dynamically extending the TCP and MPTCP stacks in the Linux kernel. The eBPF virtual machine has been used since it provides a safe and efficient execution environment of user code inside the kernel. In Chapters 5 and 6, we leveraged this eBPF execution environment to support user-defined TCP and MPTCP options - the standard way to extend the protocol. We illustrated the usage of these option frameworks with several practical use cases in each chapter. Then, as presented in Chapter 7, the eBPF infrastructure also allowed us to support user-defined path managers. Four path managers have been implemented as eBPF programs. Compared to the existing Netlink-based solution, it avoids the cost of switching context between user space and kernel space and does not rely on unreliable message passing.

A similar work to ours has been proposed by Frömmgen et al. [84] that based on eBPF to support application-defined packet schedulers, showing again that using eBPF for extending the Linux MPTCP stack is a promising approach. The principal difference of this work to ours is that they proposed

a highly-abstracted programming model. This allows application developers to easily describe the scheduler logic using a high-level syntax. However, this pushes more complexity into the kernel. In contrary, we implement basic, low-level facilities in the MPTCP stack to minimize code modifications in the kernel.

OPEN PROBLEMS

In Part II, we have revealed some inefficiencies of Multipath TCP implementations. We believe that it is important to build a *functional and performance test suite* to automatically detect potential issues and ensure the quality of each implementation. This is a step towards improving the protocol and its implementations so that it can reach the same level of stability and efficiency that TCP reached after decades of usage.

In Part III, our work has not yet made the TCP and Multipath TCP stacks fully extensible, but rather a step towards this ultimate goal. While the eBPF virtual machine is currently available only in Linux, its proven benefits have motivated some efforts to port it into other operating systems, e.g. FreeBSD [96].

At the time of this writing, a team of kernel developers are actively working to implement a clean-slate version of Multipath TCP for upstreaming to the mainline Linux kernel [151]. This implementation is expected to introduce fewer changes to the regular TCP stack and provide a much cleaner separation between the subflow layer and the multipath layer. Therefore, it is expected that adding eBPF-based extensions to this new implementation will be more straightforward than in the current one.

The standardization of the second version of Multipath TCP at the IETF has been finalized and is expected to be ratified soon. This version proposed several reliability and security improvements that protect Multipath TCP from various types of attacks. However, the protocol is still relying on plain TCP options, raising privacy concerns. This will be more important when the protocol becomes popular and MPTCP-enabled hosts are the norm. It may be necessary to design and experiment a security-enhanced version [116] of Multipath TCP which uses TLS crypto context to encrypt the MPTCP options. An eBPF-based approach could potentially be used to implement and test such a new prototype.

The eBPF approach could also be applied to extend other protocols. For example, De Coninck et al. [57] have recently proposed to extend the QUIC protocol and its implementations using a fork of the eBPF virtual machine in user space.

BIBLIOGRAPHY

[1] Alexander Afanasyev, Neil Tilley, Peter Reiher, and Leonard Kleinrock. "Host-to-host congestion control for TCP." In: *Communications Surveys & Tutorials, IEEE* 12.3 (2010), pp. 304–342. DOI: 10.1109/SURV.2010.042710.00114.

[2] *Agentbot - Automatic customer support with Artificial Intelligence. Link: https://aivo.co/en/agentbot/*. en-US. URL: https://aivo.co/en/agentbot/ (visited on 02/25/2018).

[3] Özgü Alay et al. "Measuring and assessing mobile broadband networks with MONROE." In: *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2016 IEEE 17th International Symposium on A*. IEEE. 2016, pp. 1–3.

[4] M. Allman, V. Paxson, and E. Blanton. *TCP Congestion Control*. RFC 5681 (Draft Standard). RFC. Fremont, CA, USA: RFC Editor, Sept. 2009. DOI: 10.17487/RFC5681. URL: https://www.rfc-editor.org/rfc/rfc5681.txt.

[5] Mark Allman, Sally Floyd, and Craig Partridge. "RFC 3390: Increasing TCP's initial window." In: *IETF-Request for Comments* (2002).

[6] S. Amante, B. Carpenter, S. Jiang, and J. Rajahalme. *IPv6 Flow Label Specification*. RFC 6437 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Nov. 2011. DOI: 10.17487/RFC6437. URL: https://www.rfc-editor.org/rfc/rfc6437.txt.

[7] Nadav Amit and Michael Wei. "The design and implementation of hyperupcalls." In: *2018 USENIX ATC*. 2018, pp. 97–112.

[8] Apple. "iOS: Multipath TCP Support in iOS 7." http://support.apple.com/en-us/HT201373.

[9] B. Arzani, A. Gurney, Shuotian Cheng, R. Guerin, and Boon Thau Loo. "Impact of Path Characteristics and Scheduling Policies on MPTCP Performance." In: *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*. 2014, pp. 743–748. DOI: 10.1109/WAINA.2014.121.

[10] Mehdi Assefi, Mike Wittie, and Allan Knight. "Impact of network performance on cloud speech recognition." In: *Computer Communication and Networks (ICCCN), 2015*. IEEE, 2015, pp. 1–6.

[11] Mehdi Assefi, Guangchi Liu, Mike P. Wittie, and Clemente Izurieta. "An experimental evaluation of Apple Siri and Google speech recognition." In: *Proccedings of the 2015 ISCA SEDE* (2015).

[12] Daniel O Awduche. "MPLS and traffic engineering in IP networks." In: *IEEE Communications magazine* 37.12 (1999), pp. 42–47.

[13] Matthieu Baerts and Gregory Detal. *[PATCH mptcp_trunk v8 0/5] mptcp: new generic Netlink-based PM*. Jan. 2019. URL: https://sympa-2.sipr.ucl.ac.be/sympa/arc/mptcp-dev/2019-01/msg00084.html (visited on 02/24/2019).

[14] Praveen Balasubramanian. "IETF96: Transports advancements in the Windows network stack." In: Berlin: IETF, 2016. URL: https://datatracker.ietf.org/meeting/96/materials/slides-96-tcpm-4.

[15] S. Barre. "Implementation and assessment of Modern Host-based Multipath Solutions." PhD thesis. UCLouvain, 2011.

[16] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 41–46.

[17] Jeffrey P. Bigham et al. "On How Deaf People Might Use Speech to Control Devices." en. In: ACM, 2017, pp. 383–384. ISBN: 978-1-4503-4926-0. DOI: 10.1145/3132525.3134821. (Visited on 02/25/2018).

[18] Ethan Blanton et al. "A Roadmap for Transmission Control Protocol (TCP) Specification Documents." In: *RFC7414* (2015).

[19] Luca Boccassi, Marwan M. Fayed, and Mahesh K. Marina. "Binder: A System to Aggregate Multiple Internet Gateways in Community Networks." In: *Proceedings of the 2013 ACM MobiCom Workshop on Lowest Cost Denominator Networking for Universal Access*. LCDNet '13. Miami, Florida, USA: ACM, 2013, pp. 3–8. ISBN: 978-1-4503-2365-9. DOI: 10.1145/2502880.2502894.

[20] Carl Boettiger. "An introduction to Docker for reproducible research." In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79.

[21] Olivier Bonaventure, Benjamin Hesmans, and Mohamed Boucadair. *Experimental Multipath TCP option*. Internet-Draft draft-bonaventure-mptcp-exp-option-00. Work in Progress. Internet Engineering Task Force, June 2015. 8 pp.

[22] Olivier Bonaventure, Christoph Paasch, and Gregory Detal. *Experience with Multipath TCP*. Internet-Draft draft-ietf-mptcp-experience-01. I-D Exists. IETF Secretariat, Mar. 2015. URL: http://tools.ietf.org/html/draft-ietf-mptcp-experience-01.

[23] Olivier Bonaventure and SungHoon Seo. "Multipath TCP deployments." In: *IETF Journal* 12.2 (2016), pp. 24–27.

[24] Olivier Bonaventure et al. *o-RTT TCP Convert Protocol*. Internet-Draft draft-ietf-tcpm-converters-10. Work in Progress. Internet Engineering Task Force, Aug. 2019. 48 pp.

[25] Daniel Borkmann. *BPF support for global data*. 2019. URL: https://patchwork.ozlabs.org/cover/1082775/.

[26] Timm Böttger et al. "Open connect everywhere: A glimpse at the Internet ecosystem through the lens of the netflix CDN." In: *ACM SIGCOMM Computer Communication Review* 48.1 (2018), pp. 28–34.

[27] Lawrence S. Brakmo and Larry L. Peterson. "TCP Vegas: End to end congestion avoidance on a global Internet." In: *IEEE Journal on selected Areas in communications* 13.8 (1995), pp. 1465–1480.

[28] Lawrence Brakmo. *Linux Kernel patchset: bpf: BPF support for sock_ops*. 2017. URL: https://www.spinics.net/lists/netdev/msg443170.html.

[29] Lawrence Brakmo. *Linux Kernel patchset: bpf: add support for BASE_RTT*. 2017. URL: https://patchwork.ozlabs.org/project/netdev/list/?series=9392.

[30] Lawrence Brakmo. "TCP-BPF: Programmatically tuning TCP behavior through BPF." In: *NetDev 2.2* (2017).

[31] Lawrence Brakmo. *[net-next,v6,13/16] bpf: Sample BPF program to set initial cwnd.* 2017. URL: https://patchwork.ozlabs.org/patch/783031/.

[32] Lawrence Brakmo. *Linux Kernel patchset: bpf: More sock_ops callbacks*. 2018. URL: https://patchwork.ozlabs.org/project/netdev/list/?series=25441.

[33] Yu Cao, Mingwei Xu, and Xiaoming Fu. "Delay-based congestion control for Multipath TCP." In: *Network Protocols (ICNP), 2012 20th IEEE International Conference on*. 2012, pp. 1–10. DOI: 10.1109/ICNP.2012.6459978.

[34] Neal Cardwell. *Linux Kernel: Merge branch: fix stretch ACK bugs in TCP CUBIC and Reno*. 2015. URL: https://patchwork.ozlabs.org/patch/434249/.

[35] Neal Cardwell et al. "BBR: congestion-based congestion control." In: *Communications of the ACM* 60.2 (2017), pp. 58–66.

[36] Erik Carlsson and Eirini Kakogianni. Spotify Labs. *Smoother Streaming with BBR*. Aug. 2018. URL: https://labs.spotify.com/2018/08/31/smoother-streaming-with-bbr/.

[37] L. Caviglione. "A first look at traffic patterns of Siri." en. In: *Transactions on Emerging Telecommunications Technologies* 26.4 (2013), pp. 664–669. ISSN: 21613915. DOI: 10.1002/ett.2697. URL: http://doi.wiley.com/10.1002/ett.2697 (visited on 11/06/2017).

[38] Vinton Cerf and Robert Kahn. "A protocol for packet network intercommunication." In: *IEEE Transactions on communications* 22.5 (1974), pp. 637–648.

[39] Michael Chan and David R Cheriton. "Improving Server Application Performance via Pure TCP ACK Receive Optimization." In: *USENIX Annual Technical Conference*. 2013, pp. 359–364.

[40] Yung-Chih Chen, Don Towsley, and Ramin Khalili. "MSPlayer: Multi-Source and multi-Path LeverAged YoutubER." en. In: ACM Press, 2014, pp. 263–270. ISBN: 978-1-4503-3279-8. DOI: 10.1145/2674005.2675007. (Visited on 12/17/2015).

[41] Yung-Chih Chen et al. "A Measurement-based Study of MultiPath TCP Performance over Wireless Networks." In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC '13. Barcelona, Spain: ACM, 2013, pp. 455–468. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504751.

[42] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. *TCP Fast Open*. RFC 7413. RFC. Fremont, CA, USA: RFC Editor, Dec. 2014. DOI: 10.17487/RFC7413. URL: https://www.rfc-editor.org/rfc/rfc7413.txt.

[43] Yuchung Cheng. *Linux Kernel: commit 9f9843a751do. tcp: properly handle stretch acks in slow start*. 2013. URL: https://patchwork.ozlabs.org/patch/287581/.

[44] Yuchung Cheng. *bpf: fix SO_MAX_PACING_RATE to support TCP internal pacing.* Jan. 2019. URL: https://www.spinics.net/lists/netdev/msg545080.html.

[45] Yuchung Cheng, Neal Cardwell, N. Dukkipati, and P. Jha. "RACK: a time-based fast loss detection algorithm for TCP. draft-ietf-tcpm-rack-04." In: (2018). URL: https://tools.ietf.org/html/draft-ietf-tcpm-rack-04.

[46] Gregory Detal Christoph Paasch. *Modified net-tools to support MPTCP.* Sept. 2019. URL: https://github.com/multipath-tcp/net-tools.

[47] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis. *Increasing TCP's Initial Window*. RFC 6928 (Experimental). RFC. Fremont, CA, USA: RFC Editor, Apr. 2013. DOI: 10.17487/RFC6928. URL: https://www.rfc-editor.org/rfc/rfc6928.txt.

[48] David D Clark et al. "An analysis of TCP processing overhead." In: *IEEE Communications magazine* 27.6 (1989), pp. 23–29.

[49] Lorenzo Colitti et al. "Evaluating IPv6 adoption in the Internet." In: *International Conference on Passive and Active Network Measurement*. Springer. 2010, pp. 141–150.

[50] J Corbet. "Pluggable congestion avoidance modules." In: *Linux Weekly News* (2005).

[51] Xavier Corbillon et al. "Cross-layer scheduler for video streaming over MPTCP." In: *Proceedings of the 7th International Conference on Multimedia Systems*. ACM. 2016, p. 7.

[52] Team Cymru. *IP to ASN mapping*. http://www.team-cymru.org/IP-ASN-mapping.html. Jan. 5.

[53] Renzo Davoli. "VDE: Virtual Distributed Ethernet." In: *Tridentcom 2005*. IEEE. 2005, pp. 213–220.

[54] Quentin De Coninck and Olivier Bonaventure. "Multipath QUIC: Design and evaluation." In: *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM. 2017, pp. 160–166.

[55] Quentin De Coninck and Olivier Bonaventure. "Tuning Multipath TCP for Interactive Applications on Smartphones." In: *IFIP Networking 2018*. May 2018.

[56] Quentin De Coninck, Matthieu Baerts, Benjamin Hesmans, and Olivier Bonaventure. "Observing real smartphone applications over Multipath TCP." In: *IEEE Communications Magazine* 54.3 (2016), pp. 88–93.

[57] Quentin De Coninck et al. "Pluginizing QUIC." In: *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM*. 2019, pp. 19–24.

[58] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 1883 (Proposed Standard). RFC. Obsoleted by RFC 2460. Fremont, CA, USA: RFC Editor, Dec. 1995. DOI: 10.17487/RFC1883. URL: https://www.rfc-editor.org/rfc/rfc1883.txt.

[59] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. "WiFi, LTE, or Both?: Measuring Multi-Homed Wireless Internet Performance." In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC '14. Vancouver, BC, Canada: ACM, 2014, pp. 181–194. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663727. URL: http://doi.acm.org/10.1145/2663716.2663727.

[60] Gregory Detal, Benjamin Hesmans, Olivier Bonaventure, Yves Vanaubel, and Benoit Donnet. "Revealing Middlebox Interference with `tracebox`." In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC '13. Barcelona, Spain: ACM, 2013, pp. 1–8. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504757.

[61] Amogh Dhamdhere and Constantine Dovrolis. "The Internet is flat: modeling the transition from a transit hierarchy to a peering mesh." In: *Proceedings of the 6th International COnference*. ACM. 2010, p. 21.

[62] J. Dike. "User Mode Linux." In: *Proceedings of the 5th Anual Linux Showcase and Conference*. ALS'01. USENIX Association. 2001, pp. 3–14.

[63] Jon Dugan et al. "iPerf3, tool for active measurements of the maximum achievable bandwidth on IP networks." In: *URL: https://github.com/esnet/iperf* ().

[64] Nandita Dukkipati et al. "An argument for increasing TCP's initial congestion window." In: *Computer Communication Review* 40.3 (2010), pp. 26–33.

[65] Nandita Dukkipati et al. "Tail loss probe (TLP): An algorithm for fast recovery of tail losses." In: *draft-dukkipati-tcpm-tcploss-probe-01. txt* (2013).

[66] Adam Dunkels. "Design and Implementation of the lwIP TCP/IP Stack." In: *Swedish Institute of Computer Science* 2 (2001), p. 77.

[67] Adam Dunkels. "The uIP embedded TCP/IP stack." In: *The uIP* 1 (2006).

[68] Korian Edeline and Benoit Donnet. "A Bottom-Up Investigation of the Transport-Layer Ossification." In: *Network Traffic Measurement and Analysis (TMA) Conference 2019*. 2019.

[69] Korian Edeline, Mirja Kühlewind, Brian Trammell, Emile Aben, and Benoit Donnet. "Using UDP for internet transport evolution." In: *arXiv preprint arXiv:1612.07816* (2016).

[70] L. Eggert and F. Gont. *TCP User Timeout Option*. RFC 5482 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Mar. 2009. DOI: 10.17487/RFC5482. URL: https://www.rfc-editor.org/rfc/rfc5482.txt.

[71] G. Fairhurst (Ed.), B. Trammell (Ed.), and M. Kuehlewind (Ed.) *Services Provided by IETF Transport Protocols and Congestion Control Mechanisms*. RFC 8095 (Informational). RFC. Fremont, CA, USA: RFC Editor, Mar. 2017. DOI: 10.17487/RFC8095. URL: https://www.rfc-editor.org/rfc/rfc8095.txt.

[72] G. Fairhurst and M. Welzl. *The Benefits of Using Explicit Congestion Notification (ECN)*. RFC 8087 (Informational). RFC. Fremont, CA, USA: RFC Editor, Mar. 2017. DOI: 10.17487/RFC8087. URL: https://www.rfc-editor.org/rfc/rfc8087.txt.

[73] John Fastabend. *BPF control flow, supporting loops and other patterns*. 11-2018. URL: http://vger.kernel.org/lpc-bpf.html.

[74] S. Ferlin-Oliveira, T. Dreibholz, and O. Alay. "Tackling the challenge of bufferbloat in Multi-Path Transport over heterogeneous wireless networks." In: *Quality of Service (IWQoS), 2014 IEEE 22nd International Symposium of*. 2014, pp. 123–128. DOI: 10.1109/IWQoS.2014.6914310.

[75] S. Ferlin, O. Alay, O. Mehani, and R. Boreli. "BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks." In: *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. May 2016, pp. 431–439. DOI: 10.1109/IFIPNetworking.2016.7497206.

[76] Simone Ferlin, Thomas Dreibholz, and Özgü Alay. "Multi-Path Transport Over Heterogeneous Wireless Networks: Does It Really Pay Off?" In: *Proceedings of the IEEE GLOBECOM*. Austin, Texas/U.S.A.: IEEE, Dec. 2014. DOI: 10.1109/GLOCOM.2014.7037567.

[77] Joel Fernandes. *Extending the Kernel with Built-in Kernel Headers | Linux Journal*. URL: https://www.linuxjournal.com/content/extending-kernel-built-kernel-headers (visited on 08/06/2019).

[78] A. Finamore, M. Mellia, M. Meo, M.M. Munafo, and D. Rossi. "Experiences of Internet traffic monitoring with tstat." In: *Network, IEEE* 25.3 (May 2011), pp. 8–14. ISSN: 0890-8044. DOI: 10.1109/MNET.2011.5772055.

[79] S. Floyd, M. Allman, A. Jain, and P. Sarolahti. *Quick-Start for TCP and IP*. RFC 4782 (Experimental). RFC. Fremont, CA, USA: RFC Editor, Jan. 2007. DOI: 10.17487/RFC4782. URL: https://www.rfc-editor.org/rfc/rfc4782.txt.

[80] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824. Internet Engineering Task Force, 2013.

[81] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. *TCP Extensions for Multipath Operation with Multiple Addresses*. Internet-Draft draft-ietf-mptcp-rfc6824bis-03. I-D Exists. IETF Secretariat, Oct. 2014. URL: http://tools.ietf.org/html/draft-ietf-mptcp-rfc6824bis-03.

[82] Armando Fox et al. "Above the clouds: A berkeley view of cloud computing." In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28.13 (2009), p. 2009.

[83] Xavier de Foy et al. *Considerations for MPTCP operation in 5G*. Internet-Draft draft-defoy-mptcp-considerations-for-5g-01. Work in Progress. Internet Engineering Task Force, June 2018. 13 pp.

[84] Alexander Froemmgen et al. "A Programming Model for Application-defined Multipath TCP Scheduling." In: *ACM/IFIP/USNIX Middleware*. 2017.

[85] Alexander Frömmgen, Jens Heuschkel, and Boris Koldehofe. "Multipath TCP scheduling for thin streams: Active probing and one-way delay-awareness." In: *2018 IEEE International Conference on Communications (ICC)*. IEEE. 2018, pp. 1–7.

[86] Alexander Frommgen et al. "ReMP TCP: Low latency multipath TCP." In: *2016 IEEE International Conference on Communications (ICC)*. IEEE. 2016, pp. 1–7.

[87] Kensuke Fukuda. "An analysis of longitudinal TCP passive measurements." In: *TMA Workshop*. Springer. 2011, pp. 29–36.

[88] Ben Greear and Daniel Baluta. *Linux Kernel: [RFC] TCP: Support configurable delayed-ack parameters.* 2012. URL: https://patchwork.ozlabs.org/patch/165626/.

[89] Brendan Gregg. *Linux Performance [Online]*. 2018. URL: http://www.brendangregg.com/linuxperf.html.

[90] S. Guha (Ed.), K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. *NAT Behavioral Requirements for TCP*. RFC 5382 (Best Current Practice). RFC. Updated by RFC 7857. Fremont, CA, USA: RFC Editor, Oct. 2008. DOI: 10.17487/RFC5382.

[91] Haryadi S Gunawi et al. "Deploying Safe User-Level Network Services with icTCP." In: *OSDI*. 2004, pp. 317–332.

[92] Yihua Ethan Guo et al. "Accelerating multipath transport through balanced subflow completion." In: *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. ACM. 2017, pp. 141–153.

[93] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: a new TCP-friendly high-speed TCP variant." In: *ACM SIGOPS operating systems review* 42.5 (2008), pp. 64–74.

[94] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. "MP-DASH: Adaptive video streaming over preference-aware multipath." In: *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM. 2016, pp. 129–143.

[95] Jiangping Han et al. "Measurement and Redesign of BBR-based MPTCP." In: *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*. SIGCOMM Posters and Demos '19. Beijing, China: ACM, 2019, pp. 75–77. ISBN: 978-1-4503-6886-5. DOI: 10.1145/3342280.3342312.

[96] Yutaro Hayakawa. "eBPF Implementation for FreeBSD." In: *BSDCan 2018*. The BSD Conference. 2018.

[97] B. Hesmans, F. Duchene, C. Paasch, G. Detal, and O. Bonaventure. "Are TCP Extensions Middlebox-proof?" In: *CoNEXT Workshop HotMiddlebox*. 2013. DOI: 10.1145/2535828.2535830.

[98] B. Hesmans, H. Tran-Viet, R. Sadre, and B. Bonaventure. "A First Look at Real Multipath TCP Traffic." In: *7th International Workshop on Traffic Monitoring and Analysis (TMA)*. 2015.

[99] Benjamin Hesmans and Olivier Bonaventure. "Tracing Multipath TCP Connections." In: *SIGCOMM Comput. Commun. Rev.* 44.4 (Aug. 2014), pp. 361–362. ISSN: 0146-4833. DOI: 10.1145/2740070.2631453. URL: http://doi.acm.org/10.1145/2740070.2631453.

[100] Benjamin Hesmans and Olivier Bonaventure. "An enhanced socket API for Multipath TCP." en. In: *ANRW*. ACM, June 2016, pp. 1–6. ISBN: 978-1-4503-4443-2. DOI: 10.1145/2959424.2959433. (Visited on 07/29/2016).

[101] Benjamin Hesmans, Fabien Duchene, Christoph Paasch, Gregory Detal, and Olivier Bonaventure. "Are TCP Extensions Middlebox-proof?" In: *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. HotMiddlebox '13. Santa Barbara, California, USA: ACM, 2013, pp. 37–42. ISBN: 978-1-4503-2574-5. DOI: 10.1145/2535828.2535830.

[102] Benjamin Hesmans, Hoang Tran-Viet, Ramin Sadre, and Olivier Bonaventure. "A First Look at Real Multipath TCP Traffic." In: *Traffic Monitoring and Analysis*. Vol. 9053. LNCS. Springer International Publishing, 2015, pp. 233–246. ISBN: 978-3-319-17171-5. DOI: 10.1007/978-3-319-17172-2_16.

[103] Benjamin Hesmans, Gregory Detal, Raphaël Bauduin, Olivier Bonaventure, et al. "SMAPP: Towards Smart Multipath TCP-enabled APPlications." In: *CoNEXT'15*. 2015.

[104] Benjamin Hesmans et al. "SMAPP: Towards smart Multipath TCP-enabled applications." In: *Proceedings of the 11th ACM CONEXT*. ACM. 2015, p. 28.

[105] R. Hinden (Ed.) and S. Deering (Ed.) *IP Version 6 Addressing Architecture*. RFC 1884 (Historic). RFC. Obsoleted by RFC 2373. Fremont, CA, USA: RFC Editor, Dec. 1995. DOI: 10.17487/RFC1884. URL: https://www.rfc-editor.org/rfc/rfc1884.txt.

[106] Toke Høiland-Jørgensen et al. "The eXpress data path: fast programmable packet processing in the operating system kernel." In:

[107] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. "Is It Still Possible to Extend TCP?" In: *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*. IMC '11. Berlin, Germany: ACM, 2011, pp. 181–194. ISBN: 978-1-4503-1013-0. DOI: 10.1145/2068816.2068834.

[108] Michio Honda et al. "Is it still possible to extend TCP?" In: *Proceedings of the 2011 ACM SIGCOMM*. ACM. 2011, pp. 181–194.

[109] Michio Honda et al. "Rekindling network protocol innovation with user-level stacks." In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), pp. 52–58.

[110] Christian Huitema. *QUIC Multipath Requirements*. Internet-Draft draft-huitema-quic-mpath-req-01. Work in Progress. Internet Engineering Task Force, Jan. 2018. 13 pp.

[111] *IETF Minutes. https://datatracker.ietf.org/doc/minutes-99-tcpm/*. Prague, 2017. URL: https://datatracker.ietf.org/doc/minutes-99-tcpm/.

[112] DPDK Intel. *Data plane Development Kit*. 2014. URL: https://dpdk.org/.

[113] Janardhan R Iyengar, Paul D Amer, and Randall Stewart. "Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths." In: *IEEE/ACM Transactions on networking* 14.5 (2006), pp. 951–964.

[114] V. Jacobson, R. Braden, and D. Borman. *TCP Extensions for High Performance*. RFC 1323 (Proposed Standard). RFC. Obsoleted by RFC 7323. Fremont, CA, USA: RFC Editor, May 1992. DOI: 10.17487/RFC1323. URL: https://www.rfc-editor.org/rfc/rfc1323.txt.

[115] Van Jacobson. "Congestion avoidance and control." In: 18.4 (1988), pp. 314–329.

[116] Mathieu Jadin, Gautier Tihon, Olivier Pereira, and Olivier Bonaventure. "Securing multipath tcp: Design & implementation." In: *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE. 2017, pp. 1–9.

[117] EunYoung Jeong et al. "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems." In: *NSDI*. Vol. 14. 2014, pp. 489–502.

[118] Jim Keniston et al. "Kernel probes (kprobes)." In: *Documentation provided with the Linux kernel sources (v2. 6.29)* (2016).

[119] Nicolas Keukeleire, Benjamin Hesmans, and Olivier Bonaventure. "Increasing broadband reach with Hybrid Access Networks." In: *arXiv preprint arXiv:1907.04570* (2019).

[120] Ramin Khalili et al. "MPTCP is not pareto-optimal: performance issues and a possible solution." In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 1–12. (Visited on 05/20/2015).

[121] Ali Safari Khatouni et al. "Speedtest-like Measurements in 3G/4G Networks: the MONROE Experience." In: *Teletraffic Congress (ITC 29), 2017 29th International*. Vol. 1. IEEE. 2017, pp. 169–177.

[122] Adam Langley et al. "The QUIC Transport Protocol: Design and Internet-Scale Deployment." en. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 183–196. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098842.

[123] Adam Langley et al. "The QUIC transport protocol: Design and Internet-scale deployment." In: *Proceedings of the ACM SIGCOMM*. ACM. 2017, pp. 183–196.

[124] Bob Lantz, Brandon Heller, and Nick McKeown. "A network in a laptop: rapid prototyping for software-defined networks." In: *Proceedings of the 9th ACM SIGCOMM HotNets Workshop*. 2010, p. 19.

[125] David Law et al. "Evolution of Ethernet standards in the IEEE 802.3 working group." In: *IEEE Communications Magazine* 51.8 (2013), pp. 88–96.

[126] HyunJong Lee, Jason Flinn, and Basavaraj Tonshal. "RAVEN: Improving interactive latency for the connected car." In: *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. ACM. 2018, pp. 557–572.

[127] Li Li et al. "A measurement study on multi-path TCP with multiple cellular carriers on high speed rails." In: *Proceedings of the 2018 Conference of the ACM SIGCOMM*. ACM. 2018, pp. 161–175.

[128] Yeon-sup Lim, Yung-Chih Chen, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. "How Green is Multipath TCP for Mobile Devices?" In: *Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications, & Challenges*. Chicago, Illinois, USA: ACM, 2014, pp. 3–8. ISBN: 978-1-4503-2990-3. DOI: 10.1145/2627585.2627596. URL: http://doi.acm.org/10.1145/2627585.2627596.

[129] Yeon-sup Lim et al. "Design, Implementation, and Evaluation of Energy-Aware Multi-Path TCP." In: *CoNEXT'15*. 2015. (Visited on 01/09/2016).

[130] Yeon-sup Lim, Erich M Nahum, Don Towsley, and Richard J Gibbens. "ECF: An MPTCP path scheduler to manage heterogeneous paths." In: *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM. 2017, pp. 147–159.

[131] Ioana Livadariu, Simone Ferlin, Ozgu Alay, Thomas Dreibholz, Amogh Dhamdhere, and Ahmed Mustafa Elmokashfi. "Leveraging the IPv4/IPv6 Identity Duality by using Multi-Path Transport." In: *Proceedings of the 18th IEEE Global Internet Symposium (GI)*. Hong Kong/People's Republic of China, Apr. 2015. URL: https://www.simula.no/sites/www.simula.no/files/publications/files/gis2015_0.pdf.

[132] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*. RFC 2018 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Oct. 1996. DOI: 10.17487/RFC2018. URL: https://www.rfc-editor.org/rfc/rfc2018.txt.

[133] Jeffrey Mogul et al. "Unveiling the transport." In: *ACM SIGCOMM Computer Communication Review* 34.1 (2004), pp. 99–106.

[134] Quentin Monnet. *tools: bpftool: add probes for system and device*. 2019. URL: https://patchwork.ozlabs.org/cover/1026736/.

[135] Andrii Nakryiko. *CO-RE offset relocations*. 2019. URL: https://patchwork.ozlabs.org/cover/1143704/.

[136] Akshay Narayan et al. "Restructuring endpoint congestion control." In: *Proceedings of the SIGCOMM 2018*. ACM. 2018, pp. 30–43.

[137] Juniper Networks. *Carrier-Grade NAT Implementation: Best Practices - TechLibrary*. May 2018. URL: https://www.juniper.net/documentation/en_US/junos/topics/concept/nat-best-practices.html (visited on 06/04/2019).

[138] K. Nguyen, M. G. Kibria, K. Ishizu, F. Kojima, and H. Sekiya. "An Evaluation of Multipath TCP in Lossy Environment." In: *2019 IEEE PerCom Workshops*. 2019, pp. 573–577. DOI: 10.1109/PERCOMW.2019.8730695.

[139] Kien Nguyen, Mirza Golam Kibria, Kentaro Ishizu, Fumihide Kojima, and Hiroo Sekiya. "An Approach to Reinforce Multipath TCP with Path-Aware Information." In: *Sensors* 19.3 (2019), p. 476.

[140] E. Nordmark and M. Bagnulo. *Shim6: Level 3 Multihoming Shim Protocol for IPv6*. RFC 5533 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2009. DOI: 10.17487/RFC5533. URL: https://www.rfc-editor.org/rfc/rfc5533.txt.

[141] Shawn Ostermann. *Tcptrace*. http://tcptrace.org. 2005.

[142] C. Paasch, S. Barre, et al. "Multipath TCP implementation in the Linux kernel." available from http://www.multipath-tcp.org. 2014.

[143] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure. "Exploring Mobile/WiFi Handover with Multipath TCP." In: *ACM SIGCOMM CellNet workshop*. Helsinki, Finland, 2012, pp. 31–36. ISBN: 978-1-4503-1475-6. DOI: 10.1145/2342468.2342476. URL: http://doi.acm.org/10.1145/2342468.2342476.

[144] 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*. CSWS '14. Chicago, Illinois, USA: ACM, 2014, pp. 27–32. ISBN: 978-1-4503-2991-0. DOI: 10.1145/2630088.2631977.

[145] Christoph Paasch. "Improving Multipath TCP." PhD thesis. UCL, 2014. URL: http://inl.info.ucl.ac.be/publications/improving-multipath-tcp.

[146] Christoph Paasch. "Network support for TCP Fast Open." In: *Presentation at NANOG* 67 (2016).

[147] Christoph Paasch. *[multipathtcp] Multipath TCP Address advertisement 4/5 - Priorities.* Nov. 2016. URL: https://mailarchive.ietf.org/arch/msg/multipathtcp/ZsE8IXqPBhnLkOTttqglHRNI8tk (visited on 06/17/2019).

[148] Christoph Paasch. *[multipathtcp] Regarding rate control at a subflow level.* May 2019. URL: https://mailarchive.ietf.org/arch/msg/multipathtcp/fyhIpWnXCr0sImRECGh8Lx1zWdc (visited on 06/17/2019).

[149] Christoph Paasch, Sebastien Barre, et al. "Multipath TCP in the Linux Kernel." available from http://www.multipath-tcp.org.

[150] Christoph Paasch, Ramin Khalili, and Olivier Bonaventure. "On the Benefits of Applying Experimental Design to Improve Multipath TCP." In: *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '13. Santa Barbara, California, USA: ACM, 2013, pp. 393–398. ISBN: 978-1-4503-2101-3. DOI: 10.1145/2535372.2535403. URL: http://inl.info.ucl.ac.be/publications/benefits-applying-experimental-design-improve-multipath-tcp.

[151] Christoph Paasch, Mat Martineau, Peter Krystad, and Matthieu Baerts. "How hard can it be? Adding Multipath TCP to the upstream kernel." In: *Netdev*. Nov. 2018.

[152] Christoph Paasch et al. *WWDC: Advances in Networking, Part 1*. June 2019. URL: https://developer.apple.com/videos/play/wwdc2019/712/.

[153] Giorgos Papastergiou et al. "De-ossifying the Internet transport layer: A survey and future perspectives." In: *IEEE Communications Surveys & Tutorials* 19.1 (2016), pp. 619–639.

[154] Parveen Patel et al. "Upgrading transport protocols using untrusted mobile code." In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 1–14.

[155] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. *Known TCP Implementation Problems*. RFC 2525 (Informational). RFC. Fremont, CA, USA: RFC Editor, Mar. 1999. DOI: 10.17487/RFC2525. URL: https://www.rfc-editor.org/rfc/rfc2525.txt.

[156] V. Paxson, M. Allman, J. Chu, and M. Sargent. *Computing TCP's Retransmission Timer*. RFC 6298 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2011. DOI: 10.17487/RFC6298. URL: https://www.rfc-editor.org/rfc/rfc6298.txt.

[157] Q. Peng, A. Walid, J. Hwang, and S.H. Low. "Multipath TCP: Analysis, Design, and Implementation." In: *Networking, IEEE/ACM Transactions on* PP.99 (2015), pp. 1–1. ISSN: 1063-6692. DOI: 10.1109/TNET.2014.2379698.

[158] C. Perkins (Ed.), D. Johnson, and J. Arkko. *Mobility Support in IPv6*. RFC 6275 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, July 2011. DOI: 10.17487/RFC6275. URL: https://www.rfc-editor.org/rfc/rfc6275.txt.

[159] Miguel Peón-Quirós et al. "Results from Running an Experiment as a Service Platform for Mobile Networks." en. In: *Proceedings of the 11th Workshop on Wireless Network Testbeds, Experimental evaluation and Characterization*. ACM, 2017, pp. 9–16. ISBN: 978-1-4503-5147-8. DOI: 10.1145/3131473.3131485. (Visited on 11/09/2017).

[160] Christopher Pluntke, Lars Eggert, and Niko Kiukkonen. "Saving Mobile Device Energy with Multipath TCP." In: *Proceedings of the Sixth International Workshop on MobiArch*. MobiArch '11. Bethesda, Maryland, USA: ACM, 2011, pp. 1–6. ISBN: 978-1-4503-0740-6. DOI: 10.1145/1999916.1999918. URL: http://doi.acm.org/10.1145/1999916.1999918.

[161] J. Postel. *Transmission Control Protocol*. RFC 793. RFC. Fremont, CA, USA: RFC Editor, Sept. 1981. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/rfc/rfc793.txt.

[162] IO Visor Project. *Userspace eBPF VM*. 2018. URL: https://github.com/iovisor/ubpf.

[163] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. "LKL: The Linux kernel library." In: *Roedunet International Conference (RoEduNet), 2010 9th*. 2010, pp. 328–333.

[164] Octavian Purdila et al. "LKL: The Linux kernel library." In: *Roedunet International Conference (RoEduNet), 2010*. 2010, pp. 328–333.

[165] Bruno Quoitin, Cristel Pelsser, Louis Swinnen, Olivier Bonaventure, and Steve Uhlig. "Interdomain traffic engineering with BGP." In: *IEEE Communications magazine* 41.5 (2003), pp. 122–128.

[166] Sivasankar Radhakrishnan et al. "TCP Fast Open." In: *Proceedings of the Seventh CONEXT*. ACM. 2011, p. 21.

[167] C. Raiciu, M. Handley, and D. Wischik. *Coupled Congestion Control for Multipath Transport Protocols*. RFC 6356. Internet Engineering Task Force, 2011.

[168] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. "Improving Datacenter Performance and Robustness with Multipath TCP." In: *ACM SIGCOMM 2011*. Toronto, Ontario, Canada, 2011. ISBN: 978-1-4503-0797-0. DOI: 10.1145/2018436.2018467. URL: http://doi.acm.org/10.1145/2018436.2018467.

[169] Costin Raiciu, Dragos Niculescu, Marcelo Bagnulo, and Mark James Handley. "Opportunistic Mobility with Multipath TCP." In: *Proceedings of the Sixth International Workshop on MobiArch*. MobiArch '11. Bethesda, Maryland, USA: ACM, 2011, pp. 7–12. ISBN: 978-1-4503-0740-6. DOI: 10.1145/1999916.1999919. URL: http://doi.acm.org/10.1145/1999916.1999919.

[170] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark 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*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 29–29. URL: http://inl.info.ucl.ac.be/publications/how-hard-can-it-be-designing-and-implementing-deployable-multipath-tcp.

[171] Costin Raiciu et al. "How hard can it be? Designing and implementing a deployable multipath TCP." In: *Proceedings of the 9th USENIX NSDI*. 2012, pp. 29–29.

[172] Philipp Richter et al. "A multi-perspective analysis of carrier-grade NAT deployment." In: *Proceedings of the 2016 Internet Measurement Conference*. ACM. 2016, pp. 215–229.

[173] Jan Rüth, Christian Bormann, and Oliver Hohlfeld. "Large-scale scanning of TCP's initial window." In: *Proceedings of the IMC 2017*. London, United Kingdom: ACM Press, 2017, pp. 304–310. ISBN: 978-1-4503-5118-8. DOI: 10.1145/3131365.3131370. (Visited on 11/25/2018).

[174] Jan Rüth and Oliver Hohlfeld. "Demystifying TCP Initial Window Configurations of Content Distribution Networks." In: *2018 TMA Conference*. IEEE. 2018, pp. 1–8.

[175] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. *Linux Netlink as an IP Services Protocol*. RFC 3549 (Informational). RFC. Fremont, CA, USA: RFC Editor, July 2003. DOI: 10.17487/RFC3549. URL: https://www.rfc-editor.org/rfc/rfc3549.txt.

[176] Jerome H Saltzer, David P Reed, and David D Clark. "End-to-end arguments in system design." In: *Technology* 100 (1984), p. 0661.

[177] Golam Sarwar et al. "Mitigating receiver's buffer blocking by delay aware packet scheduling in multipath data transfer." In: *2013 27th International Conference on Advanced Information Networking and Applications Workshops*. IEEE. 2013, pp. 1119–1124.

[178] Gregory Vander Schueren et al. "TCPSnitch: Dissecting the Usage of the Socket API." In: *preprint arXiv:1711.00674* (2017).

[179] Justine Sherry, Sylvia Ratnasamy, and Justine Sherry At. "A Survey of Enterprise Middlebox Deployments." In: (2012).

[180] Hang Shi et al. "{STMS}: Improving {MPTCP} Throughput Under Heterogeneous Networks." In: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 2018, pp. 719–730.

[181] Tanya Shreedhar, Nitinder Mohan, Sanjit K Kaul, and Jussi Kangasharju. "QAware: A Cross-Layer Approach to MPTCP Scheduling." In: *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE. 2018, pp. 1–9.

[182] J. Solomon. *Applicability Statement for IP Mobility Support*. RFC 2005 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Oct. 1996. DOI: 10.17487/RFC2005. URL: https://www.rfc-editor.org/rfc/rfc2005.txt.

[183] Dug Song. *Dsniff*. 2000.

[184] Fei Song, Hong-Ke Zhang, Anthony Chan, and Anni Wei. *One Way Latency Considerations for MPTCP*. Internet-Draft draft-song-mptcp-owl-06. Work in Progress. Internet Engineering Task Force, June 2019. 10 pp.

[185] Alexei Starovoitov. *bpf: bounded loops and other features*. 2019. URL: https://patchwork.ozlabs.org/cover/1116447/.

[186] Alexei Starovoitov. *bpf: improve verifier scalability*. 2019. URL: https://patchwork.ozlabs.org/cover/1073775/.

[187] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. *Stream Control Transmission Protocol*. RFC 2960 (Proposed Standard). RFC. Obsoleted by RFC 4960, updated by RFC 3309. Fremont, CA, USA: RFC Editor, Oct. 2000. DOI: 10.17487/RFC2960. URL: https://www.rfc-editor.org/rfc/rfc2960.txt.

[188] *Talk To Me: The Present & Future of In-Car Speech Recognition | Globalme. Link: www.globalme.net/blog/the-present-and-future-of-in-car-speech-recognition*. URL: https://www.globalme.net/blog/the-present-and-future-of-in-car-speech-recognition (visited on 02/25/2018).

[189] J. Touch. *Shared Use of Experimental TCP Options*. RFC 6994 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Aug. 2013. DOI: 10.17487/RFC6994. URL: https://www.rfc-editor.org/rfc/rfc6994.txt.

[190] Brian Trammell et al. "Tracking transport-layer evolution with pathspider." In: *Proceedings of the Applied Networking Research Workshop*. ACM. 2017, pp. 20–26.

[191] Viet-Hoang Tran and Olivier Bonaventure. *Poster: Multipath TCP in action: A view from the server side*. TMA 2016. 2016.

[192] Viet-Hoang Tran and Olivier Bonaventure. *Poster: Towards crowd-based MPTCP measurements*. IMC 2017. 2017.

[193] Viet-Hoang Tran and Olivier Bonaventure. "Beyond socket options: making the Linux TCP stack truly extensible." In: *The IFIP Networking 2019 Conference*. IFIP. IFIP Digital Library, May 2019.

[194] Viet-Hoang Tran and Olivier Bonaventure. *Making the Linux TCP stack more extensible with eBPF*. Netdev 0x13. 2019.

[195] Viet-Hoang Tran and Olivier Bonaventure. *Multipath TCP Inactivity Time Option*. Internet-Draft draft-hoang-mptcp-inactivity-time-00. Work in Progress. Internet Engineering Task Force, July 2019. 8 pp.

[196] Viet-Hoang Tran and Olivier Bonaventure. *Multipath TCP Subflow Rate Limit Option*. Internet-Draft draft-hoang-mptcp-sub-rate-limit-00. Work in Progress. Internet Engineering Task Force, July 2019. 6 pp.

[197] Viet-Hoang Tran, Ramin Sadre, and Olivier Bonaventure. "Measuring and Modeling Multipath TCP." In: *IFIP International Conference on Autonomous Infrastructure, Management and Security*. Springer. 2015, pp. 66–70.

[198] Viet-Hoang Tran, Quentin De Coninck, Benjamin Hesmans, Ramin Sadre, and Olivier Bonaventure. "Observing real Multipath TCP traffic." In: *Computer Communications* 94 (2016), pp. 114–122.

[199] Viet-Hoang Tran, Hajime Tazaki, Quentin De Coninck, and Olivier Bonaventure. "Voice-activated applications and Multipath TCP: A good match?" In: *MNM Workshop (TMA)*. IEEE. 2018, pp. 1–6.

[200] Martino Trevisan et al. "Five years at the edge: watching internet from the ISP network." In: *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. ACM. 2018, pp. 1–12.

[201] Cheng-Chun Tu, Joe Stringer, and Justin Pettit. "Building an extensible Open vSwitch datapath." In: *ACM SIGOPS Operating Systems Review* 51.1 (2017), pp. 72–77.

[202] Mauricio Vasquez. *Implement queue/stack maps*. 2018. URL: https://patchwork.ozlabs.org/cover/985889/.

[203] Tobias Viernickel et al. "Multipath QUIC: A deployable multipath transport protocol." In: *2018 IEEE International Conference on Communications (ICC)*. IEEE. 2018, pp. 1–7.

[204] Amerigo Cong Wang. *Linux Kernel: tcp: introduce a per-route knob for quick ack*. 2013.

[205] Xiao Sophia Wang. *Eploud*. Dec. 2018. URL: http://wprof.cs.washington.edu/spdy/tool/.

[206] Xiao Sophia Wang et al. "How Speedy is SPDY?" In: *11th USENIX NSDI*. Seattle, WA: USENIX Association, 2014, pp. 387–399. ISBN: 978-1-931971-09-6.

[207] Nicholas Weaver, Christian Kreibich, Martin Dam, and Vern Paxson. "Here Be Web Proxies." In: *Proceedings of the 15th PAM - Volume 8362*. PAM 2014. Los Angeles, CA, USA: Springer-Verlag New York, Inc., 2014, pp. 183–192. ISBN: 978-3-319-04917-5. DOI: 10.1007/978-3-319-04918-2_18.

[208] Yuchung Cheng Wei Wang Neal Cardwell and Eric Dumazet. *IETF draft: TCP Low Latency Option*. Prague, 2017. URL: https://tools.ietf.org/html/draft-wang-tcpm-low-latency-opt-00.

[209] N. Williams, P. Abeysekera, N. Dyer, H. Vu, and G. Armitage. *Multipath TCP in Vehicular to Infrastructure Communications*. Tech. rep. 140828A. CAIA, Swinburne University of Technology, 2014. URL: http://caia.swin.edu.au/reports/140828A/CAIA-TR-140828A.pdf.

[210] D. Wing and A. Yourtchenko. "Happy Eyeballs: Success with Dual-Stack Hosts." RFC 6555. 2012. URL: http://www.rfc-editor.org/rfc/rfc6555.txt.

[211] Damon Wischik, Mark Handley, and Marcelo Bagnulo Braun. "The Resource Pooling Principle." In: *SIGCOMM Comput. Commun. Rev.* 38.5 (Sept. 2008), pp. 47–52. ISSN: 0146-4833. DOI: 10.1145/1452335.1452342. URL: http://doi.acm.org/10.1145/1452335.1452342.

[212] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. "Design, Implementation and Evaluation of Congestion Control for Multipath TCP." In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 99–112. URL: http://dl.acm.org/citation.cfm?id=1972457.1972468.

[213] Florian Wohlfart et al. "Leveraging interconnections for performance: the serving infrastructure of a large cdn." In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM. 2018, pp. 206–220.

[214] Gary R Wright and W Richard Stevens. *TCP/IP illustrated, volume 2: The implementation*. Addison-Wesley Professional, 1995.

[215] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. "Leveraging eBPF for programmable network functions with IPv6 Segment Routing." In: *Proceedings of the 14th CONEXT*. ACM. 2018.

[216] Xing Xu et al. "Investigating transparent web proxies in cellular networks." In: *International Conference on Passive and Active Network Measurement*. Springer. 2015, pp. 262–276.

[217] Yusuke Yamada et al. "Development and Evaluation of Julius-Compatible Interface for Kaldi ASR." In: *International Conference on Intelligent Information Hiding and Multimedia Signal Processing*. Springer. 2017, pp. 91–96.

[218] Fan Yang, Qi Wang, and Paul D Amer. "Out-of-order transmission for in-order arrival scheduling for multipath TCP." In: *2014 28th International Conference on Advanced Information Networking and Applications Workshops*. IEEE. 2014, pp. 749–752.

[219] Jing Zuo, Jianjian Zhu, and Wei Liu. *A Path-Aware Scheduling Scheme for Multipath Transport Protocols*. Internet-Draft draft-zuo-mptcp-scheduler-01. Work in Progress. Internet Engineering Task Force, Mar. 2018. 10 pp.