Beyond socket options: Towards fully extensible Linux transport stacks

Viet-Hoang Tran, Olivier Bonaventure
hoang.tran@uclouvain.be,olivier.bonaventure@uclouvain.be

ICTEAM, Université catholique de Louvain
Louvain-la-Neuve, Belgium

Abstract

The Transmission Control Protocol (TCP) is 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 a TCP option that identifies this extension. In practice, deploying a TCP extension is 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 if not a decade to actually deploy a TCP option widely. 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. In this paper, 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. For this purpose, we leverage the extended Berkeley Packet Filter (eBPF), which is a recently developed in-kernel infrastructure to enable high performance and safe programmability to the Linux kernel space.

Multipath TCP (MPTCP) is a major TCP extension that enables more capabilities and has richer semantics than regular TCP. We implemented a similar methodology in the Linux MPTCP stack to support new use-cases through custom MPTCP options. Moreover, an eBPF-based framework for user-defined path managers is also proposed, given that subflow management is an important task in Multipath TCP.

Keywords: transport layer, TCP, multipath, eBPF, extensible

1. Introduction

Transmission Control Protocol (TCP) [1] remains one of the core protocols in today's Internet. 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 [2], Selective Acknowledgements [3], TCP Fast Open [4], Multipath TCP [5] 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 [6] and the Timestamp option is still not enabled by the Microsoft stacks [7]. More recently, middlebox interference became an important concern [8, 9] 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 [10]. When Van Jacobson wrote his seminal paper on congestion avoidance and control [11], 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 [12], Multipath TCP [13] and others. The TCP stack in Linux 1.0 in 1994 contained 3k lines. It grew to 18k lines in v2.6 (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 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.

For Multipath TCP, the ability to use multiple paths enables even more use cases to extend the protocol than regular TCP [14]. Moreover, Multipath TCP has richer semantics than regular TCP [5, 13]. For example, path management and packet scheduling are two new tasks that did not exist with legacy TCP.

In short, the main contributions of this paper are as follows:

1. We propose and implement a lightweight eBPF-based framework that enables users to easily add support for new TCP options in the Linux TCP stack.
2. We propose and implement a similar framework that enables custom MPTCP options in the Linux MPTCP stack.
3. Following a similar approach, we implement a framework to support user-defined path managers in the Linux MPTCP stack.
4. For each framework, four use cases are discussed and implemented to illustrate the usage of these frameworks in practice.

The remainder of this paper is organized as follows: Section 2 discusses the related work and gives an overview of the eBPF infrastructure in the Linux kernel. The Multipath TCP protocol and its implementation in the Linux kernel are depicted in Sections 4 and 5. We present the design and implementation of our TCP option framework in Section 3, the MPTCP option framework in Section 6, and the MPTCP path manager framework in Section 7. In each of these sections, four use cases are presented to illustrate how these frameworks could be used. Section 8 discusses the insights and future work. Finally, Section 9 concludes the paper and provides links to the artefacts of our work.

2. State of the art

Transport protocols such as TCP can be implemented inside the operating systems' kernel [10] or as an application library. 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 [15]. 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 [16].

2.1. In-kernel approaches

Several researchers have proposed solutions to simplify the extension of in-kernel implementations. STP [17] was an initial 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. However, there was no follow-up work of STP to deal with the main challenge - that is to ensure both safety and performance of the sandbox mechanism.

[1] See `https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt`

[2] See `https://man7.org/linux/man-pages/man7/socket.7.html`

The idea of exposing and allowing applications to set internal state variables of TCP connections was proposed earlier [18, 19]. This permits the control plane of TCP congestion control to be moved from kernel to userland [18, 20]. This also enables adding new non-intrusive features [18], 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.

2.2. Userland approaches

Besides kernel stacks, there are complete userspace TCP stacks [16, 21, 22]. Their nature makes them be easier to be modified by application developers than 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 [23] 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%) [23] or rate-limit UDP traffic.

Linux Kernel Library (LKL)[24] 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 applications to use new features (e.g. TCP Fast Open, MPTCP) even if updating the host kernel is not possible or not desirable[25]. However, it currently induces some memory overhead and the dynamicity of the network stack has not yet been considered.

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 [26]. Second, `Netlink` [27] establishes channels between kernel space and user space. It has been used to support user-level control plane for MPTCP path manager [28]. However, this approach requires the addition of a lot of code into both kernel and userspace, causing both memory and processing overhead.

One 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 the 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 [17]. However, there was no follow-up work since the original article. 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.

2.4. 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 of popular tools such as `tcpdump` or `Wireshark`. Since the BPF filter runs in the kernel, 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 packets (e.g. only capture TCP packets 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 massively extended and renamed the extended BPF (eBPF). It supports several use cases such as sandboxing system calls (`seccomp`), tracing kernel events [29], implement hyperupcalls [30]. Several networking use cases already leverage eBPF. For example, XDP uses it for fast packet processing [31], IPv6 Segment Routing uses it to support network programming [32] and it improves the extensibility of Open vSwitch [33]. Motivated by its success in the Linux

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

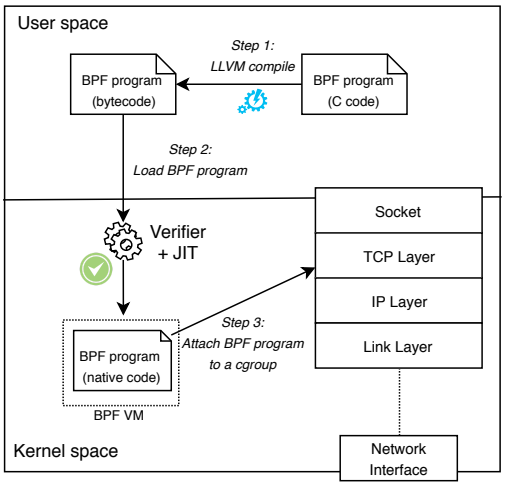


Figure 1: A user application compiles and injects eBPF program into kernel

kernel, there are ongoing works to port eBPF to userspace [34] or to the FreeBSD kernel [35].

2.4.1. 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. 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 Linux version 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 can interact with only one or a subset of the 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.

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

2.4.3. 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.

2.4.4. 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.

2.5. TCP-BPF framework

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 [36] 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.

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 [37, 38, 39] into mainstream kernel in versions 4.13 through 4.15. It was mainly designed to help network administrators 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 [36] adds several callbacks (also called hooks by the authors) to call BPF programs at different stages of a TCP connection, as shown in Table 1. In this paper, 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 application. However, this is not a permanent requirement, rather it should be considered as an implementation caveat which can be changed later.

3. Supporting user-defined TCP options

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 [40], 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. [13, 12]). However, users are

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

forced to recompile their kernels with those patches to support the proposed extension. This severely limits their deployment.

3.1. Why supporting custom TCP options in Linux kernel is hard

Our goal is to support Linux users to deploy new TCP options dynamically and efficiently. This includes inserting new TCP options at the sender side and parsing them at the receiver side, along with the corresponding logic implemented by users. This is difficult for several reasons. The first reason is the complexity of the Linux TCP stack. As a part of the Linux kernel, it was implemented to support a large number of use cases and to handle various corner cases correctly. Second, Linux TCP stack is also highly optimized for performance, e.g. TCP Segmentation Offload (TSO) feature complicates the TCP option insertion process as we discuss in 3.2.1. Third, since the stack is in the kernel space, a small mistake could cause a security hole or corrupt the whole system. Therefore, every change in the Linux TCP stack requires careful considerations. Fourth, until recently, there lacks proper infrastructure in the Linux kernel to support general programmability.

3.2. Methodology

Our approach is to leverage as much as possible the eBPF execution environment which is available on recent Linux kernels. Thanks to eBPF, any application can inject code inside the underlying TCP stack to modify its behaviour (as shown in Fig. 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

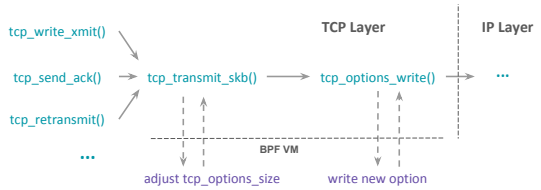


Figure 2: Insert TCP options to outgoing packets

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.

3.2.1. 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 2 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. 2.

We first add into the function `tcp_transmit_skb()` a new 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

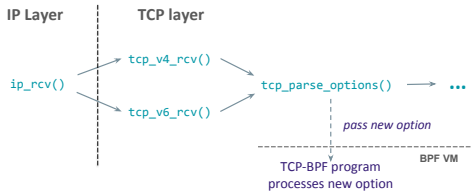


Figure 3: Pass unknown TCP options of incoming packets to BPF program

current option pointer. To reduce 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 the to-be-added option into consideration. This is performed by a hook with the same `op` type as the above hook (which adjusts `tcp_options_size`) that is added to `tcp_current_mss()` and thus is completely transparent to the BPF programs.

3.2.2. At the receiver: Parsing new 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 add a hook to pass these unknown options to the BPF program, as shown in Fig. 3. This hook, once activated, passes 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()`.

3.2.3. 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 (IP addresses and ports) to only influence the interesting connections. These operations are already supported by the vanilla kernel so no kernel change is required.

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.

3.2.4. 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 3 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 [41] proposed by Ben Greear and Daniel Baluta, is reasonably larger.

3.2.5. 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 [42] test between two servers over a 10 Gbps link. Each server 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

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 2: New BPF hooks added by TCP option framework

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

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 apply actions using two helper functions `bpf_setsockopt()` and `bpf_getsockopt()`

Each measurement lasts 40 seconds and each scenario is repeated 20 times. Figure 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 Figures 4b and 4c.

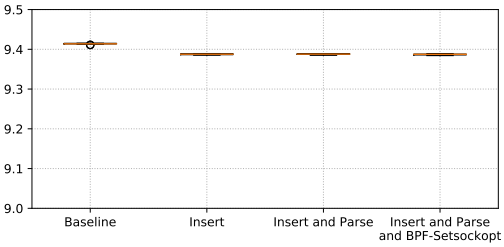
3.3. Use Cases

In this section, we demonstrate with a variety of use cases how it is possible to leverage BPF programs to extend the Linux TCP stack. We start in Section 3.3.1 with the TCP User Timeout Option [43] that has not been implemented in the Linux TCP stack. We then propose and implement in Section 3.3.2 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 3.3.3 and discuss how eBPF code can be used to tune the acknowledgement strategy in Section 3.3.4.

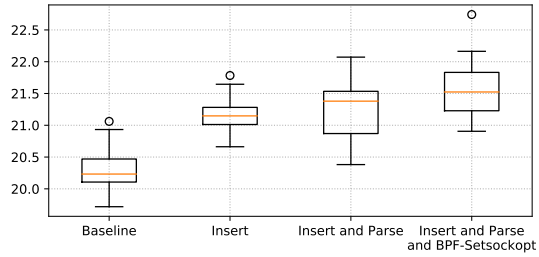
3.3.1. TCP User Timeout Option

The TCP User Timeout (UTO) option [43] 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 a longer UTO value. 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.

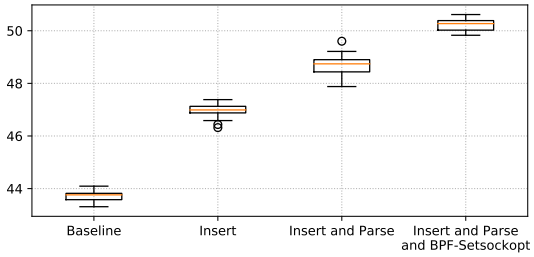
The UTO option [43] carries the suggested timeout value. It is sent unreliably, typically inside a TCP ACK. In contrast with most TCP extensions, the utilisation of this option is not negotiated during the three-way handshake. It is simply used once the connection has been established. Linux allows applications 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



(a) Average Throughput (Gbps)



(b) Sender's CPU Usage (%)



(c) Receiver's CPU Usage (%)

Figure 4: Benchmarking results: iPerf3 test over a 10Gbps link

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.

3.3.2. TCP Congestion Control Option

The Linux TCP stack supports a dozen of pluggable congestion control modules [44]. Depending on its configuration, a Linux host may directly support two to three TCP congestion control schemes, e.g. NewReno [45], CUBIC [46], or Vegas [47] or BBR [48]. Content Distribution Networks (CDN) often tune their congestion control scheme to better serve their customers [49]. 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 environ-

ment similar[4] to Mininet [50]. We set up separate network namespaces for client and server, a Linux bridge in-between, and use 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 [45], CUBIC [46], Vegas [47] and BBR [48] 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 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.

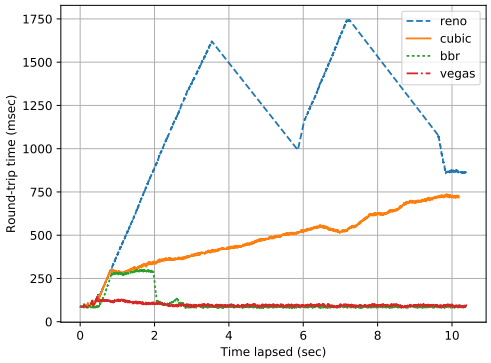


Figure 5: Congestion Control Option test: RTT on the server (8 Mbps bandwidth, 40 ms link delay)

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 [23] 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.

[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.

3.3.3. 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 [51] and later 10 MSS[52, 53] 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 [54, 55] show that while most web servers use the default values of their TCP stacks, CDN operators usually apply much larger values of IW [55]. These measurements also suggest that some CDNs customize their IW configuration based on the network and/or content type.

Brakmo suggested [36] to heuristically select the IW based on the IP prefix using TCP-BPF, with a simple example [56]. 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. [57] with the `epload` software [58]. 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 3.3.2. 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 6 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 increases. 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, RFC4782 [59] has proposed an IP option for signaling the optimal initial sending rate. However, IP options have more risk to be filtered than the TCP options.

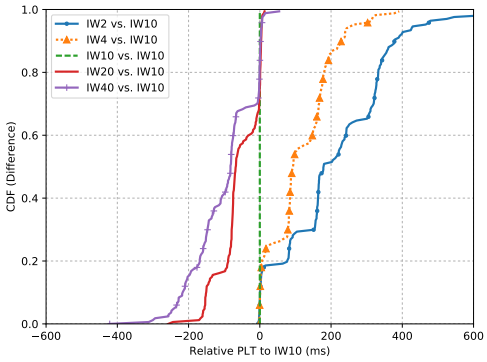


Figure 6: Initial Window Option test: Page Load Time relatively to IW=10 (40 Mbps bandwidth, 40 ms link delay)

3.3.4. Tuning the acknowledgement strategy

As a reliable protocol, TCP crucially relies on 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 [60]. 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) [61] 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 local connections, but too small for intercontinental connections. The only customization supported by Linux is to disable the delayed ACK mechanism for each route [62]. 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 Retransmission Time Out (RTO) calculation is based on smoothed RTT (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 at least 40 ms in Linux. While Linux tries to guess delayed ACK to exclude from RTT sampling, there is no reliable way to do this.

Meanwhile, modern networking stacks have adopted ACK-stretching techniques. 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 [63, 64]. Furthermore, the Recent ACK (RACK) [65] (subsumed Tail-Loss Probe (TLP) [66]) mechanism which is being standardized and deployed in Linux and Windows [67]. 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 [68] to negotiate a custom delayed ACK timeout during the three-way handshake. However, as discussed during the IETF99 TCPM WG meeting[69], 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.

4. Multipath TCP Protocol

In this section, we present a summary of Multipath TCP. Detailed explanations of the protocol can be found in [70, 5]. The second version of Multipath TCP [71] improves the first version [5] with reliable connection setup and various security enhancements. In Section 4.5, we discuss main use cases of using Multipath TCP. Subflow management (in Section 4.6) and packet scheduling (in Section 4.7) are two new tasks that did not exist with legacy TCP.

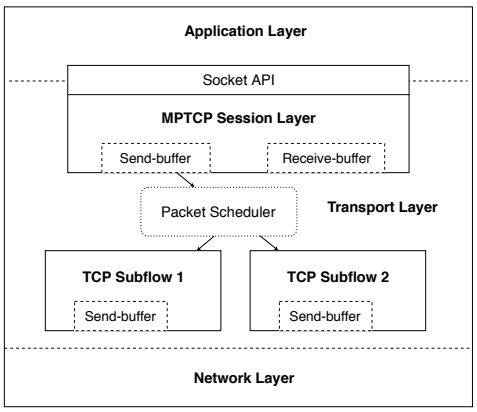


Figure 7: MPTCP Protocol Stack

The conceptual view of the Multipath TCP protocol stack is depicted in Fig. 7. Multipath TCP enables hosts to exchange the packets that belong to one connection over different interfaces or paths. To deal with middleboxes, it is necessary to establish a separate TCP connection (called *subflow*) along each path so that packets could be sent on each path reliably. 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. All operations of Multipath TCP use TCP options for signaling[5]. 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.

4.1. 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. MPTCP

uses *Data Sequence Signal (DSS)* option mainly to specify explicit mapping between this DSN and SSN. To know 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.

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. 7). This design decision was made to avoid a deadlock scenario [70].

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 closes the affected subflow. If this is the only subflow of the connection, MPTCP falls back to regular TCP so that the middlebox could change the payload without breaking the connection.

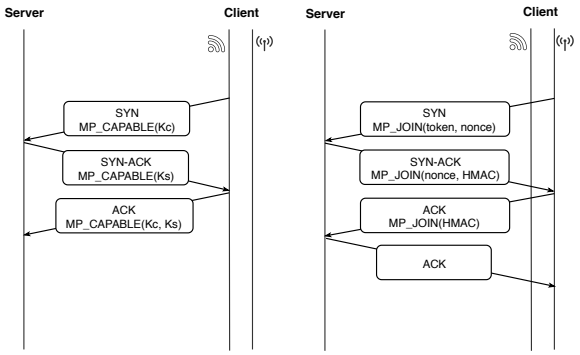
4.2. Initial Subflow setup

The MPTCP connection establishment relies on the regular TCP connection handshake. As shown in Fig. 8a, 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.

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 expires.

4.3. Additional Subflow setup

After the initial subflow established and some data has been exchanged, additional subflows can 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



(a) Initial subflow (also MPTCP connection)

(b) Additional subflow

Figure 8: MPTCP connection/subflow establishments

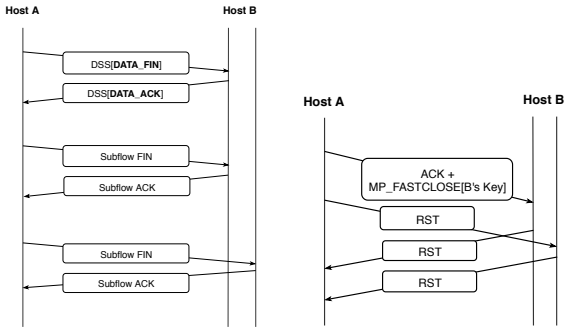
the `MP_JOIN` options. However, different from the connection setup, the subflow establishment is only done after a four-way handshake, as shown in Fig. 8b. The first SYN `MP_JOIN` option carries the 32-bit connection token, which is inferred from the connection key, which the receiver could use to quickly look up the connection. Both sides send a nonce challenge, requiring each other peer to authenticate itself with the corresponding HMAC. The option also contains the ID of the local IP address and optional flags (not shown in the figure).

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.

4.4. 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



(a) Graceful Close (b) Abrupt Close

Figure 9: MPTCP Connection Close

peer that there is no more data to send on this MPTCP connection, as illustrated in Fig. 9a. For simplicity, Fig. 9a 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_ACK`ed 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. 9b) and goes to the `FASTCLOSE_WAIT` state. The new version of MPTCP (defined in RFC 6824bis [71]) allows the sender to attach the option to the TCP-RST segment (which may be lost) and go directly to the CLOSED state without further delay.

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

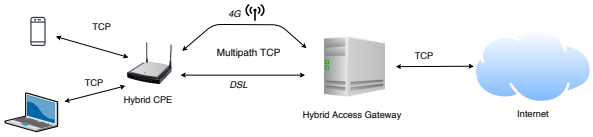


Figure 10: Hybrid access networking: One typical use case of Multipath TCP is combining cellular and xDSL paths

peers the exact reason.

4.5. Main use cases

Many use cases for Multipath TCP are proposed in the literature and are deployed commercially [72, 14]. 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 [73] or successively to support mobility [74, 75]. 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. 10. 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 combine Wi-Fi and 4G for high-end smartphones [14]. One type of this explicit proxy has been generalized for other TCP extensions and is actively standardized within the IETF TCPM working group [76].

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 [77]. 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 [78].

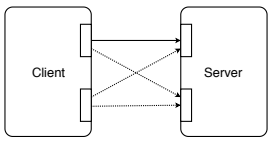


Figure 11: Fullmesh path manager tries to establish subflows along all possible paths

4.6. 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 subflow neither advertise available addresses. However, it accepts subflow requests from its peer. The `fullmesh` path manager, as its name suggested, creates all possible subflows between two hosts, forming a fully-meshed connection. Figure 11 illustrates the behavior of this path manager when each host has two interfaces. The third path manager is `ndiffports` which creates multiple subflows on the same address pair, trying to exploit the path diversity in ECMP-enabled datacenters [79]. The fourth path manager is `binder` [80] 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.

4.7. 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` [81]). 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 [82]. Pinedo et al. and Frommgen et al. [83] independently proposed and implemented redundant schedulers which duplicate data on multiple subflows to minimize the latency and increase the robustness. The DEMS [84] and RAVEN [85] schedulers selectively duplicate data to reduce the latency for interactive traffic and to probe the status of inactive subflows.

Schedulers may take a diverse range of input parameters to take the decision. The DAPS [86], OTIAS [87] and BLEST [88] schedulers use subflow-level information as input data. The ECF [89] and DEMS [84] and Cross-Layer [90] schedulers rely on application layer information, e.g. data chunk boundary and delivery deadline, to minimize the flow completion time. The MP-DASH solution [91] takes a step further and couples the operation of the MPTCP scheduler with the 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 [92].

One common problem of Multipath TCP is that the difference of RTT among subflows causes out-of-order delivery of packets. This imposes a requirement for large receive buffers. Also, 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 [81] mitigates this problem with an 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

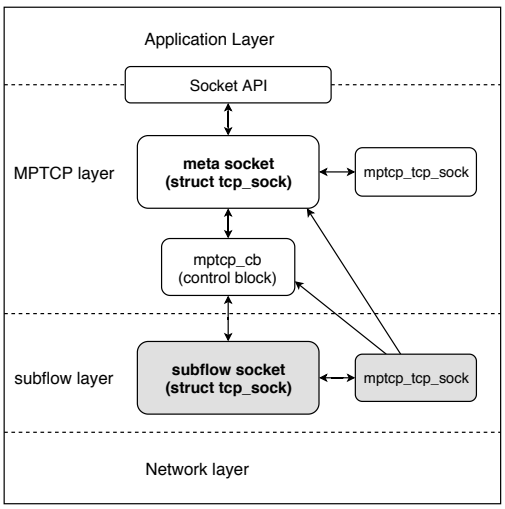


Figure 12: Main data structures of the Multipath TCP implementation in the Linux kernel

scheduling and the congestion control mechanism, causing suboptimal behaviors. It does not fully prevent sending data on the slow subflow when it hurts performance. Meanwhile, when the quality of this subflow gets better, the low congestion window prevents this subflow being used efficiently. Instead, the BLEST [88] scheduler or MPTCP-LA [93] temporarily disable a slow subflow when they predict 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 [86] and OTIAS [87]. Meanwhile, the STMS scheduler [94] 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.

5. Multipath TCP implementation in The Linux Kernel

In this section, we briefly present how Multipath TCP is implemented in the Linux kernel. More details of this implementation can be found in Chapters 5 and 6 of Christoph Paasch's thesis [95]. However, many details of the implementation have been changed since then, notably on the server side.

5.1. Data structures

Following the architectural design of the Multipath TCP protocol, its implementation in Linux is also logically divided into two sublayers (Figure 12). 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.

5.2. 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 13.

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 a host to quickly verify the uniqueness of newly created tokens.

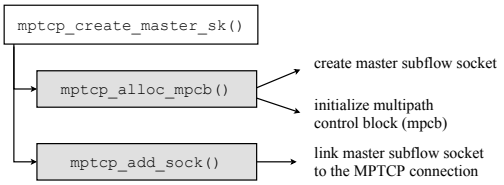


Figure 13: On both the client and server, MPTCP control block and master subflow socket is created only after 3-way handshake finished

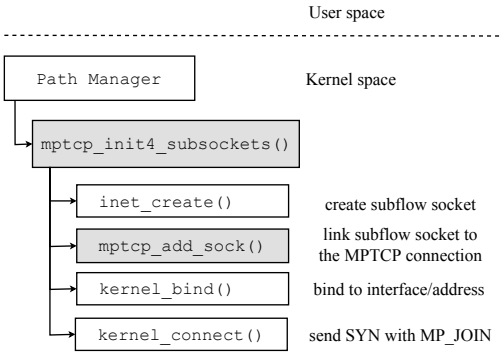


Figure 14: On the client side, a new subflow socket is created and added to the MPTCP connection before the 4-way JOIN handshake

5.3. Subflow setup

After the establishment of the initial subflow and some data has been exchanged, hosts are allowed to create additional subflows. As mentioned in Section 4.6, 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 sockets, the subflow sockets are added to the MPTCP connection by the `mptcp_add_sock()` function *before* the handshake, as shown in Figure 14. 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 15).

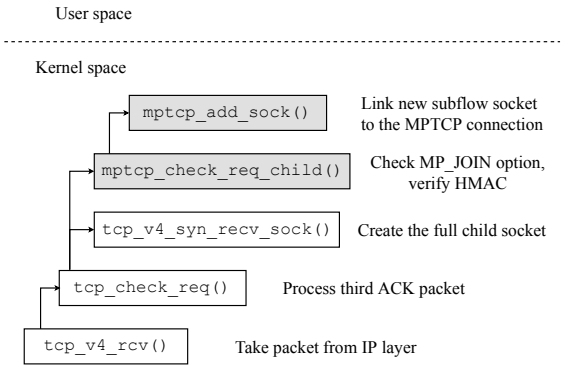


Figure 15: On the server side, a full subflow socket is created and added to the MPTCP connection after the third JOIN ACK packet arrives

5.4. Data transfer

Data sending process starts with the application passing data to the kernel stack using 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 selects 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

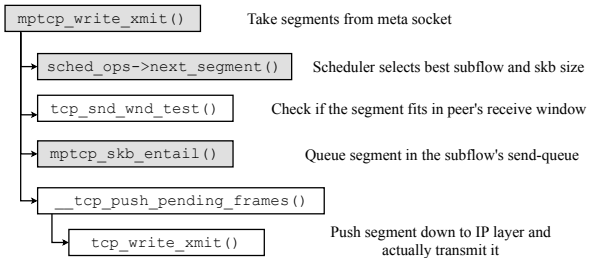


Figure 16: MPTCP sending path

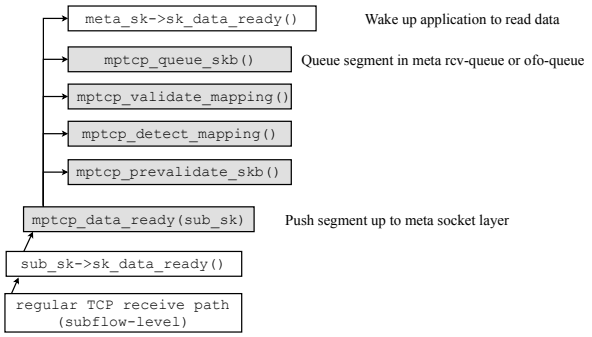


Figure 17: MPTCP receiving path

stored in the subflow out-of-order queue. When an incoming segment arrives in order, it (and out-of-order packets whose sequence numbers 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 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.

5.5. 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 4.4. 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. Extending Linux MPTCP with User-Defined Options

Similar to the TCP option framework in the previous section, we implement a generic eBPF framework to support user-defined MPTCP options. The methodology and implementation details are presented in Section 6.1. Then we illustrate the usage of user-defined MPTCP options with four different use cases in Section 6.2.

6.1. Methodology

To support user-defined MPTCP options, we 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 6.1.1 describes how we track the MPTCP subflows and Section 6.1.2 discusses how to access MPTCP-specific information from BPF programs. The implementation details of the MPTCP options handling are presented in Sections 6.1.3 and 6.1.4.

6.1.1. 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 add new callbacks with MPTCP-specific information. Since these callbacks are located in the MPTCP-specific code path, they are

not called for 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 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 arrives, 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 kernel function `mptcp_rcv_synsent_state_process()` is called to finish the subflow establishment. Therefore, we only 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.

6.1.2. Accessing MPTCP metadata

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

One issue is that the TCP-BPF callbacks 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 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 those of the `tcp_sock` structure without any data copy operation.

6.1.3. At the Sender: Inserting New MPTCP Options

To insert new MPTCP options, we could reuse our facilities for TCP (in Section 3.2). 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 18. 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.

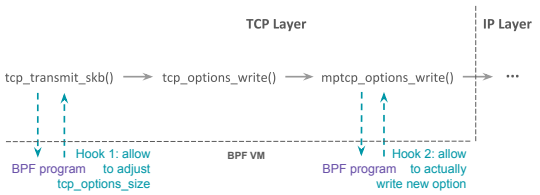


Figure 18: New hooks to insert user-defined MPTCP options

6.1.4. 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 3.2) 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 19). Similar to the option-insertion callback above, this option-parsing callback is disabled by default and activated on demand by the BPF program.

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 4: New TCP-BPF callbacks for tracking MPTCP subflows

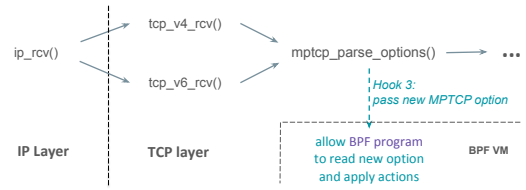


Figure 19: New hook to parse user-defined MPTCP options

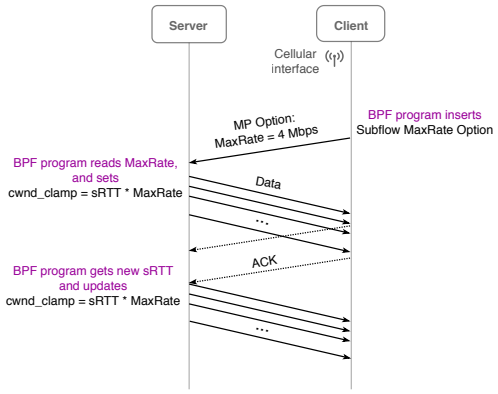


Figure 20: Usage of an MPTCP option to signal the maximum rate of the cellular subflow

6.2. Use Cases

In this part, we leverage the MPTCP option framework to implement four use cases and collect experimental results. The first option allows a client to request the server to limit the sending rate on a subflow (Section 6.2.1). The second option enables a host to request its peer to select a packet scheduler for the connection (Section 6.2.2). 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 6.2.3). Finally, a host may request its peer to set a desired timeout to remove the MPTCP connection state after the termination of the last subflow (Section 6.2.4).

6.2.1. Subflow Rate-Limit Option

Motivation: Most 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 mobile data quota. Clients could limit their upstream traffic, however, most of the traffic is sent by servers. For these applications, rate-limit should be controlled on the server side. This is more important when the mobile clients are roaming abroad where the monetary cost for cellular data is usually very high. The clients could request the servers at the application layer to limit the maximum throughput on the cellular network subflow. However, applying the mechanism at the transport layer would be more generic and could be done automatically by the system itself.

As discussed on the `multipathtcp` IETF mailing list [96], 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 would have to 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 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

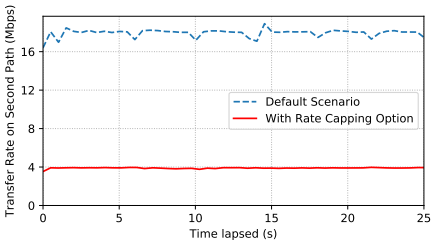


Figure 21: Throughput on the second path before and after capping at 4 Mbps

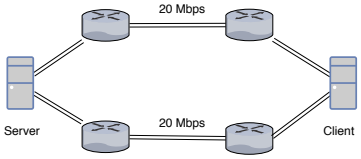


Figure 22: Topology for the MPTCP options experiments

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 14), 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 6.1.1.

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`) 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) 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 [97]. 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 3.3. The topology consists of two hosts that communicate through two paths as shown in Figure 22. The bandwidth capacity of each path is 20 Mbps and the one-way delay is 20 ms. Figure 21 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.

6.2.2. 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 [88, 82]. But normally receivers cannot control it. Most of the web traffic is in the downstream direction. It would be useful for the clients to request the scheduler algorithm on the server side. E.g. if the client wants to reduce the bufferbloat issue, an optimized scheduler like BLEST [88] 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 [83].

At the moment, the Linux MPTCP stack supports multiple packet schedulers, but it only allows the users to select the scheduler before the connection 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 3.3.2. 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 with 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 22. Each path has a capacity of 20 Mbps but the delay varies in the range of 20 ms

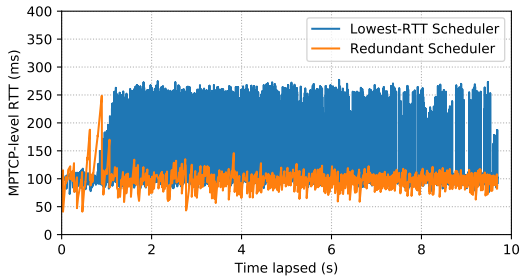


Figure 23: MPTCP-level Round-trip Time: the redundant scheduler achieves lower latency than the default one

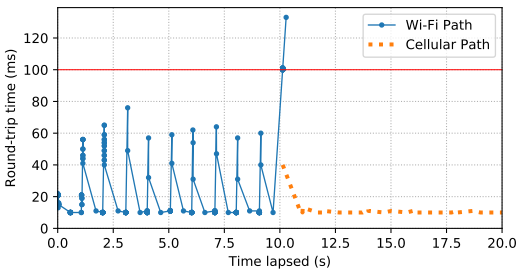


Figure 24: The server switches to the Cellular path after the experiencing high latency on the WiFi path

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 23 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 most of the time. The exception is the spike at the beginning of the connection in the case of the 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 the redundant scheduler because it sends more traffic per path than the default scheduler.

6.2.3. 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 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. [98]. While the rate-limiting option which we mentioned above focused on the heavy streams, this use case is mostly about the thin streams. The detailed mechanism is explained 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 connected to the server via two paths: Wi-Fi and cellular. The topology remains the same as in Figure 22 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. 24. Then when the network conditions on Wi-Fi become worse (RTT surpasses the 100 ms threshold), the server switches to the cellular path to satisfy the low latency requirement.

6.2.4. MPTCP Inactivity Timeout Option

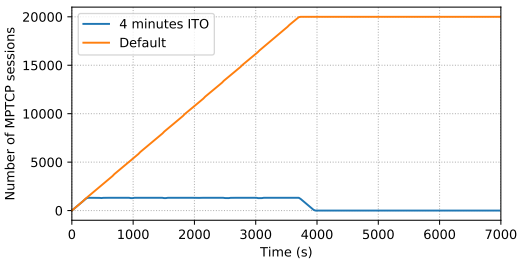
Motivation: Hosts need to maintain the state of Multipath TCP connections for some time after all established subflows have been closed, as mentioned in RFC 6824 [5]:

> "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-

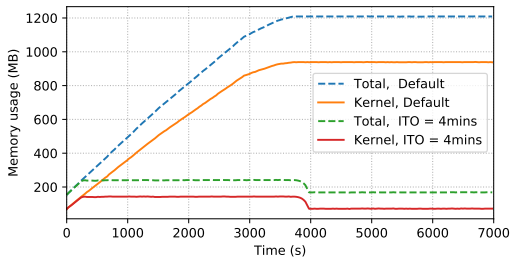
make" scenarios where connectivity is lost on all subflows before a new one can be re-established."

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`). When the idle timeout fires, the stack closes the MPTCP session and reports 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 connection will be closed shortly. For regular TCP, extending the connection lifetime by increasing the inactivity timeout on both ends is usually not sufficient 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.



(a) Number of established MPTCP sessions on the server



(b) Memory usage on the server

Figure 25: Inactivity Timeout option helps reducing significantly the server resource usage

For evaluation, we set up a local testbed in which the client and the server are two dedicated machines connected via intermediate routers. The server is equipped with 8 GB of memory and a 4-core Intel Xeon X3440 2.53 GHz CPU. In our experiments, the `curl` tool on the client initiates an MPTCP session to download a file which is served by the `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 `TCP_RST` packets but the MPTCP session still persists. The client creates 20000 of such sessions in one hour. Figure 25 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 [99] 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 both the 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.

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. [82] 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 [28]. It has been recently merged in the `mptcp-trunk` branch [101]. 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

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

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 `cgroup`s 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. Design and Implementation

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 6). Since these callbacks are inserted at the same places as the netlink-based Path Manager solution does [101], 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 1), and new subflow-specific hooks (as described in Section 6.1.1). The way we pass MPTCP-specific metadata to the BPF programs has been mentioned in Section 6.1.2.

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. 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.

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 6: New TCP-BPF callbacks for generic PM framework

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 `sockaddr`s, 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). 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 mainstream 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

The *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 [79]. Due to its simplicity, we implemented *ndiffports* program using only around 20 LoCs, as shown in Listing 2. 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 is left to pass this information to the helper function.

```
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 source and destination addresses
     * will be 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 2: ndiffports path manager as a BPF program

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).

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 ISPs to deal with the exhaustion of their public IPv4 address pools [102]. Usually, idle connections are 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 [103]. However, the timeout in practice could be as low as 30 seconds - the default value on Juniper equipments [104]. 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 various traffic shaping strategies. For example, they could apply 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` networking tool suite [105]) to regularly inject TCP RST on existing TCP connections. As shown in Figure 26, 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

Several measurement works [106, 107] indicated that MPTCP hosts usually 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.

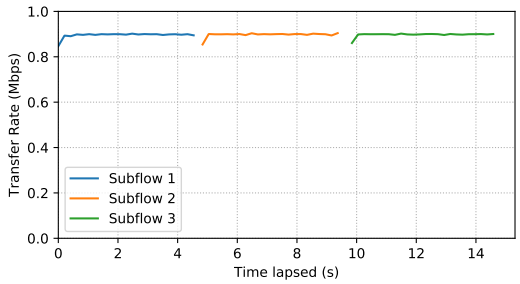


Figure 26: Subflow-Refreshing path manager opens a new subflow if the current subflow is abruptly closed

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.

7.4. Next Steps

Our framework prototype allows custom path management mechanisms to be deployed or replaced on the fly. Currently, 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 [108] 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. Discussion

TCP and MPTCP were designed to be extensible by using TCP options. However, the last decades

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

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 [40] to avoid the middlebox interference, TCP implementations such as the Linux TCP stack are monolithic and difficult to extend. In this paper, we have leveraged the eBPF virtual machine in the Linux kernel to demonstrate that it becomes possible to incrementally extend the Linux TCP and MPTCP stacks. Our work has shown that, with little changes to the kernel code, it is possible to quickly implement a range of new TCP features.

By default, we will piggyback the new TCP option onto an existing outgoing TCP packet to avoid dispatching extra packets. The main drawback of this method is the limitation of the TCP option space, which cannot be larger than 40 bytes. If there is not enough option spsace, we could wait for other outgoing TCP packet for piggybacking, or create a new TCP packet if the timeliness is important.

Currently, a technical limitation of our framework is that it now only supports TCP options of four bytes. This is because we pass the option data directly through the TCP-BPF callback's argument, whose size is four bytes. Instead, we may 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 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 [109] was added in Linux kernel since 4.20. After several efforts [110], BPF subsystem maintainers have recently implemented constrained loops support [111], supported global data [112] and extended the program size limit to one million instructions [113].

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 [114], using BTF type information and layout [115], or embedding kernel header within the running kernel image [116].

While we are preparing the camera-ready version of this paper, the Facebook network team has proposed to the netdev mailing list an eBPF-based TCP option framework [117] with a similar design to ours. The main improvement compared to ours is the support of multiple concurrent eBPF programs which handle different TCP options. These come with the expense of larger change to the kernel codebase.

For extending the out-of-tree Linux MPTCP stack, a challenge comes from the stack's design which tightly couples the multipath layer and the subflow layer. However, the mainstream Linux has recently merged another clean-slate MPTCP stack [118]. This stack was designed with a clean separation between the two layers - which should facilitate implementing major extensions like ours.

9. Conclusion

This paper has proposed three frameworks to improve the flexibility of the Linux TCP/MPTCP stack. The work should be considered as a first step to make the in-kernel networking stacks truly extensible. The results described in this paper 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 [8]. 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 [14]or in the 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.

Artefacts

The implementation of our TCP/MPTCP extension frameworks in the Linux kernel, different use cases and the experiment scripts, analysis scripts, plot scripts and datasets are publicly available at `https://github.com/hoang-tranviet/mptcp-ebpf-resources`.

Acknowledgements

This work was supported by the ARC-SDN project and the WALInnov MQUIC project. We thank Olivier Tilmans for explaining to us the eBPF infrastructure and giving useful suggestions, and thank Matthieu Baerts for reviewing the MPTCP path manager work.

References

[1] J. Postel, Transmission Control Protocol, RFC 793 (Sep. 1981). doi:10.17487/RFC0793.

[2] V. Jacobson, R. Braden, D. Borman, TCP Extensions for High Performance, RFC 1323 (Proposed Standard), obsoleted by RFC 7323 (May 1992). doi:10.17487/RFC1323.

[3] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, TCP Selective Acknowledgment Options, RFC 2018 (Proposed Standard) (Oct. 1996). doi:10.17487/RFC2018.

[4] Y. Cheng, J. Chu, S. Radhakrishnan, A. Jain, TCP Fast Open, RFC 7413 (Dec. 2014). doi:10.17487/RFC7413.

[5] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, TCP Extensions for Multipath Operation with Multiple Addresses, RFC 6824 (January 2013).

[6] K. Fukuda, An analysis of longitudinal TCP passive measurements, in: TMA Workshop, Springer, 2011, pp. 29–36.

[7] M. Honda, et al., Rekindling network protocol innovation with user-level stacks, ACM SIGCOMM Computer Communication Review 44 (2) (2014) 52–58.

[8] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, H. Tokuda, Is it still possible to extend TCP?, in: Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference, IMC '11, ACM, New York, NY, USA, 2011, pp. 181–194. doi:10.1145/2068816.2068834.

[9] K. Edeline, B. Donnet, A Bottom-Up Investigation of the Transport-Layer Ossification, in: Network Traffic Measurement and Analysis (TMA) Conference 2019, 2019.

[10] G. R. Wright, W. R. Stevens, TCP/IP illustrated, volume 2: The implementation, Addison-Wesley Professional, 1995.

[11] V. Jacobson, Congestion avoidance and control 18 (4) (1988) 314–329.

[12] S. Radhakrishnan, et al., TCP Fast Open, in: Proceedings of the Seventh CONEXT, ACM, 2011, p. 21.

[13] C. 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.

[14] O. Bonaventure, S. Seo, Multipath TCP deployments, IETF Journal 12 (2) (2016) 24–27.

[15] D. D. Clark, et al., An analysis of TCP processing overhead, IEEE Communications magazine 27 (6) (1989) 23–29.

[16] E. Jeong, et al., mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems, in: NSDI, Vol. 14, 2014, pp. 489–502.

[17] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, T. Stack, Upgrading transport protocols using untrusted mobile code, ACM SIGOPS Operating Systems Review 37 (5) (2003) 1–14.

[18] H. S. Gunawi, et al., Deploying Safe User-Level Network Services with icTCP, in: OSDI, 2004, pp. 317–332.

[19] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, J. Moore, Unveiling the transport, ACM SIGCOMM Computer Communication Review 34 (1) (2004) 99–106.

[20] A. Narayan, et al., Restructuring endpoint congestion control, in: Proceedings of the SIGCOMM 2018, ACM, 2018, pp. 30–43.

[21] A. Dunkels, Design and Implementation of the lwIP TCP/IP Stack, Swedish Institute of Computer Science 2 (2001) 77.
[22] A. Dunkels, The uIP embedded TCP/IP stack, The uIP 1.
[23] A. Langley, et al., The QUIC transport protocol: Design and Internet-scale deployment, in: Proceedings of the ACM SIGCOMM, ACM, 2017, pp. 183–196.
[24] O. Purdila, et al., LKL: The Linux kernel library, in: Roedunet International Conference (RoEduNet), 2010, 2010, pp. 328–333.
[25] V.-H. Tran, H. Tazaki, Q. De Coninck, O. Bonaventure, Voice-activated applications and multipath tcp: A good match?, in: 2018 Network Traffic Measurement and Analysis Conference (TMA), IEEE, 2018, pp. 1–6.
[26] G. V. Schueren, et al., TCPSnitch: Dissecting the Usage of the Socket API, preprint arXiv:1711.00674.
[27] J. Salim, H. Khosravi, A. Kleen, A. Kuznetsov, Linux Netlink as an IP Services Protocol, RFC 3549 (Informational) (Jul. 2003). doi:10.17487/RFC3549.
[28] B. Hesmans, et al., SMAPP: Towards smart Multipath TCP-enabled applications, in: Proceedings of the 11th ACM CONEXT, ACM, 2015, p. 28.
[29] B. Gregg, Linux Performance [Online] (2018).
[30] N. Amit, M. Wei, The design and implementation of hyperupcalls, in: 2018 USENIX ATC, 2018, pp. 97–112.
[31] T. Høiland-Jørgensen, et al., The eXpress data path: fast programmable packet processing in the operating system kernel.
[32] M. Xhonneux, F. Duchene, O. Bonaventure, Leveraging eBPF for programmable network functions with IPv6 Segment Routing, in: Proceedings of the 14th CONEXT, ACM, 2018.
[33] C.-C. Tu, J. Stringer, J. Pettit, Building an extensible Open vSwitch datapath, ACM SIGOPS Operating Systems Review 51 (1) (2017) 72–77.
[34] I. V. Project, Userspace eBPF VM (2018).
[35] Y. Hayakawa, eBPF Implementation for FreeBSD, in: BSDCan 2018, The BSD Conference, 2018.
[36] L. Brakmo, TCP-BPF: Programmatically tuning TCP behavior through BPF, NetDev 2.2.
[37] L. Brakmo, Linux Kernel patchset: bpf: BPF support for sock_ops (2017).
[38] L. Brakmo, Linux Kernel patchset: bpf: add support for BASE_RTT (2017).
[39] L. Brakmo, Linux Kernel patchset: bpf: More sock_ops callbacks (2018).
[40] J. Touch, Shared Use of Experimental TCP Options, RFC 6994 (Proposed Standard) (Aug. 2013). doi:10.17487/RFC6994.
[41] B. Greear, D. Baluta, Linux Kernel: [RFC] TCP: Support configurable delayed-ack parameters. (2012).
[42] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, K. Prabhu, iperf3, tool for active measurements of the maximum achievable bandwidth on ip networks, URL: https://github.com/esnet/iperf.
[43] L. Eggert, F. Gont, TCP User Timeout Option, RFC 5482 (Proposed Standard) (Mar. 2009). doi:10.17487/RFC5482.
[44] J. Corbet, Pluggable congestion avoidance modules, Linux Weekly News.
[45] M. Allman, V. Paxson, E. Blanton, TCP Congestion Control, RFC 5681 (Draft Standard) (Sep. 2009). doi:10.17487/RFC5681.
[46] S. Ha, I. Rhee, L. Xu, CUBIC: a new TCP-friendly high-speed TCP variant, ACM SIGOPS operating systems review 42 (5) (2008) 64–74.
[47] L. S. Brakmo, L. L. Peterson, TCP Vegas: End to end congestion avoidance on a global Internet, IEEE Journal on selected Areas in communications 13 (8) (1995) 1465–1480.
[48] N. Cardwell, et al., BBR: congestion-based congestion control, Communications of the ACM 60 (2) (2017) 58–66.
[49] E. Carlsson, E. K. S. Labs, Smoother Streaming with BBR (8 2018).
[50] B. Lantz, B. Heller, N. McKeown, A network in a laptop: rapid prototyping for software-defined networks, in: Proceedings of the 9th ACM SIGCOMM HotNets Workshop, 2010, p. 19.
[51] M. Allman, S. Floyd, C. Partridge, RFC 3390: Increasing TCP's initial window, IETF-Request for Comments.
[52] N. Dukkipati, et al., An argument for increasing TCP's initial congestion window., Computer Communication Review 40 (3) (2010) 26–33.
[53] J. Chu, N. Dukkipati, Y. Cheng, M. Mathis, Increasing TCP's Initial Window, RFC 6928 (Experimental) (Apr. 2013). doi:10.17487/RFC6928.
[54] J. Rüth, C. Bormann, O. Hohlfeld, Large-scale scanning of TCP's initial window, in: Proceedings of the IMC 2017, ACM Press, London, United Kingdom, 2017, pp. 304–310. doi:10.1145/3131365.3131370.
[55] J. Rüth, O. Hohlfeld, Demystifying TCP Initial Window Configurations of Content Distribution Networks, in: 2018 TMA Conference, IEEE, 2018, pp. 1–8.
[56] L. Brakmo, [net-next,v6,13/16] bpf: Sample bpf program to set initial cwnd. (2017).
[57] X. S. Wang, et al., How speedy is SPDY?, in: 11th USENIX NSDI, USENIX Association, Seattle, WA, 2014, pp. 387–399.
[58] X. S. Wang, Eload (12 2018).
[59] S. Floyd, M. Allman, A. Jain, P. Sarolahti, Quick-Start for TCP and IP, RFC 4782 (Experimental) (Jan. 2007). doi:10.17487/RFC4782.
[60] M. Chan, D. R. Cheriton, Improving Server Application Performance via Pure TCP ACK Receive Optimization., in: USENIX Annual Technical Conference, 2013, pp. 359–364.
[61] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, B. Volz, Known TCP Implementation Problems, RFC 2525 (Informational) (Mar. 1999). doi:10.17487/RFC2525.
[62] A. C. Wang, Linux Kernel: tcp: introduce a per-route knob for quick ack (2013).
[63] Y. Cheng, Linux Kernel: commit 9f9843a751d0. tcp: properly handle stretch acks in slow start (2013).
[64] N. Cardwell, Linux Kernel: Merge branch: fix stretch ACK bugs in TCP CUBIC and Reno (2015).
[65] Y. Cheng, N. Cardwell, N. Dukkipati, P. Jha, RACK: a time-based fast loss detection algorithm for TCP. draft-ietf-tcpm-rack-04.
[66] N. Dukkipati, et al., Tail loss probe (TLP): An algorithm for fast recovery of tail losses, draft-dukkipati-tcpm-tcploss-probe-01. txt.
[67] P. Balasubramanian, IETF96: Transports advancements in the Windows network stack, IETF, 2016.
[68] W. Wang, N. Cardwell, Y. Cheng, E. Dumazet, IETF

draft: TCP Low Latency Option (2017).
[69] IETF Minutes. https://datatracker.ietf.org/doc/minutes-99-tcpm/ (2017).
[70] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, M. Handley, How hard can it be? Designing and implementing a deployable Multipath TCP, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 29–29.
[71] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, TCP Extensions for Multipath Operation with Multiple Addresses, Internet-Draft draft-ietf-mptcp-rfc6824bis-03, IETF Secretariat, i-D Exists (Oct. 2014) [cited 10-Jan-2015].
[72] O. Bonaventure, C. Paasch, G. Detal, Experience with Multipath TCP, Internet-Draft draft-ietf-mptcp-experience-01, IETF Secretariat, i-D Exists (Mar. 2015) [cited 10-Jan-2015].
[73] D. Wischik, M. Handley, M. B. Braun, The resource pooling principle, SIGCOMM Comput. Commun. Rev. 38 (5) (2008) 47–52. doi:10.1145/1452335.1452342.
[74] C. Raiciu, D. Niculescu, M. Bagnulo, M. J. Handley, Opportunistic mobility with Multipath TCP, in: Proceedings of the Sixth International Workshop on MobiArch, MobiArch '11, ACM, New York, NY, USA, 2011, pp. 7–12. doi:10.1145/1999916.1999919.
[75] C. Paasch, G. Detal, F. Duchene, C. Raiciu, O. Bonaventure, Exploring Mobile/WiFi Handover with Multipath TCP, in: ACM SIGCOMM CellNet workshop, 2012, pp. 31–36. doi:10.1145/2342468.2342476.
[76] O. Bonaventure, M. Boucadair, S. Gundavelli, S. Seo, B. Hesmans, 0-RTT TCP Convert Protocol, Internet-Draft draft-ietf-tcpm-converters-15, Internet Engineering Task Force, work in Progress (Feb. 2020).
[77] C. Paasch, et al., WWDC: Advances in Networking, Part 1 (Jun. 2019).
[78] C. Paasch, M. Martineau, P. Krystad, M. Baerts, How hard can it be? Adding Multipath TCP to the upstream kernel, 2018.
[79] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, M. Handley, Improving Datacenter Performance and Robustness with Multipath TCP, in: ACM SIGCOMM 2011, 2011. doi:10.1145/2018436.2018467.
[80] L. Boccassi, M. M. Fayed, M. K. Marina, Binder: A system to aggregate multiple internet gateways in community networks, in: Proceedings of the 2013 ACM MobiCom Workshop on Lowest Cost Denominator Networking for Universal Access, LCDNet '13, ACM, New York, NY, USA, 2013, pp. 3–8. doi:10.1145/2502880.2502894.
[81] C. Paasch, S. Ferlin, O. Alay, O. Bonaventure, Experimental evaluation of Multipath TCP schedulers, in: Proceedings of the 2014 ACM SIGCOMM Workshop on Capacity Sharing Workshop, CSWS '14, ACM, New York, NY, USA, 2014, pp. 27–32. doi:10.1145/2630088.2631977.
[82] A. Froemmgen, et al., A Programming Model for Application-defined Multipath TCP Scheduling, in: ACM/IFIP/USNIX Middleware, 2017.
[83] A. Frommgen, T. Erbshäußer, A. Buchmann, T. Zimmermann, K. Wehrle, Remp tcp: Low latency multipath tcp, in: 2016 IEEE International Conference on Communications (ICC), IEEE, 2016, pp. 1–7.
[84] Y. E. Guo, A. Nikravesh, Z. M. Mao, F. Qian, S. Sen, 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.
[85] H. Lee, J. Flinn, B. 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.
[86] G. Sarwar, R. Boreli, E. Lochin, A. Mifdaoui, G. Smith, 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.
[87] F. Yang, Q. Wang, P. 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.
[88] S. Ferlin, O. Alay, O. Mehani, R. Boreli, BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks, in: 2016 IFIP Networking Conference (IFIP Networking) and Workshops, 2016, pp. 431–439. doi:10.1109/IFIPNetworking.2016.7497206.
[89] Y.-s. Lim, E. M. Nahum, D. Towsley, R. 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.
[90] X. Corbillon, R. Aparicio-Pardo, N. Kuhn, G. Texier, G. Simon, Cross-layer scheduler for video streaming over mptcp, in: Proceedings of the 7th International Conference on Multimedia Systems, ACM, 2016, p. 7.
[91] B. Han, F. Qian, L. Ji, V. 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.
[92] T. Shreedhar, N. Mohan, S. K. Kaul, J. Kangasharju, Qaware: A cross-layer approach to mptcp scheduling, in: 2018 IFIP Networking Conference (IFIP Networking) and Workshops, IEEE, 2018, pp. 1–9.
[93] K. Nguyen, M. Golam Kibria, K. Ishizu, F. Kojima, H. Sekiya, An approach to reinforce multipath tcp with path-aware information, Sensors 19 (3) (2019) 476.
[94] H. Shi, Y. Cui, X. Wang, Y. Hu, M. Dai, F. Wang, K. Zheng, {STMS}: Improving {MPTCP} throughput under heterogeneous networks, in: 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18), 2018, pp. 719–730.
[95] C. Paasch, Improving Multipath TCP, Ph.D. thesis, UCL (November 2014).
[96] C. Paasch, [multipathtcp] Regarding rate control at a subflow level. (May 2019).
[97] Y. Cheng, bpf: fix SO_MAX_PACING_RATE to support TCP internal pacing. (Jan. 2019).
[98] C. Paasch, [multipathtcp] Multipath TCP Address advertisement 4/5 - Priorities. (Nov. 2016).
[99] C. Paasch, G. Detal, Modified net-tools to support MPTCP. (Sep. 2019).
[100] B. Hesmans, O. Bonaventure, An enhanced socket API

for Multipath TCP, in: ANRW, ACM, 2016, pp. 1–6. doi:10.1145/2959424.2959433.
[101] M. Baerts, G. Detal, [PATCH mptcp_trunk v8 0/5] mptcp: new generic Netlink-based PM. (Jan. 2019).
[102] P. Richter, F. Wohlfart, N. Vallina-Rodriguez, M. Allman, R. Bush, A. Feldmann, C. Kreibich, N. Weaver, V. Paxson, A multi-perspective analysis of carrier-grade nat deployment, in: Proceedings of the 2016 Internet Measurement Conference, ACM, 2016, pp. 215–229.
[103] S. Guha (Ed.), K. Biswas, B. Ford, S. Sivakumar, P. Srisuresh, NAT Behavioral Requirements for TCP, RFC 5382 (Best Current Practice), updated by RFC 7857 (Oct. 2008). doi:10.17487/RFC5382.
[104] J. Networks, Carrier-Grade NAT Implementation: Best Practices - TechLibrary (May 2018).
[105] D. Song, Dsniff (2000).
[106] V.-H. Tran, Q. De Coninck, B. Hesmans, R. Sadre, O. Bonaventure, Observing real Multipath TCP traffic, Computer Communications 94 (2016) 114–122.
[107] Q. De Coninck, M. Baerts, B. Hesmans, O. Bonaventure, Observing real smartphone applications over Multipath TCP, IEEE Communications Magazine 54 (3) (2016) 88–93.
[108] X. de Foy, M. Perras, U. Chunduri, K. Nguyen, M. G. Kibria, K. Ishizu, F. Kojima, Considerations for MPTCP operation in 5G, Internet-Draft draft-defoy-mptcp-considerations-for-5g-01, Internet Engineering Task Force, work in Progress (Jun. 2018).
[109] M. Vasquez, Implement queue/stack maps (2018).
[110] J. Fastabend, BPF control flow, supporting loops and other patterns (11-2018).
[111] A. Starovoitov, bpf: bounded loops and other features (2019).
[112] D. Borkmann, Bpf support for global data (2019).
[113] A. Starovoitov, bpf: improve verifier scalability (2019).
[114] Q. Monnet, tools: bpftool: add probes for system and device (2019).
[115] A. Nakryiko, Co-re offset relocations (2019).
[116] J. Fernandes, Extending the Kernel with Built-in Kernel Headers | Linux Journal.
[117] M. K. Lau, BPF TCP header options (2020).
[118] M. Martineau, Multipath TCP: Prerequisites (Jan. 2020).