# Making the Linux TCP stack more extensible with eBPF

Viet-Hoang Tran
*ICTEAM, UCLouvain*

Olivier Bonaventure
*ICTEAM, UCLouvain*

## Abstract

This work aims to make the Linux TCP stack more extensible by leveraging Lawrence Brakmo's TCP-BPF framework. We implement new eBPF callbacks to support user-defined TCP options, and present various use cases.

The first one is to implement the TCP User Timeout Option, which has been standardized in RFC 5482 but not yet implemented in the Linux kernel.

Since RFC 6994, TCP supports experimental options. We use eBPF to implement a new experimental option that enables a client to request a server to use a specific congestion control scheme or set the initial congestion window. We also demonstrate how eBPF code can be used to tune the acknowledgment strategy.

Multipath TCP (MPTCP) is another interesting use case for eBPF with its scheduler and path manager. We present a MPTCP path-manager framework prototype which is based on eBPF and use it to implement the two well-known *ndiffports* and *fullmesh* path managers as eBPF programs.

## 1 Introduction

TCP 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 includes TCP options that can be placed in the extended TCP header. However, deploying a new TCP option takes time. It needs to be defined, accepted by the IETF and eventually implemented by the major TCP stacks. Measurements show that Selective Acknowledgements took more than a decade to be deployed [23] and the Timestamp option is still not enabled by the Microsoft stacks [28]. More recently, middlebox interference became an important concern [27].

The second factor is the extensibility of the TCP implementations. The Linux TCP stack is highly optimised for the most common use cases, but it has limited ability to *adapt* to a changing environment of network conditions, workloads or user requirements. Many TCP aspects can be tuned through a myriad of `sysctl` parameters: delayed ACK timeout, ACKing strategy, congestion control scheme, and so on. However, the `sysctl` interface only allows changing system-wide behaviors, not per-connection policies. Some of these parameters and others are exposed as socket options that can be set by applications on a per-connection basis. The socket options themself are not suitable for system-level control though.

Another interesting use case is Multipath TCP [22], which is a major extension to TCP and brings up a new dimension of required control plane. Managing the subflow paths is such a control logic which should be flexible enough to support different users and applications.

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

1. Section 2 proposes and implement a light eBPF-based framework that enables users to easily add support for new TCP options in the Linux TCP stack
2. Section 3 illustrates four use cases that leverage this framework to adapt the stack to various scenarios or user requirements.
3. We implement an eBPF-based prototype to support user-defined MPTCP path managers (Section 4). Using this facility, the *ndiffports* and *fullmesh* path managers have been implemented as eBPF programs.

Section 5 discusses the insights and future work and Section 6 provides links to the artefacts of our work.

## 2 Extending TCP in the Linux kernel

The standard method to extend TCP is to define a new TCP option. The IETF has defined a format for experimental TCP

options [34]. This format has not yet been widely used, but we leverage it in this paper to propose and implement several TCP extensions.

Currently, a TCP extension can be added to the Linux kernel as a set of patches. However, users are forced to recompile their kernels with those patches to support the proposed extension. This severely limits their deployment. A better approach is to leverage as much as possible the eBPF virtual machine. 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.

In 2017, Lawrence Brakmo proposed the TCP-BPF framework [8] 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. Since TCP-BPF was designed 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.

## 2.1 Supporting 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 add support for a new TCP option. Table 1 summarizes the new hooks added by our framework and their meaning.

Let us first analyse the sender side. When sending packets, the `tcp_transmit_skb()` function 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 these places (Fig. 1).

The first hook (in `tcp_transmit_skb()`) calls a BPF program to adjust the provisioned size of all TCP options (`tcp_options_size`). It is also verified to not exceeded 40 bytes - the maximum value. Then, at the end of `tcp_options_write()`, a second hook calls a BPF program which passes the new option data to the kernel. The kernel is then responsible for writing the new option data at the current option pointer.

To limit the overhead on 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 remains one thing that 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

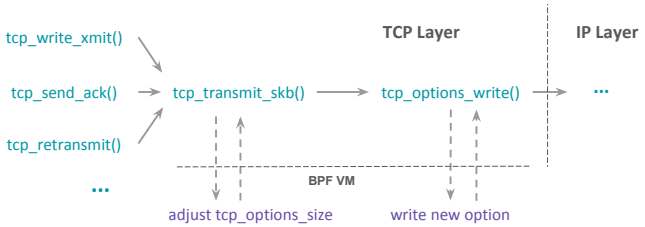



Figure 1: Insert TCP options to outgoing packets

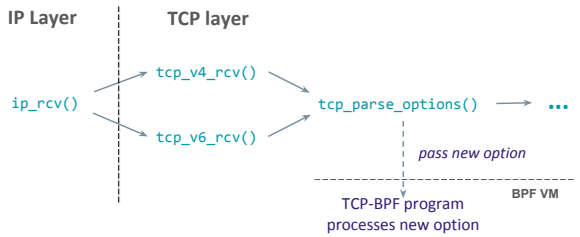



Figure 2: Pass unknown TCP options of incoming packets to BPF program

and could be fragmented on the wire. We need to update the `tcp_current_mss()` function to take the length of to-be-added option into the 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 is thus completely transparent to the BPF programs.

On the receiver side, the extension is simpler. Linux TCP parses the options of incoming TCP packets in `tcp_parse_options()`, in which all unknown options are ignored. At the end of this function, we added a hook to pass these unknown options to the BPF program, as shown in Fig. 2. This hook, once activated, will pass the option data along with option kind and length to the BPF program. The hook could also pass multiple new options of the same TCP packet to one or multiple 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()`.

## 2.2 How to select the desired packets for inserting 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 to only take care of the interesting connections. These operations have already been supported by the vanilla kernel so no kernel change is required.

The second question is how to insert new TCP options in the desired packets only. To mark when the program wants to

| Hook | In kernel function | Passed arguments | Meaning |
|---|---|---|---|
| BPF_TCP_OPTIONS_SIZE_CALC | tcp_transmit_skb<br>tcp_current_mss | Length of all TCP options | Call BPF program to adjust the length of all TCP options |
| BPF_TCP_OPTIONS_WRITE | tcp_options_write | - | Call BPF program to insert new TCP option |
| BPF_TCP_PARSE_OPTIONS | tcp_parse_options | Option kind, len, and value | Pass unknown TCP option to BPF program |

Table 1: New BPF hooks added by TCP option framework

actually insert new options, we need to add a new flag. TCP-BPF already has 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 changed code. 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 skbs of the same TCP connection, and unset the flag at another hook (e.g.: when RTO fires) to disable option writing from this point.

## 2.3 Implementation status

By building on top of TCP-BPF, we can implement our framework with modest changes to the kernel (75 LoCs). TCP option insertion support requires around 60 LoCs, while the TCP parsing support requires only 15 LoCs since it is much simpler as explained above. Table 2 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 values directly by TCP-BPF programs, while the 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 RFC patch [24], is reasonably larger.

| | 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 2: Lines of code (LoC) of the framework and each use case

## 2.4 Performance Overhead

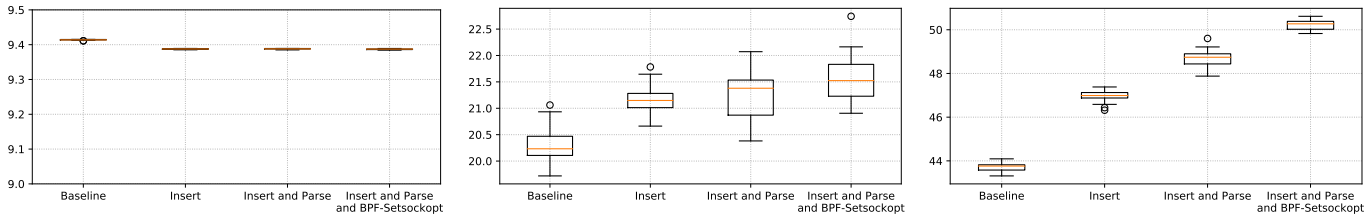
Linux TCP is a high-performance stack. Any proposed extension should take the performance impact into consideration. To evaluate the performance impact of our BPF extensions, we run the iPerf3 [18] test between two dedicate machines over a 10 Gbps link. Each machine is equipped with an Intel Xeon X3440 2.53GHz CPU and 16 GB RAM. Our framework is implemented in Linux kernel version 4.17-rc5. We use different TCP-BPF programs that are called to manipulate *each* transmitted packet. We consider four different experiments.

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

Each measurement lasts 40 seconds and each scenario is repeated 20 times. Figure 3 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 Fig. 3b and Fig. 3c.

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

These results suggest that most of the overhead of the framework comes from the call-backs to the BPF program, not from the execution of the BPF program itself.




(a) Average Throughput (Gbps) (b) Sender's CPU Usage (%) (c) Receiver's CPU Usage (%)

Figure 3: Benchmarking results:
iPerf3 client and server test over 10Gbps link.


(a) Average Throughput (Gbps) (b) Average RTT (micro-seconds)

Figure 4: Benchmarking results:
iPerf3 client and server are on the same host, connected via loopback interface.

## 3 TCP Use Cases

In this section, we illustrate with several different use cases how it is possible to leverage BPF programs to extend the Linux TCP stack. We start in Section 3.1 with the TCP User Timeout [21] that has not been implemented in the Linux TCP stack. We then propose and implement in Section 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 and finally one to request a specific acknowledgement strategy in Section 3.4.

### 3.1 Use case: TCP User Timeout Option

The TCP User Timeout option [21] 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 two typical use cases for this option. First, an application that wants to survive transient failures would select a very large User Timeout. Second, a mobile interactive application that is used on smartphones equipped with Wi-Fi and cellular interfaces could use a short User Timeout (e.g. one second) to quickly detect connectivity problems and switch to the other network interface.

The TCP User Timeout Option (UTO) [21] 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 `SO_RCVTIMEO` and `SO_SNDTIMEO` socket options. However, it does not support the UTO 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 applications to terminate the connection.

On the client side, we implement the TCP User Timeout Option with a simple BPF program (76 lines of C code) using the option-writing hooks described in the previous section. On the server side, when it receives a TCP User Timeout 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 `bpf_getsockopt()` helper function to query the current User Timeout value of the connection.

### 3.2 Use case: TCP Congestion Control Option

The Linux TCP stack supports a dozen of pluggable congestion control modules [17]. Depending on its configuration, a Linux host may directly support two to three TCP congestion control schemes, e.g. NewReno [4], CUBIC [25], or Vegas [9] or BBR [11]. Content Distribution Networks

(CDN) often tune their congestion control scheme to better serve their customers [12]. 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 of 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 an emulation environment similar[1] to Mininet [30]. We set up separate network namespaces for client and server, a Linux bridge in-between, and using Traffic Control (TC) with HTB qdisc to restrict link bandwidth to 8 Mbps and 40 ms delay per direction. Our emulated client downloads the same large file using the `curl` software. We use our BPF program to insert in the third ACK packet the TCP congestion control option to request the utilisation of a specific congestion control scheme by the server.

We consider NewReno [4], CUBIC [25], Vegas [9] and BBR [11] 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 RTT from increasing. Among all, NewReno performs worse in terms of delay.

In this example, we used the congestion control option to exchange the identifier of the congestion control scheme

[1]We do not use Mininet 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.

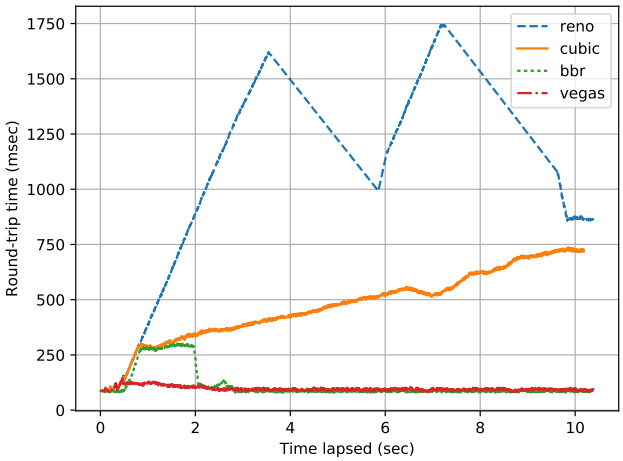



Figure 5: Congestion Control Option test: RTT on the server (8 Mbps bandwidth, 40 ms link delay)

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 [29] 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 could apply to HTTP/2 running over TCP.

### 3.3 Use case: Option to Request Initial Congestion Window

While the congestion control algorithm has a significant impact on the performance of long flows, 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 been increased over the years from 2 MSS to 4 MSS [3] and later 10 MSS [16, 19] 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 [32, 33] show that while most web servers use the default values of their TCP stacks, CDN operators usually apply much larger values of IW [33]. This research also suggests that some CDNs customize their IW configuration based on the network type and/or the content type.

Brakmo suggested [8] to heuristically select the IW based on the IP prefix using TCP-BPF, with a simple example [1]. 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 attack, 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 BPF program. Our 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. Listing 1 presents the simplified version of our BPF program used to write IW option.

```
struct tcp_option {
    __u8 kind;
    __u8 len;
    __u16 data;
}
SEC("sockops")
int bpf_insert_option(struct bpf_sock_ops *skops)
{
    struct tcp_option opt = {
        .kind = 66,      // option kind
        .len = 4,        // of this option struct
        .data = 20,      // # MSS
    };
    int rv = 0;
    int option_buffer;

    switch (skops->op) {
    case BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB:
        // activate option writing flag
        rv = bpf_sock_ops_cb_flags_set(skops,
                    BPF_SOCK_OPS_OPTION_WRITE_FLAG);
        break;
    case BPF_TCP_OPTIONS_SIZE_CALC:
        // adjust total option len, not over 40 Bytes
        int option_len = sizeof(opt);
        int total_len = skops->args[1];
        if (total_len + option_len <= 40)
            rv = option_len;
        break;
    case BPF_TCP_OPTIONS_WRITE:
        // put struct option into reply field
        memcpy(&option_buffer, &opt, sizeof(int));
        rv = option_buffer;
        // will not insert option after 1st data packet
        if (skops->data_segs_in > 1)
            bpf_sock_ops_cb_flags_set(skops, 0);
        break;
    default:
        rv = -1;
    }
    skops->reply = rv;
    return 1;
}
```

Listing 1: BPF program to write IW option

To demonstrate the impact of tuning the initial congestion window with web traffic, we use the methodology proposed by Wang et al. [37] with the `epload` software [36]. This enables us to emulate real web contents and gather web page download times.

We set up a similar testbed to the one of the previous use case 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 against each website is repeated three times.

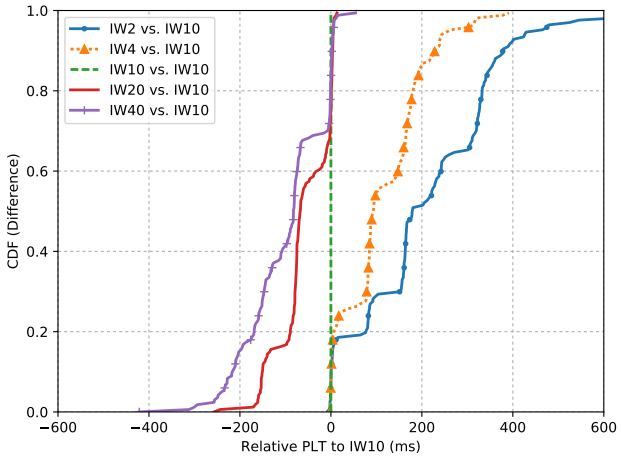



Figure 6: Initial Window Option test: Page Load Time relatively to IW=10 (40 Mbps bandwidth, 40 ms link delay)

Figure 6 shows the relative Page Load Time (PLT) results of 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, and a few of sites suffered from a higher value of IW. With high network capacity in the experiment, we did not observe much congestion; however, the results should change if the network resources are 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.

### 3.4 Use case: Tuning the acknowledgement strategy

As a reliable protocol, TCP crucially relies on the ACK packets to detect losses and control the data transfer. Sending ACKs too frequently may impose too much overhead in wireless networks or on fat pipes. On heavily loaded servers, the ACK processing may consume as much as 20% of the CPU cycles [13]. 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 the congestion window during the slow-start phase, trigger bursty transmissions, overestimate RTT and RTO, or prevent

Fast/Early Retransmit recovers from real losses.

For these reasons, the IETF (RFC2525 section 2.13 [31]) recommended a trade-off: do not delay ACKing for more than 500 ms and immediately send an ACK for every second packet. Linux follows this recommendation and the minimum and maximum values of the delayed ACK timeout are hard-coded at 40 ms and 200 ms.

However, such fixed values cannot adapt to connections having very different delay, bandwidth and loss characteristics. They may be too large for local connections, but too small for inter-continental ones. The only customization supported by Linux is to disable the delayed ACK mechanism for each route [35]. However, there is no way for a sender to know the acknowledgement strategy that is used by its peer.

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

Meanwhile, modern networking stacks have adapted to the stretch ACK technique. First, popular networking stacks support pacing, which helps to avoid the bursty transmission issue, a side-effect of the interaction between the stretched ACKing and the classical congestion controls. Second, the congestion control implementations were adapted to increase the congestion window properly with stretch ACKs [10, 14]. Furthermore, the Recent ACK (RACK) [15] (subsumed Tail-Loss Probe (TLP) [20]) mechanism is being standardized and deployed in Linux and Windows [6]. This allows TCP senders to quickly detect losses based on a per-packet timer instead of using duplicated ACKs, reducing the impact of ACK stretching.

Google proposed a TCP Option [38] to negotiate a custom delayed ACK timeout during the three-way handshake. However, as discussed in IETF99 TCPM WG meeting [2], 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 Supporting User-Defined MPTCP Path Managers

Multipath TCP [22] is a major extension of TCP. By decoupling TCP from IP and enabling resource pooling, it brings several benefits: allow seamless handover, aggregate bandwidth of multiple paths, increase resiliency, reduce latency and so on [7]. One of the major tasks in an MPTCP implementation is to set up paths based on current situation and the user requirements, which is handled by path managers. The path managers need to decide which path should be created or removed, and which addresses should be announced. The Linux implementation of MPTCP (which is still out-of-tree at `github.com/multipath-tcp/mptcp`) supports different path managers implemented as kernel modules. However, this task should be controlled by applications or users who have a wide range of requirements. Therefore, a netlink-based PM framework has been introduced to support control plane in user space [26], and has been recently merged in `mptcp-trunk` branch [5].

While using netlink is a natural approach that provides a clean separation between control plane and data plane, it is not without issues. It has a certain amount of 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. Moreover, this approach requires separated facilities to support various but maybe necessary features: `setsockopt` or `getsockopt` (e.g. access subflow-level info), TCP state change notification, apply a custom policy to refuse the establishment of a subflow. The last one may be difficult to implement.

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

- Performance: Once a BPF program is loaded into the kernel, it is not necessary to switch between user space and kernel space for every operation like netlink-based approach. We can also avoid the overhead due to sending and receiving netlink messages
- TCP-BPF has built-in support for TCP state tracking and TCP socket manipulation

- Easy to apply custom accepting/refusal policies on the subflow establishment

However, it is expected that this approach would have its own limitations. First, eBPF programs are restricted by current eBPF limits: no loop supported, each BPF program cannot have more than 4096 instructions. Second, compared to the netlink-based approach, the layering separation is less clean. Third, since BPF programs can be called from different contexts, the locking mechanism is probably trickier than userspace solutions like the netlink one.

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

## 4.1 Tracking events

In order to give decisions, the path managers must know which MPTCP-related events happen and the associated metadata. It is possible to perform these operations using BPF programs of the `kprobe` type without introducing any kernel change. However, we also need to carry actions on the connection (e.g. create or delete a subflow) which typically requires TCP-BPF programs of `sock_ops` type. This means that we need two different BPF programs to fulfill the task: `BPF_PROG_TYPE_KPROBE` and `BPF_PROG_TYPE_SOCK_OPS`. Connecting these two BPF programs and synchronizing shared data would be complex (ugly if not difficult).

For this reason, we added new TCP-BPF callbacks to track important events for the path managers (Table 3). Since these callbacks are inserted at the same places as the netlink-based Path Manager solution does [5], we do not present these locations in the table.

One issue is that the TCP-BPF callbacks only support at most three arguments accompanying each call. This limits the amount of MPTCP metadata that could be passed through these calls. For this reason, we extended the object context of the TCP-BPF programs (struct `bpf_sock_ops`) to keep track of common metadata per MPTCP session (`mptcp_loc_token`, `mptcp_rem_token`, `mptcp_loc_key`, `mptcp_rem_key`, `mptcp_flags`). This brings more MPTCP metadata to BPF programs and also simplifies new MPTCP callbacks.

At the moment, to track the subflow-level events we reused the available TCP-BPF hooks for regular TCP stack: e.g. `BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB` `BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB`, or the more generic one `BPF_SOCK_OPS_STATE_CB`. In the future to support more advanced path managers, we could either extend these hooks to pass more MPTCP-specific metadata or add new dedicated MPTCP-only hooks to keep the API clean.

## 4.2 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 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 context information.

## 4.3 Opening a subflow

Since opening subflows is an action that changes the behavior of the kernel stack, we followed the eBPF common practice by implementing it via a new helper function (`mptcp_open_subflow()`). This helper function takes five arguments as input:

- `bpf_sock_ops` (BPF socket context): The function uses `bpf_sock_ops` 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, `mptcp_open_subflow()` uses the existing or kernel-assigned values when creating the subflow.
- The associated lengths of above `sockaddr`s, as required by eBPF when passing memory regions.

There is however one subtle issue here: BPF programs can be called from different contexts. Usually, we are in softirq context, therefore this helper function cannot open subflows directly. Instead, we created a custom global `work queue` first. Every time this helper function is called, it schedules a `work` into this `work queue` to delegate the actual task in the future. Since we cannot add our custom parameters into the work itself, we need to embed the work and 4-tuple 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 structure `bpf_pm_priv` and linked to the relevant mptcp control block (`mptcp_cb`). Then, when the kernel scheduler wakes up the worker thread, the work handler actually opens the requested subflow by calling `mptcp_init4_subsockets()` (or `mptcp_init6_subsockets()` for IPv6 subflow).

## 4.4 Use cases

To illustrate the usage of this path manager framework, two minimal path managers were implemented as BPF programs (Table 4).

| Callbacks | Events | Passed arguments |
|---|---|---|
| BPF_MPTCP_NEW_SESSION | A new MPTCP Session is created | - |
| BPF_MPTCP_FULLY_ESTABLISHED | An MPTCP Session is established | flag: is this master_sk? |
| BPF_MPTCP_CLOSE_SESSION | An MPTCP Session is closed (including fallbacks to regular TCP) | - |
| BPF_MPTCP_ADD_RADDR | Remote IP address is added | IP, port, address ID |
| BPF_MPTCP_REM_RADDR | Remote IP address is removed | address ID |

Table 3: New TCP-BPF callbacks for Generic PM Framework

#### 4.4.1 *ndiffports* path manager

*ndiffports* path manager creates multiple subflows on the same source and destination IP addresses, only differing in port numbers. It is designed to exploit the path diversity to avoid the bottlenecks in ECMP-enabled datacenters. Due to its simplicity, we implemented *ndiffports* in 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.

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

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

        /* when passing (NULL, 0): existing addresses
        * will be used to set up new subflow */
        /* Call twice to open two 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

#### 4.4.2 *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. The local addresses are global and loaded into an array map (`local_addr_map`) by a user daemon. Meanwhile, `add_addr_map` stores the remote addresses are per connection, with the MPTCP token as the key. The map value is the structure that contains remote IP addresses and their correspondent address IDs. The remote address list is updated every time the host receives an `ADD_ADDR` or `REM_ADDR` from remote peer. On the MPTCP session closing event, the remote address list of that session is removed from `add_addr_map`.

| | Lines of code |
|---|---|
| Generic PM Framework (in kernel) | ~300 |
| ndiffports PM (BPF program) | ~20 |
| fullmesh PM (BPF program) | ~200 |

Table 4: The implementation size of Generic PM Framework and two path managers

### 4.5 Next steps

At the moment, several features have not been implemented in this first prototype:

- Handle events of local IP address changed. Need to send events to each BPF program in each `cgroup-v2`. However, for TCP-BPF program type, we need to pass the appropriate `sock` struct which contains the cgroup information. An 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 BPF programs.

- Subflows removal support. This feature should be implemented as a helper function, in a similar way to `mptcp_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 kernel. Therefore, this feature is only needed in other cases: e.g. to close additional subflows under memory pressure or when cellular traffic quota is 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 `MPTCP_INFO` socket option).

- Only IPv4 is supported so far. Dual-stack support may be similar to the implementation of `bpf_bind()`, which should not be too difficult.

- Multiple path managers support. This may be necessary, for example, when MPTCP proxies may want to use different PMs for upstream and downstream traffic.

In comparison, current netlink-based solution supports different PMs, one per network namespace.

## 5 Discussion

TCP was designed to be extensible by using TCP options. However, the last decades have shown that it remains very difficult to extend TCP by defining such a new option. While the IETF has reserved a set of option types for experimental options, 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 stack. Our work has shown that, with little changes to the kernel code, it is possible to leverage eBPF programs to quickly implement a range of new TCP features. The main drawback of this method is the limitation of TCP option space, which cannot be larger than 40 bytes. On the other hand, it is the first step to make the Linux TCP stack truly extensible.

The results described in this paper open different directions for future work. A first direction is improving the eBPF support in the Linux kernel. Our implementation is based on the TCP-BPF framework which currently relies on `cgroup` version 2. It could be interesting to remove this restriction.

A second 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 [27]. 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 [7] or between edge servers and core servers of CDNs. Furthermore, there is anecdotal evidence that large content providers use a tuned version of the Linux TCP stack that has diverged from the mainline Linux kernel over the years. This implies that either they frequently need to backport new features of the Linux kernel or do not use these improvements in their stack. Using eBPF would enable them to both completely tune their Linux TCP stack and still benefit from the community improvements.

A third 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 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.

## 6 Software Artefacts

The implementation of our TCP option extension framework, the MPTCP path manager framework, different use cases and the experiment scripts are available at `https://github.com/hoang-tranviet/tcp-options-bpf`. Our customized Epload is available at `https://github.com/hoang-tranviet/Epload`. Our analysis and plot scripts is available at `https://github.com/hoang-tranviet/tcp-options-bpf-analysis`. Experiment results and related resources can be found at `https://www.info.ucl.ac.be/~tranviet/`.

## 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] [net-next,v6,13/16] bpf: Sample bpf program to set initial cwnd. https://patchwork.ozlabs.org/patch/783031/.

[2] IETF Minutes IETF99. https://datatracker.ietf.org/doc/minutes-99-tcpm/, 2017.

[3] ALLMAN, M., FLOYD, S., AND PARTRIDGE, C. RFC 3390: Increasing TCPs initial window. *Internet Eng. Task Force (IETF)-Request for Comments* (2002).

[4] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009.

[5] BAERTS, M., AND DETAL, G. [PATCH mptcp_trunk v8 0/5] mptcp: new generic Netlink-based PM. https://sympa-2.sipr.ucl.ac.be/sympa/arc/mptcp-dev/2019-01/msg00084.html/, Jan. 2019.

[6] BALASUBRAMANIAN, P. IETF96: Transports advancements in the Windows network stack. IETF.

[7] BONAVENTURE, O., AND SEO, S. Multipath TCP deployments. *IETF Journal 12*, 2 (2016), 24–27.

[8] BRAKMO, L. TCP-BPF: Programmatically tuning TCP behavior through BPF. *NetDev 2.2* (2017).

[9] BRAKMO, L. S., AND PETERSON, L. L. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications 13*, 8 (1995), 1465–1480.

[10] CARDWELL, N. Linux Kernel: Merge branch: fix stretch ACK bugs in TCP CUBIC and Reno, 2015.

[11] CARDWELL, N., ET AL. Bbr: congestion-based congestion control. *Communications of the ACM 60*, 2 (2017), 58–66.

[12] CARLSSON, E., AND LABS, E. K. S. Smoother Streaming with BBR, 8 2018.

[13] CHAN, M., AND CHERITON, D. R. Improving Server Application Performance via Pure TCP ACK Receive Optimization. In *USENIX Annual Technical Conference* (2013), pp. 359–364.

[14] CHENG, Y. Linux Kernel: commit 9f9843a751d0. tcp: properly handle stretch acks in slow start, 2013.

[15] CHENG, Y., CARDWELL, N., DUKKIPATI, N., AND JHA, P. Rack: a time-based fast loss detection algorithm for tcp. draft-ietf-tcpm-rack-04.

[16] CHU, J., ET AL. RFC 6928: Increasing TCP's initial window. Tech. rep., 2013.

[17] CORBET, J. Pluggable congestion avoidance modules. *Linux Weekly News* (2005).

[18] DUGAN, J., ELLIOTT, S., MAH, B. A., POSKANZER, J., AND PRABHU, K. iperf3, tool for active measurements of the maximum achievable bandwidth on ip networks. *URL: https://github.com/esnet/iperf*.

[19] DUKKIPATI, N., ET AL. An argument for increasing TCP's initial congestion window. *Computer Communication Review 40*, 3 (2010), 26–33.

[20] DUKKIPATI, N., ET AL. Tail loss probe (TLP): An algorithm for fast recovery of tail losses. *draft-dukkipati-tcpm-tcploss-probe-01. txt* (2013).

[21] EGGERT, L., AND GONT, F. TCP User Timeout Option. RFC 5482 (Proposed Standard), Mar. 2009.

[22] FORD, A., RAICIU, C., HANDLEY, M., AND BONAVENTURE, O. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, Jan. 2013.

[23] FUKUDA, K. An analysis of longitudinal TCP passive measurements. In *TMA Workshop* (2011), Springer, pp. 29–36.

[24] GREEAR, B., AND BALUTA, D. Linux Kernel: [RFC] TCP: Support configurable delayed-ack parameters., 2012.

[25] HA, S., RHEE, I., AND XU, L. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review 42*, 5 (2008), 64–74.

[26] HESMANS, B., ET AL. SMAPP: Towards smart Multipath TCP-enabled applications. In *Proceedings of the 11th ACM CONEXT* (2015), ACM, p. 28.

[27] HONDA, M., ET AL. Is it still possible to extend TCP? In *Proceedings of the 2011 ACM SIGCOMM* (2011), ACM, pp. 181–194.

[28] HONDA, M., ET AL. Rekindling network protocol innovation with user-level stacks. *ACM SIGCOMM Computer Communication Review 44*, 2 (2014), 52–58.

[29] LANGLEY, A., ET AL. The QUIC transport protocol: Design and Internet-scale deployment. In *Proceedings of the ACM SIGCOMM* (2017), ACM, pp. 183–196.

[30] LANTZ, B., HELLER, B., AND MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (2010), ACM, p. 19.

[31] PAXSON, V., ALLMAN, M., DAWSON, S., FENNER, W., GRINER, J., HEAVENS, I., LAHEY, K., SEMKE, J., AND VOLZ, B. Known TCP Implementation Problems. RFC 2525 (Informational), Mar. 1999.

[32] RÜTH, J., BORMANN, C., AND HOHLFELD, O. Large-scale scanning of TCP's initial window. In *Proceedings of the IMC 2017* (London, United Kingdom, 2017), ACM Press, pp. 304–310.

[33] RÜTH, J., AND HOHLFELD, O. Demystifying TCP Initial Window Configurations of Content Distribution Networks. In *2018 TMA Conference* (2018), IEEE, pp. 1–8.

[34] TOUCH, J. Shared Use of Experimental TCP Options. RFC 6994 (Proposed Standard), Aug. 2013.

[35] WANG, A. C. Linux Kernel: tcp: introduce a per-route knob for quick ack, 2013.

[36] WANG, X. S. Eload. http://wprof.cs.washington.edu/spdy/tool/, 12 2018.

[37] WANG, X. S., ET AL. How speedy is SPDY? In *11th USENIX NSDI* (Seattle, WA, 2014), USENIX Association, pp. 387–399.

[38] WEI WANG, NEAL CARDWELL, Y. C., AND DUMAZET, E. IETF draft: TCP Low Latency Option, 2017.