Mutipath TCP Upstreaming

Mat Martineau[1] and **Matthieu Baerts**[2]
[1] Intel, United States of America – mathew.j.martineau@intel.com
[2] Tessares SA, Belgium – matthieu.baerts@tessares.net
https://github.com/multipath-tcp/mptcp_net-next/wiki

Abstract

Multipath TCP is a recent TCP extension that allows devices like today's smartphones or laptops to send and receive data over multiple interfaces for better resource utilization, throughput, and reaction to failures and handover. While interest is growing in this new TCP extension, this protocol is not currently supported by the upstream Linux kernel. A community project is underway to add Multipath TCP to the upstream Linux kernel, with baseline features arriving first and plans to continue maintenance and development in later kernel versions.

Introduction

Multipath TCP (MPTCP) is an increasingly popular protocol that members of the kernel community are actively working to upstream. An out-of-tree Linux kernel implementing the protocol has been developed and maintained since March 2009 [1]. While there are some large MPTCP deployments using this custom kernel [2], an upstream implementation will make the protocol available on Linux devices of all flavors.

MPTCP is closely coupled with TCP, but an implementation does not need to interfere with operation of normal TCP connections. The MPTCP Linux Upstream roadmap begins with the server use case, where both initial MPTCP connections and additional path connections are generally initiated by peer devices. This will start with RFC 6824 [3] compliance, but with a minimal feature set to limit the code footprint for initial review and testing.

The MPTCP upstreaming community has shared an RFC patch set on the netdev list that shows their progress and how they plan to build around the TCP stack. This paper explains the consensus decisions that this community has reached and shares the roadmap for how this patch set will evolve before submission.

With baseline MPTCP code merged, community members have further plans to develop more advanced features for managing subflow creation (path management), scheduling outgoing packets across TCP subflows, and other capabilities important for client devices that initiate connections. This includes making use of a userspace path manager, which is already available as an alpha release. Support for additional TCP features and optimization of MPTCP performance are also expected as wider availability of MPTCP in Linux leads to more feedback from kernel users.

Unfortunately, Multipath TCP is in the end a complex protocol that poses certain challenges when integrating with a TCP stack [5]. It has to be in order to work in a world full of middleboxes that can be hostile to new TCP extensions [4]. Some concepts and terminology are also specific to MPTCP. To establish a foundation with the MPTCP protocol to help understand the features discussed later in the paper, the next section has an overview of MPTCP, as also explained in the previous paper by these authors [5]. After that, Section 2 details how the first patch set will be composed. Before the conclusion, Section 3 describes the features that will further evolve MPTCP for Linux to meet the needs of more devices and users.

1 Multipath TCP Overview

On both wired and wireless networks TCP is the dominant transport protocol. At 45 years old TCP has evolved continuously and includes various optimizations. Multipath TCP is a recent major extension to TCP [3, 6] that allows devices to exchange data for a single connection over different paths simultaneously.

1.1 Use-cases

Multipath TCP was standardized in early 2013, almost 30 years after the standardization of TCP [7]. Even though this extension is young, it is already used to support several commercial services [2]. The first and most significant deployment is at Apple. In September 2013, Apple enabled Multipath TCP on iPhones and iPads in support of the Siri voice-recognition application. Their main motivation for using Multipath TCP was to provide smooth handovers between Wi-Fi and cellular while users are physically moving while interacting with Siri. Given the success of this deployment, Apple enabled Multipath TCP for all applications on iOS in September 2017 and is going to use it for more of their own applications in 2019 [8].

The second significant deployment is on Android smartphones. Since July 2015, the out-of-tree implementation of Multipath TCP in a Linux kernel [1] is used in high-end smartphones from Samsung and LG to bond Wi-Fi and

4G/LTE to achieve higher bandwidth. Korea Telecom has reported download speeds of 800 Mbps and more.

Another important use case for Multipath TCP is with Hybrid Access Networks [9]. To provide faster Internet services, notably in rural areas, several network operators have decided to combine their xDSL network and their LTE network together. One of the solutions standardized by the Broadband Forum to create such hybrid networks uses Multipath TCP proxies [10]. Several operators have already deployed this solution.

Given all of these use cases and their scope it can be expected that Multipath TCP traffic will grow in the coming years. Also significant is selection of MPTCP by the 3GPP organization to play an important role in the emerging 5G network architectures[11] where it will be used to provide the Access Traffic Steering, Switching & Splitting (ATSSS) core networking function.

1.2 Differences in requirements and features from existing implementation

Multipath TCP has five known independent implementations [12]. These are on the following platforms: GNU/Linux with a fork of the Linux kernel [1], Apple iOS and MacOS [13], Citrix load balancers, FreeBSD [14], and Oracle Solaris. The first three implementations are known to interoperate. Three are open source (Linux kernel, FreeBSD and Apple's iOS and MacOS). Apple's implementation is widely deployed.

The existing out-of-tree Linux kernel implementation of Multipath TCP started off as a research project during the standardization phase. With an evolving specification, the community defining the MPTCP protocol needed an open implementation that could support experiments and rapid changes. During that time the code was not designed for easy upstream inclusion but rather to quickly meet the needs of the standardization community and showcase the benefits and potential of MPTCP. Over time, this implementation has been used as the basis for several of the abovementioned MPTCP deployments [12].

However, it should be noted that those deployments all target a specialized infrastructure and configuration where the platform has tight control over how the kernel is being used. The MPTCP Linux kernel is deployed either on smartphones where the vendors specifically configured the system for this use-case, or on embedded platforms and servers where the vendor fully controls the configuration and environment of the deployment.

The design of the out-of-tree implementation [1] is not well-suited for the more generic upstream Linux kernel. It significantly changes core TCP code and data structures as it was initially built to add multipath functionality to every TCP connection. In contrast, an upstream design must fit within and around the existing TCP stack, not introduce any performance regressions, be maintainable, and be straightforward to configure and use in a variety of deployments.

1.3 Protocol concepts

MPTCP uses the multiple network interfaces of a host by creating separate TCP connections, called TCP subflows, on each of those interfaces. Those subflows are then used to transmit and receive data. The data that is sent from the application is dynamically split across those subflows. As MPTCP still guarantees in-order delivery of the data stream, the data that is sent on each subflow needs an additional sequence number to allow the receiver to reconstruct the application stream. This so-named data sequence number is put in the TCP option space. All of the signaling in MPTCP is done in the TCP option space. Examples of this signaling are capability negotiation, authentication, address announcement, and error signaling.

Using the TCP option space and separate TCP subflows that look like normal TCP connections enables MPTCP to be globally deployed on the current Internet despite the presence of various stateful middleboxes and firewalls [4]. MPTCP is thus in some sense a middle layer between the transport (TCP) and the application, while using TCP options for signaling.

However, this design also implies that the MPTCP middle layer requires very tight control over how the TCP stack behaves. MPTCP needs to direct TCP to include specific options in the TCP header associated with specific data, and it needs to force the TCP stack to send specific segments like ACK, FIN, or RST.

The most notable areas of low-level TCP control and semantic modifications are:

Coupled receive windows across TCP subflows The receive window is shared across all the subflows involved in a MPTCP connection. The flow-control exercised by the application needs to apply to the overall MPTCP connection across all TCP subflows [6]. Where a regular TCP connection has exclusive control over its own receive window, MPTCP subflows need to safely access shared data in the MPTCP middle layer to manage and advertise the window size.

Sending of ACKs to signal options MPTCP announces subflow priorities or changes of addresses to the peer, e.g. when a mobile host disconnects from Wi-Fi, MPTCP can signal this via the cellular interface [15]. As MPTCP uses TCP options for such signaling, the MPTCP layer needs an interface to trigger the sending of a TCP ACK segment that includes the specific TCP option.

Reception of ACKs with signaling options Upon reception of TCP ACKs with an MPTCP signal, the MPTCP layer needs access to the MPTCP option values in order to take action. For example, if a smartphone moves out of the Wi-Fi range, it will send a "*remove-address*" message to the server inside a TCP ACK that includes the MPTCP option in the TCP option space. The smartphone will send this TCP ACK on the TCP subflow that is routed via the cellular interface. Upon reception of this segment, the MPTCP stack on the server-side will need to tear down the subflow that is linked to the Wi-Fi interface.

Data-stream MPTCP utilizes a data-sequence number (DSN) that maps every byte sent on each subflow to the application-level byte-stream [3]. The data-sequence number is sent as part of a TCP option with outgoing data segments. During the generation of the TCP header, the TCP stack needs to access metadata associated with the payload.

The same holds true on the receiver side. Incoming data also has its DSN stored in the TCP option space. When receiving data, the contents of TCP options are not exposed outside the TCP layer as traditionally TCP options only have a meaning for the TCP state machine. MPTCP breaks with this assumption as the data sequence number needs to be communicated up to the MPTCP layer to reconstruct the byte stream.

1.4 RFC 6824bis, aka MPTCP v1

A new version of Multipath TCP protocol is nearing release [17]. It has been submitted to IESG for publication. This document is currently called RFC 6824bis and will be assigned a new number when the standard is released. It introduces some behavioral changes in the protocol which necessitates the version changing from v0 to v1. The current version [3] is (and will remain) *Experimental* while the new one [17] will be *Standard*.

The differences between the two versions are listed in Appendix E of RFC 6824bis [17]. The major change that impacts the implementation is the way the initialization is done: the initiator's key is no longer included in the SYN packet. Among other advantages, this reduces the number of TCP option bytes consumed by MPTCP, allowing other TCP options like TCP Fast Open Cookie Request to be present. There are also some changes in some signaling options, e.g. it is possible to add a reason when removing one path and the `ADD_ADDR` is now reliable. Other changes also help simplify the implementation like the *fast-close*: with the new version, it is possible to send a RST on all subflows and stop maintaining a state for potential retransmissions of the `MP_FASTCLOSE` signal included in a ACK. This allows the host to tear down the subflows and the connection immediately which makes more sense for a *fast-close* when resources need to be freed quickly.

For the moment, no known implementation supports this new version of the protocol. The out-of-tree MPTCP implementation [1] has some RFC patches posted [18] but they still need to be finalized.

2 First Patch Set Upstreaming Roadmap

A community group consisting of developers from different companies is working on upstreaming Multipath TCP to the Linux kernel. This group has already published various patch sets, with some code only relevant to MPTCP but other patches being of benefit to other parts of the network subsystem.

2.1 Guidelines

The community group has a set of guidelines for the design and implementation of MPTCP in the Linux kernel. First, it is critical to maintain the performance and reliability of the TCP code. MPTCP is built on top of TCP and involves a limited number of semantic changes to TCP, but can be built in such a way that regular TCP continues to operate as it does today.

Second, applications that use TCP today will continue to use TCP by default even after MPTCP is upstream. Multipath TCP will be "opt-in", requiring programs to explicitly request multipath connections for each socket.

Finally, the specifications for MPTCP describe both required and optional features, which allows us to initially target a limited feature set that will interoperate with other MPTCP peers. This will keep the initial patch set size from becoming unwieldy and allow us to more thoroughly test the first batch of features.

2.2 Protocol compliance

Multipath TCP is defined in RFC 6824 [3] as an *Experimental* version while a new *Standard* version is on its way to be published soon [17]. The relative complexity of the protocol stems from the need to have MPTCP packets look like *normal* TCP on the wire in order to pass through the various middleboxes present on today's Internet.

What to support? Like any protocol some behaviors are mandatory (*MUST*) and some are not (*SHOULD*). It is not necessary to wait for support of all features before proposing MPTCP upstream. Not even all mandatory features need to be present. Indeed, the protocol has some useful escapes in case some features are not supported. The most common technique is to fallback to TCP. Using this approach, certain features can be developed later, e.g.

- The DSS checksum is an optional field in the DSS MPTCP option. This feature is useful to detect corruption in the MPTCP option space and is negotiated at connection time. In controlled environments, this CPU-costly feature is not required.
- Only the client or server side could be supported initially.
- Even if MPTCP sits on top of the IP layer, some code paths for IPv4 and IPv6 can be different. If an MPTCP request arrives in an IPv6 packet, the server does not have to reply that MPTCP is supported for this connection if IPv6 is not supported with Multipath TCP in the first place. Instead, the server can reply that it does not support MPTCP, simply by not adding an MPTCP option in the SYN-ACK.

Another way to add MPTCP to the upstream Linux kernel with a minimal number of bugs is to forcus on clarity and readability over very high optimization. This is not to say that things like memory usage and CPU efficiency should be ignored, instead it is a recognition that it is appropriate for MPTCP to meet the expectations of normal kernel code but not necessarily have the same attention to fine-tuned clock cycle counts and data structure cache line tweaks that TCP has gained from years of measurement and incremental improvement.

Which version to support? The upstreaming community's current intent is to implement the MPTCP v1 protocol from RFC 6824bis [17] as described in section 1.4. This will simplify the code since there will be no need to handle both the experimental v0 protocol and the standardized v1. There is a tradeoff in terms of interoperability, as all other public MPTCP stacks deployed today only support v0. The changes between the two standards were made after seeing where v0 needed improvement when used in production, and once the final standard is published v1 deployments should take over as the experimental protocol is deemphasized. Using only the final v1 standard in the upstream Linux kernel will also help obsolete the experimental protocol faster.

2.3 Interfaces with userspace

As mentioned in section 2.1, Multipath TCP will be "opt-in". Some other aspects of MPTCP will also be under control of the userspace applications, which will therefore require a way to interact with it.

Defining a new internal IP protocol number When creating a socket, programs will need to explicitly request multipath connections by using the `IPPROTO_MPTCP` protocol number:

```
socket(AF_INET, SOCK_STREAM, IPPROTO_MPTCP);
```

The `IPPROTO_MPTCP` protocol number will not be visible on the wire. Currently, it is set to the value:

```
IPPROTO_TCP | 0x100
```

Only 8 bits are available in the IP header to identify the next level protocol. The protocol number is mostly plumbed through the kernel as an `int`, so it works to define a 16-bit protocol number for use in the `socket()` call. This keeps `IPPROTO_MPTCP` out of the IANA-managed 8-bit range of values, and choosing a value near today's `IPPROTO_MAX` minimizes impact on the few arrays sized using that value.

Socket options `setsockopt()` and `getsockopt()` system calls are another aspect of the userspace API for sockets. Ultimately, applications will be able to control Multipath TCP connections as they already do with TCP. Many options continue to make sense for MPTCP but some would have limited scope:

- only for the subflows: `SO_MARK`, `TCP_CONGESTION`, etc.
- others only for the global MPTCP connection: `TCP_NODELAY`, `SO_LINGER`, etc.
- and others that can be valid for both: some users would prefer to use `SO_KEEPALIVE` for the whole MPTCP connection, others for each subflow individually.

For the moment, options can be set and retrieved only when no subflow has been created, allowing us more time to settle on an API. More details are available in Section 3.6.

Information exposed in `procfs` `/proc` has been traditionally used to expose information from the kernel space to the user space. `sock_diag(7)` is now the preferred mechanism for sharing information about sockets. Given the tight link between MPTCP and TCP, it might be necessary to expose certain data like TCP does in `/proc/net/tcp`, but nothing is planned in this area. To ease debugging, MPTCP counters will be added in `/proc/net/netstat`. More details are given in Section 2.6.

Regarding `/proc/sys`, some behaviors can be changed via `sysctl`. These parameters will affect all connections in a network namespace. For the moment, it is possible to control the creation of new MPTCP sockets per namespace. The sysctl is set to `0` by default to maximize security, rejecting the creation of MPTCP sockets. This leaves more time for the community to test and inspect the code for security issues.

2.4 TCP features

Not every TCP feature will be usable in combination with MPTCP at first. System defaults will be used for TCP timestamps and congestion control. Features that change connection time behavior or use a significant amount of option space, like TCP Fast Open, are not planned to be supported initially.

The present RFC patch set supports IPv4. IPv6 is crictical to have early in the upstream life of MPTCP, but the community is soliciting feedback about adding IPv6 soon after an initial MPTCP merge to simplify the initial code.

2.5 Server use case

One common way Multipath TCP is used is in a traditional client/server model, where the server has an internet-accessible network interface and numerous clients initiate connections to the server. By focusing on the server functionality in the initial upstreaming patch set, the scope of kernel changes can be limited while still serving a broad set of users and creating a base for building further MPTCP client, proxy, or peering features.

In this server use case, the client is responsible for sending `MP_JOIN` requests in order to add subflows to the MPTCP connection. This means the initial kernel code does not have to track interface availability or have configurable policies for subflow establishment. For a server it is sufficient to have a limit on the number of open subflows, and if a peer tries to go beyond the limit, the additional TCP subflow can be immediately closed. This can have a straightforward implementation within the kernel without requiring any interaction with userspace or other kernel subsystems. In addition, since the client can create subflows from its various network interfaces to a single internet-accessible server IP address, it is not necessary for a server to announce addresses with an `ADD_ADDR` signal.

Many applications can also be served with a simple transmit packet scheduler. In MPTCP, a scheduler is responsible for choosing one or more subflows to use to transmit each segment of data. What is proposed by the community is a simple, server-oriented scheduler that

would transmit on the non-backup subflow that most recently received data from the peer. If only backup subflows are available it would use the subflow that most recently received data.

2.6 Diagnostics and Tests

The kernel self tests for MPTCP currently create multiple namespaces and `veth` interfaces, which are then used in several configurations. Coverage includes MPTCP/MPTCP, MPTCP/TCP, and TCP/MPTCP connections, packet loss, reordering, and variations in routing.

Features that are useful for MPTCP development, and are therefore useful to implement early, are SNMP counters for tracking MPTCP behavior and diag interface extensions that give visibility into subflow sockets.

A background project is in progress to add MPTCP support to Packetdrill [21] in order to help test the upstream MPTCP code as it is created. Like the MPTCP out-of-tree kernel [1], an out-of-tree Packetdrill project was initiated in 2013 to add MPTCP support [22]. Also like the MPTCP out-of-tree kernel, this project cannot be upstreamed as-is. It is based on an old version of Packetdrill and the architectural choices do not fit upstream expectations.

2.7 Phasing in MPTCP

As Multipath TCP code is introduced, it makes sense to be careful about new code that has not yet had time in the wild to be exposed to hostile traffic, misuse, or a wide variety of systems and configurations. `CONFIG_MPTCP` will default to N, as is customary for new features. Linux distributions may choose to enable the feature to allow use of the protocol and discovery of any issues. A further option for phasing in MPTCP is to require `CAP_NET_ADMIN` to set a sysctl to allow applications in the namespace to create MPTCP sockets, as described in section 2.3. This would make MPTCP broadly available in kernel binaries while still making sure users are opting in to the new functionality.

2.8 Summary of TCP changes

While implementing the features above, as much code as possible has been isolated in `net/mptcp/`. An MPTCP socket is a new socket type, with code that is independent from TCP. This MPTCP socket creates and manages in-kernel TCP sockets that do share code with TCP and use ULP to customize socket behavior. Even with as much MPTCP-specific code as possible isolated in the MPTCP socket and its ULP functions, a few targeted changes have been needed in the TCP code:

ULP In order to clone a listening socket while using ULP, a hook has been added for ULP to set `icsk_ulp_data` immediately after cloning. Without this modification, under some conditions the clone is deleted early and corrupts the listener's ULP state.

TCP functions exported MPTCP needs low-level access to `tcp_push()`, `tcp_send_mss()`, and `struct tcp_request_sock_ops`.

TCP coalesce/collapse SKBs with MPTCP extensions can't be coalesced or collapsed because per-SKB extension data would be lost. Additional checks in the code verify that the MPTCP SKB extension is not used before coalescing or collapsing SKBs.

TCP option parsing When parsing TCP options, there's an additional switch case to handle the MPTCP option.

TCP option writing The option header writing code will write the MPTCP option when needed. When MPTCP needs option space, it takes priority over SACK when allocating space in the TCP option field.

MPTCP socket flag `struct tcp_sock` and `struct tcp_request_sock` gain an `is_mptcp` bit each for tracking whether the socket is an MPTCP subflow.

TCP minisocks One additional if statement to handle subflow connections differently in `tcp_check_req()`

Received SKB handling Call out to MPTCP code from `tcp_data_queue()` to attach the MPTCP SKB extension and process MPTCP options on ACK packets before they are freed.

Additional structure members for receive options `struct tcp_options_received` has grown to hold MPTCP option values after they are parsed from the header. This can utilize some unions to make more efficient use of memory.

Coupled Receive window Receive window sharing between subflows has not been implemented yet. This is required by the RFC as described in Section 1.3 and changes to TCP receive window code will be needed in order to send the correct window value.

2.9 Changes already upstreamed

Some changes to support MPTCP also benefit other parts of the `net` subsystem, and are good candidates for merging before MPTCP.

SKB Extensions The SKB Extensions feature has already been merged to add less-frequently used extended information to the `sk_buff` structure [19]. This uses a pointer and some flags to manage an extra block of storage and has already helped shrink the `sk_buff` structure by removing two pointers. SKB Extensions are used by MPTCP to store information to be read from or written to a TCP option associated with the data payload. More specifically, it is needed to map logical MPTCP sequence numbers to the TCP sequence numbers used by individual subflows. When a socket is using MPTCP this DSS mapping is read from TCP option space on receive, and written to TCP option space of transmitted packets. Regular TCP skbs do not change in size when MPTCP is configured or in use on other sockets.

TCP ULP diagnostic The TCP ULP framework has been extended [20] to allow userspace tools like `ss` to retrieve some information. Today this is used by kTLS [16] to expose the protocol version and the cipher in use.

3 Advanced Features Roadmap

Once the initial patchset described in section 2 is reviewed, revised, and merged, the community group will focus on more advanced features. Here is a list of features that the MPTCP upstreaming community has planned for. Other good ideas are certain to arise in the future, and will be prioritized as work continues.

3.1 Userspace path manager

The MPTCP specification describes how two peers share information about the IP addresses and ports they have available for additional subflow connections. The peers are allowed full flexibility in determining how many subflows are created and which TCP endpoints are involved. The component that handles subflows creation and acceptance for an existing MPTCP connection is known as a "path manager".

One MPTCP peer can inform the other of additional IP addresses (and, optionally, port numbers) to connect to, using the `ADD_ADDR` TCP option subtype. It can cancel an advertised address using `REMOVE_ADDR`. In either case, the information is provided to the configured path manager so it can be used in combination with local policy to initiate subflow connections to the peer. Once an MPTCP connection is made, either peer may initiate a new subflow using the `MP_JOIN` option subtype.

In practice, many MPTCP connections involve a server with one or more internet-accessible IP addresses and a client. The server will likely handle a large number of incoming connections to one IP address, while the client will make outgoing connections, have multiple network interfaces, and is likely to be be behind a NAT (network address translator) and firewall. This topology lends itself to implementing the path manager in userspace, since a client device will be responsible for initiating new subflows through the NAT/firewall and will have a smaller volume of userspace/kernel communication than a server.

As the community group is focusing on the server use case for the initial upstream patch set, a full-featured path manager will not be included in this patch set. A server can implement a simple policy such as a configurable limit on subflows-per-connection without calling out to userspace. Looking ahead to later features, a generic netlink API for communication between the kernel and the path manager has already been defined. This API has been adopted by the out-of-tree Linux MPTCP kernel [1, 23] and is used in production [24]. A Multipath TCP userspace Daemon (`mptcpd`) has been open-sourced [25].

`mptcpd` has very few dependencies and can run on systems ranging from embedded devices up to large servers. There are quite a few network management components in use on Linux-based platforms, and the community group envisions `mptcpd` as a reference implementation that can be used as-is or as an example for integration of similar capabilities in to other open source projects.

3.2 Packet scheduling

An MPTCP scheduler selects the subflow on which to transmit a segment of data. Sending the same segment on only one subflow makes more efficient use of bandwidth, but data can be transmitted in parallel on multiple subflows for redundancy or error recovery. An advanced scheduler might make decisions based on round trip time, latency, congestion, throughput, or radio information. Each host also sends a per-subflow "backup" flag to its peer to indicate that a specific subflow is only to be transmitted on if no other paths are available. Other MPTCP implementations have made alternate schedulers available as kernel modules. Another approach would be to add eBPF-based schedulers to allow MPTCP transmissions to be tuned for specific applications without building kernels or kernel modules. These architectural options are not mutually exclusive: the kernel could have a couple of simple, default schedulers built in and also have customization available with eBPF.

3.3 Allowing unmodified binaries to use MPTCP

Some users would like to use MPTCP with existing binaries built for TCP. A kernel that defaulted to using MPTCP for all TCP sockets would accommodate this need, but might also add unintended overhead to other connections. In order to run only certain binaries with MPTCP substituted for TCP, a cgroup-specific eBPF hook could be introduced to modify the parameters for the `socket()` call within a given cgroup. This is similar to hooks provided for `bind()` and `connect()`.

3.4 Performance optimizations

With a baseline implementation available to a broad user base, it will work well to gather performance data from the developers and user community and focus optimization efforts on those important areas.

There will also be opportunities to fine-tune MPTCP-specific network performance. For example, when multiple subflows are active, then different scheduling choices for retransmissions may impact throughput in the presence of errors.

TCP Fast Open also requires special handling to co-exist with MPTCP. While it is not expected to have this in the initial feature set, it will be important to add.

3.5 Break-before-make

While a typical MPTCP connection has at least one subflow, it is possible for both peers to keep a connection active even if all of the subflows are closed with regular TCP FINs. The connection can exist in this zero-connection state for a brief period of time, during which either side can send `MP_JOIN`s to re-establish subflows and continue the MPTCP session. This is known as "break-before-make". The length of time in this MPTCP-level `TIME_WAIT` state is up to the implementation, and may be zero. While the initial implementation may close the MPTCP socket immediately in the interest of simplicity, it is expected to add break-before-make in a later iteration.

3.6 Subflow information and configuration

An MPTCP-aware application may want to make per-subflow configuration choices as introduced in Section 2.3.

The MPTCP-level socket manages the set of TCP subflow sockets that it creates, so userspace does not have direct `setsockopt()` / `getsockopt()` access for per-subflow configuration or information retrieval. If set directly by the application at the subflow level, some options could also interfere with the operation of MPTCP, by consuming too much TCP option space, or by changing buffering behavior in a way that is not compatible with MPTCP's transmit data scheduling. Given those considerations, it is planned for the MPTCP socket to act as an intermediary for per-subflow socket options. It can provide an interface for setting a socket option for a specific subflow, and only allow access to whitelisted socket options.

3.7 kTLS

Kernel TLS [16] offload would require significant work at multiple layers to be used with MPTCP, and may not support some types of hardware acceleration.

While MPTCP sockets are deliberately made to behave as much like TCP sockets as possible to simplify application compatibility, MPTCP and TCP sockets consist of entirely different code. The ULP hooks defined for TCP do not work with non-TCP sockets, so both a new generic MPTCP ULP layer and MPTCP-oriented ULP hooks would have to be added.

kTLS operating in `TLS_SW` mode appears feasible. On the transmit side, record framing and encryption could be performed before handing data off to subflows. This allows MPTCP to correctly set the sequence numbers in DSS mappings. When receiving data, MPTCP would reassemble the encrypted data stream across all the subflows, which could then be handed off to the TLS decryption layer.

There are two main challenges with `TLS_HW` and MPTCP. The first is that record framing bytes inserted or removed by hardware would interfere with MPTCP DSS mappings, which assume knowledge of TCP sequence numbers. Another issue is that the MPTCP data stream is potentially split across multiple network interfaces, creating problems with receiving parts of a TLS record on different subflows and with skipped or repeated TLS records in one specific subflow.

Given these considerations, the MPTCP upstream community is interested in feedback on demand for `TLS_SW`-only kTLS for MPTCP before spending time on the project.

Conclusion

With time and effort the MPTCP Upstream community group believes that Multipath TCP can be successfully integrated with the networking subsystem. Focusing on essential features first will lead to the best feedback from the wider upstream community, and result in a more maintainable and robust MPTCP implementation as work continues on both TCP and MPTCP.

Linux MPTCP Upstream project

The goal of this community is to add an implementation of the MPTCP protocol to the upstream Linux kernel. This project is open to everybody. Discussions happen on a dedicated mailing list and during weekly meetings on an opened platform. Please use these addresses and sites to give feedback and start discussions about this paper and the MPTCP upstreaming project.

- Website and Wiki: https://github.com/multipath-tcp/mptcp_net-next/wiki
- Mailing list: https://lists.01.org/mailman/listinfo/mptcp
- Git repository: https://github.com/multipath-tcp/mptcp_net-next

People actively working on this project are, in alphabetical order:

- Paolo Abeni (Red Hat)
- Matthieu Baerts (Tessares)
- Davide Caratti (Red Hat)
- Peter Krystad (Intel)
- Mat Martineau (Intel)
- Ossama Othman (Intel)
- Christoph Paasch (Apple)
- Florian Westphal (Red Hat)

Acknowledgments

We would like to thanks Tim Froidcoeur, David Verbeiren, Peter Krystad and Ossama Othman for their useful reviews.

References

[1] Paasch, C., Barre, S., et al. Multipath TCP in the Linux Kernel. Available from http://www.multipath-tcp.org.

[2] Bonaventure, O. and Seo, S. Multipath TCP Deployments. Available from https://www.ietfjournal.org/multipath-tcp-deployments/.

[3] Ford, A., Raiciu, C., Handley, M., and Bonaventure, O. TCP Extensions for Multipath Operation with Multiple Addresses, version 0. RFC 6824, January 2013.

[4] Hesmans, B., Duchene, F., Paasch, C., Detal, G. and Bonaventure, O. Are TCP extensions middlebox-proof?. In *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization* (pp. 37–42), ACM, 2013.

[5] Martineau, M., Baerts, M., Paasch, C. and Krystad, P. How hard can it be? Adding Multipath TCP to the upstream kernel Available from https://www.files.netdevconf.org/d/4de7ab0bd1d543eb95b9/.

[6] Raiciu, C., et al. How hard can it be? Designing and implementing a deployable Multipath TCP. In *NSDI'12* (Berkeley, CA, USA), USENIX Assoc., pp. 29–29.

[7] POSTEL, J. Transmission control protocol. RFC 793, 1981.

[8] BONAVENTURE, O. Apple pushes Multipath TCP further Available from https://www.tessares.net/highlights-from-advances-in-networking-part-1-wwdc19/.

[9] WIKIPEDIA CONTRIBUTORS Hybrid Access Networks Available from https://en.wikipedia.org/wiki/Hybrid_Access_Networks, visited on 30rd of August.

[10] BROADBAND FORUM MEMBERS Hybrid Access Broadband Network Architecture, TR-348 Available from https://www.broadband-forum.org/technical/download/TR-348.pdf.

[11] BONAVENTURE, O. AND MEN, N. Opening the way for convergence in the 5G era with MPTCP Available from https://www.tessares.net/news-room/papers/.

[12] BONAVENTURE, O., PAASCH, C., AND DETAL, G. Use Cases and Operational Experience with Multipath TCP. RFC 8041, January 2017.

[13] APPLE XNU KERNEL DEVELOPERS. Multipath TCP in Apple's XNU Kernel. Available from https://github.com/apple/darwin-xnu/tree/master/bsd/netinet.

[14] FREEBSD KERNEL DEVELOPERS. Multipath TCP in FreeBSD Kernel. Available from https://www.freebsdfoundation.org/project/multipath-tcp-for-freebsd/.

[15] PAASCH, C., DETAL, G., DUCHENE, F., RAICIU, C., AND BONAVENTURE, O. Exploring mobile/Wi-Fi handover with multipath TCP. In *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design* (pp. 31–36), ACM, August 2012.

[16] WATSON, D. KTLS: Linux Kernel Transport Layer Security Available from https://netdevconf.org/1.2/papers/ktls.pdf.

[17] FORD, A., RAICIU, C., HANDLEY, M., BONAVENTURE, O., AND PAASCH, C. TCP Extensions for Multipath Operation with Multiple Addresses Available from https://datatracker.ietf.org/doc/draft-ietf-mptcp-rfc6824bis/.

[18] PAASCH, C. RFC6824bis handshake implementation Available from https://sympa-2.sipr.ucl.ac.be/sympa/arc/mptcp-dev/2019-04/msg00003.html.

[19] WESTPHAL, F. `sk_buff`: add extension infrastructure Available from https://lore.kernel.org/netdev/20181210145006.19098-1-fw@strlen.de/#t.

[20] CARATTI, D. net: tls: add socket diag Available from https://lore.kernel.org/netdev/cover.1567158431.git.dcaratti@redhat.com/#t.

[21] CARDWELL, N., CHENG, Y., BRAKMO, L., MATHIS, M., RAGHAVAN, B., DUKKIPATI, N., CHU, H., TERZIS, A. AND HERBERT, T. packetdrill: Scriptable Network Stack Testing, from Sockets to Packets Available from https://www.usenix.org/conference/atc13/packetdrill-scriptable-network-stack-testing-sockets-packets.

[22] SCHILS, A., CRECIUN, E. ET AL. Multipath TCP Packetdrill version Available from https://github.com/multipath-tcp/packetdrill_mptcp.

[23] DETAL, G. ET AL. Multipath TCP Netlink API Available from https://github.com/multipath-tcp/mptcp/blob/mptcp_trunk/include/uapi/linux/mptcp.h.

[24] HESMANS, B., DETAL, G., BARRE, S., BAUDUIN, R. AND BONAVENTURE, O. SMAPP: Towards Smart Multipath TCP-enabled APPlications Available from https://www.tessares.net/wp-content/uploads/2016/06/SMAPP-Towards-Smart-Multipath-TCP-enabled-APPlications.pdf.

[25] OTHMAN, O. ET AL. The Multipath TCP Daemon: `mptcpd` Available from https://github.com/intel/mptcpd.