Towards Evaluating Multipath TCP using Linux Tools and Utilities

Suvam Mukherjee, Abhinaba Rakshit, Dayma Khan, Mohit P. Tahiliani
Wireless Information Networking Group (WiNG)
National Institute of Technology Karnataka, Surathkal, Mangalore, Karnataka - 575025, India
(suvammukherjee.217cs011,abhinaba.212cs002,daymakhan.212cs006,tahiliani)@nitk.edu.in

ABSTRACT

Multipath TCP (MPTCP) is a transport protocol standardized in RFC 8684 and is an ongoing research topic at IETF. The implementation of MPTCP is under active development in the Linux network community. Performing experiments with MPTCP using the Linux network stack requires installing external tools and utilities. This paper analyzes the tools and utilities that are popularly used for MPTCP experimentation using the Linux network stack. Subsequently, using these tools and utilities, we perform a deep-dive analysis of the implementation of the connection establishment phase of MPTCP in the Linux kernel by conducting experiments and comparing whether this implementation is in line with the RFC. We leverage Linux network namespaces to perform experiments because they provide a lightweight alternative to setting up physical testbeds or virtual machines. Moreover, it makes our experiments easily reproducible across different Linux platforms. Our experiments show that the Linux implementation of the connection establishment phase of MPTCP closely follows RFC 8684.

CCS CONCEPTS

• **Networks** → **Transport protocols**; **Network experimentation**.

KEYWORDS

Multipath TCP, Path Manager, Linux Network Namespaces

ACM Reference Format:
Suvam Mukherjee, Abhinaba Rakshit, Dayma Khan, Mohit P. Tahiliani. 2023. Towards Evaluating Multipath TCP using Linux Tools and Utilities. In *24th International Conference on Distributed Computing and Networking (ICDCN 2023), January 4–7, 2023, Kharagpur, India.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3571306.3571426

1 INTRODUCTION

Over the years, there have been multiple ways of improving the performance of TCP. Extending the support of multi-homed and mobile devices has been one such effort. However, traditional TCP uses a single interface per connection for communication. Hence, to leverage the capability of using multiple interfaces simultaneously to obtain a higher throughput, approaches such as VPN bonding [1], Linux bonding [1], and Multipath TCP (MPTCP) [2] can be leveraged. L2 VPN and 802.3ad/LACP (Link Aggregation Control Protocol) features are combined in VPN bonding. LACP is used for Ethernet aggregation. VPN bonding allows VPN tunnels to be added into one logical link and aggregate the total bandwidth of all the VPN links. VPN bonding increases throughput for a single stream and for numerous streams of connection. In Linux bonding, administrators can combine multiple network interfaces into a single logical interface, called a channel bonding interface. The aggregated network interfaces are called controlled devices, and the bonded logical interface is called the controller device [1]. Using Linux bonding, we can see that the controlled device collaboratively works so that the controller device gets more throughput than it used to get in the network. MPTCP also uses the available interfaces between the client and the server and gets more throughput than using a single interface. However, the main difference between MPTCP, VPN bonding, and Linux bonding is that the interfaces in case of MPTCP can be added and removed dynamically. But in the case of Linux bonding, controlled devices have to be aggregated (or bonded) before the data transfer begins. The same holds true for VPN bonding as well. This is one of the primary reasons we explore more about the MPTCP approach in this work. It allows us to simultaneously use multiple interfaces between two end hosts. Using MPTCP, we can use the cumulatively added-up bandwidth of various interfaces, thus providing enhanced performance.

Multipath TCP is a transport layer protocol and its implementation exists in Linux Kernel, which is still an ongoing project. There exist two implementations of MPTCP, namely *out-of-tree* and *upstream*. The out-of-tree implementation supports both MPTCPv0 (RFC 6824) and MPTCPv1 (RFC 8684) [8]. This implementation makes significant modifications to Linux's TCP implementation, and hence is challenging to maintain. Thus, the community designed a new upstream implementation based on MPTCPv1. It has been merged in the Linux kernel from v5.6 onwards [8]. Therefore, we have considered the upstream implementation for this work.

Despite being natively supported in the Linux kernel, enabling and configuring MPTCP for running experiments is still a challenging task. Several tools and supporting packages have been developed for Linux, and are still being designed by the community for the ease of running MPTCP on Linux platforms. In this work, we analyze the tools and utilities that are popularly used for MPTCP experimentation using the Linux network stack. Subsequently, using these tools and utilities, we perform an in-depth analysis of the implementation of the connection establishment phase of MPTCP in the Linux kernel by carrying out experiments and comparing whether it is in line with the RFC. We leverage Linux network namespaces to perform experiments because they provide a lightweight alternative to setting up physical testbeds or virtual machines. Moreover, it makes our experiments easily reproducible across different Linux platforms. Our experiments show that the Linux implementation of the connection establishment phase of MPTCP closely follows RFC 8684.

ICDCN 2023, January 4–7, 2023, Kharagpur, India

ACM ISBN 978-1-4503-9796-4/23/01…$15.00
https://doi.org/10.1145/3571306.3571426

This paper makes the following three contributions: (i) We discuss the popular tools and utilities required for experimenting with MPTCP (mptcp-tools, mptcpize, ip mptcp, and mptcpd) using Linux kernel, briefly explaining their working details, use-cases, and the difference between them. (ii) We practically demonstrate how these tools can be used for MPTCP socket creation and path management while performing MPTCP-related experiments using the Linux network namespaces, which are lightweight and scalable alternatives to a physical testbeds or virtual machines. (iii) We verify whether the connection establishment phases of the upstream implementation of MPTCP is in compliance with RFC 8684 [2].

2 LINUX TOOLS AND UTILITIES FOR MPTCP

This section discusses the working of popular tools and utilities that are required for experimenting with MPTCP using Linux network stack. The list includes: mptcp-tools, and mptcpize, which enable native applications to create MPTCP sockets. In addition, tools like ip mptcp and mptcpd (MPTCP Daemon) allow users to configure MPTCP for their sub-flows. Besides, we briefly describe the in-kernel and userspace Path Manager and highlight their differences.

2.1 mptcp-tools and mptcpize

MPTCP in the Linux kernel can be enabled using *sysctl* variables. By default, 'net.mptcp.enabled' is set in the Linux kernel. However, this is not sufficient to use MPTCP because the applications generally, by default, create TCP sockets. In the Linux kernel, the *socket* API is used to create sockets, and it expects three parameters to be passed. One of the parameters passed to *socket* API is 'Protocol Number'. Legacy TCP-only applications use the 'Protocol Number' as - IPPROTO_TCP (6) or IPPROTO_IP (0). Popularly used tools for traffic generation in Linux, like iperf3 and netperf, can create UDP or TCP traffic by calling the socket API with Protocol numbers 5 (IPPROTO_UDP) and 6 (IPPROTO_TCP). Similarly, they support several other protocols using the IANA standardized protocol numbers. The Linux network community has assigned protocol number 262 for MPTCP, declared as a variable IPPROTO_MPTCP, [3]-[4], using which an application can create an MPTCP socket. However, currently, iperf3 and netperf do not support the creation of MPTCP sockets. Hence, an external application is required to create the MPTCP sockets explicitly. The tools such as mptcp-tools and mptcpize allow the applications to create MPTCP sockets instead of TCP sockets. The flowchart describing the working of mptcp-tools and mptcpize is shown in Fig. 1.

Since mptcp-tools is an external utility not directly linked with iperf3/netperf, we need to use the *usemptcp.sh* script, which is provided by the developers of mptcp-tools, to start using mptcp-tools. On executing *usemptcp.sh* script, a C file wrapper is compiled, generating a library file *usemptcp.so* as an output. The *wrapper* file redefines the socket API, which transparently replaces the creation of any TCP sockets by MPTCP sockets. This library file is loaded in the runtime environment of iperf3/netperf using LD_PRELOAD. Once it sets LD_PRELOAD to the path of the object module (library *usemptcp.so*), this library is loaded into the program before any other library. It also overrides any other standard C libraries. Hence, subsequently, any call to the *socket* API would make a call to the custom API defined in the wrapper file.

As a result, if there is a request to create TCP sockets (Protocol Number = 6) or default sockets (Protocol Number = 0), the API updates the protocol number to (256 + IPPROTO_TCP) and executes a system call for MPTCP socket creation. For protocols other than TCP, it will create a socket as specified without any update.

Unlike mptcp-tools, mptcpize comes as a package that can be installed into the kernel. Installing it adds a binary into the */usr/bin* directory, so it can be used as a command and executed from any location in the system. mptcpize provides several options (such as run, enable and disable) to be used along with it. The *run()* API is used to enable an application to create MPTCP sockets. It achieves the objective of converting the TCP socket creation to MPTCP sockets using the same technique of LD_PRELOAD as mentioned earlier. mptcpize also provides a pre-compiled library file called *libmptcpwrap.so* in its package, which internally redefines the *socket* API. This library file is loaded by the *run()* API to LD_PRELOAD. Finally, the application is executed by *execvpe()*. This causes any call to *socket* API to be redirected to the custom API, which then undertakes the task of creating MPTCP sockets similar to mptcp-tools as shown in the Fig. 1. The primary difference between mptcp-tools and mptcpize is that mptcp-tools is to be used explicitly whereas mptcpize can be installed into the kernel.

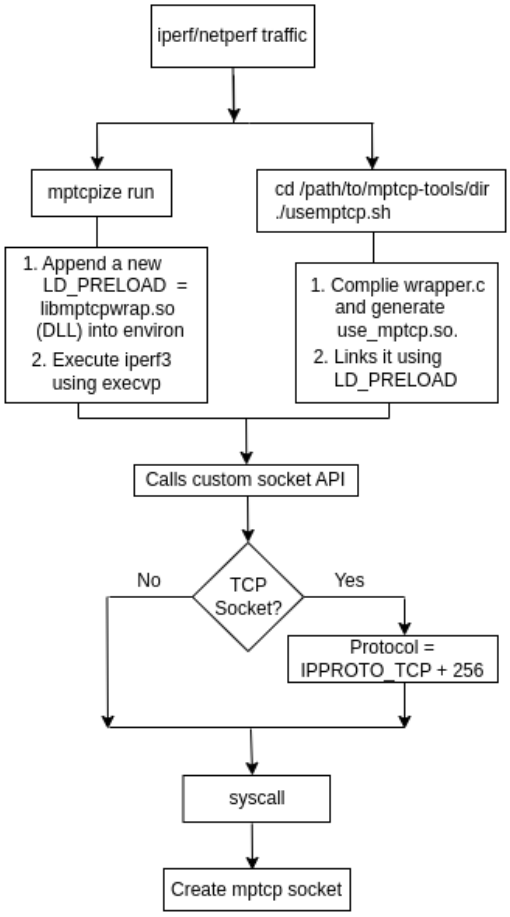

Figure 1: Flow-chart of mptcpize and mptcp-tools

2.2 ip mptcp and mptcpd

One of the essential aspects of MPTCP is that it generates multiple sub-flows, which need to be managed efficiently. To achieve this, the MPTCP path manager plays a crucial role in setting up numerous sub-flows. Path manager (PM) is an essential component of MPTCP which determines when the additional sub-flows are to be created or removed, and controls the advertisement of available addresses to the peers. It also determines the addresses used to create the sub-flows [9]. mptcpd [7] and ip mptcp are the tools that can configure several aspects of the MPTCP path manager. The upstream implementation of MPTCP supports two types of PM: in-kernel and userspace. A new *sysctl* variable called 'pm_type' has been introduced in the kernel version 5.19, which specifies the type of PM to be used. For example, 'pm_type = 0' (in-kernel path manager) and 'pm_type = 1' (userspace path manager). Fig. 2 shows the difference between the in-kernel and userspace PM.

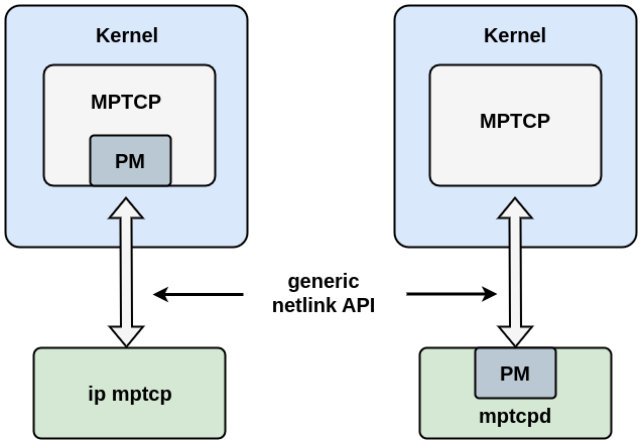

Figure 2: In-kernel vs Userspace Path Manager

In-kernel PM is implemented in the kernel-space and covers the common use cases (e.g., default, full-mesh). In case of an in-kernel path manager, the kernel applies the same configuration to all the MPTCP connections. It is controllable via ip mptcp. ip mptcp is a part of the 'iproute2' suite in Linux. It can be used to configure the number of sub-flows a host can create or accept using the limits object, and can also initiate the sub-flow creation and signal the availability of new addresses using the 'ADD_ADDR' subtype.

In order to avoid maintaining multiple path managers in the kernel space, the customization part is moved into the userspace. This makes it easier to develop custom path managers without affecting the stability of the kernel. Userspace Path Manager can communicate with the kernel using MPTCP generic 'Netlink APIs' [4]. Using this, the userspace application, like mptcpd, can ask the kernel to create new sub-flows per connection. It gives users the flexibility to cover more specific use cases.

mptcpd is a tool that allows the users to implement and use userspace path managers. It makes use of plugins to be loaded during the run-time. The communication between the userspace path manager and kernel takes place using MPTCP generic 'Netlink APIs'. The kernel informs the plugins about the kernel events (e.g., addition of new sub-flow, advertisement of new sub-flow) [4]. The plugins can also command the kernel to carry out path management related activities based on these events. Currently, mptcpd supports *sspi* (single-subflow-per-interface) and *addr_adv* (address-advertisement) plugins and several features in it are still under development [7]. Users can implement their own plugins as per the required behavior and use it as PM logic.

3 EMULATING MPTCP USING LINUX KERNEL

In this section, we evaluate the connection establishment phase of MPTCP using the Linux kernel. We use the tools and utilities discussed in the previous section to perform experiments with MPTCP. Network emulations can be performed in different ways. The most commonly used approach is to set up a physical testbed or by using Virtual Machines (VMs) that resembles the desired network. However, they are very expensive in terms of resources, and hence not preferred for large network topologies. Linux network namespaces are a cost-effective and scalable alternative to physical testbeds and VMs. Using Linux network namespaces, we can quickly replicate the network components and reproduce the results.

In our experiments, we use the topology as shown in Fig. 3 containing two distinct paths between the client and server. Hence, it provides an ideal set up for verifying the working of MPTCP.

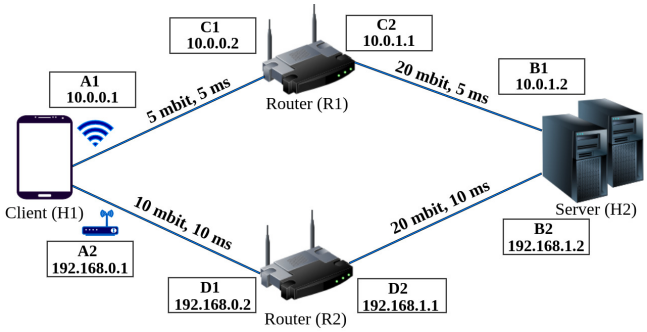

Figure 3: Network Topology

Prior to running experiments, MPTCP should be enabled in the kernel of both end hosts. Besides, *reverse path filtering* is enabled by default in the Linux kernel. It must be disabled to prevent the kernel from dropping packets that are not routable through the given interface so that packets reach their destination interface. Hence, we configured the 'rp_filter' variable using *sysctl* to disable the reverse path filtering.

In the case of MPTCP, a routing table needs to be created for each address. Hence, we created two routing tables for both: the client and the server [5]. Client (H1) consists of two routing tables, table 1 for address - 10.0.0.1 (A1) and table 2 for address - 192.168.0.1 (A2), such that any packet with source address A1 will refer to routing table 1 for forwarding the packet, and packets with source address A2 will refer to routing table 2. Similarly, server (H2) consists of two routing tables, table 3 for address - 10.0.1.2 (B1) and table 4 for address - 192.168.1.2 (B2). The script developed by us for setting up this topology is provided in this GitHub repository[1].

[1]https://github.com/abhinaba-fbr/mptcp_linux_kernel_experiments

Initially, we used mptcp-tools to force the application to create an MPTCP socket. However, we noticed that the same thing can be achieved using mptcpize in a more straightforward way. Hence, we switched to using mptcpize subsequently. iperf3 is used for generating traffic between the client and the server. After running the experiment, we observe the following behavior, as depicted in the following figures: Fig. 4, Fig. 6 and Fig. 7.

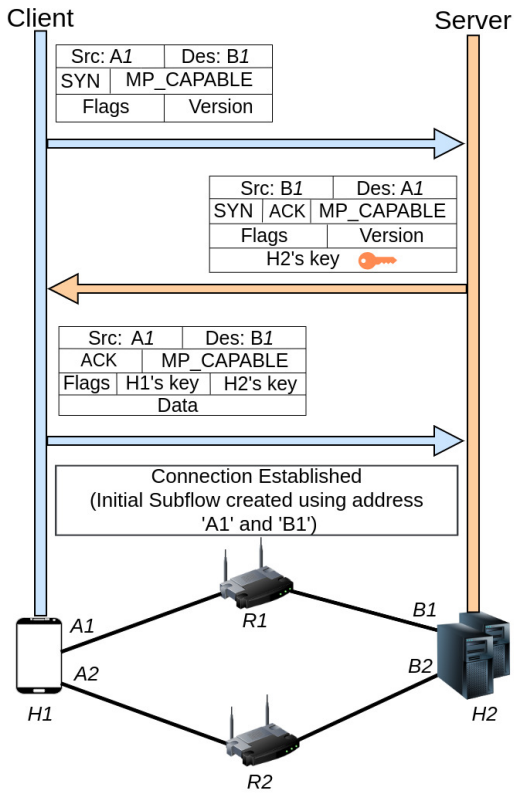

Figure 4: MPTCP connection establishment phase

The primary distinction between TCP and MPTCP connection establishment is that TCP options field is used for MPTCP as shown in the Fig. 5. Any subsequent MPTCP related communication is done through the MPTCP option. The Kind field is 8 bits, and a value of 30 has been assigned for the MPTCP option.

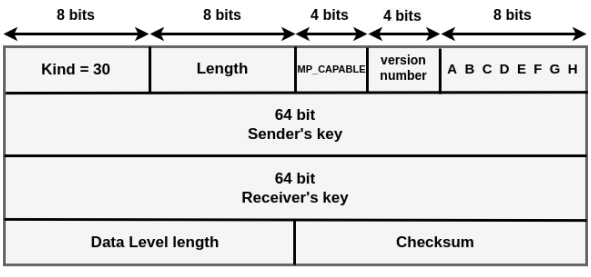

Figure 5: MPTCP option with MP_CAPABLE subtype

The initial MPTCP connection establishment process is described in Fig. 4. When the client wishes to connect, it sends a SYN packet to the server. According to RFC 8684, IANA maintains a new sub-registry using the 4-bit subtype field in the MPTCP option. The subtype field in the initial SYN packet consists of MP_CAPABLE. The MP_CAPABLE option is mainly used for two purposes. First, it checks whether both endpoints support MPTCP or not. Second, it is used to exchange authentication keys and cryptographic materials between peers. Hence, in the SYN packet, along with MP_CAPABLE, the version number and certain flags are also shared with the server.

On receiving the SYN packet, the server generates a random key (H2's key) to be sent along with the SYN+ACK packet. The key generation process could be implementation specific. A 32-bit token is generated using this key. This token is subsequently used to identify the connection for all the future sub-flows.

On receiving the SYN+ACK packet, the client knows whether the server is also MPTCP capable or not, and it gets the server's key if it is MPTCP capable. Subsequently, the client makes use of the ACK packet to share its own generated random key (H1's key) with the server, which is later used for authentication purposes. The client can piggyback data, along with the ACK packet, if it has some data to be sent immediately.

The client and server perform a three-way shake to initiate the MPTCP connection by adding the MP_CAPABLE subtype in its header. In our experiment, addresses 10.0.0.1 (client) and 10.0.1.2 (server) are used for setting up the initial sub-flow (also called master sub-flow). The exchange of packets observed during the initial connection establishment phase is shown in Fig. 4 on capturing the traffic in Wireshark. The observed behavior is the same as defined in RFC 8684.

When new sub-flows are set up between the same sender and receiver, the keys exchanged in the MP_CAPABLE option during the initial connection establishment play an important role in authenticating the connection. The additional sub-flows also begin with the same SYN, SYN+ACK, and ACK packets, but there is a difference in the MPTCP subtype that is used while setting up additional sub-flows. Instead of using the MP_CAPABLE subtype, a separate MPTCP option subtype called MP_JOIN is used. Firstly, it is used to authenticate the connection using the tokens and authentication algorithms. Once the authentication is successful, a new subflow is set up between the client and the server.

In order to use the additional interfaces, we tried to create an additional sub-flow using the MP_JOIN subtype. There can be two scenarios for setting up a new sub-flow: (1) Server advertises its unused IP address to the client. Subsequently, the client tries to initiate a new sub-flow creation using this advertised address. (2) Client can initiate a new sub-flow creation itself using its unused IP address. However, in both scenarios, we observed a failure in setting up a new sub-flow, as shown in Fig. 6 and Fig. 7. The reason for failure is explained below.

The Fig. 6 demonstrates Scenario 1 in detail. Once the initial connection is established, and the master flow is created between address A1 (on the client side) and address B1 (on the server side), the server identifies its additional address and advertises it to the client. Here, H2 (server) has an unused address, B2. So, H2 advertises the unused address to the client using the ADD_ADDR subtype. The ADD_ADDR subtype is used for advertising addresses to peers.

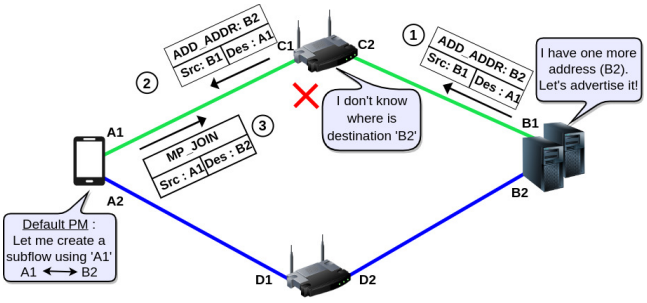

Figure 6: First failure of sub-flow addition

The address B2 is encapsulated along with the ADD_ADDR subtype in the options field and sent via the initial subflow.

The client maintains a list of all the available IP addresses of the server. On receiving the advertised address, the client includes address B2 in this list. It then tries to create a new subflow using this advertised address, using MP_JOIN. Hence, B2 is used as a destination address in the IP header.

However, the other address (source address) is determined based on a special component of MPTCP called the Path Manager. Since the path manager requires explicit configuration, the Default Path Manager is used in Linux. The Default Path Manager does not actively manage any subflows. It picks the other address from the addresses used in the initial subflow setup. Therefore, address A1 is selected as the source address in this case. The source address 10.0.0.1 (A1) is routed as per routing table 1 to router R1. R1 does not find any route for the destination address B2, so it drops the packet and sends back an ICMP (destination unreachable) message. This is the primary cause of failure in Scenario 1.

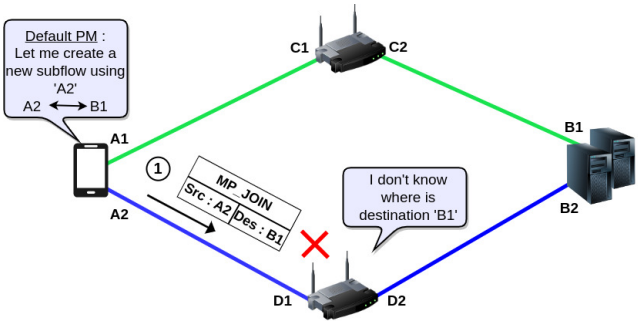

Figure 7: Second failure of sub-flow addition

In Scenario 2, the client proactively initiates a new subflow without any indication from the server. In this case, we configured the client to initiate a new subflow using the unused address A2. So, the client sends a MP_JOIN packet with A2 as the source address. But, similar to the previous scenario, the Default Path Manager will select the initial sub-flow address B1 as the destination address for subflow creation. Due to the source address being 10.0.0.1 (A2), as per routing table 2, the packet is routed to R2. Again, R2 does not find any route for the destination address B1, and the packet is dropped, thus leading to the failure of Scenario 2. This clearly shows that configuring the MPTCP path manager is a crucial step for the appropriate functioning of MPTCP.

There are two ways of configuring the path manager. In-kernel PM can be configured using ip mptcp and userspace PM can be configured using mptcpd. Currently, in-kernel path manager supports 'fullmesh', the most commonly used PM. It tries to create sub-flows to all the known server IP addresses using the client's specified address. The support for configuring 'fullmesh' PM using ip mptcp has been added recently. On setting the 'fullmesh PM' for sub-flow creation, we observed the expected behavior as shown in the Fig. 8.

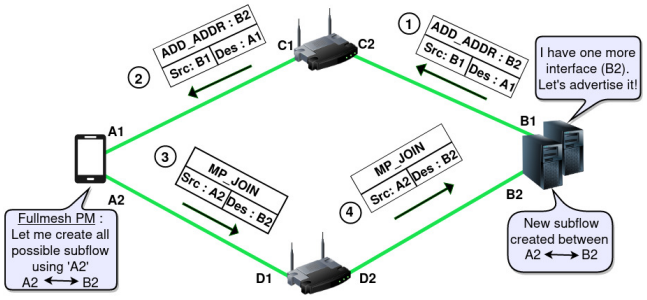

Figure 8: Successful joining of additional sub-flow

Once the initial connection is established, the server advertises its unused address (B2) to the client through the initial connection. The client stores the address in its address list. Since the Full-mesh PM is configured with A2 as the source address, the client now tries to initiate subflow to both the addresses B1 and B2. In this situation, two MP_JOIN packets will be sent with the same source address (A2) and different destination addresses (B1 and B2). Since the source address is the same, both will be routed to R2. The packet having destination address B1 will be dropped (destination unreachable). However, the packet with destination address B2 will be correctly routed to the server H2. Subsequently, a new subflow is created between addresses A2 and B2 successfully.

Since two subflows are now set up via different routes, their cumulative bandwidth can be used to achieve higher throughput.

Inferences

From our experiments, we draw the following inferences: first, we observed an increase in throughput on running the above experiments using MPTCP. We obtained an average throughput of 13.4 Mbits (89.3% of bottleneck bandwidth) on running the experiment for 20 iterations, which is higher than what is obtained using traditional TCP. This shows the usefulness of MPTCP in terms of aggregating the bandwidth available on multiple paths.

Table 1: Throughput observed using iperf3

Interval (Sec)	Bitrate (Mbits/sec)	Host
0.00-10.00	16	Sender
0.00-10.70	13.4	Receiver

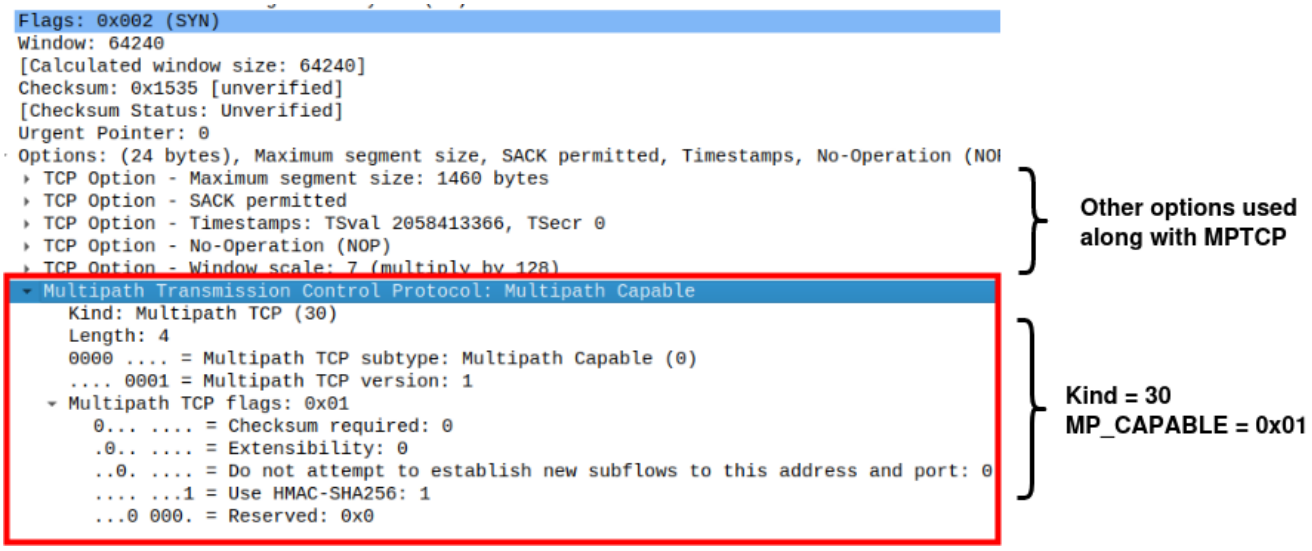

Figure 9: Snapshot of a SYN packet with MPTCP option

Second, we validated the MPTCP option fields in the packets transmitted between the client and the server using Wireshark. We specifically validated whether the connection establishment phase of MPTCP in the Linux kernel is in compliance with the RFC. As per RFC 8684, the kind value is 30 in the MPTCP option field, and MP_CAPABLE (0x01) is used as a MPTCP subtype message to specify initial connection establishment. The packet shown in the Fig. 9 is an instance of a SYN packet for the MPTCP connection establishment phase captured during our experiment.

Lastly, we verified the negotiation of several flags used in the connection establishment phase. MP_CAPABLE consists of flags from A-H. According to RFC 8684 [2], flag A is used for checksum negotiation, B is currently reserved for extensibility, and flag C set indicates the sender will not accept further sub-flow additions using this address. Flags from D-H indicate the use of cryptographic algorithms for authentication. Currently, D-G is unused, and H refers to the use of the HMAC algorithm. Fig. 9 shows a SYN packet having flags A and C set to 0. These flags can also be configured using the sysctl variables (checksum_enabled and allow_join_initial_addr_port).

4 CONCLUSIONS AND FUTURE WORK

In this work, we explored various popular tools and utilities that can be used to experiment with MPTCP in the Linux kernel. Specifically, we discussed four tools and their working details and even showed practical experimentation using them. The scripts developed for these experiments have been made openly available in the GitHub repository[2]. We leveraged Linux network namespaces to demonstrate a convenient way to perform MPTCP-related experiments. This verifies that MPTCP could indeed be emulated in the Linux network namespaces. Subsequently, we verified the connection establishment phase of MPTCP Linux kernel implementation and checked its compliance with RFC 8684. Our experimentation shows that upstream Linux kernel implementation of connection establishment phase of MPTCP is in line with the RFC 8684. We believe that this work can help the research community to quickly understand the tools and utilities related to MPTCP in Linux, and effectively perform experiments using them.

Furthermore, the userspace path manager could be configured using mptcpd. As part of our future work, we plan to implement plugins so that the topology used in our work can be emulated using mptcpd as well. Lastly, we can explore the cryptographic keys, flags, and authentication algorithms which plays an essential role in the sub-flow addition process.

[2]https://github.com/abhinaba-fbr/mptcp_linux_kernel_experiments

ACKNOWLEDGMENTS

We would like to thank Addhyan Malhotra, Ishaan Singh, Rakshita Varadarajan and Gaurang Velingkar from National Institute of Technology Karnataka, Surathkal, India for helping us with the preliminary scripts to perform the experiments.

REFERENCES

[1] S. Aust, J. -O. Kim, P. Davis, A. Yamaguchi and S. Obana, *Evaluation of Linux Bonding Features*, "2006 International Conference on Communication Technology", 2006, pp. 1-6, doi: 10.1109/ICCT.2006.341935
[2] A. Ford, C.Raiciu, M.Handley, O.Bonaventure, C.Paasch *TCP Extensions for Multipath Operation with Multiple Addresses.* "RFC 8684", United States, 2020
[3] B. Hesmans, O. Bonaventure *An Enhanced Socket API for Multipath TCP.*
[4] M. Martineau and O. Othman *Using Upstream MPTCP in Linux Systems.* "Intel", United States of America
[5] Paasch, C., Barre, S., ET AL. Multipath TCP in the Linux Kernel. Available from https://www.multipath-tcp.org/
[6] P Abeni, mptcp-tools Available from https://github.com/pabeni/mptcp-tools.
[7] Intel Multipath TCP Daemon, Available from https://github.com/intel/mptcpd.
[8] Upstream Linux kernel implementation Available from https://github.com/multipath-tcp/mptcp_net-next/wiki/.
[9] Martineau, Mat, and Matthieu Baerts. *Multipath TCP Upstreaming.* "In Indico". 2019.
[10] Server Security Reverse Path Forwarding, Security Guide, Red Hat Enterprise Linux Available from Reverse Path Forwarding
[11] iperf3 Manual, Oracle Available from https://docs.oracle.com/cd/E88353_01/html/E37839/iperf3-1.html