Using Upstream MPTCP in Linux Systems

Mat Martineau and Ossama Othman

Intel
United States of America
mathew.j.martineau@intel.com, ossama.othman@intel.com

Abstract

Multipath TCP patches have been merged for inclusion in the Linux kernel, v5.6 and later. A community effort has been ongoing to develop an upstream-friendly kernel patch set, with patches merged in 2020 that implement MPTCP capability according to RFC 8684 (MPTCP v1). Complementing this set of kernel patches is a userspace reference implementation for the MPTCP generic netlink API called the Multipath TCP Daemon, i.e. "mptcpd". The Multipath TCP Daemon provides an extensible framework for MPTCP path management.

Keywords

MPTCP, Multipath, TCP

Introduction

Multipath TCP (MPTCP) has been added to the upstream Linux kernel during the first half of 2020. This community development effort has involved contributors from Red Hat, Tessares, Apple, and Intel.

Using MPTCP in a Linux system involves kernel configuration, a new type of socket, and system-level runtime configuration. Programs using regular TCP will not be affected by the presence of MPTCP unless they specifically opt in to the multipath protocol.

The addition of this protocol, with its close ties to the TCP subsystem, presented some upstreaming challenges that similar projects can learn from.

The Linux MPTCP community is growing and work continues on more advanced MPTCP features that will address a wider variety of use cases. One area of focus is a userspace component to control how multiple paths are used on a particular system.

Multipath TCP: Protocol and Use Cases

Multipath TCP makes use of multiple, linked TCP connections to carry a single data stream. It is a protocol layer above and intermingled with TCP. RFC 8684 defines how TCP option kind 30 is used to manage MPTCP connections and data sequence numbering. [1] Each of the linked TCP connections is known as a “subflow”.

The most straightforward layering of MPTCP above TCP is in the area of data segmentation and reassembly. The MPTCP option header is used to carry an extra set of sequence numbers and acknowledgements with each TCP data stream. The MPTCP layer provides both outgoing data and the necessary sequence numbers to the TCP stack to use when populating the MPTCP option headers at transmit time. For incoming packets, the subflow data stream and the corresponding MPTCP-level sequence numbers are passed up to the MPTCP layer where the combined data stream is reassembled. Even though the TCP subflows deliver independent reassembled data streams to the MPTCP layer, those streams may contain redundant or out-of-order data for MPTCP to use when reassembling at the upper layer.

The MPTCP option is also used while establishing connections, for the initial subflow or when adding subflows to an existing MPTCP connection. Additional MPTCP-related information is exchanged during the SYN/SYN-ACK/ACK handshake in either case. If the two peers do not both support the same MPTCP version during the initial connection process, a regular TCP connection is created.

During the life of an MPTCP connection, the peers can also exchange information about IP addresses that are available for new connections or that should no longer be used. These messages are also exchanged using the MPTCP header option whether on a data packet or an ACK.

There are some changes to TCP semantics for MPTCP subflows. Most notably, the receive window is shared across all subflows in an MPTCP connection. The use of ACK packets to exchange MPTCP options also requires that duplicate ACKs are not considered a congestion signal.

Use Cases

There are three categories of MPTCP use cases. The first of these are “steering” cases, where MPTCP chooses between available subflows to optimize for the specific application in use. In some cases higher bandwidth may be preferred, while in others low latency is more important. MPTCP can use low-level TCP connection information to choose the best subflow as each packet is transmitted.

When used for “switching”, MPTCP provides quick, seamless handover between multiple networks. For example, the connection can use Wi-Fi when an associated access point is in range and then switch to a LTE or 5G network as the device moves out of Wi-Fi range.

"Splitting" use cases take advantage of the combined bandwidth of multiple network interfaces. Some internet providers use this to combine DSL and LTE for "Hybrid Access" networks that leverage always-on wired networks while having LTE available for peak bandwidth demands.

The 3GPP has recognized MPTCP's usefulness in these areas and specified MPTCP for use in the 5G Access Traffic Steering, Switching, and Splitting (ATSSS) standard.

Protocol Versions and Specifications

The IETF Multipath TCP working group produced two versions of the MPTCP protocol.

MPTCP was initially specified in the experimental RFC 6824 in 2013. [2] This is referred to as "MPTCP v0". While this was not a standards-track document, it was used in major deployments that helped identify how MPTCP could be refined. As of August 2020, most existing deployments use the v0 protocol.

The standards-track RFC 8684 for "MPTCP v1" was published in March 2020. [1] This version modifies the connection handshake to better support TCP Fast Open, makes use of the SHA256 HMAC instead of SHA1, reliably exchanges additional IP address information, and adds an MPTCP-level fast close option using TCP reset packets. The modified connection handshake is incompatible between MPTCP v0 and v1. If the two peers do not support the same MPTCP version, the connection will still succeed as regular TCP. A device that supports both versions can make a second connection attempt using MPTCP v0 if a v1 connection attempt falls back to regular TCP.

The upstream Linux kernel currently supports only the standards-track MPTCP v1. This simplifies the implementation, and this version of the protocol is expected to be the most popular for future deployments.

Feature Support in Kernel Releases

MPTCP was upstreamed in stages, so successive kernel versions have supported additional features.

Kernel v5.6 was the first upstream release to include MPTCP code. Only the foundational code had been reviewed and merged when v5.6 was released, so only single-subflow connections are supported. This does not offer benefits over TCP, so it is better to continue using regular TCP sockets with v5.6 kernels.

Multiple subflow creation was added in Linux v5.7. The initiation and acceptance of additional subflows is controlled by an in-kernel "path manager". The path management parameters are controlled with a generic netlink socket. This kernel version also extends the inet_diag interface used by the '`ss`' command to retrieve information about the TCP subflows within an MPTCP connection.

Linux v5.8 improves MPTCP performance and reliability. The shared receive window handling is significantly improved over v5.7.

Upstreaming Lessons

As the progressive introduction of features starting with kernel v5.6 shows, the introduction of initial MPTCP features into released kernel code took place over several months starting at the beginning of 2020. Preparations for upstream submission began long before that. This process itself has some helpful lessons for open source contributors planning for similarly scoped projects.

Projects that stand to learn most from MPTCP upstreaming will involve significant new functionality that did not fit in the kernel's `drivers/staging/` tree. There will also be close coupling with critical existing kernel code. In the case of MPTCP, this was the TCP subsystem. Also consider that MPTCP was a community effort, with contributors working for several organizations of various sizes.

Cautionary Tales

As with any significant project, the MPTCP upstreaming effort had some learning experiences that didn't accomplish what was desired at the time, but did result in feedback that was helpful in later decisions.

One early example was an attempt to upstream a new framework for extensible TCP options. [3] One goal in the early upstream MPTCP design was to minimize changes to the TCP code. The handling of TCP options for TCP-MD5 and SMC (Shared Memory Communications) were similar to what MPTCP would require, so the MPTCP team crafted a suitable framework and refactored TCP-MD5 and SMC to use it. This patch set was rejected. [4]

In the case of this TCP option framework, the lesson was to not try to design a generic framework up front, but to instead directly code the new functionality. Afterward, if the code would be improved (for example, made simpler or faster) by a generic framework, then such a framework could be designed and upstreamed.

An extension of this lesson is to not spend a lot of time guessing what maintainers do and don't want. Request For Comment (RFC) patch sets can help significantly in this area to get direct answers. Sometimes a brief email posted to the relevant mailing list will get answered, but including code can often help provide something concrete to review. It is still important to strike a balance: feel welcome to engage on the mailing list, but also avoid creating an overwhelming amount of list traffic.

Building An Upstream Community

A project within one organization can have a team assigned by management, but a community-based project often starts small and grows over time. It's worth the effort to make your project known very early on. Reach out on relevant mailing lists. Attend a related conference or, better yet, propose a talk. You may be surprised at who gets involved or cheers you on.

Be aware of the multiple roles that need to be filled by community members. MPTCP benefitted from the participation of domain experts for the new feature,

experienced contributors for the existing upstream code, and automation builders. Projects also need communicators who can keep project members informed and share with the larger community through conference talks and other efforts.

The MPTCP upstreaming team has weekly meetings that have helped strengthen the sense of community and accountability. Face-to-face meetings, even once or twice a year, have been very valuable.

Tool Tips

Communication, patch handling, and continuous integration tools have all played an important role for MPTCP community members working together.

Crafting a patch set, or for MPTCP a group of patch sets, for upstreaming requires frequent rebasing on the upstream git tree. The new patches that are being prepared are also likely to be rearranged and revised over time. The 'topgit' tool was used to good effect by the MPTCP community to manage not-yet-upstream patch series. [5]

The MPTCP team utilizes typical communication tools for an open source project, including a dedicated mailing list (mptcp@lists.01.org), an IRC channel (#MPTCPUpstream on freenode.net), and an issue tracker. [6]

For continuous integration, testing with syzkaller and the kernel's kselftests have been extremely valuable.

Patch Set Partitioning

The baseline multipath TCP code was too large to be submitted and reviewed all at once. The networking maintainer recommended that each patch set sent to netdev be a maximum of 12-20 patches. Most individual patches were under 10 kilobytes in size, a few were on the order of 20 to 30 kilobytes.

For MPTCP, there were four upstreaming phases.

The first phase was to upstream independent building blocks. An example of this is the `skb_ext` code, with related declarations and structures in the Linux kernel source file `include/linux/skbuff.h` . [7] This code was required for MPTCP to build upon, and was also useful for removing some members from `struct sk_buff` . This phase can involve multiple patch sets.

Phase two was a single patch set covering the prerequisite changes to existing code. In the case of MPTCP these were changes to TCP and the networking core. This allowed the maintainers to focus on changes to code they were already familiar with and were likely to have feedback on. It was the patch set with the most revisions required before merging.

After that, the main contributions could begin. The third phase contributed foundational new code. This MPTCP patch set added code that allowed single-subflow MPTCP connections to work. Even though this is not very useful in practice, it met the requirements for patch set size and merged working code that could be expanded upon.

The patch set for phase four established the baseline functionality for MPTCP, allowing creation of multiple subflows. With the time required to prepare, review, and revise patch sets for phases two and three, there was not time to contribute this multipath patch set before kernel v5.6 was released. As a result the changes for this phase first appear in kernel v5.7.

Using MPTCP on Linux

Making use of MPTCP in a Linux system requires proper kernel configuration, application support, and system-level runtime setup

MPTCP is not enabled by default kernel configurations, but only two Kconfig options are used to build MPTCP: `CONFIG_MPTCP` and, optionally, `CONFIG_MPTCP_IPV6`. If MPTCP IPv6 support is enabled, it is not compatible with `CONFIG_IPV6=m` (IPv6 as a kernel module).

Socket Usage

A program selects MPTCP when creating a socket, with a system call very similar to what would be used for TCP:

```
socket(AF_INET6, SOCK_STREAM, IPPROTO_MPTCP);
```

The program then uses the returned file descriptor with the typical socket-oriented system calls to manage connections (`connect()`, `bind()`, `listen()`, `accept()`), and the send/receive functions.

While the intent is for MPTCP sockets to mimic TCP sockets as much as possible to minimize code changes required for existing programs, there are differences between a MPTCP socket and a TCP socket.

As of kernel v5.8, zerocopy is not yet supported for MPTCP sockets. It's a goal to support zerocopy in a future kernel version.

The main area of possible code incompatibility is with socket options. Kernels released to date (up to v5.8) do not support `SOL_TCP` socket options with MPTCP sockets, except when the underlying connection has reverted to regular TCP. `SOL_SOCKET` options are handled in generic socket code in the kernel, so the `setsockopt()` and `getsockopt()` system calls will be handled for all `SOL_SOCKET` options. However, `SO_REUSEPORT` and `SO_REUSEADDR` settings do not propagate to the underlying TCP socket used by MPTCP in kernels up to and including v5.8, so the actual port binding behavior is not changed. A fix for this required some modifications to the core socket code and will arrive in kernel v5.9.

Support for individual `SOL_TCP` options is planned to be added incrementally in future kernel versions. A MPTCP socket creates and controls a set of TCP sockets to handle multiple subflows, and these TCP sockets may be added and removed over time. This creates a number of challenges for managing socket option values set on the overall connection. Some `SOL_TCP` options, or certain values for them, may prevent proper operation of MPTCP. When using `getsockopt()`, some values are straightforward to aggregate from multiple TCP subflows, but some are not. Certain options may need to be cached at

the MPTCP layer and replayed to new in-kernel TCP sockets at the correct time. Given these issues, each option will be separately assessed and implemented in a way that produces sensible MPTCP-level behavior.

Kernel v5.9 will add support for the `SOL_IPV6 IPV6_V6ONLY` option.

The MPTCP community also has plans for enabling per-MPTCP-connection configuration through the addition of `SOL_MPTCP` socket options in future kernel versions.

System-Level Runtime Configuration

Making full use of multipath TCP requires some configuration after a Linux system is booted. If a kernel is configured with MPTCP enabled, by default MPTCP sockets may be created and used. These sockets will only allow single-subflow connections without further configuration. MPTCP can also be selectively disabled in each network namespace using the `net.mptcp.enabled` sysctl.

Path Manager Configuration

Using multiple subflows requires configuring a "path manager", a component of MPTCP that determines when additional subflows are requested or accepted, and that controls advertisement of available addresses for subflows. An in-kernel path manager is available today in the v5.7 and v5.8 kernels, and a future kernel release will add support for userspace path managers.

The in-kernel path manager is configured through a generic netlink socket. The '`ip mptcp`' command can be used by a privileged user to interact with the in-kernel path manager and is included in `iproute2-ss200602` and later releases.

One typical MPTCP use case has a server accepting incoming connections from mobile peers, where the mobile devices are responsible for initiating new connections and adding subflows. Here, the path manager needs to be told to increase the limit for the number of additional subflow connections it will accept per MPTCP connection:

```
sudo ip mptcp limits set subflow 4
```

Most peers will use only a couple of subflows (for example, one for Wi-Fi and one for cellular), so this sample command uses a limit of '4' to limit resource use per MPTCP socket but still allow flexibility for failed subflow connections and attempts to re-establish subflows by the peer device.

To use multiple subflows on a Linux system acting as a client device, making outgoing connections, subflow limits and interface information need to both be updated:

```
sudo ip mptcp limits set subflow 2
sudo ip mptcp endpoint add 192.0.2.10 subflow
```

The first command is the same "subflows per MPTCP connection" limit set in the first use case, which controls the total number of allowed subflows, whether they are initiated locally or by the peer. The second command allows the in-kernel path manager to initiate a new subflow using 192.0.2.10 as the source address. The destination address will be the same one used for the initial connection.

More details on the ip mptcp command are found in the "ip mptcp" man page. [8]

The in-kernel path manager offers limited client-side functionality. Userspace path managers will offer significantly more capability for client devices, but initial development has been focused on server use cases.

Userspace Path Management

Depending on the use case, several trade-offs exist that could impact the decision of where to implement MPTCP path management, in the kernel or in userspace. Such trade-offs include implementation complexity, performance, scalability, and flexibility.

Userspace vs. Kernel Tradeoffs

While the bulk of the MPTCP protocol may be implemented in the kernel, less performance- or latency-critical aspects of the protocol, such as path management, may be implemented in userspace. Implementing MPTCP path management in userspace provides a number of advantages.

Platform Integration. Path management in userspace instead of the kernel allows for more straightforward integration for Linux distributions, mobile platforms, and carriers, as well as making it possible to use a variety of data sources for endpoint policy and history.

Simpler Kernel-side Code. Moving path management to userspace means less code in the kernel, simplifying the kernel implementation, and reducing the chances of introducing kernel-side bugs. For instance, network interface and address tracking for MPTCP path management purposes may be implemented in userspace. Similarly, path management strategies may be added or removed as needed in userspace without impacting the kernel through reboots or module reloading. Custom path management strategies in userspace can be maintained when kernels are updated without rebuilding kernel modules.

Disadvantages of the Userspace Approach. Under heavy connection load a userspace path manager may become a bottleneck due to increased kernel-to-userspace interaction over generic netlink, as well as increased resource utilization (e.g. memory). Scalability becomes a concern in the presence of a large number of MPTCP connections. For example, a server receiving thousands of connection requests in a short period of time could trigger a large number of MPTCP-related events being sent to userspace, and path management commands sent from userspace to the kernel. Given these issues, path management in userspace may not be suitable for servers with heavy loads.

MPTCP Generic Netlink API

As MPTCP path management tasks would be delegated to userspace, a suitable mechanism to propagate related events and commands from and to the kernel, respectively, is necessary. In the case of userspace path management, The Linux kernel generic netlink framework is ideal.

For each path management event that potentially occurs over an MPTCP connection, a corresponding generic netlink message would be issued. The userspace listens for these events, and handles them accordingly. Typical path management related events include: new connection, connection established, connection closed, new subflow, subflow closed, new peer address, removed peer address, along with the related netlink attributes like address family, IPv4 or IPv6 address, address ID, etc.

A generic netlink API suitable for userspace path management has not yet been added as of kernel v5.8, and the earliest it could be added is v5.10.

Similarly, the MPTCP generic netlink API exposes several commands that allow the userspace path manager to instruct the kernel to perform path management tasks and configuration changes. Typical commands include, but are not limited to, advertising of local addresses, setting resource limits. and changing subflow priority.

The implementation of the API in the Linux kernel is provided by the "`pm_netlink`" path manager, which is responsible for bridging kernel and userspace path management interactions.

MPTCP generic netlink API message and attribute types exposed by the kernel are found in the `include/uapi/linux/mptcp.h` header file. Userspace implementations would include `<linux/mptcp.h>`, accordingly.

Connection managers found in Linux distributions, such as NetworkManager, ConnMan, and wicd, could leverage the MPTCP generic netlink API as part of their MPTCP connection management once MPTCP is supported by them.

Multipath TCP Daemon

The Multipath TCP Daemon – mptcpd – is a daemon for Linux based operating systems that performs MPTCP path management related operations in the userspace. It interacts with a Linux kernel through the MPTCP generic netlink API to track and manipulate per-connection information (e.g. available peer addresses), available network interfaces, request new MPTCP subflows, handle requests for subflows, etc. Mptcpd is a reference userspace MPTCP path manager implementation. It is not meant to replace existing connection managers like NetworkManager and ConnMan. Ideally such connection managers will implement their own MPTCP support based on the MPTCP generic netlink API described earlier. Mptcpd currently operates with an out-of-tree kernel for development purposes, and will be updated to work with a future upstream kernel release when the generic netlink API for userspace path management is completely settled.

Mptcpd provides an extensible MPTCP path management framework that allows path management strategy – plugin – developers to focus solely on path management related tasks without having to worry about MPTCP related kernel/userspace interaction, and network interface and address monitoring.

In addition to the path management plugin framework, other components found mptcpd are (1) the path manager, responsible for dispatching all MPTCP path management and network monitoring events to loaded plugins, (2) the network monitor, responsible for listening for changes to network interfaces and addresses, and (3) the mptcpd library – `libmptcpd` – implements several function that allow mptcpd plugins to issue MPTCP path management commands to the kernel.

Plugins. MPTCP path management in the Multipath TCP Daemon is implemented through plugins. Concrete path management strategies implement the mptcpd plugin API defined in the header file `<mptcpd/plugin.h>`. One such example is the *single-subflow-per-interface* "`sspi`" reference plugin that ships with mptcpd.

Plugins define each of the mptcpd plugin API functions found in the `mptcpd_plugin_ops` structure, and register those path management operations with mptcpd in the plugin `init` function by calling the `mptcpd_plugin_register_ops()` function, e.g.:

```
static void foo_new_connection(…) { … }

static
struct mptcpd_plugin_ops const pm_ops = {
  .new_connection = foo_new_connection,
  …
};

static int foo_init(void)
{
  static char const name[] = "foo";

  return !mptcpd_plugin_register_ops(
           name, &pm_ops);
}
```

These functions each correspond to a specific MPTCP event (e.g. *new connection*, *new subflow*, etc.) defined in the Linux kernel MPTCP generic netlink API.

There are times when a plugin may need to alter MPTCP connections or subflows, such as explicitly advertising new network addresses. Mptcpd exposes a path management command API, corresponding to the commands defined in the MPTCP generic netlink API, that may be leveraged by plugins. The path management command functions are declared in the `<mptcpd/path_manager.h>` header.

Mptcpd monitors all MPTCP-capable network interfaces. Plugins may examine those network interfaces, along with associated network addresses, by retrieving a pointer to the network monitor through the `mptcpd_pm_get_nm()` function found in the mptcpd path manager API defined in `<mptcpd/path_manager.h>`.

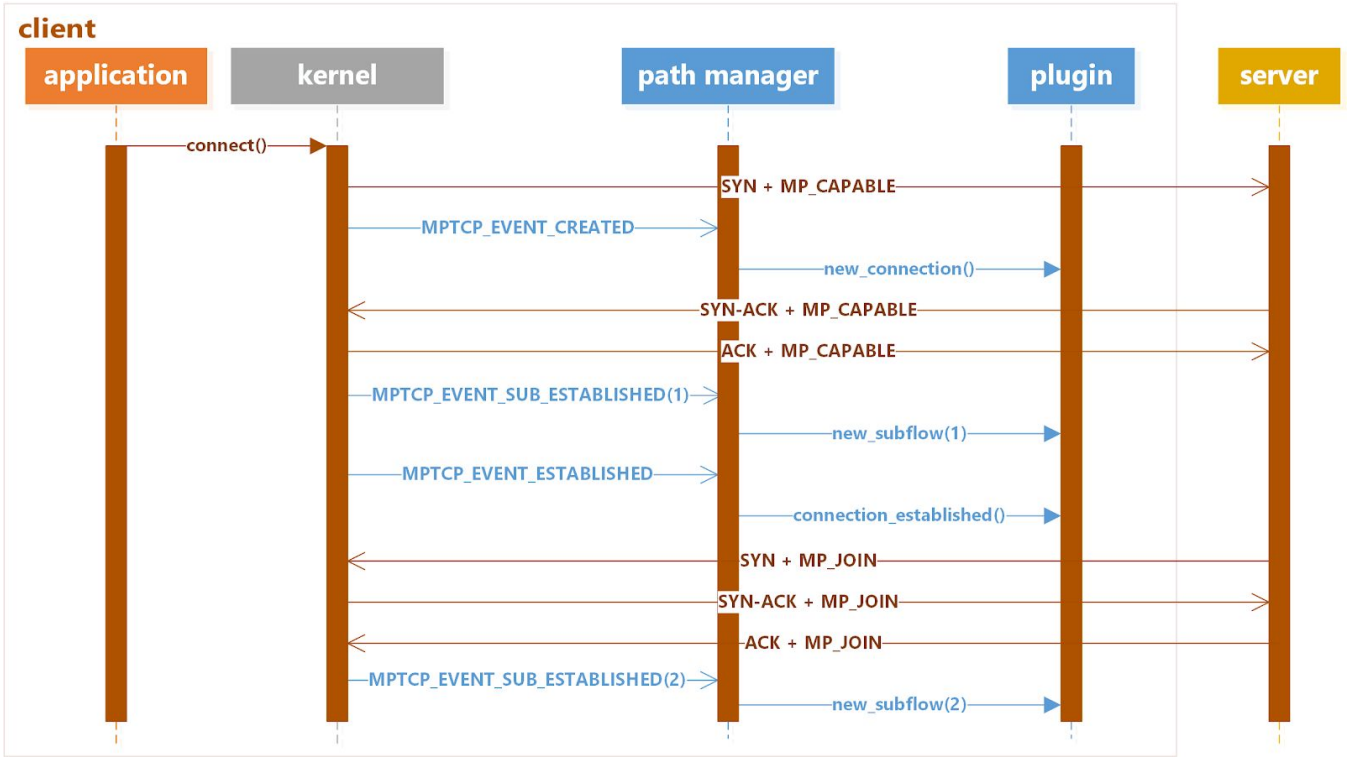

Figure 1. MPTCP event propagation from the kernel to mptcpd plugins.

Iteration over the monitored network interfaces may then be performed through the `mptcpd_nm_foreach_interface()` function declared in the `<mptcpd/network_monitor.h>` header. They may also register network monitoring related callbacks found in the `mptcpd_plugin_ops` structure to be notified dynamically of changes in local network interfaces and addresses.

Event Handling. Figure 1 depicts the sequence of operations that may occur when a new MPTCP connection is initiated by a client, along with a subflow that was added (joined) to the connection by the server side. The kernel multicasts related MPTCP event messages, which the path manager component listens for, and ultimately propagates those events to the path management plugin.

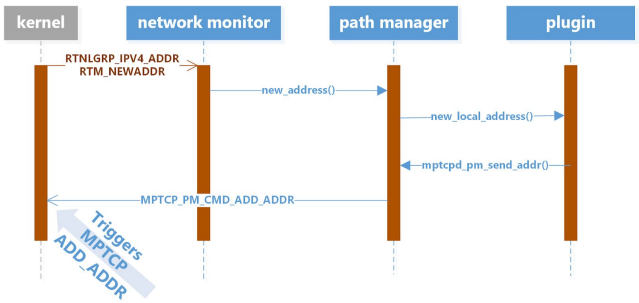

Figure 2. Network monitoring event propagation to mptcpd plugins.

Similarly, the path manager component also propagates network monitoring events to the plugins, as depicted in Figure 2. Network monitoring events correspond to those network link and address related events found in the kernel's `rtnetlink` interface.

Deployment. Since mptcpd leverages GNU Autotools for its build infrastructure, the build procedure may be as simple as the canonical "`./configure; make`". The Embedded Linux Library – ELL – is the only build dependency. [9] Installation and packaging is also straightforward due to the existence of the typical GNU style `Makefile` targets and variables.

Run-time requirements include needing a MPTCP-enabled kernel that supports the expected MPTCP generic netlink API for userspace path management. Mptcpd will attempt to detect these features at process start.

As mptcpd may potentially need to alter MPTCP connections, the `CAP_NET_ADMIN` capability is required. No other capabilities are explicitly necessary, nor is it necessary to run mptcpd as `root` if `CAP_NET_ADMIN` is granted. Assuming systemd is available on the host platform and detected by mptcpd during a `configure`

script run, a systemd unit file, `mptcp.service`, will be installed that grants the CAP_NET_ADMIN to the mptcpd process, as well enabling systemd dynamic user support to make mptcpd run with a dynamic generated unprivileged "`mptcp`" user. This helps limit potential escalation of privilege. Lastly, mptcpd will also verify that its global configuration file, e.g. `/etc/mptcpd/mptcpd.conf`, and plugin directory, e.g. `/usr/lib/mptcpd`, do not have global write permission to further mitigate potential security vulnerabilities.

Plugins should be installed in the mptcpd "`pkglib`" directory, e.g. `/usr/lib/mptcpd`. Mptcpd currently only supports loading plugins at process start, and does not support dynamic loading of plugins installed after mptcpd is started.

Project Information

The MPTCP upstream community is continuing work on enhancing the Linux Multipath TCP implementation. Current work on upstream MPTCP is in these areas:

– More features from RFC 8684 [1]
– Generic netlink interface for userspace path management and adaptation of mptcpd for this interface
– Better utilization of multiple subflows
– SYN cookie support
– TCP Fast Open
– Support more TCP socket options
– Configurable packet scheduler
– Improve performance

The team is distributed across different companies and locations throughout the world, with an open online presence for communication and project records:

– Github project (kernel work): https://github.com/multipath-tcp/mptcp_net-next/wiki
– Github project (mptcpd): https://github.com/intel/mptcpd
– Mailing List Address: mptcp@lists.01.org
– Mailing List Subscription: https://lists.01.org/postorius/lists/mptcp.lists.01.org/
– IRC: #MPTCPUpstream on https://freenode.net/

Acknowledgements

Many thanks to the MPTCP upstreaming community members, especially for editing and review assistance by Matthieu Baerts.

References

1. A. Ford, C. Raiciu, M. Handley, O Bonaventure, C. Paasch, "RFC 8684: TCP Extensions for Multipath Operation with Multiple Addresses", IETF website, accessed 2020-07-20, https://tools.ietf.org/html/rfc8684
2. A. Ford, C. Raiciu, M. Handley, O Bonaventure, "RFC 6824: TCP Extensions for Multipath Operation with Multiple Addresses", IETF website, accessed 2020-07-20, https://tools.ietf.org/html/rfc6824
3. Christoph Paasch and Mat Martineau, "Generic TCP-option framework and adoption for TCP-SMC and TCP-MD5", netdev@vger.kernel.org archive website, accessed 2020-07-20, https://lore.kernel.org/netdev/20180201000716.69301-1-cpaasch@apple.com/
4. David Miller, "Re: [RFC v2 02/14] tcp: Pass sock and skb to tcp_options_write", netdev@vger.kernel.org archive website, accessed 2020-07-20, https://lore.kernel.org/netdev/20180201.101153.762442261858533127.davem@davemloft.net/
5. Kyle McKay, "TopGit Github Project", GitHub website, accessed 2020-07-20, https://github.com/mackyle/topgit
6. N/A, "MPTCP Upstreaming Community Issue Tracker", GitHub website, accessed 2020-07-20, https://github.com/multipath-tcp/mptcp_net-next/issues
7. Alan Cox, Florian La Roche, Florian Westphal, and others, "skbuff.h Header File, Linux kernel v5.8", Linux Git Repository website, accessed 2020-08-02, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/skbuff.h?h=v5.8
8. Paolo Abeni, "ip-mptcp Man Page", Linux man pages website, accessed 2020-07-20, https://man7.org/linux/man-pages/man8/ip-mptcp.8.html
9. Marcel Holtmann, Denis Kenzior, and others, "ELL Project Page", 01.org website, accessed 2020-07-20, https://01.org/ell