Multipath TCP

Tex Altstaetter

Texas A&M Computer Science & Engineering

***Abstract*—Multipath TCP (MPTCP) can greatly impact a device's network performance by using multiple interfaces to transmit data across a network. Multiple interfaces provide several potential uses. It can improve latency, network resilience, and network capacity by transmitting data on by using multiple TCP connections simultaneously. This is an exploratory study of MPTCP's architecture, design decisions, configuration, and how it can be used to improve data flow through a network.**

I. Introduction

Multipath TCP (MPTCP) is an extension of the TCP protocol that uses multiple network interfaces. Current mobile devices may support several wireless interfaces and MPTCP can be used to improve the overall user-experience in several ways by simultaneously using of multiple TCP paths between hosts. MPTCP is able to pool the resources of different network interfaces to act as a single connection and leverage the existing benefits of TCP. There is a need to increased data throughput and also be able to handle user-interactive low-latency applications in a mobile setting. MPTCP attempts to address these problems transparently at the kernel level without requiring any changes to the user applications.

An MPTCP connection is a set of one or more TCP subflows combined to provide a single MPTCP service to an application. Each subflow consists of a TCP segment operating over an individual path and forms part of the larger MPTCP connection. Each interface (WiFi, Ethernet, LTE, etc.) that MPTCP utilizes opens an additional subflow. These subflows can be used in several ways. The subflows can function as a load balancer to better distribute data over a network, it may improve the overall network capacity by increasing the bandwidth capacity, or may add resilience by sending data redundantly over each subflow hence providing the shortest round trip time (RTT) to protect against network variability. The main complexity inherent to MPTCP has to do with the underlying TCP protocol and how it manages each subflow. TCP was designed as a single independent connection, however MPTCP needs to share information among these subflows to handle issues with congestion control, path scheduling, and data reordering. The congestion control algorithm most significantly impacts the performance of MPTCP since transmission timeouts due to congestion drastically degrade performance. Existing congestion control algorithms attempt to solve issues of responsiveness, throughput, fairness, and resilience. Also, since data from multiple subflows are transmitted from a single host, the data may arrive out of order due to the variability of each subflow's RTT. This is cause for reordering at the endpoint and may present issues of head-of-line blocking or deadlock in the receive buffer.

There are several existing congestion control algorithms though none have been proven optimal. Equal Weighted TCP (EWTCP) is the most responsive to network changes and equally distributes data across each subflow. EWTCP is a very simple congestion control algorithm but suffers from inefficient use of the network since it assumes a constant RTT for each subflow and packets can be dropped due to congestion when TCP subflows share the same bottleneck when traversing the network. The Coupled algorithm, also known as the Linked-Increase Algorithm (LIA), more efficiently sends data on good performing subflows by always sending data on the least congested subflow. It does so by measuring the subflow's expected loss rate and functions as a load balancer among the network. The Coupled algorithm is able to equalize congestion and total throughput in ways that EWTCP cannot. The Coupled algorithm forces a tradeoff between balancing congestion and subflow responsiveness. The Semicoupled algorithm is an improvement on the Coupled algorithm in that the former does not assume equal RTTs among subflows and forces traffic onto every subflow to prevent getting trapped on a single subflow which can happen in Coupled if a single subflow is always the least congested. If this occurs, the network isn't able to get feedback when the conditions on other subflows improve. Weighted Vegas (wVegas), is a delay-based congestion control algorithm. The Coupled algorithm uses packet loss events to determine subflow quality, whereas wVegas uses packet queuing delay to estimate per-subflow congestion levels. Wvegas is more sensitive to network congestion and shifts network traffic with a faster convergence rate. Balanced Link-Adaptation (BALIA) control, helps to solve the Coupled algorithm's unfriendliness to single-path TCP and unresponsiveness. The tradeoff between the friendliness of a TCP flow and the responsiveness is unavoidable, but BALIA is proven to provide a unique equilibrium point that is asymptotically stable. The friendliness of an algorithm determines how well an MPTCP connection shares bandwidth with other TCP connections. Since MPTCP is at the Transport Layer, the network isn't aware that multiple TCP subflows belong to a single connection and allocates disproportionate traffic to MPTCP connections which can starve single-path TCP connections. This is why resource pooling is used to aggregate multiple subflows together so that the Network Layer treats it as a single connection.

MPTCP is implemented in Linux and requires a custom kernel. MPTCP connections are easily configurable via system control variables. MPTCP performs a similar function to another multi-stream protocol Stream Control Transmission Protocol (SCTP). SCTP is a message-oriented protocol as opposed to MPTCP which is stream-oriented. SCTP doesn't suffer from head-of-line blocking or delayed transmission as MPTCP does but this comes at the cost of higher transfer overhead in the packets. MPTCP requires a 4-way handshake due to the possibility of middleboxes stripping out MPTCP packet options while SCTP is a 3-way handshake. SCTP is rarely deployed because it isn't supported by many middleboxes. One key difference in function is that SCTP uses uncoupled congestion control for each path and does not allocate resources fairly to competing single-path traffic

The main contributions can be summarized as:

- Create MPTCP socket programs and analyze how to configure and enhance the user-experience through the MPTCP options available via the MPTCP Linux kernel
- Identify key design considerations of MPTCP

II. State of the Art

For congestion control, the Max MPTCP algorithm balances the trade offs between responsiveness and maximizing throughput [1]. By incorporating variables from the Coupled and EWTCP algorithms, Max MPTCP effectively responds to different types of network access patterns while maintaining fairness at bottlenecks and considers subflows with varying RTT.

For the MPTCP scheduler the Decoupled Multipath Scheduler (DEMS) optimizes the packet scheduling for decoupled subflows and results in shorter download time for independent data chunks and has shown to reduce download time by up to 74% and 21% for 256 KB and 4 MB data chunks, respectively [2]. For highly mobile network conditions as seen in a vehicular-to-infrastructure (V2I) network RAVEN has improved median RTT by over 300% and 95% tail response times are 2-11 times faster than MPTCP's default MinRTT scheduler [3].

To integrate SCTP support with TCP, [4] discusses how SCTP uses concurrent multipath transfer (CMT) and resource pooling (RP) as an extension to SCTP called CMT/RP-SCTP which implements MPTCP compatible congestion control [5]. It also addresses issues of sending data over paths with different characteristics, e.g. bandwidth, delays, queuing behaviors, etc.

III. Problem Formulation

This paper will identify key design choices made when implementing MPTCP.

Design Criteria:

1. MPTCP must work with the Internet's middleboxes and firewalls
2. MPTCP should be as usable and perform at least as well as single-path TCP, i.e. maintain in-order, reliable data transmission
3. MPTCP must optimally use different paths and handle heterogeneous environments
4. MPTCP must be secure

The Transport Layer was selected as the ideal place to implement MPTCP for several reasons. For one, the Transport Layer is aware of path characteristics for multipath scheduling. The Transport Layer is concerned with data transmission between the source and the endpoint which allows for flexibility to update and extend the interface on top of the existing internet architecture. The Transport Layer is also close enough to the Application Layer for performance improvements to be noticeable to the user. In addition, MPTCP is able to safely revert to TCP if an MPTCP connection fails and its overhead of a failed connection doesn't propagate any further than the initial handshake.

Building MPTCP at the Transport Layer requires careful design since it must handle the higher-level socket API, work with the lower-level network stack, and not be detrimental to the existing internet traffic routing. One such issue has to do with the middleboxes, firewalls, and NAT that interfere with network traffic. Middleboxes may disrupt MPTCP packet options and discard them, thus preventing an established MPTCP connection. Due to middleboxes, MPTCP requires a 4-way handshake to verify no MPTCP packet options were discarded during the handshake. This uncertainty about establishing connections and the lack of MPTCP servers is why MPTCP was designed to optimize for the most common case and doesn't allocate the MPTCP data structures until after a connection has been established (as shown in Fig. 5.2 and 5.3). This is solely an artifact of maintaining existing TCP performance. Another issue at the Transport Layer is that IP addresses are intrinsically linked to TCP connections, this has impacts in how each TCP connection is allocated resources and is the requires the need for resource pooling. Ideally, a higher level abstraction would already exist combine multiple IP addresses into a single connection.

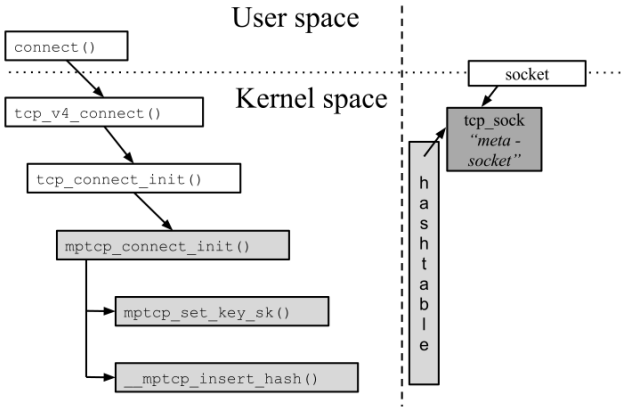


Figure 5.2: Upon a `connect()`, the client generates a random key but does not allocate additional data structures.

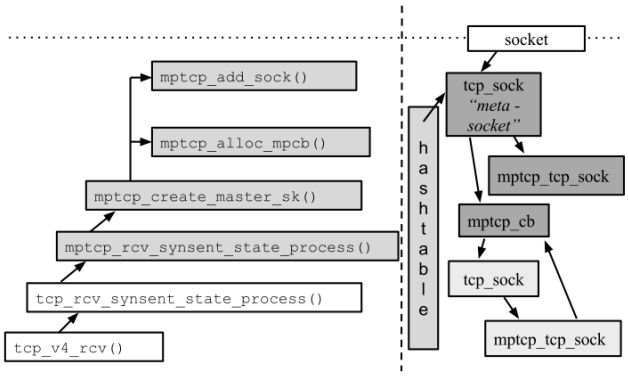
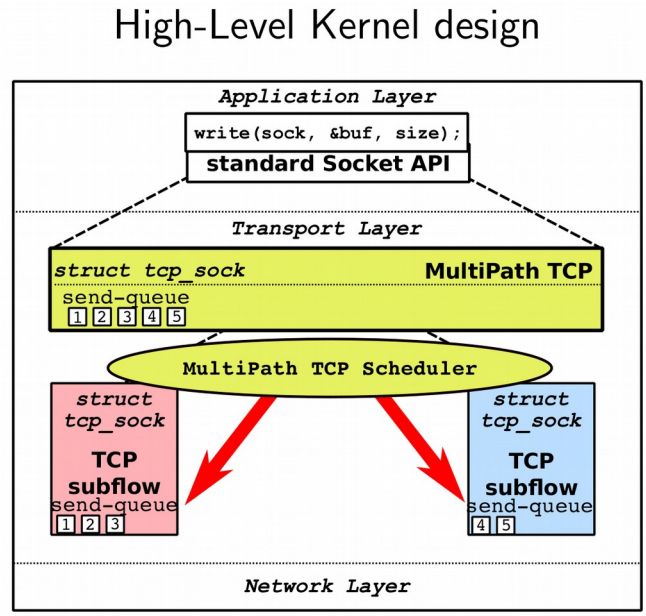


Figure 5.3: During the processing of the SYN/ACK containing an MP_CAPABLE option, the required data structures are allocated and linked together.

To ensure that MPTCP is as usable as single-path TCP it needs to handle the higher-level network stack as well. MPTCP uses the existing Socket API so that it is compatible with all existing programs. Since MPTCP is implemented at the kernel level, one key requirement for its success is acceptance by the Linux community. MPTCP should be entirely transparent to users, and most importantly, fail safely by reverting to TCP. To do this, MPTCP creates wrappers for all the standard function calls that operate differently for MPTCP. This allows MPTCP to be highly extensible at the cost of MPTCP requiring tight integration into the Linux's implementation of the Network Stack. MPTCP is compatible with the existing Socket API, and also extends the API by adding additional MPTCP socket options.

One constraint intrinsic to multiple subflows is the need to ensure in-order delivery. Since TCP subflows may be established or removed at any time during an MPTCP connection, MPTCP must be able to detect a lost subflow. Since data is striped across multiple subflows a single queue on top of the subflows is required to retransmit lost packets on an alternate interface and ensure in-order delivery. This makes packet acknowledgment more complex. An explicit data acknowledgment field is added in the TCP header in addition to the subflow acknowledgment. Acknowledgments can be inferred from subflow acknowledgments, however dropped packets may occur due to difficulty in determining the trailing edge of the receive window. This will occur anytime RTTs differ such that acknowledgments arrive in a different order than they were sent.



MPTCP must optimally use different paths

Congestion control is highly important to the design of MPTCP to optimize paths.

MPTCP Congestion Control Requirements:

- A MPTCP flow should give a connection at least as much throughput as it would get with single-path TCP on the best of its paths. This ensures there is an incentive for deploying multipath.
- A multipath flow should take no more capacity on any path or collection of paths than if it was a single-path TCP flow using the best of those paths. This guarantees it will not unduly harm other flows at a bottleneck link, no matter what combination of paths passes through that link.

Congestion control must consider fairness at shared bottlenecks that compete with single-path TCP, choosing efficient subflows, adapting to load changes, and adapting to heterogeneous paths with RTT mismatches.

MPTCP Path Limitations:

- Must rely on standard internet routing mechanisms
- Cannot selectively choose which subflow to use
- The TCP subflow can only control the traffic of that specific path

The internet routing already attempts to select efficient paths and balance congestion. Inefficiencies arise when both the end-to-end systems and the Internet Core attempt to balance traffic. MPTCP leaves the intermediate routing to the Internet Core.

Security

MPTCP has in-built security features when establishing the initial connection and any subsequent subflows. It does so by passing tokens and HMAC keys that are verified and acknowledged on each endpoint. This is essential to prevent side-channel attacks from hijacking a subflow or adding a subflow.

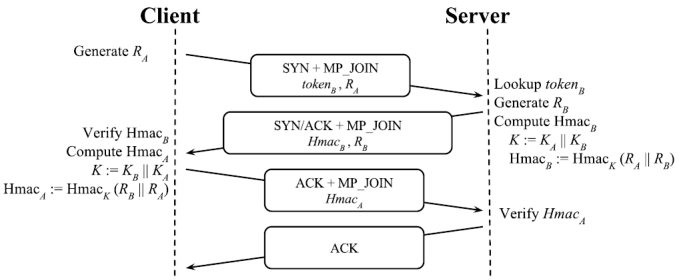


Figure 2.4: The token serves as a connection identifier, while the HMAC-exchange allows to authenticate the end hosts, while creating additional subflows.

III. Evaluation

There were several design challenges involved when implementing MPTCP.

The most notable were:

- Handling middleboxes that operate in a non-standard manner. This was solved by adding a 4-way handshake and including explicit DSS acknowledgments in the TCP header.
- Writing the Data-sequence number in the TCP-options. This was solved inside the MPTCP-scheduler by writing the data-sequence on top of the payload
- Handling socket options. There are many socket options in the TCP/IP stack, some of which are for the MPTCP-level, while others should get passed onto all subflows This requires many changes in unrelated TCP functions (e.g., do_ip_setsockopt)

MPTCP attempts a “make-before-break” handover mechanism to establish a new subflow before dropping an MPTCP connection with a single subflow. However, if its not possible, it is able to “break-before-make” and re-establish a TCP connection when one becomes available

Conclusion

What separates MPTCP from SCTP is the architecture of MPTCP. It takes a holistic approach that maintains transparency at the Application Layer, handles interference from middleboxes and NAT, has its own scheduler, creates and break TCP subflows dynamically, optimizes for congestion control, and safely reverts to single-path TCP when MPTCP connections fail.

There is still future work that could improve MPTCP. For example, lowering the setup overhead, improving security measures, and determining the best time to initiate additional subflows. Currently, it uses heuristics to determine when to create, destroy, or re-establish a subflow. This could be further studied to improve MPTCP. Once an initial subflow has been established and data exchanged, only the client is able to initiate additional subflows in the current MPTCP implementation.This is in case both the client and server initiate new subflows across the same IP address pairs, however, in a mobile ad-hoc network it might be desirable that either mobile device initiate a connection.

References

[1] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. “Design, Implementation and Evaluation of Congestion Control for Multipath TCP”. in Proc. Usenix NSDI 2011

[2] Guo, Y. E., Nikravesh, A., Mao, Z. M., Qian, F., and Sen, S. Accelerating Multipath Transport Through Balanced Subflow Completion. In Proceedings of the 23rd International Conference on Mobile Computing and Networking (2017).

[3] HyunJong Lee, Jason Flinn, and Basavaraj Tonshal. 2018. “RAVEN: Improving Interactive Latency for the Connected Car”. In Proceedings of the 24th Annual International Conference on Mobile Computing and Networking. ACM, 557-572.

[4] Adhari, Hakim & Dreibholz, Thomas & Becke, Martin & P. Rathgeb, Erwin & Tüxen, Michael. (2011). “Evaluation of Concurrent Multipath Transfer over Dissimilar Paths.” 708-714. 10.1109/WAINA. 2011.92.

[5] J. R. Iyengar, P. D. Amer, and R. Stewart, “Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths,” IEEE/ACM Transactions on Networking, vol. 14, no. 5, pp. 951–964, Oct. 2006, ISSN 1063-6692.

[6] B. Hesmans, O. Bonaventure. “An Enhanced Socket API for Multipath TCP.” Proceedings of the 2016 Applied Networking Research Workshop, July 16, 2016. Berlin, Germany

[7] A. Ford, C. Raiciu, M. Handley, O. Bonaventure. “TCP Extensions for Multipath Operation with Multiple Addresses”, RFC 6824. 2013.

[8] A. Ford, C. Raiciu, M. Handley, S. Barre, J. Iyengar, “Architectural Guidelines for Multipath TCP Development”, RFC 6182, March 2011.

[9] Q. Peng, A. Walid, J. Hwang, S.H. Low, “Multipath TCP: Analysis, Design, and Implementation,” in IEEE/ACM Transactions on Networking, vol. 24, no. 1, pp. 596-609, Feb. 2016

[10] R. Stewart, “Stream Control Transmission Protocol”, RFC 4960, September 2007.

[11] “Welcome to the Linux Kernel MultiPath TCP Project.” *MultiPath TCP - Linux Kernel Implementation*, Université Catholique De Louvain, multipath-tcp.org/.

[12] Bonaventure, Olivier. “Obonaventure/Mptcp-Doc.” *GitHub*, github.com/obonaventure/mptcp-doc/.

[13] Altstaetter, Dalton. “Daltstaetter/CSE_664_MPTCP.” *GitHub*, github.com/daltstaetter/CSE_664_MPTCP.