Improving Multipath TCP

Christoph Paasch

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

October 30, 2014

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Improving Multipath TCP
by Christoph Paasch

ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was partially supported by the European-funded Trilogy, CHANGE and Trilogy2 project.

Preamble

The 20th century has seen the digital revolution, introducing the change from analog to digital technology. It is continuing right into the 21st century, interconnecting billions of digital devices over the Internet. Different types of devices and ways of communication have been designed. From smartphones to personal computers; from small servers connected to the Internet to huge content delivery networks serving millions of users simultaneously. The scale of the Internet and the mass of traffic it is transporting is continuously increasing [LIJM+11].

In order to cope with this increasing amount of traffic, redundancy and load-balancing is the rule on the Internet. Indeed, there often exist many different paths between any two pair of hosts connected to the Internet. Internet Service Providers (ISPs) extensively use redundant paths to be resilient against link-failures. The load along these redundant paths is often balanced by using techniques like Equal Cost Multipath which distribute the traffic across multiple paths. Data centers have a similar incentive like ISPs to balance the load and be resilient against link failures. Finally, end-user devices also employ such kind of redundancy. Smartphones are typically equipped with a WiFi and a 3G/4G interface. This allows to remain connected to the Internet, even if the next WiFi access point is far away.

It becomes obvious that the Internet architecture offers a large number of paths between the end hosts. There remains however one problem. The protocols in use today are not able to efficiently use these paths. The most used protocol is the Transmission Control Protocol (TCP). It allows to send a byte stream in a reliable and in-order manner between two devices. It uses the IP addresses and port numbers to identify to which connection the payload belongs to. In order to steer a packet along a specific path, the IP address (or port numbers) must change. This would effectively break TCP's way of demultiplexing, making it unable to reconstruct the byte stream.

There is a gap between the single-path transport protocols and the multi-path network.

Multipath TCP, a recently proposed TCP extension, attempts to fill this gap. It extends TCP to allow transmitting a single byte stream across different paths.

This kind of multipath transmission has several benefits. First, it increases the resilience to failures. If a link disappears, Multipath TCP can simply continue the transmission on another one - without disrupting the byte stream. Second, it allows to increase the performance by pooling the resources. Indeed, the different paths may be constrained by distinct bottlenecks, each offering a different capacity to the flow. Multiplexing a byte stream across these different bottlenecks effectively allows to pool their capacities and thus increase the overall goodput for the byte stream transmission.

This thesis is a contribution to make multipath communication not an exception, but the rule by improving Multipath TCP and its Linux Kernel implementation. More specifically, the main contributions of this thesis are:

- *Developing an experimental design approach to evaluate Multipath TCP.*

  A large number of factors influence the performance of a Multipath TCP connection. Factors like delay, loss rate and capacity have a big impact on the performance of a connection. For traffic going over the Internet, each of these factors can be within a large range. Validating that Multipath TCP works well across the Internet is very difficult, considering the sheer infinite number of possible environments.
  In Chapter 3, this thesis applies an *"Experimental Design"* to the evaluation of Multipath TCP. This approach allows to have a statistical confidence on the results of the evaluation. It provides these results by equally spreading the parameter sets for the evaluation across the considered factor-space. In this thesis, an evaluation of Multipath TCP, using this approach is presented. The evaluation measures the resource pooling as well as the load-balancing capabilities of Multipath TCP. Finally, it measures how different congestion control algorithms influence the delay perceived by the application.

- *Improving Multipath TCP in heterogeneous environments.*

  The paths used by Multipath TCP may have significantly different characteristics. As shown in Chapter 4, such a heterogeneity impacts Multipath TCP in several ways. Particularly, receive-window limitations and head-of-line blocking are accentuated by such heterogeneous environments. We propose heuristics to allow Multipath TCP to achieve high performance – even in heterogeneous environments.
  First, a reactive approach is proposed. This approach has shown very promising results and has been fine-tuned by using the experimental design

approach. Another, proactive, approach would be to schedule traffic across the paths by taking into account the characteristics of each of the paths so that all of them can be used in the most efficient way. We suggest a modular scheduler infrastructure to enable researchers to design such heuristics inside the scheduler.

- *A scalable Linux Kernel implementation of Multipath TCP.*

  The Linux Kernel implementation has been developed over the last years, constantly changing and improving. Chapter 5 explains how the implementation has been designed while keeping the following three goals in mind: minimize performance impact on regular TCP; achieve reasonable complexity inside the TCP stack; and achieve high performance for Multipath TCP.
  These goals have a major influence on certain design decisions, where sometimes the performance of Multipath TCP needs to be reduced for the benefit of regular TCP's performance. The chapter shows that even considering these design decisions, the implementation is able to achieve high performance in a scalable manner.

- *Retrospecting the design decisions of Multipath TCP.*

  During the specification of Multipath TCP in RFC 6824 [FRHB13], certain decisions were made with respect to the handshake, security, data transmission and the exchange of control information. Now, after several years of experience in implementing, testing and deploying Multipath TCP we can take a step back and revisit some of these design decisions. In Chapter 6, we revisit the computational overhead of the current handshake and suggest a low-overhead flavor for use-cases where security attacks are not of a concern. Next, we show how it is possible to leverage the application-layer security into Multipath TCP to overcome some of the residual security threats we identified. Finally, we envision a more disruptive change to Multipath TCP. A control-stream could open the door for future extensions and overcome some of the limitations of the initial design-choice of transmitting Multipath TCP's control information inside the TCP option space.

Bibliographic notes

Most of the work presented in this thesis appeared in conference proceedings, journals and IETF contributions. The list of related publications is shown hereafter:

Conference and workshop publications

1. *Multipath TCP: From Theory to Practice*
   S. Barre, C. Paasch and O. Bonaventure. In IFIP Networking. 2011.

2. *How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP*
   C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure and M. Handley. USENIX NSDI. 2012.

3. *Exploring Mobile/WiFi Handover with Multipath TCP*
   C. Paasch, G. Detal, F. Duchene, C. Raiciu and O. Bonaventure. ACM SIGCOMM workshop Cellnet. 2012

4. *On the Benefits of Applying Experimental Design to Improve Multipath TCP*
   C. Paasch, R. Khalili and O. Bonaventure. ACM CoNEXT. 2013.

5. *Evolving the Internet with Connection Acrobatics*
   C. Nicutar, C. Paasch, M. Bagnulo and C. Raiciu. ACM CoNEXT workshop HotMiddlebox. 2013.

6. *Multipath in the Middle(Box)*
   G. Detal, C. Paasch and O. Bonaventure. ACM CoNEXT workshop HotMiddlebox. 2013.

7. *Are TCP Extensions Middlebox-Proof?*
   B. Hesmans, F. Duchene, C. Paasch, G. Detal, O. Bonaventure. ACM CoNEXT workshop HotMiddlebox. 2013.

8. *Experimental Evaluation of Multipath TCP Schedulers*
   C. Paasch, S. Ferlin, O. Alay and O. Bonaventure. ACM SIGCOMM Capacity Sharing workshop (CSWS). 2013.

Journal publications

1. *Revisiting Flow-Based Load Balancing: Stateless Path Selection in Data Center Networks*

G. Detal, C. Paasch, S. van der Linden, P. Merindol, G. Avoine and O. Bonaventure. Computer Networks, 57(5):1204-1216. 2013.

2. *Multipath TCP*
   C. Paasch and O. Bonaventure. Communications of the ACM. 57(4):51-57. 2014

IETF contributions

1. *Multipath TCP - Guidelines for implementers*
   S. Barre, C. Paasch and O. Bonaventure. IETF Internet-Draft draft-barre-mptcp-impl-00. 2011.

2. *Securing the Multipath TCP handshake with external keys*
   C. Paasch and O. Bonaventure. IETF Internet-Draft draft-paasch-mptcp-ssl-00. 2012.

3. *Multipath TCP Low Overhead*
   C. Paasch and O. Bonaventure. IETF Internet-Draft draft-paasch-mptcp-lowoverhead-00. 2012.

4. *Analysis of MPTCP Residual Threats and Possible Fixes*
   M. Bagnulo, C. Paasch, F. Gont, O. Bonaventure and C. Raiciu. IETF Internet Draft draft-ietf-mptcp-attacks-02. 2014.

5. *A Generic Control Stream for Multipath TCP*
   C. Paasch and O. Bonaventure. IETF Internet Draft draft-paasch-mptcp-control-stream-00. 2014.

6. *Experience with Multipath TCP*
   O. Bonaventure and C. Paasch. IETF Internet Draft draft-bonaventure-mptcp-experience-00. 2014.

7. *Processing of RST segments by Multipath TCP*
   O. Bonaventure, C. Paasch and G. Detal. IETF Internet Draft draft-bonaventure-mptcp-rst-00. 2014.

Acknowledgments

This thesis as well as my future career would never have been possible without the support of my advisor, Olivier Bonaventure. Thanks to him I had the opportunity to work with him (and many other researchers) on this very interesting research topic. He gave me the guidance I needed so that my thesis went into the right direction, while also giving me the freedom to spend time on the implementation. His enthusiasm for networking research is infectious and pushed me (and many other PhD students) to accomplish a PhD thesis.

The members of my thesis committee Marco Canini, Ramin Khalili, Charles Pecheur, Costin Raiciu and Laurent Schumacher gave me valuable feedback during my private defense on how to improve the thesis. I would like to thank them for being part of my thesis committee and having invested the time to read and review my thesis.

I am also thankful to all my co-authors, Ozgu Alay, Gildas Avoine, Marcelo Bagnulo, Sébastien Barré, Gregory Detal, Fabien Duchêne, Simone Ferlin, Alan Ford, Fernando Gont, Mark Handley, Benjamin Hesmans, Micchio Honda, Ramin Khalili, Pascal Mérindol, Catalin Nicutar, Costin Raiciu and Simon van der Linden. I have had many fun hours with them, working until very late (or early in the morning) to meet the paper deadlines.

Among my former colleagues at the UCL, special thanks go to the MPTCP-team: Sébastien Barré from whom I took over the maintenance of the Multipath TCP implementation in the Linux Kernel; Gregory Detal, who actually gave me the incentive and pushed me to ask for a PhD position at UCL; Fabien Duchêne with whom I walked across San Francisco to the Golden Gate Bridge; Benjamin Hesmans who was a very nice office neighbor, always available for a little chat. I thank also my other former colleagues of the IP Networking Lab (Raphael Bauduin, Marco Canini, Juan Antonio Cordero, Sébastien Dawans, Benoit Donnet, Pierre Francois, Nicolas Laurent, David Lebrun, Damien Leroy, Damien Saucez, Olivier Tilmans, Hoang Tran Viet, Simon van der Linden, Virginie Van den Schrieck, Laurent Vanbever and Stefano Vissicchio) from whom I learned a lot and had lots of fun during our long coffee breaks in the afternoon. I also wish to thank the whole INGI department, academic and administrative, for all their help during those years at the UCL.

I also would like to thank all my other collaborators during this journey with which I had interesting discussions about issues in Multipath TCP, who contributed to the implementation, helped me setting up test environments or reported bugs. It is also thanks to them that I was able to accomplish my thesis. I am also grateful to Lars Eggert for giving me the opportunity of doing an internship at Nokia in Helsinki. Thanks also to Ramin Khalili and his former colleagues for welcoming me during my short visit at T-Labs in Berlin.

Finally, I would like to thank my family for the support they gave me during these last years. Many thanks go as well to the whole team of the Roller Bulls, who forced me to exercise twice a week, even if the day after I had sore muscles at work. Lastly, I would like thank Perrine, whose support helped me to eventually finish the thesis and take the necessary time off to recover.

Christoph Paasch
October 30, 2014

Contents

Chapter 1

Introduction

The Internet is a network of nodes that allow end hosts to communicate with each other. Communication is achieved through the use of a layered protocol stack. Starting at the physical layer, the subsequent link-layer interconnects local area networks over switches as well as wireless networks. The network layer is then responsible for the interconnection of these heterogeneous networks via IP version 4 [Pos81a] or version 6 [DH98]. This interconnection is achieved through the use of globally unique IP addresses for each end host. Packets emitted by one end host are forwarded by the routers based on the destination address specified inside the packet. Then, the transport layer handles the communication of a data stream between the applications running on the end hosts. This layer can offer different services like reliable or unreliable transmission, as well as in-order delivery. The most prominent transport layer protocol is the Transmission Control Protocol (TCP) [Pos81b].

In the past, end hosts were typically connected via a single interface to the Internet. An IP address is assigned to this interface and all communications of the

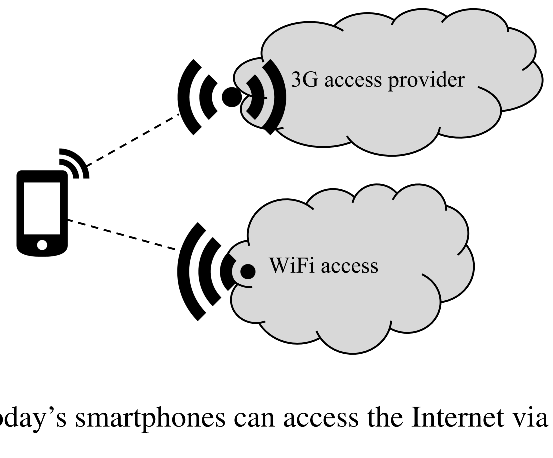


Figure 1.1: Today's smartphones can access the Internet via 3G or WiFi.

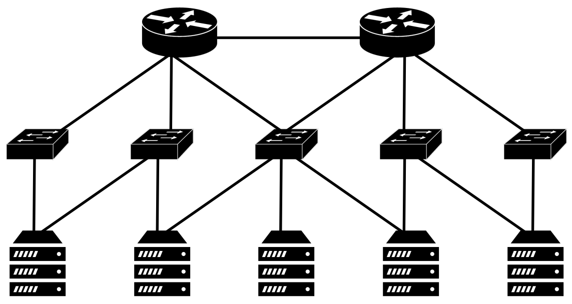


Figure 1.2: Data centers have a huge redundant infrastructure to allow for network load-balancing and resilience to failures.

host go via this specific interface. However, nowadays the end hosts often have multiple interfaces. Every notebook has a wired and a wireless interface, allowing to connect via Ethernet and WiFi. Smartphones are also equipped with multiple interfaces, allowing to access the Internet either via the mobile operator's 3G/4G network or via a WiFi access-point (as shown in Figure 1.1). The model of single interface, single IP-address per end host is effectively over in the Internet of the 21st century.

Another, very specific environment are data centers, shown in Figure 1.2. They contain a very large number of servers (several thousands and more). The servers often communicate with each other via the data center's network to replicate data, achieve a distributed computation or to request data from another host [KSG+09]. The performance of the network in these data centers is critical for a whole range of cloud-based services. To allow for this high performance network, data centers employ a huge redundant infrastructure [AFLV08, GHJ+09]. The redundancy allows to balance the load on different paths and improves resilience against link/node failures.

Additionally, the ISPs also offer multiple paths. Equal Cost Multipath (ECMP) allows routers to send traffic with the same destination across different paths, if the routing metric of these paths is of equal cost. Tools like Paris Traceroute have allowed to detect these load-balanced paths inside ISPs, and shown which fields in the packet header make a packet follow a certain load-balanced path [ACO+06]. Pelsser et al. show in [PCVB13] that across these load-balanced paths the delays are quite disperse, effectively offering multiple diverse paths between two end hosts.

However, the protocols widely deployed today are not designed to be used across such redundant, multipath networks. They have all been designed with single-path communication in mind. The identifiers used in the protocols (e.g., the IP-address in IP routing, or the 4-tuple in transport layer protocols) are also

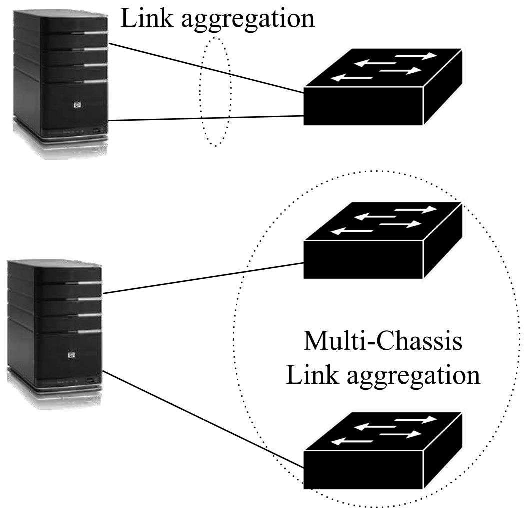


Figure 1.3: Link aggregation requires a specific configuration on the end host and the next-hop switch. Further, multi-chassis link aggregation requires a synchronization protocol between the switches.

identifiers for a specific path, limiting these protocols to being *single-path protocols*.

There is this gap between the *single-path transport* and the *multipath network*. Filling this gap would allow to pool the resources of the different available paths, known as the *"Resource pooling principle"* [WHB08]. Indeed, the bottleneck of each of the paths may be different, allowing us to actually increase the bandwidth for the end host. It also allows to be resilient against link failures. If one of the paths experiences an issue, the traffic can be moved away from this one and continue on the other, still operational paths [RNBH11].

1.1 Multipath/Multihoming solutions

This gap between the single-path transport protocols and the multipath network is far from new. Over time, different solutions have been proposed to bridge this gap at different layers with respect to their specific use case.

1.1.1 Link Layer

At the link layer, link aggregation techniques are used to aggregate the capacities of different interfaces to the same switch. These aggregation techniques only allow to pool the bandwidth of a single hop and require a specific configuration on the host and the switch (an example can be seen at the top of Figure 1.3). Both host interfaces have the same IP address and typically also the same MAC address. Traffic can be sent on either of the interfaces. As the interfaces are assigned the same IP address, upper layer protocols are not aware of the link aggregation. The link aggregation mechanism has been defined in the standard IEEE 802.3ad [IEE00]. An automatic configuration protocol has been specified and is known as the Link Aggregation Control Protocol (LACP). To avoid having a single-point-of-failure, multi-chassis link aggregation allows to aggregate the capacity of multiple switches. This kind of aggregation requires a synchronization mechanism between the switches.

There are different modes to achieve the resource pooling through link aggregation. The node may distribute the frames in a round-robin fashion across the links. This kind of approach may introduce reordering across a flow. TCP suffers from this kind of reordering as out-of-order data reception is an indication for a packet-loss, forcing TCP to retransmit data [BA02]. Another approach is to pin each flow to a specific interface. With respect to this, it is crucial what defines a flow. A flow may be a pair of source/destination MAC addresses, source/destination IP addresses or even the 4-tuple. The definition of a flow influences the granularity with which the link aggregation distributes the segments across the links. This distribution may happen in a round-robin fashion or be more "intelligent" by using an adaptive mechanism. The adaptive mechanism would take into account the loads of the link to balance the traffic in the most efficient way [Fou09].

Link layer aggregation is widely used in today's networks. Either in the ISP's to increase the bandwidth between two switches or in data centers to provide a higher network-access to the servers. However, one of its downsides is that it only allows to aggregate the capacity of the next hop. If the bottleneck of the communication is not in the next hop, link aggregation does not bring any benefit. It also requires specific configuration of the next-hop switch. Thus, end users are often unable to use link aggregation to benefit from multiple paths.

1.1.2 Network Layer

Multiple solutions have been proposed to benefit from multiple paths by using a network-layer approach. At first sight, this approach seems to be the most straight-forward and promising. Disjoint paths can often be pinned to a different pair of source/destination IP addresses. E.g., a smartphone communicating over WiFi and 3G has been attributed a different IP address for each of its interfaces. Two major solutions have been proposed to achieve multihoming at the network layer.

Mobile IP [Per97] (and Mobile IPv6 [PJA11]) allows an end host to change its

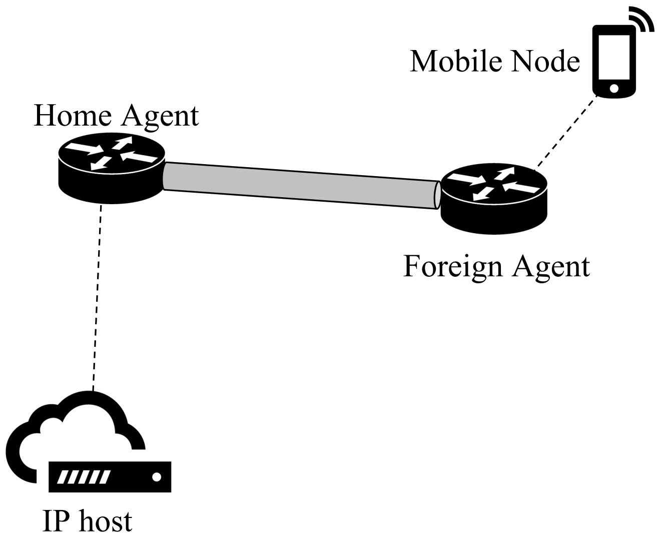


Figure 1.4: The home agent has to relay the traffic to the foreign agent so that it can deliver the data to the mobile node.

own IP address, without the need to reestablish all its TCP connections. It relies on the fact that a so-called *Home agent* (shown in Figure 1.4) is not moving and can be seen as a stable point of communication. As the mobile node is moving from one access gateway to another one (called *Foreign agent*), the home agent will relay traffic to the foreign agent which then in turn will send the data to the mobile node. This allows for undisrupted TCP connections. However, it does not allow to pool the resources, and thus increase the bandwidth. Further, it requires extensive support of the network equipment to relay the data from the home agent to the foreign agent.

A pure end-host based solution has been specified by the shim6 extension [NB09]. Shim6 - short for *Site Multihoming by IPv6 Intermediation* - is an IPv6-only solution that allows to handover traffic from one IPv6 address to another upon failure detection. The signaling is done via IPv6 extension headers and splits IP addresses in upper-layer identifiers and locators. It achieves the same goal as Mobile IP, by performing hot-handover from one interface to another without disrupting upper-layer transport level protocols. A Linux kernel implementation has been done and it has been showed that shim6 is indeed a realizable solution [Bar11].

Network-based solutions that hide address changes to the transport layer protocols seem like a straight-forward solution. However, another dimension is not taken into account in such approaches. Each path is subject to different characteristics in terms of bandwidth, delay, jitter,. . . A stateful transport layer protocol like

TCP uses an estimation of these characteristics by tracking the bandwidth-delay-product of the path. If multiple paths are used in a transparent manner to the TCP protocol, the stack has an incoherent view of the bandwidth-delay-product, resulting in performance issues and thus a bad user experience.

1.1.3 Transport layer

A multipath solution that sits at (or on top) of the transport layer has the advantage of being aware of the path characteristics, measured by the transport layer (e.g., by TCP). It allows for the multipath mechanism to take this information into account when scheduling the traffic across the different paths. The most prominent transport layer multipath solution is the Stream Control Transmission Protocol (SCTP) [SX01].

SCTP is an alternative transport protocol capable of supporting several IP addresses per connection. The first versions of SCTP used multiple addresses in failover scenarios. SCTP uses a message-based multi-streaming approach to the transmission of the data. Instead of transmitting a continuous byte stream (as is done by TCP), it splits the byte stream in chunks. Further, it allows to transmit multiple streams, separating itself from the single-stream TCP. Each stream can be sent across a different source/destination IP address pair. Finally, SCTP allows failover to different IP addresses. To achieve resource-pooling, recent extensions have enabled it to support the simultaneous use of several paths (SCTP-CMT) [IAS06]. Unfortunately, except in niche applications such as signaling in telephony networks, SCTP has not been widely deployed. One reason is that many firewalls and NAT (Network Address Translation) boxes are unable to process SCTP packets and thus simply discard them. Another reason is that SCTP exposes a different socket API to the applications. These two factors lead to the classic chicken-and-egg problem. Network manufacturers do not support SCTP in their firewalls because no application is using this protocol; and application developers do not use SCTP because firewalls discard SCTP packets. There have been attempts to break this vicious circle by encapsulating SCTP on top of UDP (User Datagram Protocol) [TS13] and exposing a socket interface to the application, but widespread usage of SCTP is still elusive.

Today's Internet is invaded by middleboxes which interfere with our traffic. TCP and UDP are the only protocols from which a reasonable acceptance by the middleboxes can be expected. Thus, the Internet's hour-glass enforced a multipath solution on top of either of these protocols. As the Transmission Control Protocol accounts for the largest proportion of the Internet's traffic, the choice felt on this one to extend it for multipath support. The following section gives an overview

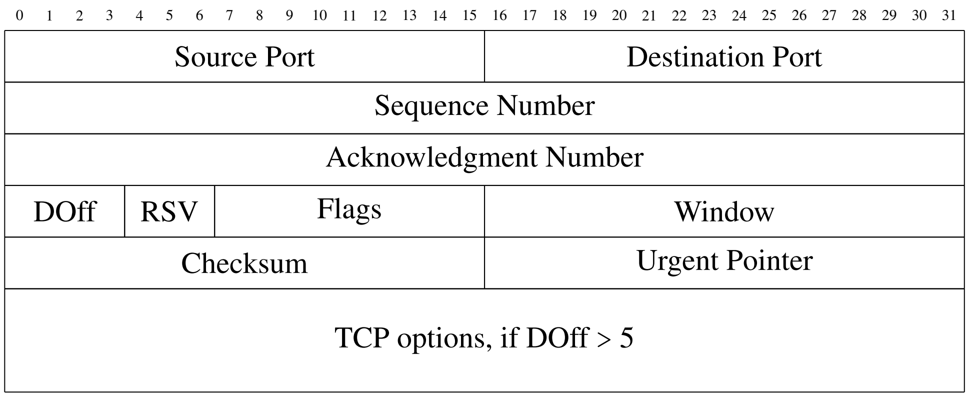


Figure 1.5: Header of the Transmission Control Protocol (TCP)

of the functioning of the reliable and in-order streaming service provided by the Transmission Control Protocol (TCP).

1.2 Transmission Control Protocol (TCP)

The service provided by TCP ensures that a byte sent on a socket of one end host will eventually reach the peer. Additionally, it is ensured that the bytes sent on the socket reach the receiver in the same order. Many applications rely on this service provided by TCP, as it abstracts the network to a pipe, where everything that enters will come out at the other side. TCP achieves this by sending segments over IP, using a specific transport header that allows to ensure the above goals. The following describes the control and the data plane of TCP.

1.2.1 Control plane

TCP splits the byte stream sent by the application in smaller segments, and transmits them over the underlying IP protocol (IPv4 or IPv6). In order to identify which service is being contacted, source and destination port numbers are added in the transport header of TCP. This 4-tuple (source/destination IP addresses and source/destination port numbers) identifies the data stream that is transmitted between the applications.

In order to start a TCP connection, the hosts go through the 3-way handshake phase, illustrated in Figure 1.6. The handshake allows the hosts to mark the beginning of a data stream. First, it allows the active opener of the connection to specify the 4-tuple in use for this stream by sending a SYN to the server. The SYN also includes the initial sequence number for the data that will be sent from

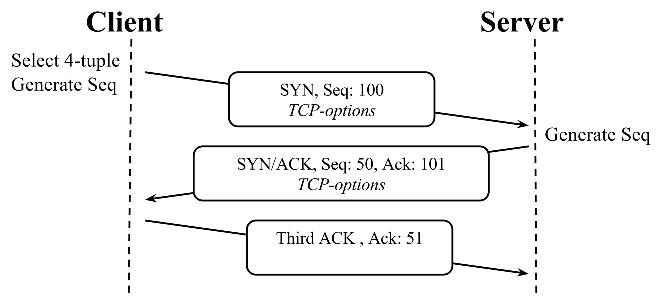


Figure 1.6: During the 3-way handshake, TCP exchanges the initial sequence numbers and negotiates optional extensions to TCP within the TCP option space

the client to the server (details about the sequence numbers are provided in the next section). If the server is willing to accept this connection, it replies with a SYN/ACK packet, indicating its own initial sequence number and acknowledging the reception of the client's SYN. Finally, the client acknowledges this packet with a final ACK.

TCP has been designed to be extensible by allowing up to 40 bytes of optional arguments inside the TCP header. These so-called *TCP options* are in a TLV-format to allow to easily specify new extensions. Support for new extensions is typically negotiated as part of the 3-way handshake. During the 3-way handshake the client signals in the TCP options field which extensions it would like to use during this connection. The server then replies inside the SYN/ACK with the chosen set of extensions that are indeed in use during this connection. Some examples of this kind of extensions are the TCP timestamp option as well as the window-scaling option that allow TCP to achieve high performance [JBB92]. An improved loss-recovery is provided thanks to the selective acknowledgements option (SACK) [MMFR96].

While the 3-way handshake allows to mark the beginning of the data stream, the end of the stream is signaled through a 4-way handshake that allows half-closed connections as shown in Figure 1.7. An end host that wants to stop sending data signals this through the FIN segment. The FIN segment represents one byte in the sequence number space and thus ensures that all prior data is correctly received. The peer must acknowledge the FIN to signal correct reception up to the sequence number of the FIN. A connection is only half-closed after such an exchange. A server is still allowed to send data to the client. As soon as it has also done the FIN-exchange, the connection is considered to be closed and the state on both sides can be freed.

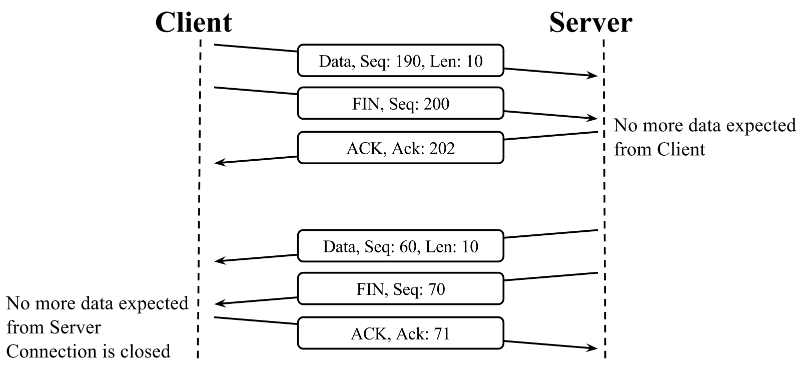


Figure 1.7: Even after the client has signaled a FIN, the server can still transmit data as the connection is only half-closed.

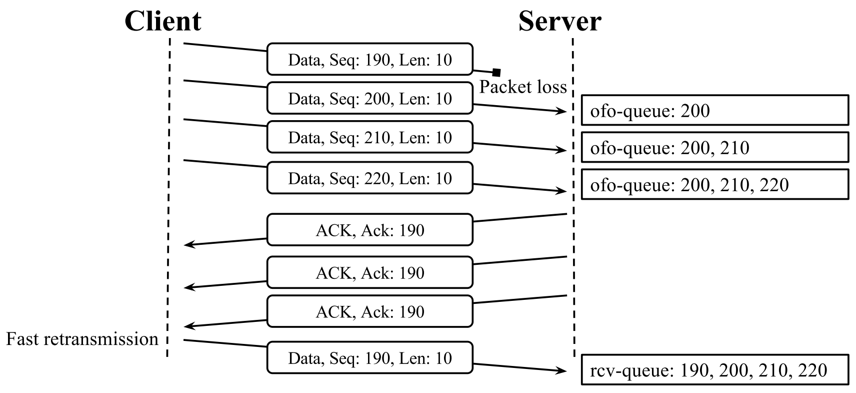


Figure 1.8: After three duplicate acknowledgments, the sender assumes that the corresponding segment has been lost and retransmits it.

1.2.2 Data plane

To ensure the reliable in-order delivery of the byte stream, TCP assigns a sequence number to each byte of the data stream. Starting from the initial sequence number (exchanged during the 3-way handshake), the sequence number increases for each byte pushed by the application on the socket. The sequence number is included inside the TCP header and maps to the first byte of the segment. This allows the receiver to actually bring the incoming segments back in order. To allow reliable transmission, the receiver sends acknowledgments back to the sender. The acknowledgment specifies the sequence number that is expected to be received next. Upon packet-loss, duplicate acknowledgments signal to the sender which packet is missing.

To recover from packet-loss, the sender counts the duplicate acknowledgments. Upon 3 duplicate acknowledgments, the sender assumes that the corresponding packet has been lost in the network (e.g., due to congestion) and retransmits it (shown in Figure 1.8). This mechanism is known as a fast-retransmission as it only takes one round-trip-time to recover from this packet loss. It is not always possible to receive sufficient duplicate acknowledgments to trigger a fast-retransmission. For example, if the last segment of a burst is lost, the receiver has no incentive to send three duplicate acknowledgments as it is not aware that one segment is still outstanding. For this purpose, the sender launches the retransmission timer which waits for the acknowledgment of the last byte sent. If after a retransmission timeout (RTO) period no acknowledgment has been received, the sender retransmits the missing packet. The RTO is typically significantly larger than the round-trip-time to avoid spurious retransmissions and is thus a much slower loss-recovery mechanism than fast-retransmissions.

On the receiver side, the TCP stack has to book a certain amount of memory, for different purposes. It may be that the application is slow at reading the data out of the receive-queue. Thus, the stack buffers the data in its receive-queue. It may also be that a packet loss occurs while more data is in-flight. As TCP ensures in-order delivery of the data, the receiver's stack must store this data in the out-of-order queue (shown in Figure 1.8) until the packet loss has been resolved. This amount of memory available to store the data in the receive-queue and the out-of-order queue must be communicated to the sender, so that this one does not attempt to send more data than memory available at the receiver. This is achieved thanks to the window-field of the TCP header. It indicates up to how many bytes (starting from the cumulative acknowledgment) the sender can emit. This process is the so-called *flow-control* of the TCP connection as it allows the receiver to throttle down the sending rate of the peer.

Finally, the TCP congestion control has been introduced in [Jac88] to prevent the congestion collapse of the Internet. It dynamically adapts the sending rate of a TCP connection in such a way that the capacity of the network is fully used, while still reducing the sending rate in case congestion is observed. These congestion events may be of different natures, depending on the congestion control algorithm. The most prominent ones are loss-based, interpreting a packet-loss as a sign of congestion [ATRK10]. Delay-based congestion controls have also been suggested [BP95].

The congestion control achieves this rate control by setting the size of the congestion window of a TCP connection. This window defines how much of in-flight data is allowed for this TCP connection. At the beginning, the initial congestion window is set to 10 segments [DRC+10]. As long as no congestion event is detected, the TCP connection goes through the slow-start phase. During this phase the congestion window is increased by one Maximum Segment Size (MSS) for each MSS of data that is being acknowledged by the receiver. As soon as a congestion event happens, the additive increase multiplicative decrease algorithm (AIMD) from [Jac88] mandates to enter the congestion-avoidance phase. The congestion window is divided by two (multiplicative decrease) and from now on the increase-rate of the window is of an additive nature: when one window worth of data has been acknowledged, the congestion window is increased by one MSS. This two-fold mechanism (slow-start and congestion-avoidance) allows to first probe the network for its congestion-point by aggressively increasing the sending rate at the beginning and then is less aggressive during the additive increase, trying to avoid congesting the network.

1.3 Conclusion

Multiple paths often exist between two end hosts on the Internet. To exploit these paths, packets must be steered alongside them. This is only possible by adapting the IP addresses and/or port numbers in the packets. This, because a different interface of the end host (e.g., WiFi and 3G on a smartphone) has been assigned another IP address by the carrier. A host striping segments of a data stream across different interfaces must use a different IP address for each interface. However, one of the limitations of TCP is that IP addresses are intrinsically linked to the TCP connection because the 4-tuple is the identifier of the connection. A host receiving such multiplexed segments is not able to identify to which data stream they belong. It is not possible to send the data stream of one TCP connection across different interfaces at the same time, effectively preventing TCP from using the multiple paths available to the end hosts.

We wish to move from a single-path Internet to one where the robustness, performance and load-balancing benefits of multipath transport are available to all applications. As has been shown in this chapter, the choice naturally fell on extending TCP for multipath capabilities. It is the right layer for multipath, as it is able to measure the characteristics of each path. Further, as most applications use TCP for their data transmission, a multipath transport should provide the same service: byte-oriented, reliable and in-order delivery.

The extension being standardized at the IETF is called Multipath TCP (MPTCP) [FRHB13] and tries to achieve four goals.

First, since Multipath TCP provides a reliable bytestream like TCP, unmodified applications should be able to use it via the standard socket API. If both endpoints support Multipath TCP, it's up to the operating system to create additional subflows, steered along each available path. Data will then be striped across these different paths.

Second, in order to make the solution deployable, Multipath TCP must work in all scenarios where regular TCP currently works. There are various types of middleboxes on the Internet that may interfere with the new protocol (e.g., blocking or modifying segments which look unfamiliar to the middlebox). Multipath TCP must be usable across all networks where regular TCP works. Therefore, a fallback mechanism must be included so that Multipath TCP can always revert back to regular TCP to preserve connectivity.

Third, Multipath TCP must be able to efficiently utilize the network at least as well as regular TCP, but without starving regular TCP. The congestion control scheme described in [WRGH11] meets this requirement, but congestion control is not the only factor that can limit throughput, as shown in this thesis.

Finally Multipath TCP must be implementable in operating systems without using excessive memory or processing and in such a way that the implementation does not negatively impact other parts of the TCP stack.

This thesis is all about **improving Multipath TCP** in such a way that it is suitable to transition from today's single-path Internet to one where hosts achieve better performance and robustness by using multiple paths. In order to achieve this, we must make sure that Multipath TCP works well in a wide range of environments. However, it is very difficult to cover this wide range of environments with such a complex system like Multipath TCP. Further, heterogeneous environments might be troublesome for Multipath TCP as different paths experience different congestion and round-trip-times. New solutions are necessary to handle such environments. Finally, deploying Multipath TCP requires a high-performing operating system implementation, tightly integrated in the existing TCP stack, without disturbing the well-functioning of TCP. An implementation must make certain design decisions and compromises to achieve such a goal.

This thesis tackles the above mentioned challenges. It starts by explaining in detail the key mechanisms used by Multipath TCP in Chapter 2. It continues in Chapter 3 by explaining how to efficiently and accurately evaluate a multipath transport protocol like Multipath TCP. How to achieve high performance in heterogeneous environments is explained in Chapter 4. A Linux Kernel prototype implementation of Multipath TCP has been brought to a high-performing, scalable and robust reference implementation. The details of this implementation are presented in Chapter 5. Finally, Chapter 6 discusses how Multipath TCP could have been designed taking into account the experience acquired during this thesis. The thesis then concludes in Chapter 7.

Chapter 2

Multipath TCP

Multipath TCP achieves resource pooling [WHB08] and increases the resilience to link failures by sending traffic across (possibly) diverse paths. This idea of using multiple paths for the transmission of data between end hosts is not new.

The previous chapter has outlined why the current Internet does not easily allow to achieve multipath transmission with today's protocols. It has also been shown that middleboxes are ubiquitous on the Internet [HNR+11, SHS+12] and must be considered while designing new protocols. Multipath TCP is designed with all these problems in mind. More specifically, the design goals for Multipath TCP are [RPB+12]:

- It should be capable of using multiple network paths for a single connection.
- It must be able to use the available network paths at least as well as regular TCP, but without starving TCP.
- It must be as usable as regular TCP for existing applications (thus, present the same API as TCP to the applications).
- Enabling Multipath TCP must not prevent connectivity on a path where regular TCP works (thus, middleboxes/firewalls must be supported).

As Multipath TCP must be as usable as regular TCP for existing applications, it means that Multipath TCP has to provide a reliable and in-order byte transmission service. This implies that Multipath TCP must be a stateful protocol, that includes a connection setup-phase, during which the start of the byte stream is being signaled. Additionally, Multipath TCP needs reliable transmission by using an acknowledgement mechanism, as well as a means for the receiver to bring incoming data back in-order in case of out-of-order reception. Further, a Multipath

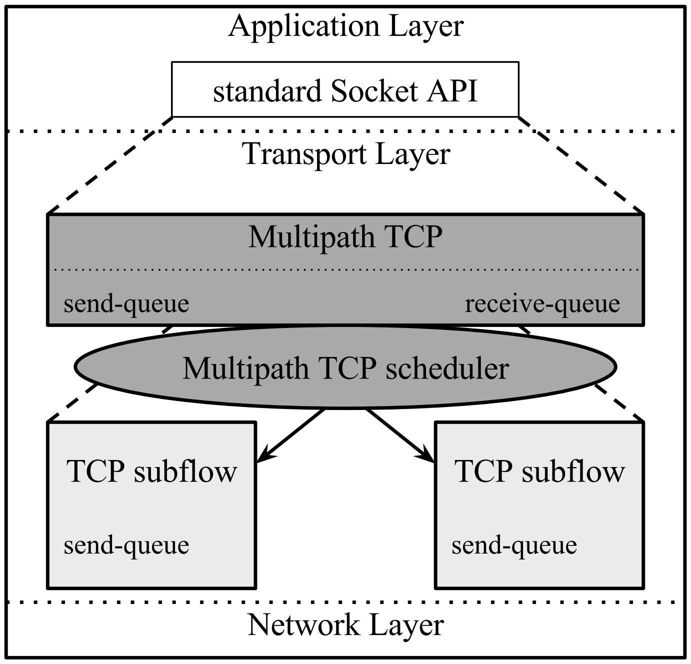


Figure 2.1: Multipath TCP creates one TCP subflow per path, so that the scheduler can distribute the data among these.

TCP connection should also allow the receiver to do flow control on the connection to prevent a sender from overwhelming the receiver with data and using up all its buffers. Finally, a proper teardown of the connection is necessary, so that end hosts can signal the end of the byte stream.

The simplest possible way to implement Multipath TCP would be to take segments coming out of the regular TCP stack and "stripe" them across the available paths. For this to work well, the sender would need to know which paths perform well and which don't: it would need to measure per path RTTs to quickly and accurately detect losses. To achieve these goals, the sender must remember which segments it sent on each path and use TCP Selective Acknowledgements to learn which segments arrive. Using this information, the sender could drive retransmissions independently on each path and maintain congestion control state.

This simple design has one fatal flaw: on each path, Multipath TCP would appear as a discontinuous TCP bytestream, which will upset many middleboxes (a study has shown that a third of paths will break such connections [HNR+11]). To achieve robust, high performance multipath operation, we need more substantial changes to TCP.

Figure 2.1 illustrates the architecture of a Multipath TCP implementation. To the application a standard stream socket interface is presented. Below, Multipath TCP is negotiated via new TCP options in the SYN packets of the TCP connection. To allow transmission across different paths, a *TCP subflow* is being created along each of these paths. They resemble regular TCP connections on the wire, including a 3-way handshake for the setup, a proper sequence number space with retransmissions and a 4-way handshake for the termination. These subflows are linked together to form the Multipath TCP connection and are used to carry the data between the end hosts. The *Multipath TCP Scheduler* is in charge of distributing the data across the different subflows - allowing to pool the resources of each subflow's path. Each subflow uses its own sequence-number space to detect losses and drive retransmissions. Multipath TCP adds connection-level sequence numbers to allow reordering at the receiver. Finally, connection-level acknowledgements are used to implement proper flow control.

In this chapter, the above outlined building blocks of Multipath TCP are explained in detail. Conceptually, Multipath TCP can be split in two parts: first, the control plane, responsible of creating and destroying subflows and signaling other connection-level control information; second, the data plane, which transmits the data between the end hosts. As a third part, the middlebox traversal of Multipath TCP is explained, which effectively allows Multipath TCP to be deployable on the Internet.

2.1 Control plane

The control information between two Multipath TCP-enabled end hosts is sent within the TCP option space. Multipath TCP uses a single TCP option type, and differentiates the control information using subtypes. The following is a description of the different Multipath TCP mechanisms that allow to create and destroy subflows as well as enable mobility and signal subflow-priorities.

2.1.1 Initial handshake

The TCP three-way handshake serves to synchronize state between the client and server[1]. In particular, initial sequence numbers are exchanged and acknowledged, and TCP options carried in the SYN and SYN/ACK packets are used to negotiate optional functionality like the maximum segment size (MSS) or support for TCP timestamps [JBB92]. During Multipath TCP's handshake the end hosts detect whether the peer actually supports Multipath TCP. Additionally, three state variables of the connection are being exchanged:

[1]The correct terms should be *active opener* and *passive opener*. For conciseness, we use the terms client and server, but we do not imply any additional limitations on TCP usage.

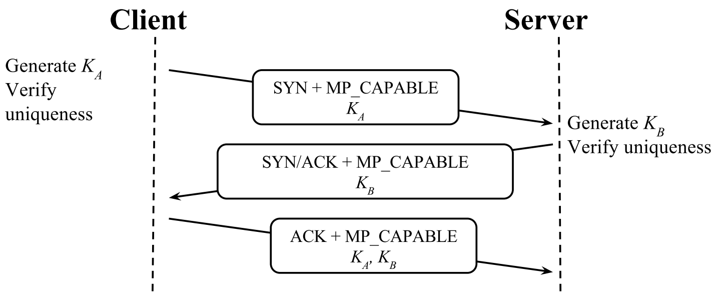


Figure 2.2: The handshake of the initial Multipath TCP subflow uses the MP_CAPABLE option to negotiate the necessary information.

1. Regular TCP identifies a connection thanks to the 5-tuple of the packets. Multipath TCP combines several TCP subflows in a single connection, each having a different 5-tuple. As new subflows may join a Multipath TCP connection, the latter must be identified. This identification is achieved thanks to a locally unique 32-bit identifier, called *token* [FRHB13].

2. In order to allow the transmission of data, a data sequence number is used. The data sequence number defines the position of a segment inside the data stream, explained in detail in the following Section 2.2. In order to start sending data, both hosts must agree on an *Initial Data Sequence Number (IDSN)*.

3. Multipath TCP tries to verify that the host opening a new subflow is indeed the same host as the one of the other TCP subflows. This is achieved thanks to two exchanged 64-bit *keys* and a cryptographic mechanism explained below. These keys serve as a shared secret and thus authenticate the end hosts.

These three elements (token, IDSN, and key) are exchanged during the handshake of the initial subflow.

Multipath TCP starts the connection by establishing an initial TCP subflow with a standard TCP 3-way handshake (Figure 2.2). It uses TCP options to add signaling into the 3-way handshake in order to detect whether the peer supports Multipath TCP and to negotiate the above described three elements. Detecting whether the peer supports Multipath TCP is done by adding the MP_CAPABLE option inside the SYN packets. This is a standard way of negotiating the support of a TCP extension between end hosts [JBB92]. Exchanging the keys is achieved by sending them in clear inside the MP_CAPABLE option. The keys are echoed

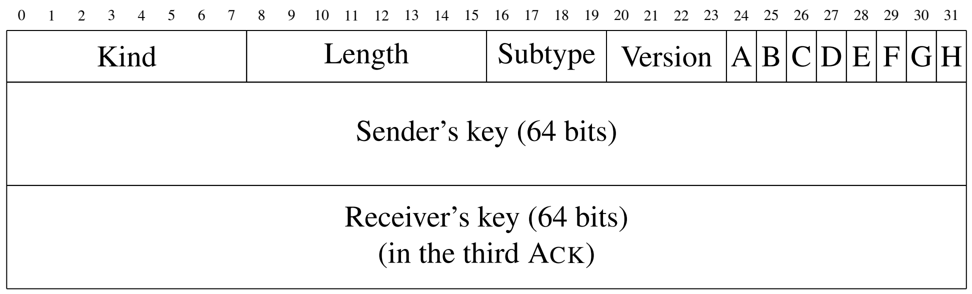


Figure 2.3: The MP_CAPABLE option exchanges the keys in clear. Token and IDSN are derived through a hash-function from these keys.

back in the third ACK of the 3-way handshake to support servers handling the TCP handshake in a stateless manner [Edd06].

One important limitation has to be considered – the maximum size of the TCP option field. No more than 40 bytes can be placed inside the TCP option space [Pos81b]. To minimize the length of the MP_CAPABLE options, the token and the IDSN are derived from the keys. The token is computed as the 32 most significant bits of the hash of the key[2]. As the token must be locally unique and since hash-collisions may happen with already existing Multipath TCP connections, its uniqueness must be verified among all established Multipath TCP connections [FRHB13]. Similarly, the IDSN is computed as the 64 least significant bits of the hash of the keys. Thus, only the keys need to be exchanged inside the MP_CAPABLE option, which is using the format shown in Figure 2.3. It can be seen that the option has a different length when being used within the third ACK to allow echoing back both keys. Thanks to this echoing, a server handling the 3-way handshake in a stateless manner can thus learn that the 3-way handshake negotiated Multipath TCP support and derive the keys being used for this connection.

2.1.2 Handshake of additional subflows

When adding a new subflow to a Multipath TCP connection, two problems must be solved. First, the new subflow needs to be associated with an existing Multipath TCP connection. The classical 5-tuple cannot be used as a connection identifier, as it may change due to NATs. Second, Multipath TCP must be robust to an attacker that attempts to add his own subflow to an existing connection. Multipath TCP solves these two problems, first by using the locally unique token, and

[2]The hashing algorithm is specified by additional bits inside the MP_CAPABLE-option, as explained in [FRHB13].

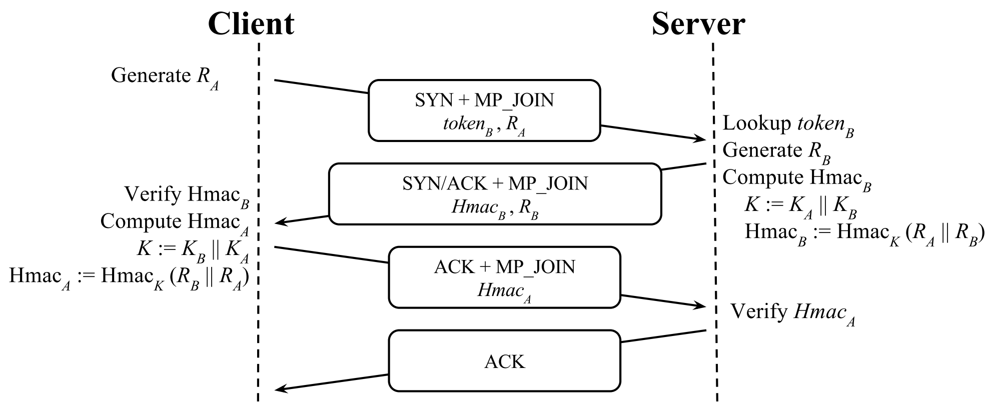


Figure 2.4: The token serves as a connection identifier, while the HMAC-exchange allows to authenticate the end hosts, while creating additional subflows.

second by computing and verifying an HMAC, using the exchanged keys within the 3-way handshake of the initial subflow.

To open a new subflow, Multipath TCP performs a new SYN exchange using the addresses and ports it wishes to use. The handshake is illustrated in Figure 2.4. A TCP option, MP_JOIN, is added to the SYN. The token included inside the SYN allows to identify the Multipath TCP connection this subflow belongs to, while the 32 random bits (R_A) will be part of the input for the HMAC computation. The server computes the HMAC$_B$, based on the two random numbers R_A and R_B and the keys exchanged in the initial handshake. Upon reception of the SYN/ACK, the client can then verify the correctness of the HMAC$_B$, which proves that the server has knowledge of the keys K_A and K_B and thus participated in the initial handshake. Next, the client generates a different HMAC$_A$ which is included inside the third ACK. This one enables the server to verify on its own that the client is the same as the one involved in the initial handshake. Finally, the server sends a duplicate acknowledgement to the client, signaling the reception of this third ACK. The client will only start sending data to the server once it has received the fourth ACK.

The subflow is now linked to the Multipath TCP connection (thanks to the token) and it has been proven that the hosts behind this subflow have participated in the key-exchange of the initial subflow (thanks to the HMAC's). Now, this subflow can be used to transmit data.

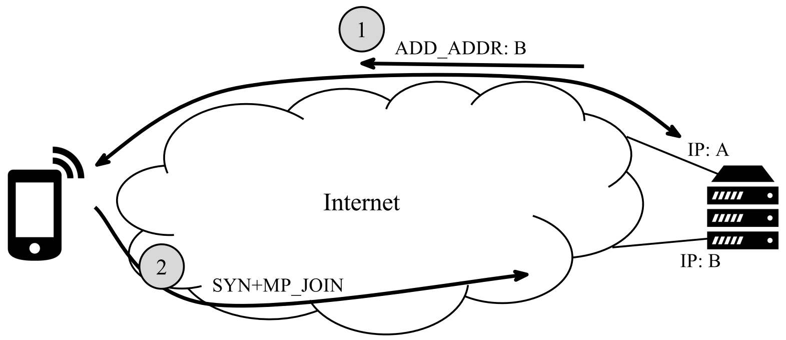


Figure 2.5: First, the server announces its additional address via an existing subflow. Then, the client can establish an additional subflow to the announced address

2.1.3 Address agility

If a client is multihomed, then it can easily initiate new subflows from any additional IP addresses it owns. However, if only the server is multihomed, the wide prevalence of NATs makes it unlikely that a new SYN will be received by a client. The solution is for the Multipath TCP server to inform the client that the server has an additional address by sending an ADD_ADDR option on a segment on one of the existing subflows (illustrated in Figure 2.5). The ADD_ADDR option is one of the subtypes of the TCP options reserved for Multipath TCP.

The client may then initiate a new subflow. This asymmetry is not inherent - there is no protocol design limitation that means the client cannot send ADD_ADDR or the server cannot send a SYN for a new subflow.

The announced address may be an IPv4 or an IPv6 address - irregardless of the IP-version of the initial subflow. This means that Multipath TCP allows a data stream to be sent via IPv4 and IPv6 simultaneously. This could be seen as a facilitator for the deployment of IPv6.

It must be noted that each address is assigned to a so-called *address-ID*. This is an 8-bit integer, that locally identifies the IP-address. It is being used within the MP_JOIN option so that a host knows which pair of address-IDs is associated to each subflow. The address-ID is also part of the ADD_ADDR option. The use of the address-ID is mainly for the REMOVE_ADDR option, which is used to inform the peer when one of its addresses has become unavailable (e.g. the mobile node is not anymore connected to the WiFi access point). It specifies the address-ID that is associated to the IP-address that should be removed. Upon reception of this option the destination closes all TCP subflows that are using this address. The address-ID allows the host to identify which subflows to close, even if these ones have their public IP-address changed due to a NAT device. The NAT would not change the address-ID and so the ID can serve as an end-to-end identifier of the IP-address.

2.1.4 Further protocol details

When Multipath TCP is being used across different interfaces on an end-host, it may be that the end user has different priorities on which he/she might want to send traffic. E.g., on a smartphone, the user might want to avoid sending traffic over the 3G interface to avoid the monetary cost of using the cellular network. Multipath TCP incorporates a mechanism that allows to specify priorities on a subflow. The MP_JOIN option includes a backup-bit that allows to signal to the peer that it should not send any data on this subflow. This requirement can be violated by the peer, if there is no other subflow available for data transmission. To signal this kind of subflow priority after the 3-way handshake, the MP_PRIO option is part of Multipath TCP. The backup-bit and MP_PRIO effectively allow a smartphone to reduce the traffic over the 3G interface, while still benefiting from Multipath TCP's handover mechanism.

TCP specifies the RST-bit to abruptly close a connection [Pos81b]. Although, a TCP-reset should only be sent upon reception of a packet whose 5-tuple does not match to an existing socket, implementations have made use of TCP-reset to force the closure of a connection (e.g., when the end host experiences memory-pressure and needs to redeem resources). Within Multipath TCP it is not sufficient to just send a TCP-reset, because this will only close the individual subflows, but not the whole Multipath TCP connection. Thus, the specification of Multipath TCP includes the fast-close option to signal the abrupt closure of the whole stream. This option includes the key of the peer to protect against attackers injecting fast-close messages and thus execute denial-of-service attacks. While sending the fast-close option, the host also sends a TCP-reset on all other subflows. Upon reception of the option, the host is aware that the connection shall be treated as if a TCP-reset has been received in regular TCP (e.g., present the `ECONNRESET` error-code to the application). Furthermore, all remaining subflows are closed with a regular TCP-reset.

2.2 Data plane

The data plane is the part of Multipath TCP in charge of the actual transmission of the byte stream sent and received by the application. The application assumes a reliable and in-order delivery of the byte stream, a requirement that influenced the design of Multipath TCP as explained in this section.

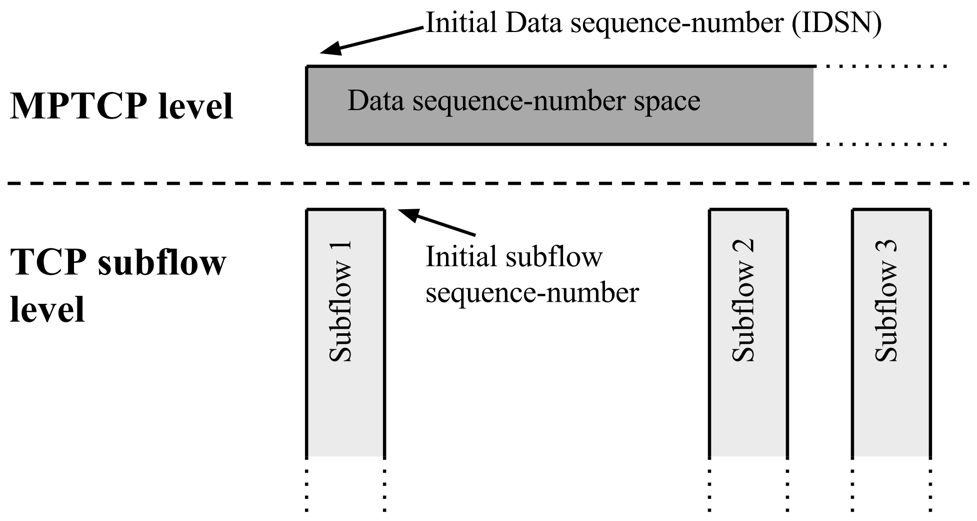

Initial Data sequence-number (IDSN)
MPTCP level
Data sequence-number space
TCP subflow level
Subflow 1
Initial subflow sequence-number
Subflow 2
Subflow 3

Figure 2.6: The Multipath TCP level has its own sequence number space, while each subflow also occupies its own sequence space.

2.2.1 A second sequence-number space

The previous section illustrates why there is a need to create one TCP subflow per path. This implies that each TCP subflow occupies its own sequence-number space, as shown in Figure 2.6. Multipath TCP being used to transmit a data stream in a reliable and in-order manner, it is necessary to enable the receiver to reorder the possibly out-of-order data segments. This can be achieved thanks to the data-sequence-number space. The data-sequence-number space maps each byte to its position within the continuous data stream that is being sent by the application.

The data sequence number space starts at the initial data sequence number (IDSN). It is 64-bit long to prevent any issues with wrapped sequence numbers. In fact, a packet sent over a high-delay path might take so long to reach the destination, that a large amount of data might already have been sent over a low-delay, high-bandwidth path. So much data, that a 32-bit sequence number space might have wrapped around. If 32-bit sequence numbers were used, the receiver would not be able to distinguish the packet with the old sequence number, from up-to-date data on the low-delay path. The 64-bit sequence number space should be sufficient to protect against wrapped sequence numbers, as 2^{64} bytes must be sent on the low-delay path in order to have this kind of wrapping.

Further, each subflow has its subflow sequence number. This one is entirely independent of the data sequence number, and its initial value is being exchanged during the 3-way handshake of each subflow. The subflow sequence number al-

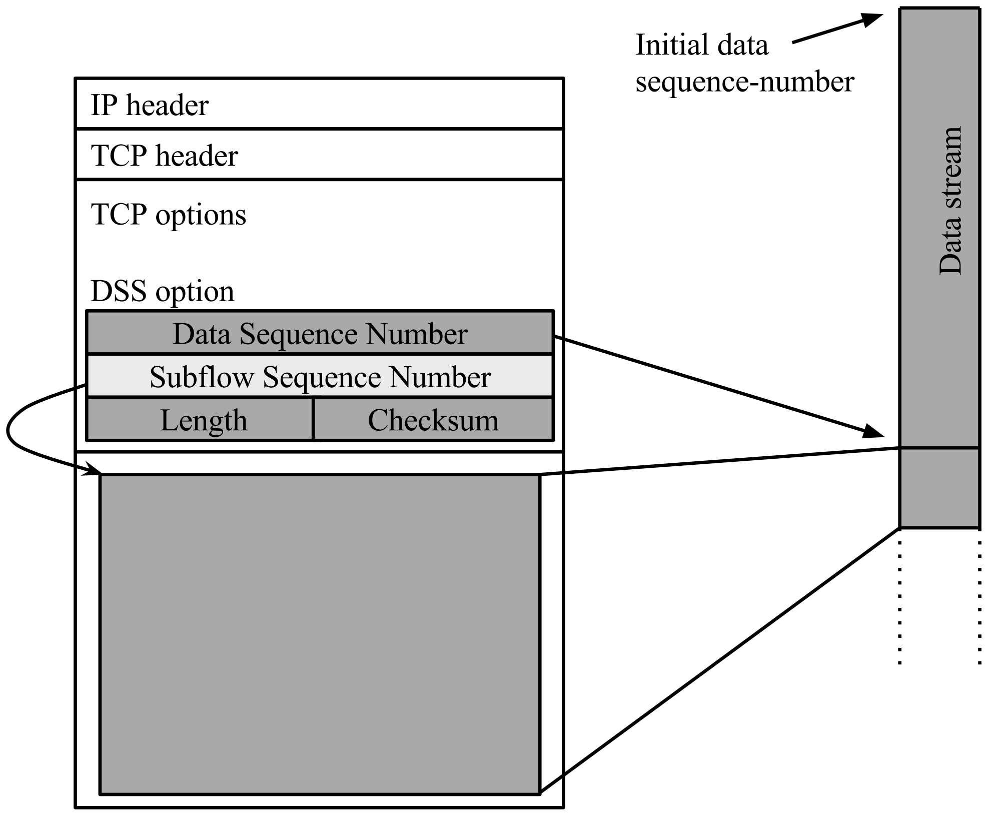


Figure 2.7: The DSS-option maps each subflow-level byte to its corresponding byte at the data-sequence level.

lows to mimic the continuous byte stream to the middleboxes. Each subflow effectively appears like a regular TCP connection, implementing a full TCP state machine together with its retransmissions.

2.2.2 Data-plane signaling

Upon receiving data on a TCP subflow, the receiver must know the data-sequence number to pass the data to the application. A simple approach would be to include the data-sequence number in a TCP option inside each segment. Unfortunately, this clean solution does not work. There are deployed middleboxes that split segments on the Internet. In particular, all modern NICs (network interface cards) act as segment-splitting middleboxes when performing TSO (TCP segmentation offloading). These NICs split a single segment into smaller pieces. In the case of TSO, CPU cycles are offloaded from the operating system to the

NIC, as the operating system handles fewer, larger segments that are split down to MTU (maximum transmission unit)-sized segments by the NIC. The TCP options, including the Multipath TCP data-sequence number of the large segment, are copied in each smaller segment. As a result, the receiver will collect several segments with the same data-sequence numbers, and be unable to reconstruct the data stream correctly. The Multipath TCP designers solved this problem by placing a mapping in the data-sequence signal option (illustrated in Figure 2.7), which defines the beginning (with respect to the subflow sequence number) and the end of the data-sequence number (indicating the length of the mapping). Multipath TCP can thus correctly work across segment-splitting middleboxes and can be used with NICs that use TCP segmentation offloading to improve performance.

Using the first Multipath TCP implementation in the Linux kernel to perform measurements revealed another type of middlebox. Since the implementation worked well in the lab, it was installed on remote servers. The first experiment was disappointing. A Multipath TCP connection was established but could not transfer any data. The same kernel worked perfectly in the lab, but no one could understand why longer delays would prevent data transfer. The culprit turned out to be a local firewall that was changing the sequence numbers of all TCP segments. This feature was added to firewalls several years ago to prevent security issues with hosts that do not use random initial sequence numbers. In some sense, the firewall was fixing a security problem in older TCP stacks, but in trying to solve this problem, it created another one. The mapping from subflow-sequence number to data-sequence number was wrong as the firewall modified the former. Since then, the mapping in the data-sequence option uses relative subflow-sequence numbers compared with the initial sequence number, instead of using absolute sequence numbers [FRHB13].

The data-sequence option thus accurately maps each byte from the subflow-sequence space to the data-sequence space, allowing the receiver to reconstruct the data stream. Some middleboxes may still disturb this process: the application-level gateways that modify the payload of segments. The canonical example is active FTP. FTP uses several TCP connections that are signaled by exchanging ASCII-encoded IP addresses as command parameters on the control connection. To support active FTP, NAT boxes have to modify the private IP address that is being sent in ASCII by the client host. This implies a modification of not only the content of the payload, but also, sometimes, its length, since the public IP address of the NAT device may have a different length in ASCII representation. Such a change in the payload length will make the mapping from subflow-sequence space to data-sequence space incorrect. Multipath TCP can even handle such middleboxes thanks to a checksum that protects the payload of each mapping. If an application-level gateway modifies the payload, then the checksum will be corrupted; Multipath TCP will be able to detect the payload change and perform a

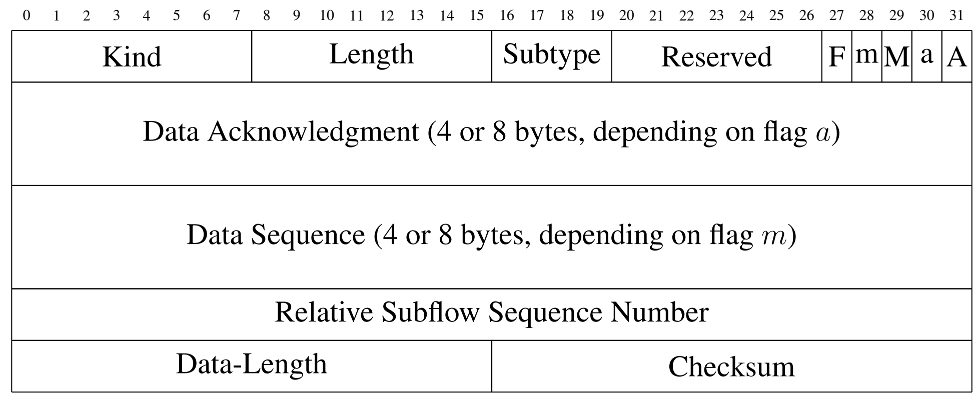

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Kind
Length
Subtype
Reserved
F
m
M
a
A
Data Acknowledgment (4 or 8 bytes, depending on flag a)
Data Sequence (4 or 8 bytes, depending on flag m)
Relative Subflow Sequence Number
Data-Length
Checksum

Figure 2.8: The DSS-option allows the receiver to detect the mapping of the data. The Data Ack allows proper flow control and reliable delivery.

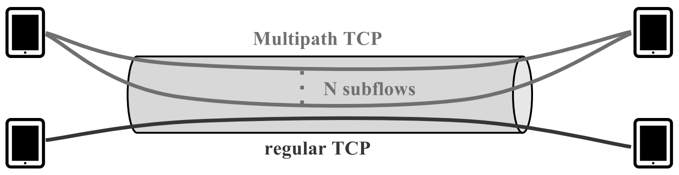


Figure 2.9: If regular TCP congestion control is being used, Multipath TCP would use much more capacity across shared bottlenecks and be unfair to regular TCP.

seamless fallback to regular TCP to preserve the connectivity between the hosts. This fallback is explained in detail in Section 2.3.

These different informations, allowing a receiver to detect the data sequence-number and protecting itself against payload-modifying middleboxes, lead to the format of the DSS-option as shown in Figure 2.8. The data sequence number may be 4 or 8 bytes long, depending on the m-flag. Further, the DSS-option also contains a field to announce the cumulative data-level acknowledgment. This allows to signal to the sender up to which data sequence-number the peer has successfully received the data. It also allows the host to do flow control, as the right edge of the window is calculated based on the cumulative data-acknowledgment.

2.2.3 Congestion control

From a congestion-control viewpoint, using several subflows for one connection leads to an interesting problem. With regular TCP, congestion occurs on one path between the sender and the receiver. Multipath TCP uses several paths. Two

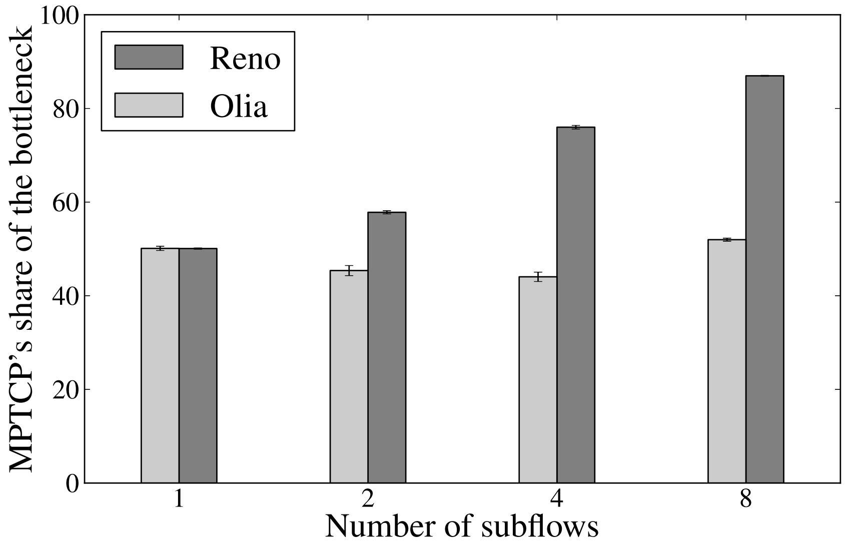


Figure 2.10: The Olia congestion control is fair to regular TCP across shared bottlenecks.

paths will typically experience different levels of congestion. A naive solution to the congestion problem in Multipath TCP would be to use the standard TCP congestion-control scheme on each subflow. This can be easily implemented but leads to unfairness with regular TCP. In the network depicted in Figure 2.9, two clients share the same bottleneck link. If the Multipath TCP-enabled client uses two subflows, then it will obtain two-thirds of the shared bottleneck. This is unfair because if this client used regular TCP, it would obtain only half of the shared bottleneck.

Figure 2.10 shows the effect of using up to eight subflows across a shared bottleneck with the standard TCP congestion-control scheme (Reno). These experiments use a setup equivalent to that shown in Figure 2.9. Two Multipath TCP capable hosts are sending data through one up to eight subflows, while two hosts using regular TCP are transmitting through the same bottleneck. The figure shows how much of the bottleneck's capacity is used by Multipath TCP. With a standard TCP congestion control, Multipath TCP uses up to 85 percent of the bottleneck's capacity, effectively starving regular TCP. This, because each subflow increases its congestion window independently of the other subflows. We call this an uncoupled congestion control.

Specific Multipath TCP congestion-control schemes have been designed to solve this problem [KGPLB13, WRGH11]. Briefly, they measure congestion on each subflow and try to move traffic away from those with the highest congestion. To do so, they modify the additive increase during the congestion avoidance phase. The congestion information of all the subflows belonging to a connection is taken into account to control the increase rate of a subflow's congestion window during the congestion avoidance phase. Thus, they are coupling the increase-phase of the subflow's congestion control. Figure 2.10 shows that the OLIA congestion-control scheme [KGPLB13] preserves fairness with regular TCP across the shared bottleneck.

2.2.4 Example

The Linux Kernel implementation of Multipath TCP[3] implements the described specification of Multipath TCP. An example session, captured with `tcpdump`, is shown in Figure 2.11. Let's have a close look at this packet exchange:

Packets 1 to 3 show the 3-way handshake of the initial subflow. The MP_CAPABLE option is inside the TCP option space, indicating the chosen key. The subflows have their own subflow-level TCP sequence number (starting at 2351712563 for the traffic from client to server and starting at 2433953981 in the reverse direction).

The data exchange can be seen in packet number 4, where 250 bytes are sent with a data sequence number of 2751382862. As one can see, the `subseq` in the DSS-option is set to 1, the same as the relative subflow sequence number in the TCP header (`tcpdump` converts the absolute sequence number in the header to a relative sequence number). The subsequent data acknowledgement in packet 5 signals the correct reception of of these 250 bytes of data. Additionally, packet 5 contains an ADD_ADDR option, signaling to the client an IPv6 address.

As the client also owns an IPv6 address, it can now establish an additional subflow to the server. This example shows that Multipath TCP is indeed able to use IPv4 and IPv6 simultaneously. Segments 6 to 9 show the handshake used to establish the additional subflow over IPv6. The MP_JOIN option is included in the SYN, SYN/ACK and the third ACK, while segment number 9 acknowledges the correct reception of the third ACK.

The data transmission is then continued over IPv6 with segments 10 and 11. The closure is initiated by the server in segment 12, containing the DATA_FIN. The DATA_FIN is inside a segment without payload. Thus, it cannot be mapped to a byte at the subflow-level and thus has subflow-sequence number 0 inside the DSS-mapping. At the data-level it consumes the sequence number 3282440237 and is acknowledged in segment 13, where the data acknowledgment is 3282440238.

[3]Available at `http://multipath-tcp.org`.

```
1:  IP 1.1.1.1.60210 > 2.2.2.2.3128: Flags [S], seq 2351712563, win 29200,
        options [mss 1460,sackOK,nop,nop,nop,wscale 7,mptcp capable csum {0xa88c1b6222961dd2}], length 0
2:  IP 2.2.2.2.3128 > 1.1.1.1.60210: Flags [S.], seq 2433953981, ack 2351712564, win 28560,
        options [mss 1460,sackOK,nop,nop,nop,wscale 7,mptcp capable csum {0x9a7a0b2d92ef9685}], length 0
3:  IP 1.1.1.1.60210 > 2.2.2.2.3128: Flags [.], ack 1, win 457,
        options [mptcp capable csum {0xa88c1b6222961dd2,0x9a7a0b2d92ef9685},
                 mptcp dss ack 3282439960], length 0
4:  IP 1.1.1.1.60210 > 2.2.2.2.3128: Flags [P.], seq 1:251, ack 1, win 457,
        options [mptcp dss ack 3282439960 seq 2751382862 subseq 1 len 250 csum 0x8adc], length 250
5:  IP 2.2.2.2.3128 > 1.1.1.1.60210: Flags [.], ack 251, win 455,
        options [mptcp add-addr id 9 2001::bbbb,mptcp dss ack 2751383112], length 0
6:  IP6 2001::aaaa.53813 > 2001::bbbb.3128: Flags [S], seq 4141089230, win 28800,
        options [mss 1440,sackOK,nop,nop,nop,wscale 7,mptcp join id 9 token 0xe71e492f nonce 0xb1d91e38], length 0
7:  IP6 2001::bbbb.3128 > 2001::aaaa.53813: Flags [S.], seq 2612309998, ack 4141089231, win 28160,
        options [mss 1440,sackOK,nop,nop,nop,wscale 7,mptcp join id 9 hmac 0x6f9d57dc75c37b8d nonce 0xad60a1ba], length 0
8:  IP6 2001::aaaa.53813 > 2001::bbbb.3128: Flags [.], ack 1, win 907,
        options [mptcp join hmac 0xfb2702289c46cd26864cb4e453a2be8abad8996c], length 0
9:  IP6 2001::bbbb.3128 > 2001::aaaa.53813: Flags [.], ack 1, win 895,
        options [mptcp dss ack 2751383112], length 0
10: IP6 2001::bbbb.3128 > 2001::aaaa.53813: Flags [P.], seq 1:229, ack 1, win 895,
        options [mptcp dss ack 2751383112 seq 3282439960 subseq 1 len 228 csum 0xe7bd], length 228
11: IP6 2001::bbbb.3128 > 2001::aaaa.53813: Flags [P.], seq 229:278, ack 1, win 895,
        options [mptcp dss ack 2751383112 seq 3282440188 subseq 229 len 49 csum 0x1db7], length 49
12: IP6 2001::bbbb.3128 > 2001::aaaa.53813: Flags [.], ack 1, win 895,
        options [mptcp dss fin ack 2751383112 seq 3282440237 subseq 0 len 1 csum 0xea44], length 0
13: IP6 2001::aaaa.53813 > 2001::bbbb.3128: Flags [.], ack 278, win 910,
        options [mptcp dss fin ack 3282440238 seq 2751383112 subseq 0 len 1 csum 0x837e], length 0
14: IP6 2001::bbbb.3128 > 2001::aaaa.53813: Flags [F.], seq 278, ack 1, win 895,
        options [mptcp dss ack 2751383112], length 0
15: IP 2.2.2.2.3128 > 1.1.1.1.60210: Flags [F.], seq 1, ack 251, win 895,
        options [mptcp dss ack 2751383112], length 0
16: IP 1.1.1.1.60210 > 2.2.2.2.3128: Flags [.], ack 2, win 910,
        options [mptcp dss ack 3282440238], length 0
17: IP6 2001::bbbb.3128 > 2001::aaaa.53813: Flags [.], ack 1, win 895,
        options [mptcp dss ack 2751383113], length 0
18: IP6 2001::aaaa.53813 > 2001::bbbb.3128: Flags [F.], seq 1, ack 279, win 910,
        options [mptcp dss ack 3282440238], length 0
19: IP 1.1.1.1.60210 > 2.2.2.2.3128: Flags [F.], seq 251, ack 2, win 910,
        options [mptcp dss ack 3282440238], length 0
20: IP6 2001::bbbb.3128 > 2001::aaaa.53813: Flags [.], ack 2, win 895,
        options [mptcp dss ack 2751383113], length 0
21: IP 2.2.2.2.3128 > 1.1.1.1.60210: Flags [.], ack 252, win 895,
        options [mptcp dss ack 2751383113], length 0
```

Figure 2.11: A sample session of Multipath TCP, created with `tcpdump` and the Linux Kernel implementation of Multipath TCP

Subsequent subflow-FIN properly clear the state of each subflow and terminate the connection.

2.3 Middleboxes

The original end-to-end principel [SRC84] of the Internet does not hold anymore in today's Internet. Middleboxes and firewalls are ubiquitous in all kinds of networks. The set-top boxes at our homes typically provide a NAT-service, cellular networks are infiltrated with all kinds of middleboxes [WQX+11] and not to speak about enterprise networks, where [SHS+12] has shown that there are as many middleboxes and firewalls as routers.

As Multipath TCP tries to be deployable on the Internet with all its middleboxes, it has been designed to work around undesirable middlebox behavior. It even falls back to regular TCP in case a middlebox does not allow a proper functioning of Multipath TCP. Some of the middlebox behaviors have already been mentioned in the previous section. The following describes in-detail the middlebox behavior that has been considered in the design of Multipath TCP and how it has been designed to live with them.

2.3.1 Middlebox behavior

TCP/IP segments are built upon the IP header, the TCP header and the payload. Each of these parts is built upon different fields as shown in Figure 2.12. The original end-to-end principle assumed that only the `TTL` and `Header checksum` of the IP-header are modified while forwarding. Additionally, if routers perform IP fragmentation the related fields of the IP-header will be modified. However, the deployment of middleboxes and firewalls within our networks has made this obsolete. [HNR+11] executed an extensive study of the Internet, to know which of these header fields still traverse the Internet unmodified.

The IP-addresses in the IP-header and the port numbers of the TCP-header may be subject to modification. The depletion of the IPv4 address space has accelerated the deployment of **Network Address Translation devices (NAT)**. A NAT device separates the network in a private, client-facing network and the public, Internet-facing side. The Internet only communicates with the public IP address of the NAT device. The NAT modifies the destination IP of incoming packets to match the IP address from the private address space on the client-facing side of the NAT. The reverse operation is being done for packets originating from the client-side to the Internet.

It has also been shown by [HNR+11] that middleboxes and firewalls may interfere with the TCP options within the TCP header. The middleboxes may **remove**

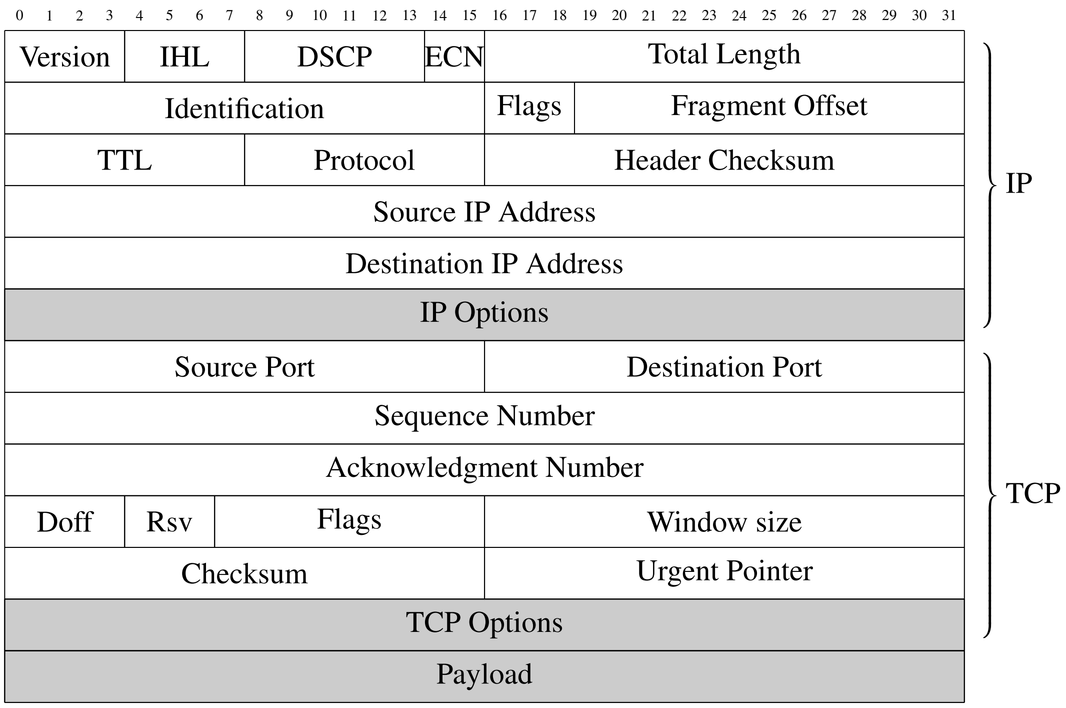


Figure 2.12: Almost every field of the TCP/IP-header, as well as the payload may be modified by a middlebox.

unknown TCP options from the SYN segments (and/or from data segments). It has even been observed that some firewalls drop SYN segments all together if an unknown TCP option is part of the TCP header.

Another field of the TCP header that was not meant to be modified in the original specification of RFC 793 is the sequence and acknowledgement number. However, a middlebox is "fixing" an old bug in a prominent TCP implementation. This implementation did not sufficiently randomize the initial TCP sequence number. Thus, the middlebox tries to protect the end hosts by **randomizing the sequence number** of each TCP flow [HDP+13]. An end host cannot assume that the sequence number it selects within its SYN is the one actually seen by the receiving host.

Network Interface Cards (NICs) are becoming more and more "intelligent", allowing the operating system's stack to offload functions to the NIC, in order to spare CPU cycles. TCP Segmentation Offloading (TSO) is one of these prominent techniques. It allows the OS to handle large TCP segments, leaving the splitting of these segments to the NIC. Thus, from the operating system's point of view, the NIC behaves like a **segment-splitting middlebox**, dividing a single segment into multiple smaller ones. There exist also middleboxes that regroup multiple segments in a single one, if the path's maximum transmission unit (MTU) provides enough space for it. Large Receive Offloading (LRO) of the NICs does a similar function, creating one bigger segment out of multiple smaller ones. For the Multipath TCP implementation on the receiver side, this looks like a **segment-coalescing middlebox**, which modifies the total length of the packets. End-hosts cannot assume that the boundaries of their emitted segments are maintained stable end-to-end.

WAN accelerators as well as intrusion detection systems may behave like a transparent TCP proxy, effectively splitting a TCP connection in two parts. A WAN accelerator **pro-actively acknowledges** data from an end-host and handles the retransmissions to the receiving end on its own. This results in shorter round-trip-times and thus in an increased TCP performance. Intrusion detection systems have to rebuild the byte stream in order to analyze the payload of the TCP connection and protect the end hosts from attackers. As these kind of devices have to buffer packets, they often modify the **window field** of the TCP header, to reduce their memory requirements. Yet another field of the TCP header which is not traversing the Internet in an unmodified way.

Application-level gateways might modify the content of the payload and in some cases even modify the length of a segment - thus shifting the whole sequence number space of a subflow. This kind of behavior can be observed in NAT devices that do support active FTP connections. When using active FTP, a client announces its IP address within the data stream to the server so that the latter can open a connection to the client for transferring the requested data. If the client is behind a NAT device, the NAT must modify the IP address inside the FTP stream as otherwise the server will see the client's private IP in the FTP stream. As the IP address is sent in ASCII-format within the payload, this modification by the FTP-aware NAT device may result in a modification of the length of the segment.

2.3.2 Protecting the control plane

Multipath TCP exchanges control information within TCP options. This is a standard way to exchange TCP-level information, like the Selective Acknowledgments (SACK) or TCP timestamps. However, as mentioned above, middleboxes may interfere with unknown TCP options. The key element of Multipath TCP's support for middleboxes is the fallback to regular TCP in case a middlebox interferes in such a way that it cannot work properly anymore. This ensures that Multipath TCP works everywhere, where regular TCP works correctly as well. The following shows how Multipath TCP protects the control plane against undesired middlebox interference.

If a middlebox removes the MP_CAPABLE option from a SYN, the server will

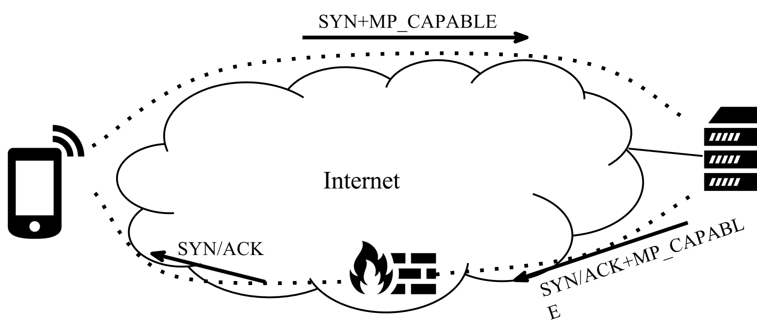


Figure 2.13: In case of asymmetric paths on the Internet, a middlebox may remove the TCP options only from the reverse-path, resulting in unsynchronized end hosts.

receive a SYN without the mentioned option. Thus, for the server the SYN appears like regular TCP. It thus replies to the client without the MP_CAPABLE option inside the SYN/ACK. As the client receives this SYN/ACK, the missing MP_CAPABLE option indicates that either the server does not support Multipath TCP or that a middlebox has removed the MP_CAPABLE. The client will thus pretend as if it never added the MP_CAPABLE option in the SYN and proceed with the 3-way handshake by sending the third ACK, falling back to regular TCP.

One particular case may be that a certain middlebox is only present on the reverse path from the server to the client as shown in Figure 2.13. If this middlebox removes the MP_CAPABLE option, both hosts will be in an unsynchronized state. The client effectively has received a SYN/ACK without the MP_CAPABLE option, believing that Multipath TCP is not being used. However, the server has received a SYN with MP_CAPABLE and replied with a SYN/ACK including the MP_CAPABLE. When finishing the 3-way handshake, the segments sent by the client will not include any Multipath TCP option. Data segments will not hold a DSS-option and acknowledgments do not include a data-acknowledgment. The server receiving these packets can thus detect that in fact Multipath TCP is not being used by the client and thus can behave as if regular TCP is being used. For this mechanism to work, the client must include a DSS-option within each of its segments at the beginning of the connection. A similar mechanism to fallback to regular TCP is being used when a middlebox removes the unknown TCP option only from non-SYN segments. In this case both hosts will trigger a seamless fallback to regular TCP, as they both do not see the DSS-option in the data segments nor in the acknowledgments.

Let's provide an anecdotal reference that the above described mechanisms work - even when facing a very unexpected behavior of a popular end host. While

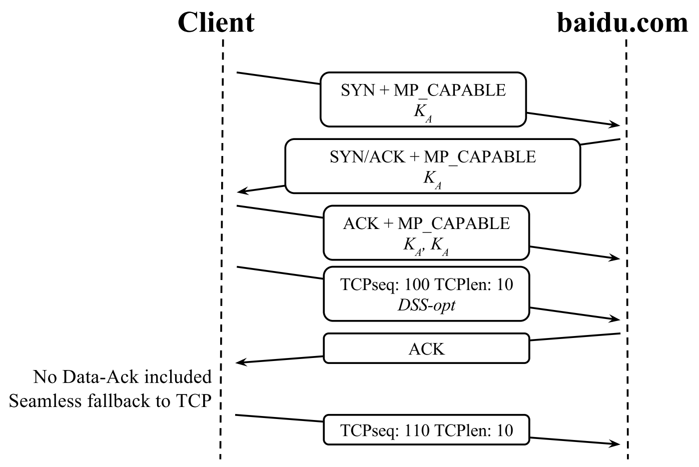


Figure 2.14: Even when facing an unusual behavior as when contacting baidu.com, Multipath TCP is able to fallback to regular TCP.

running some tests with our Multipath TCP implementation, we also established an HTTP connection to baidu.com, the popular Chinese web-search service. We observed that baidu.com replies in its SYN/ACK including the MP_CAPABLE option as shown in Figure 2.14. Looking closer at the MP_CAPABLE inside the SYN/ACK we realized that the key was exactly the same as the one chosen by our implementation when emitting the SYN. In fact, the baidu.com server (or load balancer) replies with a SYN/ACK that seems to be a copy of the SYN, with the exception that some fields have been modified. Within the TCP option space, all undesired *known* TCP options were replaced by the NOP option. However, the *unknown* MP_CAPABLE option has not been replaced by NOPs. Thus, baidu.com included inside its SYN/ACK the MP_CAPABLE option. Our Multipath TCP implementation thus believes that the server indeed supports Multipath TCP, including the DSS option inside its data segments. However, the baidu.com server will not use DSS options inside its data segments and acknowledgments. As described above, if a segment at the beginning of a Multipath TCP connection does not contain any Multipath TCP option, the host does a seamless fallback to regular TCP, behaving as if Multipath TCP has never been negotiated. Thus, after the client sent its first data segment (still including the DSS-option), the baidu-server will reply with an acknowledgement, not including a cumulative data-acknowledgment. This will trigger the seamless fallback procedure on the client, which will from now on behave as if Multipath TCP has not been negotiated. One can see that the design of Multipath TCP is able to handle even such weird behavior from a public web-server.

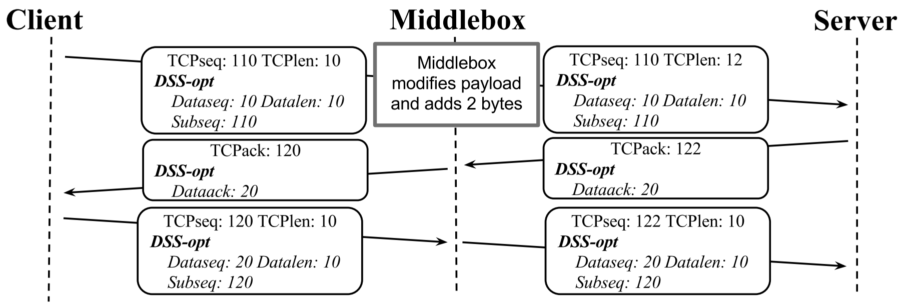


Figure 2.15: After the middlebox modified the byte-stream, the DSS-mapping is no more synchronized with the TCP subflow sequence numbers - as can be seen in the bottom right segment.

2.3.3 Protecting the data plane

The protection of the data plane has already been explained in Section 2.2. Notably, the DSS-mapping allows to protect against segment-splitting and segment-coalescing middleboxes as each subflow-level byte has its unique mapping to the data-level sequence defined within the DSS option. Further, the use of a relative subflow-sequence number allows to bypass sequence number randomizing middleboxes. The DSS-checksum allows to protect from payload-rewriting middleboxes. We now explain how the fallback to regular TCP is performed in case the DSS checksum is incorrect.

When a payload-rewriting middlebox has modified the length of a segment, the subflow-sequence space cannot be mapped accurately to the data-sequence space. This case is illustrated in Figure 2.15. The middlebox modified the payload, and while doing so added 2 bytes to the segment with TCP-sequence number 110. If no checksum would be used inside the DSS-option, transmission would continue as normal. The following segment emitted by the sender (TCP-sequence 120) will then be modified again by the middlebox. This time, the middlebox only modifies the sequence number. It previously added 2 bytes to a segment, thus it increments the sequence number to 122. The issue is that the middlebox will not modify the relative subflow-sequence number in the DSS-option. This one is still pointing to 120. The receiver is thus unable to accurately map this last segment to the correct sequence-number space and data corruption will happen. Multipath TCP tries to detect this kind of middlebox behavior by using the checksum over the payload of each mapping to the DSS-option. As soon as a checksum failure has been detected, Multipath TCP will fallback to regular TCP.

The fallback is initiated through the MP_FAIL option inside an acknowledgment sent by the host who detected the checksum failure (an example of the

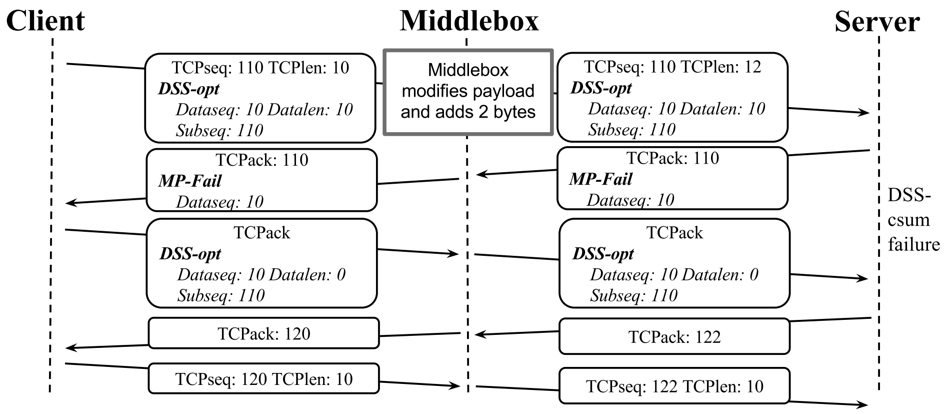


Figure 2.16: The infinite mapping announces to the receiver that from the specified subflow-sequence number on, the mapping will be implicit as all data will be sent in-order on the single subflow.

fallback is shown in Figure 2.16). The MP_FAIL option indicates which data-sequence number has caused the failure. When receiving such an MP_FAIL option, the host is aware that a middlebox has disrupted a subflow. If other subflows are operational, it can destroy the affected subflow with a TCP-reset and continue the transmission on the other subflows. However, if it is the last remaining subflow, it should fallback to the so-called *infinite mapping*. The infinite mapping is signaled within the DSS-option by setting the data length to 0. The subflow sequence number within the DSS-option specifies the start of the infinite mapping, while the data sequence number specifies where this byte must be mapped to. From this moment on, subsequent data will no more contain the DSS-option, as the mapping from subflow sequence to data sequence is implicit thanks to the infinite mapping. Multipath TCP effectively behaves now as regular TCP. It must be noted, that the hosts are no more allowed to use multiple subflows for the transmission of data.

2.4 Conclusion

This chapter provided a detailed overview of the inner workings of Multipath TCP. The control plane is achieved through an extensive use of TCP options. The data plane requires the creation of a secondary sequence number space, which allows hosts to recreate the byte stream. A mapping from subflow sequence to data sequence allows to traverse different kinds of middleboxes. It is exactly these middleboxes that have had a major impact on the design of the protocol. But, finally, this design allows Multipath TCP to be deployed on today's Internet.

Chapter 3

Evaluating Transport Protocols

A Multipath TCP implementation is a complex system, with many heuristics and algorithms influencing its performance [BPB11, RPB+12]. The congestion-control algorithm [WRGH11, KGP+12, CXF12] influences the sending rate of the individual subflows, the scheduler decides how to multiplex data among the subflows and flow control provides yet another limitation to the sending rate. Many external factors further influence the performance of Multipath TCP [RBP+11, BPB11, PDD+12, CLG+13]. Especially the network's characteristics in terms of capacity, propagation delay, etc. It is very difficult to have a clear understanding of how Multipath TCP behaves in different heterogeneous environments.

Experimental evaluation is often done in research papers to validate that a proposal works well in different environments. However, it is easy to fall into the trap of taking premature conclusions about the performance of the protocol. It is important for complex systems like Multipath TCP that the experiments are run within a wide range of environments. Also, conclusions cannot be extrapolated to other environments, due to the inherent complexity of the system.

The *"Experimental Design"* [F+49] approach defines steps to run experiments in order to answer scientific questions. This approach allows to draw solid conclusions about the performance of real-world systems. In this chapter we apply an experimental design approach to evaluate the Linux Kernel implementation of Multipath TCP in a wide range of heterogeneous environments.

3.1 Experimental Design

Experimental Design refers to the process of executing controlled experiments in order to collect information about a specific process or system [F+49]. The system under experimentation is influenced by controllable or uncontrollable factors

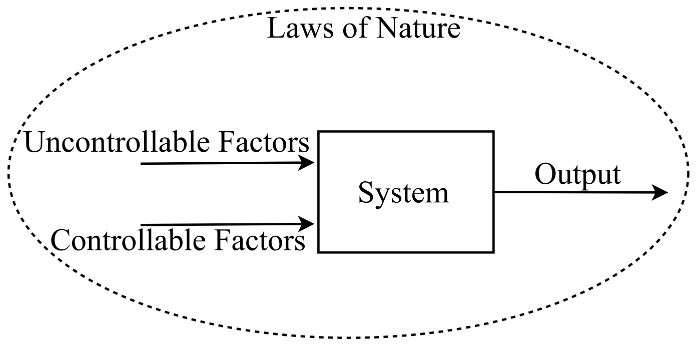


Figure 3.1: The system's output is influenced by a number of factors and obeys to the laws of nature [And12].

(see Figure 3.1). A controllable factor is one that can be influenced by the person running the experiments (e.g., temperature, pressure,...). Uncontrollable factors are those which cannot be influenced (e.g., time). It depends on the capabilities of the experimenter whether a factor is controllable or uncontrollable. The system responds to these factors according to the laws of nature. The experimenter can observe these responses by collecting one or more outputs of the system, which provide the information necessary to understand the system.

Running experiments in computer science (and networking research) is peculiar in the sense that the output is often considered to be deterministic (in absence of external factors, like temperature, time, etc.) [SWMW89]. Furthermore, depending on the system, experiments may be cheap in terms of time required, thus allowing a large number of experiments. In this section, we give an overview of the different steps of experimental design and its particularities with respect to networking experiments.

3.1.1 Objective

The first step is to define the objective pursued by the experiments. Kleijnen et al. defines in [KLC05] three different types of questions that may be answered through experimentation:

- *"Develop a Basic Understanding"* - This allows to have an overview of the system's behavior, uncover problems within the system or confirm expectations.

- *"Finding Robust Decisions or Policies"* - The goal is to find the correct configuration of the system to produce a desired output, while taking into consideration the influence of uncontrollable factors.

- *"Comparing Decisions or Policies"* - This process allows to estimate the behavior of the system with respect to a specific set of factors.

The objective defines the system that is under test and the outcome we want to measure. Once the objective is defined, the influencing factors and the way of measuring the output of the system must also be defined.

3.1.2 Factors

In experimental design, the system under test is often very complex. Many factors can influence the output of a system (see Figure 3.1). They may be of different kinds, controllable and uncontrollable. Among the controllable factors, those which influence the output of the system have to be selected. The uncontrollable factors may also influence the output of the system and are the reason for the variance of the system's output. To reduce the impact of the uncontrollable factors, an experiment should be repeated multiple times. This allows to extract the central tendency of the response by calculating the mean or median.

Optimal real-world experiments would require that the experiments are run simultaneously, to reduce the effect of the uncontrollable factors. However, in computer experiments this constraint can often be neglected as the output of the system is deterministic, allowing a sequential execution of the experiments [KLC05].

3.1.3 Design of the experiment

The design of the experiment influences the input parameters that are selected to conduct the experiments. In [BD87, MM09], the desirable properties that such parameter sets should have are discussed.

If the behavior of the system is meant to be modeled by a first-order polynomial model, fractional designs are a good fit [BD87]. These designs distribute the parameter set along the edges. Further, orthogonality is a desirable criterion of experiment designs, as it ensures that the sets are uncorrelated and thus allows to decide if a factor should be part of the model or not [KLC05].

However, sometimes it is not possible to assume a first-order polynomial model of the response. It may be that there is no prior knowledge of the system's response surface, or that the system has a rather stochastic nature. In this case, space-filling designs are good choices [MM95]. Space-filling designs don't only sample at the edges, but distribute the parameter sets equally among the whole factor space. A space-filling design can be generated with different algorithms. Santiago et al. propose in [SCBS12] the WSP algorithm which distributes the sets equally among the space. It is based on a uniform random sampling of the input parameters and eliminates excess points according to a minimum-distance criterion. The WSP algorithm has particularly good space-filling properties, even in high-dimensional spaces [SCBS12].

3.2 Evaluating transport-layer protocols

In this section we describe how the experimental design approach can be applied to the experimentation with a transport protocol like Multipath TCP. For this purpose, we need to define our objective, determine our design factors and design the experiment.

3.2.1 Objective

We are interested in a performance analysis of Multipath TCP to verify whether it fulfills its two main design goals [RHW11]:

- *Improve throughput: Multipath TCP should perform at least as well as regular TCP along the best path.*
- *Balance congestion: Multipath TCP should move traffic away from congested paths.*

Further, we evaluate the application delay of Multipath TCP. This is another, often overlooked, metric which is crucial for interactive or streaming applications. A low delay and low jitter is important to allow these applications to provide a good user experience.

We evaluate the performance of Multipath TCP for a wide range of parameters and pinpoint the scenarios where these goals are not met. We can also use this framework to validate the performance of Multipath TCP as modifications to certain algorithms within the protocol are being done.

We execute our approach by using Mininet [HHJ+12] which allows us to easily create a virtual network and run experiments between Mininet hosts using the v0.88 Linux Kernel implementation of Multipath TCP[1]. The benefit of using Mininet is that the results are reproducible and do not require a large number of physical machines. Compared to simulations, Mininet allows us to use the real Multipath TCP implementation and not a model of the protocol.

3.2.2 Factors

The performance of a transport protocol like Multipath TCP is influenced by various factors, such as bandwidth limitations, propagation delay, queuing de-

[1]Our scripts and the Mininet virtual images are available at http://multipath-tcp.org/conext2013

lay, loss, etc. [AF99]. Further, memory constraints on either host will limit the TCP window size, additionally influencing the performance [BPB11, SMM98]. Among these factors, one must distinguish between the quantitative ones (e.g., packet loss-ratio between two hosts) and the qualitative ones (e.g., congestion control algorithm being used). For each of these factors, the domain must be selected accordingly. We consider the following factors in our study:

Link Capacity Our evaluation of Multipath TCP targets environments of regular users, whose access speed may range from mobile networks to FTTH. We fix the range of our link capacities from 0.1 to 100 Mbps.

Propagation delay The propagation delay is the round-trip-time between the sender and the receiver over an uncongested path. Measurement studies have shown that the delay on the Internet may be up to 400 ms [ZNN+10] . We set the delay to a domain between 0 ms and 400 ms.

Queuing delay The buffers at the bottleneck router influence the queuing delay [GN11]. A perfect Active Queue Management (AQM) algorithm at the bottleneck router would not add any additional delay, whether a badly sized buffer may add a huge amount of queuing delay [GN11, All12]. We only consider tail-drop queues that are configured to add a queuing delay between 0 ms up to 2000 ms. We leave the evaluation of different queuing policies like RED or Codel for future work.

Loss Nguyen et al. show in [NR12] that the packet-loss probability over the Internet is very low (between 0 and 0.1%). In wireless networks, the loss probability may be considerably higher. We consider environments where the loss ranges from 0% to 2.5%.

Congestion Control We consider the three Multipath TCP congestion-control schemes: the Coupled congestion control [RHW11, WRGH11] which is the default one, Olia [KGP+12] and wVegas [CXF12]. Additionally, we use also Cubic congestion control [HRX08], to show the effect of using an uncoupled congestion control algorithm with Multipath TCP.

Application behavior Depending on the objective we want to evaluate we consider two types of applications. First, we use a bulk-transfer which tries to completely fill the pipe. Second, we use an application that transmits data at a specific rate to "simulate" a streaming application.

These factors and their ranges allow to cover the main environments that Multipath TCP might face when being used over the Internet. Additional factors and/or broader ranges are left for future work.

3.2.3 Design of the experiment

We cannot be sure of the nature of the response surface of Multipath TCP. Hence, we choose a space-filling design to cover a wide range of scenarios and correlations among the factors. It allows us to avoid making any assumptions on the behavior of Multipath TCP (cfr. Section 3.1.3). The drawback is that we need to run a large number of experiments in order to fully cover the factor space, but thanks to Mininet we are able to quickly perform these experiments. We use the WSP algorithm to generate the parameter sets in the space-filling design.

3.3 Multipath TCP evaluation

This section evaluates Multipath TCP with respect to 3 criteria. First, we look at the goodput achievable by Multipath TCP, meaning how well does it aggregates the capacity of the different interfaces. Second, we evaluate its load-balancing performance - one of the goals of the Multipath TCP congestion controls. Finally, we evaluate the burstiness of Multipath TCP and how it affects the delay as seen by the application.

3.3.1 Aggregation Benefit

We want to measure the benefit that Multipath TCP provides by pooling the resources of multiple paths. D. Kaspar provides in his thesis an expression for the aggregation benefit of a multipath protocol in terms of the capacities of the individual paths [Kas11]. This expression normalizes the aggregation benefit, allowing comparison of different environments, and takes into account the performance of Multipath TCP compared to regular TCP across the best path. If Multipath TCP performs as good as the path with the highest goodput, the aggregation benefit is equal to 0. If Multipath TCP perfectly aggregates the capacities of all paths, the aggregation benefit is equal to 1. An aggregation benefit of -1 means that Multipath TCP achieves zero goodput. Formally, the aggregation benefit is defined by the following:

Let S be a multipath aggregation scenario, with n paths. C_i is the capacity of the path i and C_{max} the highest capacity among all paths. If we measure a goodput of g with Multipath TCP, the aggregation benefit, $Ben(S)$, is given by [Kas11]:

$$Ben(S) = \begin{cases} \dfrac{g - C_{max}}{\sum_{i=1}^{n} C_i - C_{max}}, & \text{if } g \geq C_{max} \\ \dfrac{g - C_{max}}{C_{max}}, & \text{if } g < C_{max} \end{cases}$$

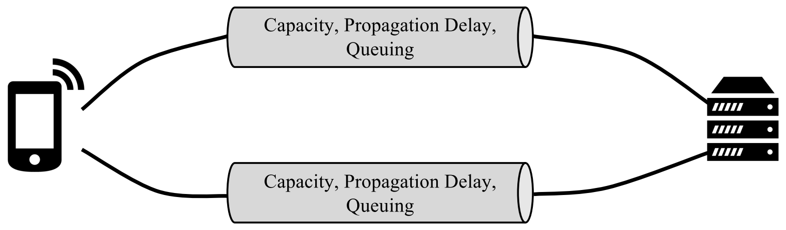

Capacity, Propagation Delay, Queuing
Capacity, Propagation Delay, Queuing

Figure 3.2: Our topology to evaluate aggregation benefit: Multipath TCP creates one subflow across each bottleneck. This allows to evaluate multipath-scenarios like a mobile phone connecting to WiFi/3G at the same time.

To provide an intuition of what this formula allows to express, let's have a look at Multipath TCP's performance in the following two scenarios where it is being used between two end hosts that have two paths between each other and they transmit data in a bulk transfer. We measure the goodput using Multipath TCP. In a first setup, both paths have a capacity of 5 Mbps and Multipath TCP has an aggregated goodput of 7 Mbps, the aggregation benefit is 0.4. If however the paths have a capacity of 7 Mbps and 3 Mbps, while Multipath TCP still achieves 7 Mbps, the aggregation benefit is 0. This shows that the above function for the aggregation benefit is able to identify the additional bandwidth Multipath TCP achieves compared to regular TCP over the path with the highest capacity.

Our setup evaluates Multipath TCP in a scenario (Figure 3.2) where the hosts establish two subflows between each other. We consider this as the common scenario (e.g., a client having two access networks like WiFi/3G). In order to measure the aggregation benefit, the Mininet-hosts create an iperf-session using the v0.88-release of Multipath TCP, which creates one subflow per bottleneck-link. The iperf-session runs for 60 seconds to allow the flows to reach equilibrium.

Initial evaluation

We study two types of environments: low Bandwidth-Delay-Products (BDP) and high-BDP. Low-BDP environments have relatively small propagation and queuing delays. In a high-BDP environment, the maximum values for the propagation and the queuing delays are very large. In a first run we only consider 3 factors per bottleneck, namely the capacity, propagation delay and queuing delay. For this first run we do not add the loss-factor as the Multipath TCP-specific congestion controls have a very specific behavior in lossy environments (as can be seen at the end of this section). The exact specifications of each environment can be found in Table 3.1.

	Low-BDP		High-BDP	
Factor	Min.	Max.	Min.	Max.
Capacity [Mbps]	0.1	100	0.1	100
Propagation delay [ms]	0	50	0	400
Queuing [ms]	0	100	0	2000

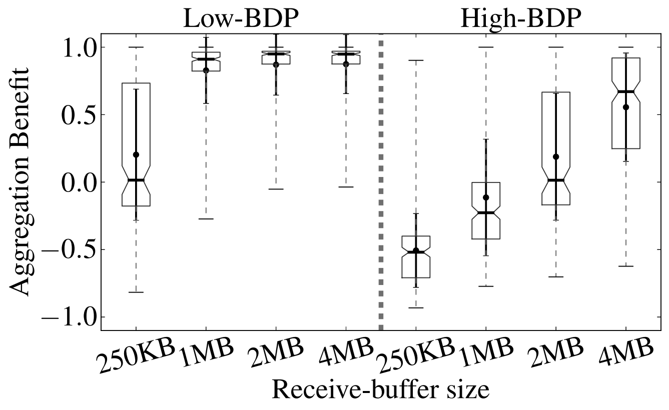
Table 3.1: Domains of the influencing factors for the measurement of aggregation benefit.

As we consider 2 paths, each being influenced by 3 factors, we have a 6-dimensional parameter space. We generate the parameter sets by using the WSP space-filling design, resulting in about 200 individual experiments. We have limited ourself to 200 experiments as our Mininet environment is able to run these within 4 hours. This allows us to quickly obtain the results of the experiments. In order to cope with possible variations, we repeat each parameter set 5 times and use the median to extract the central tendency of the aggregation benefit.

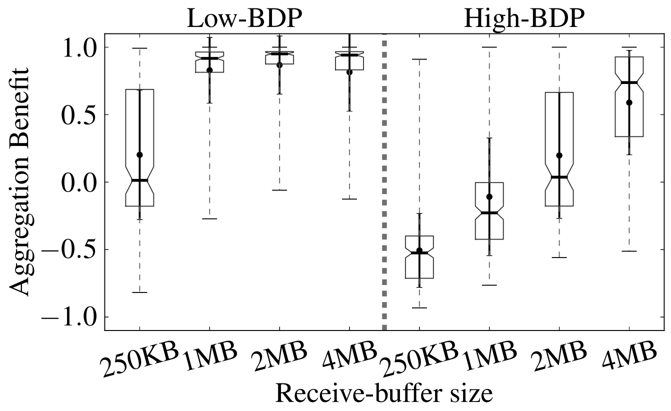
Effect of receive-buffer sizes The performance of Multipath TCP is influenced by the receive-buffer sizes of the end hosts [RPB+12]. We evaluate the impact of a fixed receive buffer on the aggregation benefit in the low-BDP and the high-BDP environments. In Figure 3.3(a) we show the aggregation benefit's mean (with its standard deviation) and the median, 25% and 75% percentiles as well as the degree of dispersion. We see that the larger the receive buffer, the larger the aggregation benefit. If the receive buffer is small, the Multipath TCP connection is receive-window limited and thus cannot use the full capacity of the links, which reduces the aggregation benefit. It can be seen that in this environments, the performance is equal, irregardless whether one uses Coupled (Figure 3.3(a)), Olia (no Figure shown because it is equivalent to Coupled and Cubic) or the uncoupled Cubic (Figure 3.3(b)) congestion control. However, using the Multipath TCP congestion control based on Vegas, called wVegas [CXF12], one can see that the aggregation benefit is very low - even below zero (Figure 3.3(c)). This is unexpected and shows that wVegas is not yet mature enough for a widespread deployment.

Small transfers

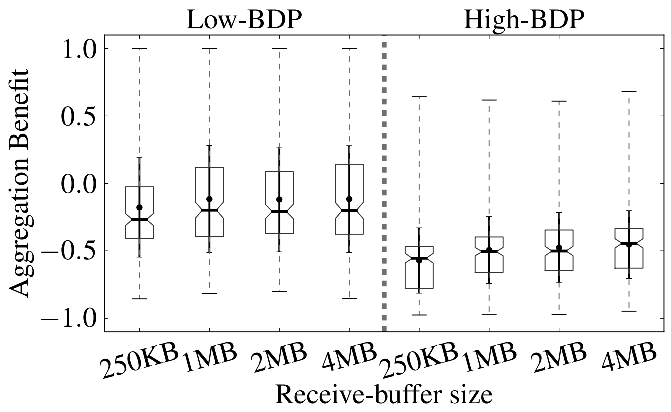
The previous measurements evaluated Multipath TCP's aggregation benefit in case of a bulk transfer by using the iperf traffic generator. Such a bulk transfer allows all subflows to probe for their path's capacity and thus enables the Multipath TCP scheduler to use the subflows in an efficient way. So, an important question is how the performance of Multipath TCP looks like when the subflows cannot fully probe the characteristics of the path. Such kind of behavior happens when



(a) Coupled congestion control



(b) Cubic



(c) wVegas

Figure 3.3: The aggregation benefit is close to perfect in the low-BDP environment. wVegas still needs some improvements before opting for widespread deployment.

the transfer is rather short. The Internet sees many of such short data transfers as many websites are built upon small HTML objects.

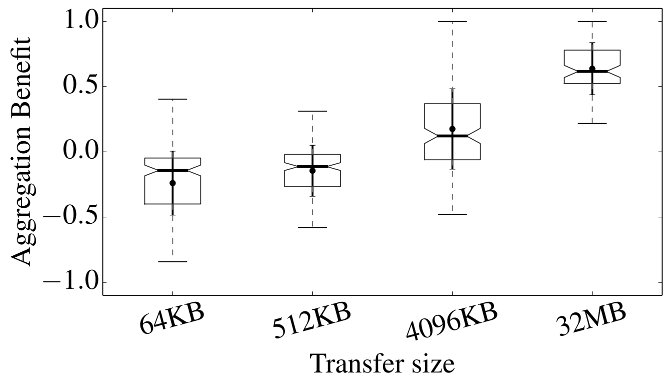
We thus measure the aggregation benefit of Multipath TCP with short data transfers by using netperf's request/response test. Using a request/response test allows to measure the time it has taken to do one such connection as the client can measure the time between it started the request and the time it received the response. We do this measurement for transfers of 64KB, 512KB, 4096KB and 32MB. In order to compute the aggregation benefit, we use Multipath TCP's flow-completion-time to compute the achieved goodput. This goodput is then compared with the goodput achieved by regular TCP with such a small transfer by using the aggregation benefit formula from above.

Figure 3.4 shows the aggregation benefit for these small transfers in the low-BDP environment with two different setups. In the first setup (shown in Figure 3.4(a)), the initial subflow of Multipath TCP is randomly chosen among the two available paths, while Figure 3.4(b) shows the results where the initial subflow is always the one where regular TCP achieved the best performance for the small transfer. If the initial subflow is chosen randomly, Multipath TCP's aggregation benefit is between 0 and -0.5 for very small transfers (64KB and 512KB). This, because most of the transfer is sent over the initial subflow, as it takes several RTTs until the second subflow is up and running. During this time, the performance of Multipath TCP is suboptimal if the path of the initial subflow is not as good as the best available path. However, as soon as the transfer becomes longer, Multipath TCP is able to pool the resources and increases the aggregation benefit above 0.

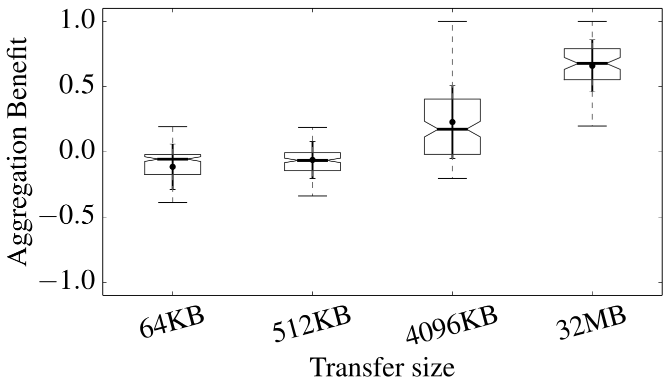
If Multipath TCP is instructed in such a way that it always uses the best available path for the initial subflow, its aggregation benefit becomes close to 0 - even for small transfers (as can be seen in Figure 3.4(b)). We observe that the initial path is of critical importance for the performance of Multipath TCP. However, it must be noted that even regular TCP has no information on which path it should start its small transfer. The path-selection problem is the same for regular TCP as well as for Multipath TCP.

Adding random packet-loss

In this part, we study the effect of packet losses on the performance of Multipath TCP. This could represent wireless environments where losses occur due to fading. We model lossy links by adding a loss factor to each bottleneck, effectively increasing the parameter space to a total of 8 factors. These loss-factors range from 0 to 2.5% and are set individually on each bottleneck. The two Multipath TCP congestion controls (Coupled and Olia [WRGH11, KGP+12]) both try to move traffic away from congested paths. As these loss-based congestion con-

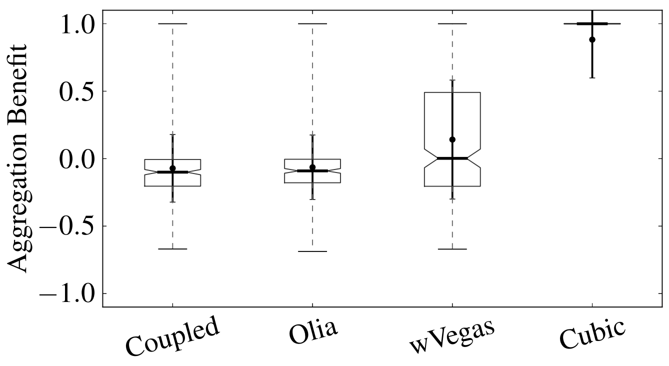


(a) Random initial path

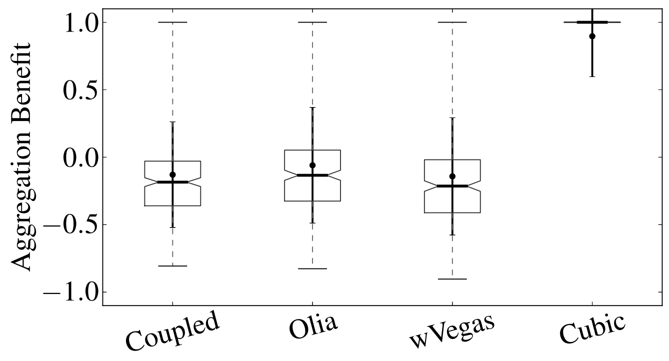


(b) Best initial path

Figure 3.4: For very small transfers, Multipath TCP does not bring a benefit compared to the best available path. Also, the choice of the initial path is important. Indeed, if Multipath TCP starts on the bad path then it suffers from the bad path's characteristics until the subflow on the better path has been established.



(a) Low-BDP environment



(b) High-BDP environment

Figure 3.5: The Multipath TCP congestion controls Coupled and Olia move traffic away from congested (aka lossy) paths. wVegas does not react upon loss as an indicator for congestion but still has issues filling the pipe, while Cubic is not moving traffic away from the lossy path and thus achieves perfect bandwidth aggregation.

trols interpret a loss as congestion, they move traffic away from lossy subflows. To compute the aggregation benefit, we consider the goodput of TCP by taking the loss into account. Thus, even if the capacity of the link is 20 Mbps, but the loss rate reduces regular TCP's goodput down to 5 Mbps, it is the 5 Mbps that are taken into account when computing the aggregation benefit.

Figure 3.5 shows that Coupled and Olia have mostly an aggregation benefit of 0. This confirms that Coupled and Olia only push traffic on the less lossy of the two subflows, thus moving almost all traffic to the best path. An uncoupled congestion control like Cubic does not take the loss-probability into account during its congestion avoidance phase. It rather aggressively increases the congestion window after a loss event. Thus, using Cubic in an environment where losses are not due to congestion but rather of a random nature brings benefits to Multipath TCP. However, as has been shown in Section 2.2.3 uncoupled congestion controls like Cubic, Reno,…have serious problems with respect to the fairness to regular TCP across shared bottlenecks.

The wVegas congestion control is purely driven by the evolution of the delay. It should not reduce its congestion window upon a random packet loss. Nevertheless, it does not seems to be able to aggregate the bandwidth of the two subflows. At this state it is yet unclear why wVegas does not aggregate the bandwidth better. It may be that RTT estimation during loss-phases is less accurate and thus might negatively influence the wVegas delay-based congestion control. Recent work in the Linux Kernel may have improved the RTT estimation upon loss events [2] but is not yet included in the 0.88 release of Multipath TCP.

3.3.2 Load balancing

In this section, we analyze whether Multipath TCP satisfies its congestion-balancing design goal. For this purpose, we study the performance of Multipath TCP in the scenario of Figure 3.6. The network contains three bottlenecks, and the end-hosts create a total of three Multipath TCP connections passing by this network. Each connection creates two subflows, one crossing a single bottleneck and the other passing through two bottlenecks. As discussed in [WRGH11], to balance the congestion and hence maximize the throughput for all Multipath TCP connections, no traffic should be transmitted over the two-hop subflows. The bottlenecks are influenced by the capacity, propagation delay and queuing delay, effectively emulating the low-BDP and high-BDP environments form Table 3.1. With three bottlenecks and three factors per bottleneck, we have effectively a 9-dimensional parameter space. We generate about 400 parameter sets with the WSP space-filling algorithm and start iperf-sessions for each Multipath TCP connection.

[2] `http://www.spinics.net/lists/netdev/msg243665.html`

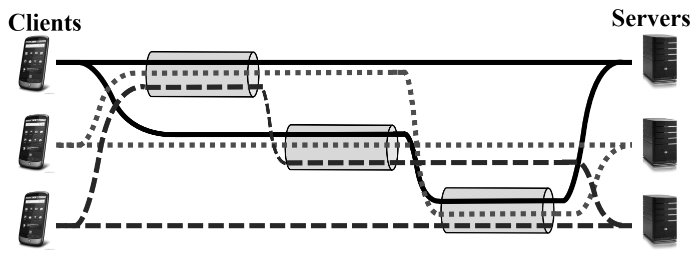


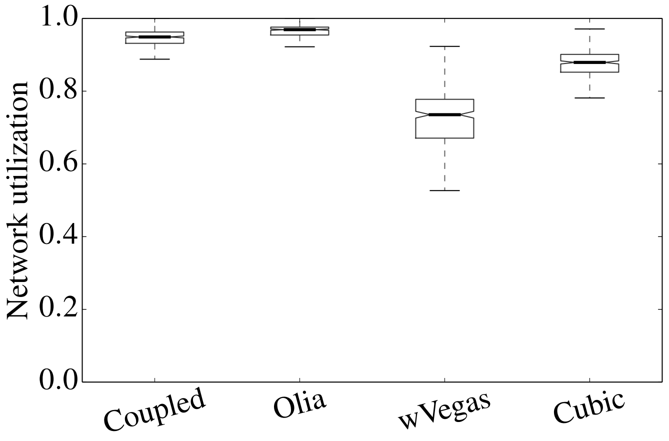
Figure 3.6: Three bottlenecks are used to evaluate Multipath TCP's load-balancing performance. Each Multipath TCP connection has a one-hop and a two-hop subflow.

We evaluate this scenario and show the relation between the aggregated goodput of all Multipath TCP connections, compared to the theoretical upper bound in Figure 3.7. We compare the performance of Coupled congestion control, Olia and wVegas with uncoupled Cubic congestion control[3]. We observe that Olia is able to move traffic away from the two-hop subflows in low-BDP environments and hence efficiently uses the capacity available in the network. There is a small difference between Coupled and Olia in terms of load-balancing, which confirms the findings of [KGP+12]. It also becomes apparent that using an uncoupled congestion control, like Cubic, does not allow Multipath TCP to efficiently pool the resources. In high-BDP environments, even Olia fails to provide a good congestion balancing. This is because the large BDP makes one of the three Multipath TCP connections become receive-window limited when the three bottlenecks have different characteristics. This flow cannot benefit from Olia's load-balancing algorithms and leads to suboptimal network utilization. wVegas again suffers from its weak bandwidth aggregation capabilities.

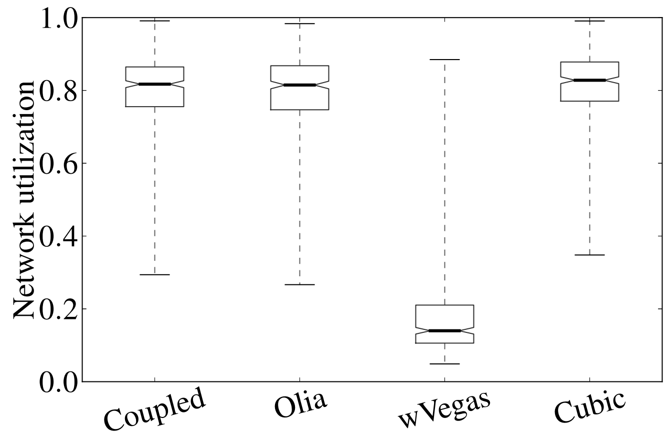
3.3.3 Application delay

Bandwidth is not the only important metric to measure user experience. Application delay defines how long it takes for a single byte to reach the peer and, more importantly, what is the variance of this delay. Streaming applications often do not fill the pipe of the network at 100%, but rather send at a steady rate. Still, they

[3]Uncoupled Cubic [HRX08] represents the case where regular TCP congestion control is used on the subflows. It increases the congestion windows of each subflow irregardless of the congestion state of the other subflows that are part of the Multipath TCP connection.



(a) Low-BDP environment



(b) High-BDP environment

Figure 3.7: In the low-BDP environments, Olia is able to efficiently move the traffic away from the congested paths. However, the difference to Cubic is only minor. In high-BDP environments, the delay is too long to efficiently act upon congestion within a reasonable time-frame.

depend on a continuous stream of data reaching the host. A low application delay and little jitter is crucial for the well functioning of such streaming applications.

We developed a tool that allows to measure the application delay of a byte stream. The application sends blocks of 8 KB of data. This transmission can be at unlimited speed, or at a predefined goodput. The application can thus "emulate" a streaming application transmitting the stream at a constant bitrate. The blocks of data include a timestamp, taken with the nano-second resolution timer. At the reception of this block, the receiver records its own timestamp ($ownts$) as well as the one in the 8 KB block ($rcvts$). and constructs a list of $< rcvts, ownts >$ pairs. The difference between $ownts$ and $rcvts$ allows to compute the one-way delay if the clocks are perfectly synchronized on the two hosts[4]. This is the case in our Mininet environment, as the nodes run all on the same operating system, separated only by Linux networking containers.

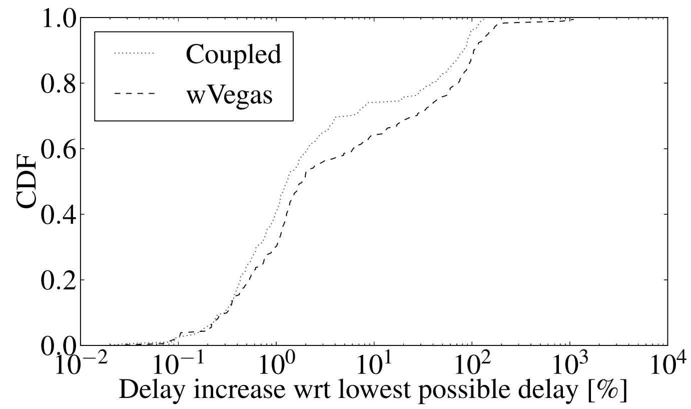
To express the delay variation, we calculate the *delay increase compared to the lowest possible delay*. This value is expressed in % and estimates how much the delay is higher for a particular block compared to the lowest one-way delay measured. E.g., if the minimum one-way delay is 20 ms, but a block of 8 KB has been received rather after 40 ms, the relative delay increase has a value of 100%.

Within Mininet we ran this application in the low- and high-BDP environment, for a total of 200 experiments. We provide the CDF of the 99th percentile of the relative delay increase. We show this 99th percentile because the tail of the latency variation is important for a well-functioning streaming application.

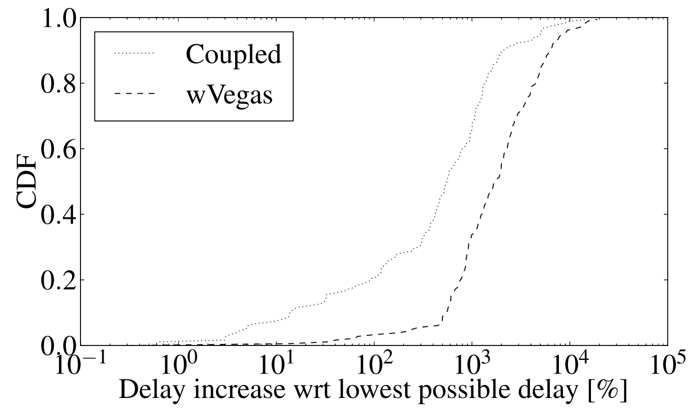
Figure 3.8 shows the results of our analysis. Our application sends at a fixed configured rate. As in our considered space-filling design a large range of capacities is considered, we have to express this rate as a percentage of the aggregate capacity among the two paths. For the low-BDP environment we show the results for a sending rate of 20% of the aggregate capacity, while in the high-BDP environment we consider a sending rate of 80%. The results are similar with other sending rates, with the high-BDP environment introducing a more significant relative delay increase. It can be seen that at a low bitrate, there is no difference between wVegas and Coupled congestion control (Olia behaves similar to Coupled). For 70% of the environments in Figure 3.8(a) the capacity available on the subflow with the lowest RTT is sufficient to transmit at 20% of the aggregated capacity. For these the environments the delay variation remains very low (less than 1%). However, for the remaining 30% of the environments (among the 200 generated through space-filling design), the delay variation increases up to 100% because traffic must be sent on the secondary subflow.

When our measurement software transmits data at a higher bitrate, closer to the

[4]If the clocks are not synchronized, the clock difference can be eliminated by taking $min(ownts - rcvts)$ as the base-delay.



(a) 20% of the capacity in low-BDP environment



(b) 80% of the capacity in high-BDP environment

Figure 3.8: Sending at a low bitrate shows almost no difference between the congestion controls. At higher bitrates, wVegas is unable to use the capacity and starts queuing data in stack's send buffer.

aggregate capacity, a difference between wVegas and Coupled becomes apparent (Figure 3.8(b)). A first observation is that generally, when transmitting at 80% of the capacity, the relative delay increase is higher than when sending at a low bitrate. This is because the secondary subflow must be used to reach 80% of the capacity. As our Multipath TCP implementation schedules traffic first on the subflow with the lowest RTT, as soon as this subflow's capacity is fully used, Multipath TCP needs to schedule on the subflow with a higher RTT, increasing the perceived application delay. It must be noted that it is difficult to compare this scenario to regular TCP. TCP would not even be able to transmit at the specified rate (it can only use the capacity of a single subflow). Finally, it can be seen that wVegas introduces a higher delay increase. This is because wVegas is not able to utilize the capacity offered by the two paths and thus is buffering data in the stack's send-queue. Thus, the delay is artificially increased up to 10000%.

3.4 Sensitivity Analysis

The number of experiments executed within the parameter space influences the accuracy of the results. The more sets explored, the better the accuracy will be. However, CPU-time constraints limit the number of experiments that can be executed. The sensitivity analysis allows to confirm that the number of experiments conducted per parameter space is sufficient to have an accurate view of Multipath TCP's performance. This can be achieved by generating different space-filling designs, and comparing the 5th, 25th, 75th and 95th percentiles and median among each of these designs.

To evaluate the aggregation-benefit we used three influencing factors per bottleneck link, effectively creating a 6-dimensional parameter space. 200 sets were generated, using the WSP space-filling algorithm. We generate 5 different space-filling designs of comparable size for the sensitivity analysis. Each design explores different sets among the parameter space. Comparing the percentiles and the median of the aggregation benefit for each of the designs has shown that the standard deviation is very low. Relative to the range of the aggregation benefit $(-1, 1)$, it ranges between 0.1% and 2.62%. We can conclude that running 200 experiments is sufficient to have a good overview of the aggregation benefit of Multipath TCP in our 6-dimensional parameter space. In the load-balancing environment we observe a similarly low standard deviation ranging from 0.008% to 1.4%.

3.5 Conclusion

Multipath transport protocols like Multipath TCP, are complex systems, whose performance is influenced by numerous factors. To explore the impact of these factors we use a space-filling design. Our first attempt at applying *”Experimental Design”* techniques to the evaluation of Multipath TCP allowed us to discover previously unknown performance issues within the Linux Kernel implementation of Multipath TCP. These issues are explained in the following chapter.

We have also shown that wVegas is far from being applicable to a widespread deployment. It is not able to fully utilize the capacity of the network, which results in a very bad aggregation benefit performance.

One benefit of Multipath TCP is its load-balancing capabilities by coupling the congestion controls. We validated that the load balancing works well in low-BDP environments. However, in high-BDP environments the load balancing does not work well as some subflows are limited by the receive buffer. This effectively prevents Olia to balance the load among the subflows.

Chapter 4

Multipath TCP in heterogeneous environments

Multipath TCP pools the resources of multiple paths, trying to aggregate the bandwidth and using them in the most efficient way. These paths may be subject to very different characteristics. For example, when using a smartphone and pooling the resources of the WiFi and the 3G interfaces, the subflows experience different round-trip-times and loss rates. As regular TCP sends traffic only over one of these paths, it optimizes its transmission behavior according to the quality. However, Multipath TCP must schedule its data across these different paths, while still achieving high performance.

In this chapter, we take a close look at the behavior of Multipath TCP in heterogeneous environments. We analyze how Multipath TCP reacts to large delay differences across the subflows and suggest heuristics to handle these cases in the most efficient way.

4.1 Problem statement

In a (hypothetical) ideal resource-pooling scenario, all the paths would experience the same congestion and delay. In such kind of environment, the paths would effectively appear as a pool of resources that behave in exactly the same way as if a single link would be present. All packets would experience the same round-trip-time and losses would occur at the same time on all subflows. Packets multiplexed in-order across the subflows would reach the peer in the exact same order.

However, in reality the paths almost never have exactly the same characteristics. The difference of these characteristics results in out-of-order delivery at the receiver, causing receive-window limitations and head-of-line blocking which

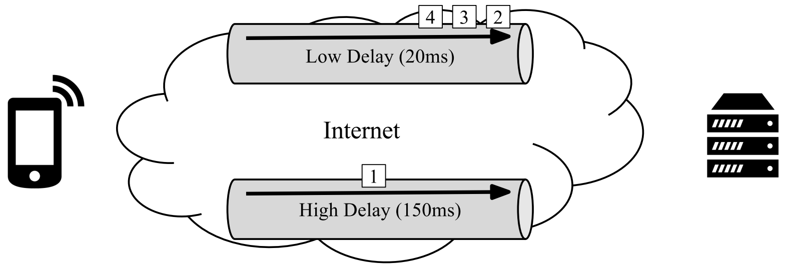


Figure 4.1: Segment number 1, sent over the high-delay path will reach the host after the ones sent over the low-delay path, resulting in out-of-order reception at the receiver.

will be discussed in detail in this section.

4.1.1 The importance of the round-trip-times

It is important to first understand what will cause out-of-order delivery at the receiver. The Multipath TCP scheduler multiplexes the data from the application to the different subflows. If it does this in an ordered way, a delay difference among the subflows will result in out-of-order delivery at the receiver, as shown in Figure 4.1. The scheduler did sent packet with sequence 1 over the high-delay path, continuing by scheduling packets 2 to 4 over the low-delay path. These latter ones will reach the server before segment 1, thus filling the out-of-order queue at the receiver as Multipath TCP ensures in-order delivery of the data to the application.

This kind of out-of-order delivery is caused principally by a delay difference among the subflows. The delay difference may be due to different factors. Some access mediums present a different base delay. For example, 3G (UMTS) has a theoretical minimum RTT of 40 down to 20 ms[1], while WiFi networks may have a much lower delay. Further, deep queues in the bottleneck router may be the reason for bufferbloat [GN11], increasing the propagation delay of data sent across the subflow. A packet loss may also temporarily affect the propagation delay of the segments over a subflow. Indeed, loss-recovery with fast retransmissions needs one round-trip-time and thus artificially increases the perceived propagation delay during the recovery period. Finally, the delay may be subject to significant jitter, for example in WiFi networks where a bad link-quality may require multiple frame retransmissions to allow a successful delivery of the data from the access point to the end host.

[1]HSPA$^+$,Rel-7: http://www.3gpp.org/specifications

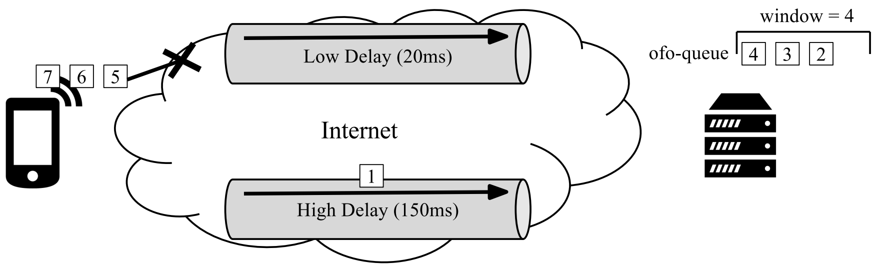


Figure 4.2: The capacity of the low-delay path is not being used because the receive window does not allow the sender to schedule segment 5. This reduces Multipath TCP's throughput significantly.

4.1.2 Receive/send-window limitations

With regular TCP, in absence of in-network reordering and packet loss, the data reaches the receiver in-order. Thus, it can immediately be delivered to the application, without using any space in the receive-buffer. However, in case of a packet-loss, out-of-order data will be delivered to the receiver. Then, the TCP stack needs to store the data until the missing packet has been received and the data can be pushed to the application. Thus, a TCP stack reserves a certain amount of memory for this out-of-order data. Typically, this is scaled to twice the bandwidth-delay-product (BDP), as it allows the sender to transmit at full speed - even in case of a loss-event. As explained above, Multipath TCP experiences reordering across the TCP subflows due to the delay differences, hence the receive buffer has to accommodate out-of-order data also at the Multipath TCP level. The size of the receive buffer is thus critical to allow high goodput.

In order to fully utilize the capacity of all paths, a receiver must provide enough buffer space so that the sender can keep all subflows fully utilized - even in the event of reordering. The recommendation for Multipath TCP's receive buffer size is defined in [BPB11]:

$$\text{Buffer} = \sum_{i}^{n} \text{bw}_\text{i} \times \text{RTT}_\text{max} \times 2$$

such a buffer allows each subflow to send at full speed ($\sum_i^n \text{bw}_\text{i}$) during the time-interval of the highest round-trip-time among all subflows (RTT_max), even if a loss event occurs (multiply by 2).

We first observe that, fundamentally, memory requirements for Multipath TCP are much higher than those for TCP, mostly because of the RTT_max term. A 3G path with a bandwidth of 2 Mbps and 150 ms RTT needs just 75 KB of receive-buffer, while a WiFi path running at 8 Mbps with 20 ms RTT needs around 40

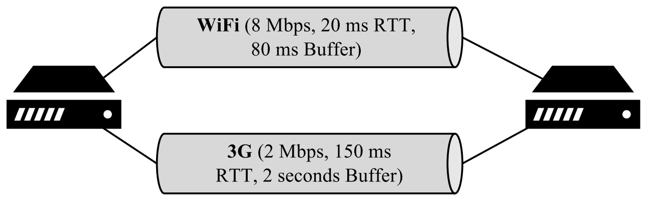

WiFi (8 Mbps, 20 ms RTT, 80 ms Buffer)
3G (2 Mbps, 150 ms RTT, 2 seconds Buffer)

Figure 4.3: We emulate a WiFi/3G aggregation scenario in our lab to test the effect of receive-window limitations.

KB. Multipath TCP running on these two paths will need 375 KB — nearly four times the sum of the path BDPs.

However, if an end host is not able to provide the necessary amount of memory for the receive buffer, the so-called *receive-window limitations* will occur. Figure 4.2 shows this kind of behavior. The receiver announced a receive window of 4. The sender scheduled segment 1 on the high-delay path, while sending segments 2 to 4 over the low-delay path. As these ones will be delivered first, the receiver has to store these in its out-of-order queue (*ofo-queue*). The sender is unable to continue its transmission over the low-delay path because the receive window does not provide the space to send segment 5. As soon as segment 1 reaches the server, data is pushed to the application and the out-of-order queue is emptied. This provides again space in the receive window and the sender can continue its transmission. Nevertheless, prior to the arrival of segment 1 the sender was unable to use the capacity of the low-delay subflow.

We used our Linux implementation to test this first behavior in our lab-setup shown in Figure 4.3. Figure 4.4 shows the goodput achieved as a function of receive window for TCP and Multipath TCP running over an emulated 8 Mbps WiFi-like path (base RTT 20 ms, 80 ms buffer) and an emulated 2 Mbps 3G path (base RTT 150 ms, 2 seconds buffer).

Multipath TCP will send a new packet on the lowest delay link that has space in its congestion window. When there is very little receive buffer, Multipath TCP sends all packets over WiFi, matching regular TCP. With a larger buffer, additional packets are put on 3G and overall goodput drops. Somewhat surprisingly, even 370 KB are insufficient to fill both pipes. This is because unnecessarily many packets are sent over 3G. This pushes the effective RTT_{max} towards 2 seconds, and so the receive-buffer requirements are exceeding the available 370 KB.

We see that a megabyte of receive-buffer (and send-buffer) is needed for a single connection over 3G and WiFi. This is a problem, and may prevent Multipath TCP from being used on busy servers and memory-scarce mobile phones. TCP

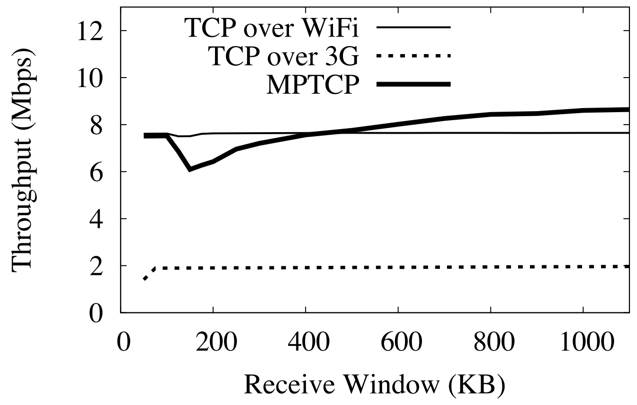


Figure 4.4: A memory-constrained end host is not able to fully utilize the capacity of the WiFi network because a large delay difference introduces receive-window limitations.

over WiFi even outperforms Multipath TCP over both WiFi and 3G when the receive buffer is smaller than 400 KB, removing any incentive to deploy Multipath TCP.

4.1.3 Head-of-Line blocking

As Multipath TCP ensures in-order delivery, the packets that are scheduled on the low-delay subflow have to "wait" for the high-delay subflow's packets to arrive in the out-of-order queue of the receiver.

Head-of-line blocking causes *burstiness* in the data stream by delaying the data delivery to the application, which is undesirable especially for interactive or streaming traffic. Interactive applications will become less reactive, resulting in a poor user-experience. Streaming applications will need to add a high amount of application-level buffering to cope with *burstiness* and provide a continuous streaming experience to the end user.

As an example, let's consider Figure 4.2. A scheduler might (for one reason or another) have decided to send packet with sequence number 1 on the bottom, high-delay subflow. Packets 2 to 4 might have been scheduled on the low-delay subflow. They will reach the destination faster than the packet with sequence number 1. As Multipath TCP ensures in-order delivery of the data, the packets 2 to 4 must "wait" in the out-of-order queue of the receiver. This phenomenon is known as *head-of-line blocking* [SK06].

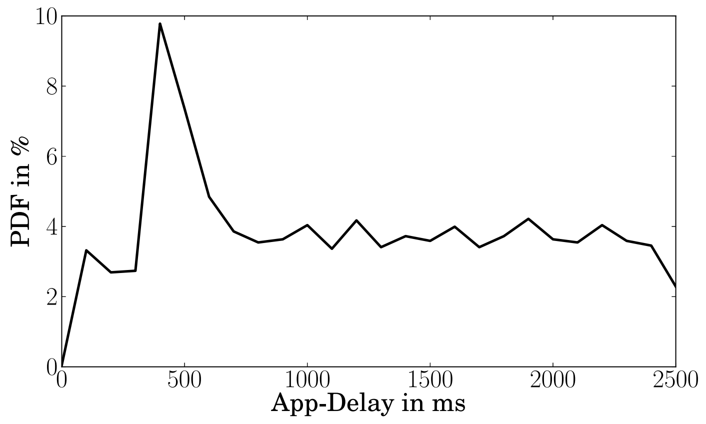


Figure 4.5: The delay perceived by the application is spread along a wide range up to 2.5 seconds. This is an indicator for head-of-line blocking issues within Multipath TCP.

Figure 4.5 shows the fraction of 8KB blocks (normalized to a probability density function) that have been received within a certain application-level delay, by using Multipath TCP in a heterogeneous environment. In this environment, the hosts were connected through two different paths. One path is similar to a 3G environment, with a high round-trip-time, up to 2 second of buffering at the bottleneck router and a relatively low capacity of 1.7 Mbps. The other one is more like a residential broadband path with a very low round-trip-time (9 ms) but still an over-buffered bottleneck router of around 500 ms buffering with a capacity of 9 Mbps. We use the application-delay application from Chapter 3, transmitting at 4 Mbps blocks of 8000 bytes. We measure the variation of the application-delay compared to the lowest observed delay. It can be seen that the delay variation is spread largely up to 2.5 seconds of variation. The round-robin scheduler has introduced head-of-line blocking issues as packets need to wait in the out-of-order queue before being pushed to the application. A packet-loss over the 3G-like path is very expensive in terms of delay, because the retransmission takes up to 2 seconds (one maximum round-trip-time on the bufferbloated 3G path).

4.2 A reactive approach

The previous section has shown that Multipath TCP has some performance issues when being used in heterogeneous environments. Effectively, receive-window limitations trigger a suboptimal usage of the network's resources and head-of-line blocking results in a high application-level delay, affecting the user's

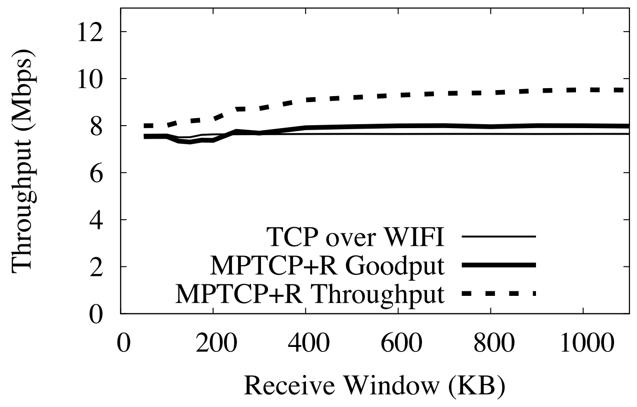


Figure 4.6: Opportunistic retransmission helps to overcome receive-window limitations. However, it is also "wasting" bandwidth as much duplicate data is being sent as can be seen from the throughput graph.

quality of experience.

In the following we show how a reactive approach can reduce the effect of receive-window limitations and is able to achieve high throughput with Multipath TCP.

4.2.1 Minimizing memory usage

Mechanism 1: Opportunistic retransmission (R)

When a subflow has a large enough congestion window to send more packets, but there is no more space in the receive window, what should it do? One option is to resend the data, previously sent on another subflow, that is holding up the trailing edge of the receive window. In our example, the WiFi subflow may retransmit some unacknowledged data sent on the slow 3G subflow.

The motivation is that this allows the fast path to send data as fast as it would with single-path TCP, even when underbuffered. If the connection is not receive-window limited, opportunistic retransmission never gets triggered.

Our Linux implementation only considers the first unacknowledged segment to avoid the performance penalty of iterating the potentially long send-queue in software interrupt context. This works quite well by itself, as shown in Figure 4.6: Multipath TCP goodput is almost always as good as TCP over WiFi, and mostly it is better.

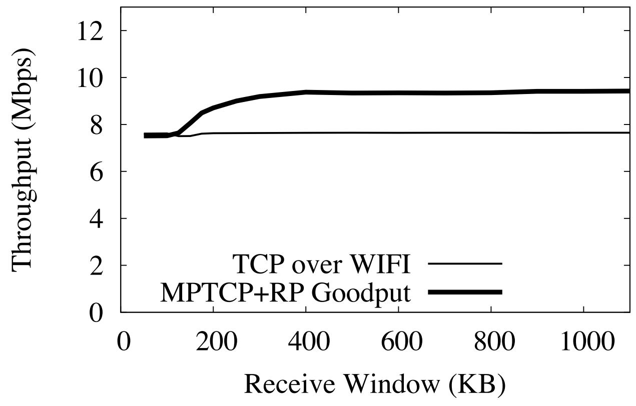


Figure 4.7: Combining opportunistic retransmission (R) with penalization (P) allows for the best possible use of the resources.

Unfortunately opportunistic retransmission is rather wasteful of capacity when underbuffered, as it unnecessarily pushes 2 Mbps traffic over 3G; this accounts for the difference between goodput and throughput in Figure 4.6.

Mechanism 2: Penalizing slow subflows (P)

Reacting to receive window stalls by retransmitting is costly; we'd prefer a way to avoid persistently doing so. If a connection has just filled the receive window, to avoid doing so again next RTT we need to reduce the throughput on the subflow that is holding up the advancement of the window. To do this, Multipath TCP can reduce that subflow's congestion window; in our tests we halved the congestion window and set the slow-start threshold to the reduced window size. To avoid repeatedly penalizing the same flow, only one reduction is applied per subflow round-trip time and only if the subflow's throughput is lower than the other one.

Penalizing and opportunistic retransmission work well together, as seen in Figure 4.7: Multipath TCP always outperforms or a least matches TCP over WiFi.

Further evaluation

To illustrate more clearly the impact of the retransmission and penalization (RP), it is worth examining a few more varied scenarios.

The first scenario we analyze is where one of the paths has extremely poor performance such as when mobile devices have very weak signal. Figure 4.8 shows

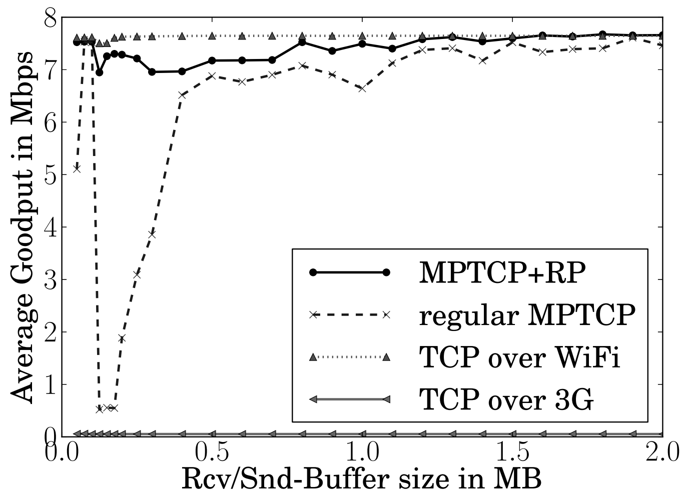


Figure 4.8: WiFi and a lossy 3G path - Retransmissions over 3G are very costly, but our mechanisms allow to overcome this limitation.

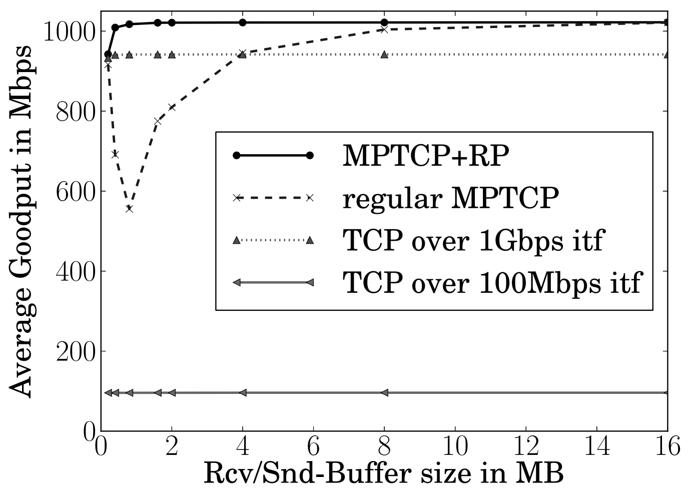


Figure 4.9: 1 Gbps and 100 Mbps links

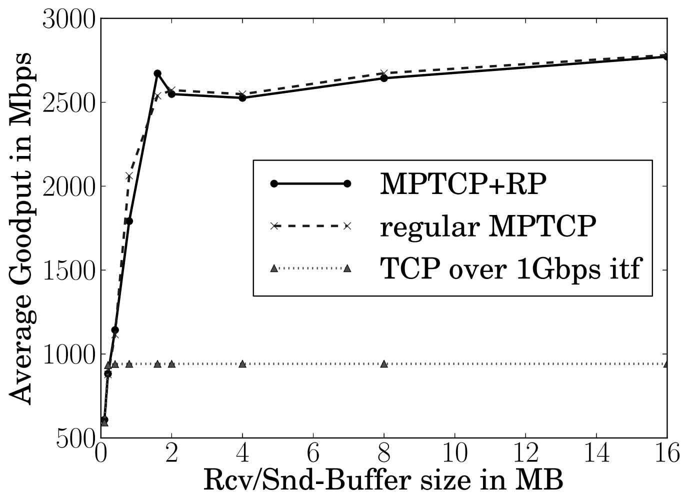


Figure 4.10: Three 1 Gbps links

the goodput achieved using our Linux implementation on an emulated WiFi path (8 Mbps, 20 ms RTT, 80 ms buffer) and an emulated very slow 3G link (50 Kbps, 150 ms RTT, 2 seconds buffer). As the link is so slow, the loss rate will be high on the 3G path, and the large network buffer means that each retransmission over 3G takes a long time. With receive buffer sizes smaller than 400 KB, whenever a loss happens on 3G, regular MPTCP ends up flow-controlled, unable to send on the fast WiFi path. MPTCP plus retransmission and penalization prevents this from becoming a persistent problem. Opportunistic retransmission allows the lost 3G packet to be re-sent on WiFi without waiting for a timeout and penalization reduces the data buffered on the 3G link, avoiding the situation repeating too frequently. With receive buffer sizes around 200 KB, these mechanisms increase Multipath TCP throughput tenfold. It is slightly below regular TCP over WiFi, because it still takes one round-trip-time for the opportunistic retransmission to reach the destination quicker over the WiFi link than over the 3G one. During this time the WiFi subflow is underutilized.

Next, we use two hosts connected by one gigabit and one 100 Mbps link to emulate inter-datacenter transfers with asymmetric links. Figure 4.9 shows that MPTCP+RP is able to utilize both links using only 250 KB of memory, while regular MPTCP under-performs TCP over the 1 Gbps interface until the receive buffer is at least 2 MB.

When the hosts are connected via symmetric links—we used three such links in Figure 4.10—both regular MPTCP and MPTCP+RP perform equally well, regardless of the receive buffer size. This is because in this scenario, when under-

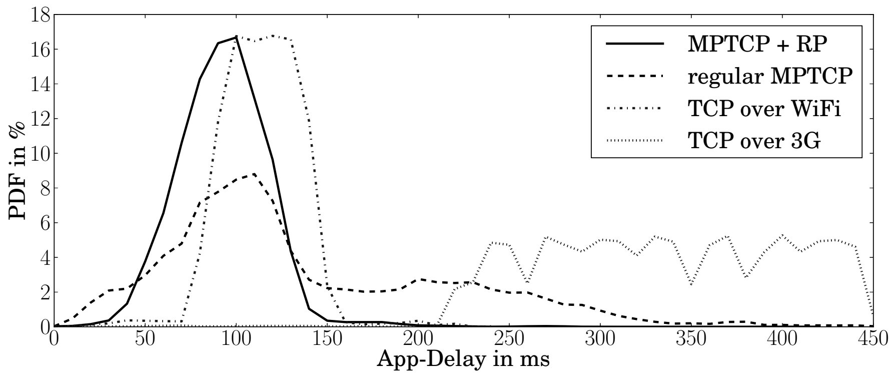


Figure 4.11: Application level latency for 3G/WiFi case

buffered, using the fastest path is the optimal strategy.

Application level latency

Goodput is not the only metric that is important for applications. For interactive applications, latency between the sending application and the receiving application can matter.

As Multipath TCP uses several subflows with different RTTs, we expect it to increase the end-to-end latency seen by the application compared to TCP on the fastest path. To test this, we use the application from Chapter 3 to measure the variation of the end-to-end delay as seen by the application.

Figure 4.11 shows the probability density function of the application-delay with a buffer-size of 200 KB running over 3G and WiFi. Retransmission and penalization do a good job of avoiding the larger latencies seen with regular Multipath TCP. Somewhat counter intuitively, the latency of TCP over WiFi is actually greater than MPTCP+RP. The reason for this is that 200 KB is more buffering than TCP needs over this path, so the data spends much of the time waiting in the send buffer. Multipath TCP's send buffer is effectively smaller because the large 3G RTT means it takes longer before DATA ACKs are returned to free space. If we manually reduce TCP's send buffer on the WiFi link, the latency can be reduced below that of Multipath TCP.

4.2.2 Problems with Penalization

The above described mechanism to reduce the memory usage includes the penalization algorithm. Penalization reduces the congestion window without actually observing a congestion event. The following analyzes this behavior in detail

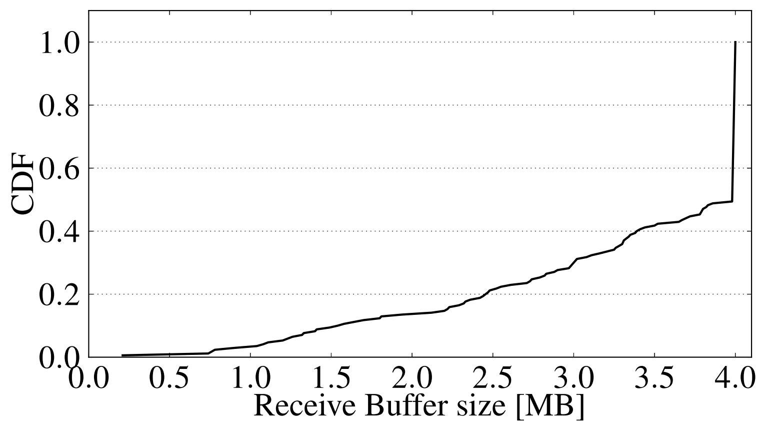


Figure 4.12: With auto-tuning, 50% of the experiments are able to consume less than 4 MB of receive buffer in the low-BDP environment.

and actually detects a particular problem with this algorithm.

To perform this analysis we use the experimental design approach, described in Chapter 3. The same low- and high-BDP environments are being used as in Chapter 3 with the influencing factors being the link's capacity, propagation delay and queue-size of the bottleneck router.

Effect of enabling auto-tuning

The Linux TCP stack does not use a fixed receive buffer. Instead, it includes an auto-tuning algorithm [SMM98] that adapts dynamically the size of the receive buffer to achieve high goodput at the lowest possible memory cost. As explained in Section 4.1.2, the current Multipath TCP implementation sets this buffer to $2 * \sum_i^n bw_i * RTT_{max}$, to allow aggregating the throughput of all the subflows [BPB11].

Enabling auto-tuning should reduce the memory requirements of each Multipath TCP connection. Figure 4.12 reports the maximum receive buffer used in the low-BDP environment for each parameter set. Indeed, we observe that on 50% of the experiments the receive buffer remains below 4 MB, effectively reducing the memory used by the connection.

However, enabling auto-tuning can also lead to a huge performance degradation with Multipath TCP. Figure 4.13 depicts the aggregate benefit of Multipath TCP v0.86 when auto-tuning is enabled, capping the buffer to Linux's default of 4 MB. We observe that in high-BDP environments the aggregation benefit is very low. Effectively, 80% of the experiments have an aggregation benefit below 0. We

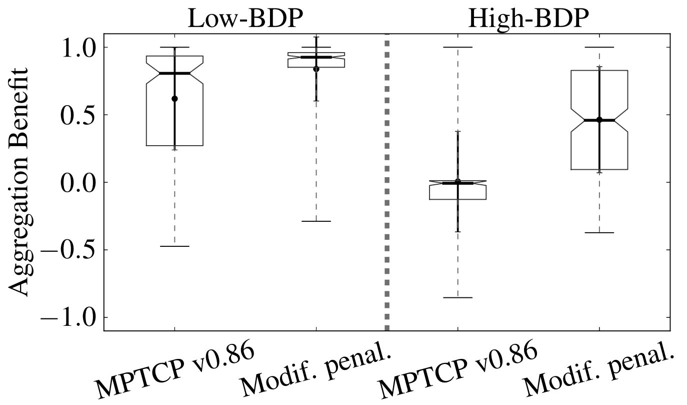


Figure 4.13: MPTCP v0.86 has a weak performance with auto-tuning. Our modification to the penalization algorithm significantly improves the aggregation benefit.

observe this performance degradation in low-BDP environments too: 25% of the experiments have an aggregation benefit below 0.2.

The auto-tuning at the receiver will make the receive buffer start at a small value at the beginning of the connection, increasing it as the sender's subflows evolve during their slow-start phase. Multipath TCP evaluates the receive-buffer size every RTT_{max} in order to estimate the sending rate of the peer. $\sum_i^n bw_i * RTT_{max}$ represents the amount of data the peer sends during one RTT_{max}-interval. Multiplying this by 2 to achieve the recommended receive buffer of [BPB11] should allow the sender to increase its sending rate during the slow-start phase. However, subflows whose RTT is smaller than RTT_{max} will more than double their sending rate during an RTT_{max}-interval, effectively making the sender limited by the receive window. As the subflows evolve through their slow-start phase, the announced window will continue increasing and eventually be large enough. Hence, these receive-window limitations are only transient and should not prevent users from achieving a high transmission rate.

Unfortunately, Multipath TCP's reaction to transient receive-window limitations is overly aggressive because of the *"penalization"* mechanism explained in Section 4.2.1. This mechanism handles receive-window limitations due to round-trip-time differences among the subflows (e.g., in WiFi/3G environments). When the flow is limited by the receive window, it halves the congestion window of the subflow which causes this receive-window limitation and sets its slow-start threshold to the current congestion window. If a Multipath TCP-connection experiences the transient receive-window limitations while one of its subflows is in slow-start, the penalization algorithm will give this subflow a false view of the path capacity

```
sub: A subflow belonging to MPTCP, ready to send data.
now: The current time.
blocksub: The subflow who sent the segment that is causing the head-of-line
    blocking.
    // Only penalize every RTT
1:  if now - sub.last_penal < sub.rtt then
2:      return ;
    // Penalize if our RTT is smaller than the one of the blocking subflow
3:  if sub.rtt < blocksub.rtt then
4:      blocksub.cwnd /= 2;
        // Do not kill slow-start
5:      if blocksub.cc_state != SLOW-START then
6:          blocksub.ssthresh /= 2;
7:      sub.last_penal = now;
```

Figure 4.14: The improved penalization algorithm in pseudo-code format. It does not interrupt the slow-start phase of the subflow being penalized.

by adjusting its slow-start threshold to a smaller value.

We propose to solve this problem by modifying the penalization algorithm, outlined in pseudo-code format in Figure 4.14. It should not adjust the slow-start threshold when a subflow is in its slow-start phase. Figure 4.13 shows how this small modification to the penalization algorithm improves the performance of Multipath TCP. In the low-BDP environments, more than 75% of the experiments achieve an aggregation benefit of 0.85 or higher. And even the experiments in the high-BDP environments have their median increased up to an aggregation benefit of around 0.5. Long data transfers with Multipath TCP are now less vulnerable to transient receive-window limitations and achieve a high aggregation benefit.

Effect of transient receive-window limitations

Our modification to the penalization algorithm mitigates the effect of the transient receive-window limitations for long flows. However, the fundamental problem is still there. Transient receive-window limitations slow down the increase rate of the congestion window during slow-start. We chose a specific set of parameters where these transient receive-window limitations are particularly apparent. The first bottleneck has a capacity of 1 Mbps, while the second has a capacity of 50 Mbps. Both bottlenecks have a high propagation delay and queuing delay. A plot of the congestion-window's evolution of the fastest path during the slow-start

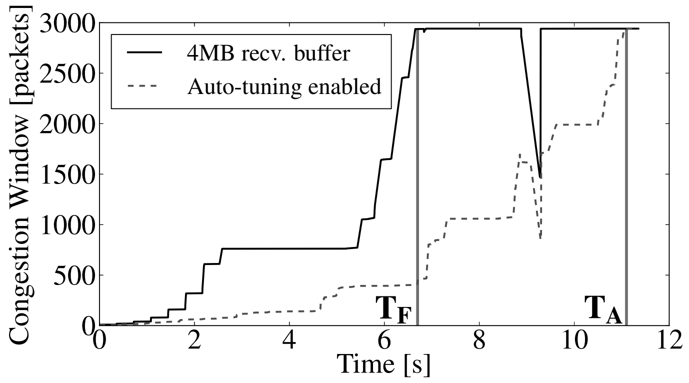


Figure 4.15: During the slow-start phase of a Multipath TCP subflow, the congestion window increases much slower when auto-tuning is enabled.

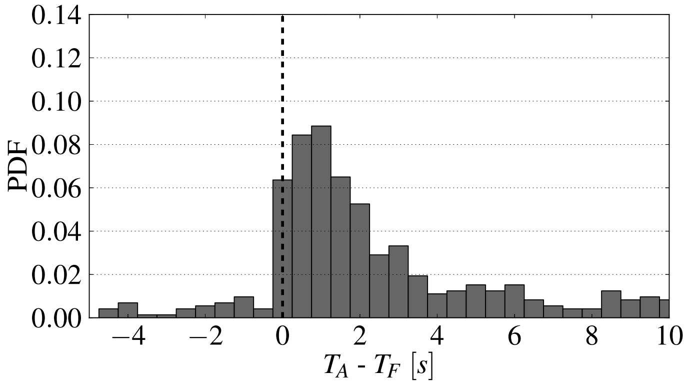


Figure 4.16: With auto-tuning enabled, Multipath TCP is slower to increase the congestion window during slow-start compared to a fixed receive buffer at 4 MB.

phase is shown in Figure 4.15[2]. When auto-tuning is enabled, the congestion window increases much slower, reaching its maximum at time T_A (indicated in the Figure), about four seconds later compared to a fixed receive buffer at 4 MB (T_F).

We measure the difference between T_A and T_F in our high-BDP environment among the 200 parameter sets generated with the space-filling design. Figure 4.16 illustrates the PDF of these differences. Ideally, the difference between T_A and T_F should be zero. However, this is only true for 6% of the experiments. For a large portion of the experiments the congestion window reaches its maximum between 500ms and 2000ms faster if auto-tuning is disabled.

This could have a negative impact on flow-completion time of short flows, as the connection is receive-window limited during its slow-start phase. An ideal auto-tuning algorithm should not prevent the sender from increasing the sending rate - even during slow-start. However, the solution to this is far from trivial. It would need to accurately estimate the round-trip-time at the receiver side in order to estimate how fast the sender will go through the slow-start phase. However, it must make sure that it is not too agressive. So, a good compromise must be found between memory utilization, responsiveness and throughput.

4.3 Scheduling

The previous section has shown how one might solve the problem of receive-window limitations in a reactive manner, meaning after the limitation happened. However, it all depends on accurately scheduling the traffic on the subflows in order to avoid these limitations. Further, the problem of head-of-line blocking has not been addressed by the retransmission and penalization algorithm. This section looks at heterogeneous environments by considering the behavior of the scheduler and how its decisions might have an impact.

The scheduler is responsible for the distribution of data over multiple paths and a wrong scheduling decision might cause head-of-line blocking or receive-window limitation, especially when the paths are heterogeneous. In such a scenario, the user will observe high delays as well as goodput degradation for its application, resulting in poor user experience. Therefore, the scheduler can have a significant impact on the performance of Multipath TCP.

In this section we propose a modular scheduler framework that allows to easily change the way data is distributed over the available subflows. Further, we

[2]The Figure shows that even the flow whose receive buffer is fixed at 4 MB experiences a stall in the congestion-window's increase rate (between 2.5 and 5.5 seconds). This is due to a very specific issue of the Linux TCP window handling and has been fixed by http://patchwork.ozlabs.org/patch/273311/. Unfortunately the Linux Multipath TCP-stack v0.86 does not include this recent fix.

evaluate different schedulers for Multipath TCP and provide an in-depth performance analysis considering both bulk data transfers and application-limited flows. We consider goodput and application delay as metrics. Our experiments include a broad evaluation within an emulated environment using the *Experimental Design*-approach of Chapter 3 [PKB13] and an evaluation using real WiFi and 3G networks within the NorNet testbed [KBE+14]. We identify the impact of the scheduling decisions on the performance of Multipath TCP and illustrate the underlying root cause for the observed behavior. We provide guidelines on the properties of a good scheduler to achieve a good performance under different scenarios (the design of such a scheduler is out of the scope and left for future work).

4.3.1 Schedulers

A wrong scheduling decision might result in head-of-line blocking or receive-window limitation, affecting the performance of Multipath TCP–as discussed in the previous section. Accurately scheduling data across multiple paths while trying to avoid head-of-line blocking or receive-window limitation is a difficult problem, in particular when using heterogeneous paths. How to schedule different SCTP streams across the different SCTP associations has been analyzed in [STR10]. However, SCTP's design is different from Multipath TCP. SCTP does not support the transmission of a single stream across different paths. [SBL+13] tries to achieve ordered delivery at the receiver by taking the delay of each path into account. While in theory this is a promising approach, it is unclear how feasible it is in a real-world kernel implementation where only a rough estimate of the path's delay is available to the scheduler.

We implemented a modular Multipath TCP scheduler framework[3]. Whenever the stack is ready to send data (e.g., an acknowledgement has freed up space in the congestion window, or the application pushed data on the send-queue), the scheduler is invoked to execute two tasks: first, choose a subflow among the set of TCP subflows and; second, decide which segment to send considering the properties of this subflow. We added callbacks from the Multipath TCP stack that invoke the functions specific to each scheduler. This allowed us to design the scheduler in a modular infrastructure, as it is done with the TCP congestion control algorithms [SH07]. A pseudo-code implementation of this behavior can be seen in Figure 4.17. A *sysctl* allows to choose the default scheduler for all Multipath TCP connections, and further socket-options allow the user-space application to explicitly set the scheduler for a particular connection.

Within our modular framework, we implement different schedulers. First, we discuss a simple round-robin scheduler. Then, considering the heterogeneous net-

[3]The code is publicly available at http://multipath-tcp.org

```
MPTCP: A Multipath TCP connection, ready to send data. MPTCP.sched
    represents a structure containing the specific callbacks.
    // Get a subflow, ready to send data
1:  subflow = MPTCP.sched.get_subflow();
2:  while subflow != NULL do
3:      data = MPTCP.sched.get_data(subflow);
        // Send data over this subflow
4:      while data != NULL do
5:          send_data(subflow, data);
6:          data = MPTCP.sched.get_data(subflow);
7:      subflow = MPTCP.sched.get_subflow();
```

Figure 4.17: Pseudo code of the modular scheduler framework, using callbacks to invoke the scheduling functions.

works where significant delay differences are observed between the subflows, we discuss delay-based schedulers. A first evaluation of schedulers has been done in [SGTG+12]. But only a limited environment has been used in the evaluation and improvements to the schedulers [RPB+12] were not yet part of the Multipath TCP implementation.

Round-Robin (RR)

The round-robin scheduler selects one subflow after the other in round-robin fashion. Such an approach might guarantee that the capacity of each path is fully utilized as the distribution across all subflows is equal. However, in case of bulk data transmission, the scheduling is not really in a round-robin fashion, since the application is able to fill the congestion window of all subflows and then packets are scheduled as soon as space is available in each subflow's congestion window. This effect is commonly known as *ack-clock* [Jac88].

Such a scheduler has already been discussed for concurrent multipath transfer SCTP [IAS06]. [DSTR10] evaluates how such a round-robin scheduler behaves in CMT-SCTP for multi-streaming compared to a scheduler that assigns each stream to a specific path.

Lowest-RTT-First (LowRTT)

In heterogeneous networks, scheduling data to the subflow based on the lowest round-trip-time (RTT) is beneficial, since it improves the user-experience. It reduces the application delay, which is critical for interactive applications. In other words, the RTT-based scheduler first sends data on the subflow with the lowest RTT estimation, until it has filled the congestion window (as it has been first described in [RPB+12]). Then, data is sent on the subflow with the next higher round-trip time.

The same way as the round-robin scheduler, as soon as all congestion windows are filled, the scheduling becomes *ack-clocked*. The acknowledgements on the individual subflows open space in the congestion window, and thus allow the scheduler to transmit data on this subflow.

As explained in Section 4.1, a delay difference triggers head-of-line blocking and/or receive-window limitation. Next, we discuss two extensions to the RTT based scheduler. The first solution reacts upon receive-window limitation, and the second solution minimizes the delay difference in the presence of bufferbloat on the individual subflows.

Retransmission and Penalization (RP)

The opportunistic retransmission and penalization algorithm, as already explained in Section 4.2.1 allows to compensate for delay differences. Opportunistic retransmission re-injects the segment causing head-of-line blocking on the subflow that has space available in its congestion window (similar to chunk rescheduling for CMT-SCTP [DBRT10]). This allows to quickly overcome head-of-line blocking situations and compensate for the RTT differences. Further, the penalization algorithm reduces the congestion window of the subflow with the high RTT, hence, reducing the sending rate and the effect of bufferbloat on the subflow.

Bufferbloat Mitigation (BM)

Another source for high RTTs are large buffers on routers and switches along the subflow's path. TCP will try to fill these buffers creating bufferbloat, resulting in very high RTTs.

The *bufferbloat-mitigation* algorithm caps the RTTs by limiting the amount of data to be sent on each subflow, hence, controlling the bufferbloat [STA14]. The goal here is not to significantly improve goodput, but instead, to improve the application delay-jitter and reduce buffer size requirements.

The main idea behind the bufferbloat-mitigation algorithm is to capture bufferbloat by monitoring the difference between the minimum smoothed RTT ($\mathrm{sRTT}_{\mathrm{min}}$), and smoothed RTT (sRTT). Whenever sRTT drifts apart from $\mathrm{sRTT}_{\mathrm{min}}$ on the same subflow, as a result of sending data, we take it as an indication of bufferbloat. Therefore, we cap the congestion window for each subflow by setting an upper bound $\mathrm{cwnd}_{\mathrm{limit}}$.

$$\mathrm{cwnd}_{\mathrm{limit}} = \lambda \times (\mathrm{sRTT}_{\mathrm{min}}/\mathrm{sRTT}) \times \mathrm{cwnd}$$

where λ determines the *tolerance* between sRTT_{min} and sRTT. Within the remainder of this chapter, we fix λ to 3, which has proven to bring the best results [STA14].

4.3.2 Evaluation

Traffic characteristics and the environment, where the TCP subflows pass, influence the schedulers' performance. This section evaluates two Multipath TCP metrics, namely goodput and application delay-jitter in emulated and real-world experiments.

Experiment and Emulation Setup

Within `Mininet` [HHJ+12] the *Experimental Design* approach [PKB13] is used, where we set both low- and high-BDP environments to evaluate the schedulers. These environments are the same as in Chapter 3 with the capacity, propagation delay and bufferbloat as influencing factors. The Multipath TCP implementation used in all our evaluations is based on release 0.88.

The real-world environment uses the `NorNet` testbed [KBE+14]. We consider the smartphone use case where we utilize both WLAN and 3G (UMTS) interfaces to evaluate the Multipath TCP performance with heterogeneous links. The WiFi connection uses a public WLAN, connecting ca. 100 people during work hours in a large office complex with several other interfering WLAN networks. On the system level, we continuously flush all cached TCP metrics to avoid any dependency between experiments. Finally, the Olia congestion control [KGP+12] is used in all experiments. Similar results were obtained with the coupled congestion control scheme [WRGH11].

Bulk-Transfer

One of the goals of Multipath TCP is to increase the application goodput [RHW11], which can be measured by transferring bulk data between two Multipath TCP capable end hosts.

Mininet Within `Mininet` we generate a bulk-transfer using iperf, where each transfer lasts for 60 seconds. Our measurements cover the same 200 different settings, classified as low-BDP and high-BDP environments, as in Chapter 3. For detailed information about the test environments as well as the system-level settings, we refer to [PKB13] and Chapter 3. Here, we measure the *aggregation benefit* which has been explained in Chapter 3.

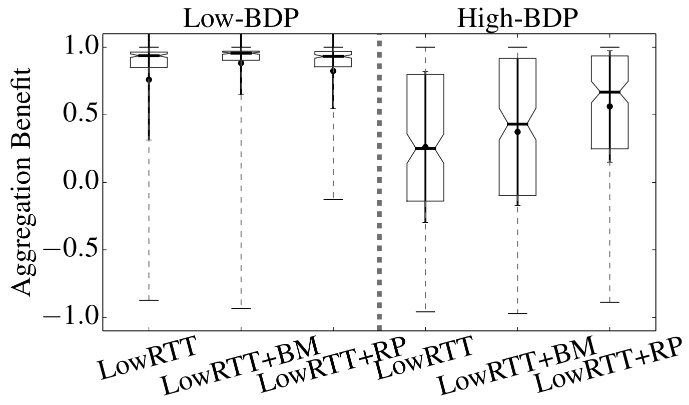


Figure 4.18: `Mininet`: In high-BDP the connection becomes receive-window limited and BM and RP show their benefits.

Figure 4.18 shows the `Mininet` results. We skip the RR scheduler since it performs similar to LowRTT. This is because in bulk-transfers the TCP subflows are saturated and are thus controlled by the *ack-clock*. Therefore, available space in the congestion window controls the way packets are multiplexed across the subflows rather than the scheduler.

In the low-BDP environment there is no significant difference between schedulers. Each of them achieves close to perfect bandwidth aggregation. Only the RP algorithm improves the worst-case result among the 200 experiments to achieve an aggregation benefit equal to the best available path. In the high-BDP environment receive-window limitations may occur. In this case the RP and BM techniques described in Section 4.2.1 improves the aggregation benefit. The RP technique has a higher benefit in the lower 25th percentile and the median. This is thanks to the retransmission of the blocking segment as the receive-window limitation in these cases is not due to bufferbloat but rather due to a difference in the baseline RTT.

NorNet Within `NorNet Edge` we tested bulk-transfers of 16 MB files in downlink with both unbounded (16 MB) and bounded (2 MB) buffers. The bounded buffers will make the connection more likely to be limited by the receive window. We repeat each measurement around 30 times for each scheduler. All measurements are performed in the same networks and at the same locations over a period of 3 weeks.

Figure 4.19 shows the Multipath TCP goodput, for all schedulers. For a bulk-transfer with unbounded buffer sizes (16 MB), the aggregation benefit of Multipath TCP across all schedulers is similar. Each scheduler is able to efficiently

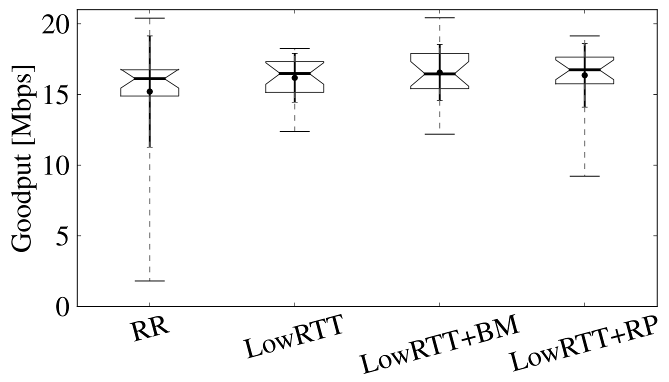


Figure 4.19: `NorNet`: With unbounded buffers (16 MB), each scheduler achieves the goodput.

aggregate the bandwidth of WLAN and 3G together within the `NorNet` testbed. The unbounded buffers allow for sufficiently large memory, so that no receive-window limitation occurs.

However, if the receive buffer is bounded (as it is often the case on a smartphone), the Multipath TCP connection may become receive-window limited. Figure 4.20 shows the Multipath TCP goodput in this case. Here, one can see that LowRTT+BM slightly outperforms the other schedulers. In case of the LowRTT+RP scheduler, the effect of bufferbloat is not optimally reduced by the penalization algorithm, as it does not manage to bring the congestion window sufficiently down so that the delay-difference is reduced. This happens, because the RP algorithm is reactive and does the penalization only after a limitation happened. The BM algorithm is proactive and prevents a high delay-difference beforehand and thus achieves a higher goodput.

Application-Limited Flows

This section evaluates the impact of the schedulers on delay-jitter with rate-limited traffic, i.e., when the application is not saturating the connection. In this case, the scheduler has available space in all subflows and should select the one that guarantees the lowest delay for each segment. We evaluate the delay-jitter by using the application described in Chapter 3.

Mininet Within Mininet we ran this application in the high-BDP environment of the *Experimental Design* approach, for a total of 200 experiments. Figure 4.21 shows the CDF of the worst-case relative delay increase. The delay increase is

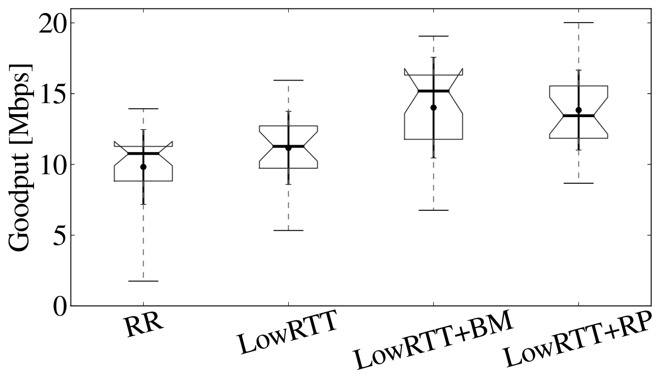


Figure 4.20: NorNet: With bounded buffers (2 MB), LowRTT+BM and LowRTT+RP achieve the best performance.

expressed in % compared to the lowest one-way delay. E.g., if the minimum one-way delay is 20 ms, but a block of 8 KB has been received rather after 40 ms the relative delay increase has a value of 100%. We show the worst delay-increase among all 8 KB blocks of each experiment as this will affect the user-experience most.

In Figure 4.21 it is visible that the RR scheduler is particularly bad in terms of application delay. 70% of the experiments using the LowRTT scheduler have a range between 10 and 100% of delay-increase. Using a RR scheduler, roughly 40% of the experiments have a delay-increase between 100 and 1500%. Such delay increases has significant impact on delay-sensitive applications since they would need to maintain large buffers to react upon these delay-spikes.

NorNet Within NorNet we evaluate the delay-jitter using rate-limited applications transmitting at 500 Kbps and 1875 Kbps in downlink, using unbounded buffers. The values for the application-limited rates are at approximately 5 and 18% of the mean goodput of the bulk-transfer. We repeat each measurement around 30 times for each configuration.

Figure 4.22 shows the variation of the application delay for all schedulers. One can see that in all application-limited scenarios, RR performs mostly worse compared to all other schedulers. More prominent, in Figure 4.22(a), RR's delay-variance shows to be up to 10 times worse compared to LowRTT. This can be explained as RR simply schedules data based on congestion-window space, which is not a limiting factor in this particular scenario. Looking into our dataset, one can see that both subflows carry a similar amount of data, thus increase head-of-line blocking.

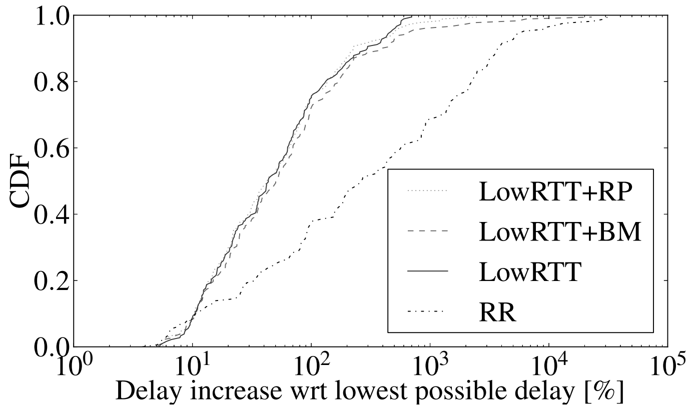


Figure 4.21: `Mininet`: Using the lowest-RTT-first scheduler greatly reduces the application-delay variance in `Mininet`.

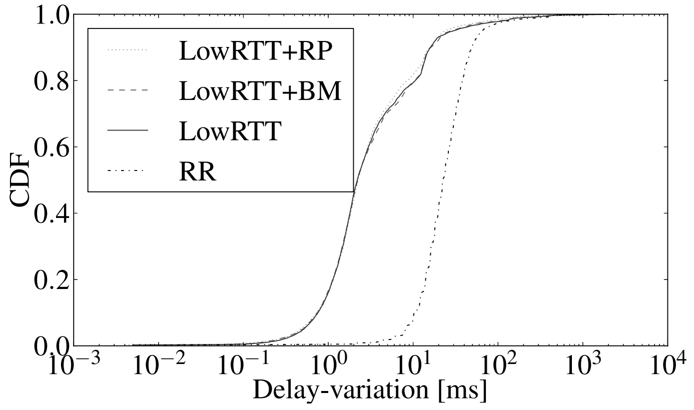
For all other schedulers we observe that for very low application rates, see Figure 4.22(a), LowRTT utilizes mainly one subflow, the subflow with lowest RTTs. This is because the congestion window space is not a limiting factor, and it has mostly enough space to carry all data available.

By increasing the application rate, see Figure 4.22(b), LowRTT performs in at least 60% of the cases up to 10 times better compared to RR. The remaining 40% can be explained as we see that the subflow with the higher RTT (3G network) contributes more compared to the scenario in Figure 4.22(a). This happens, because at a higher sending rate, occasional congestion on the WiFi network will make the LowRTT scheduler send traffic on the 3G subflow. This will introduce head-of-line blocking due to the higher delay over the 3G network and thus increases the delay-variation.

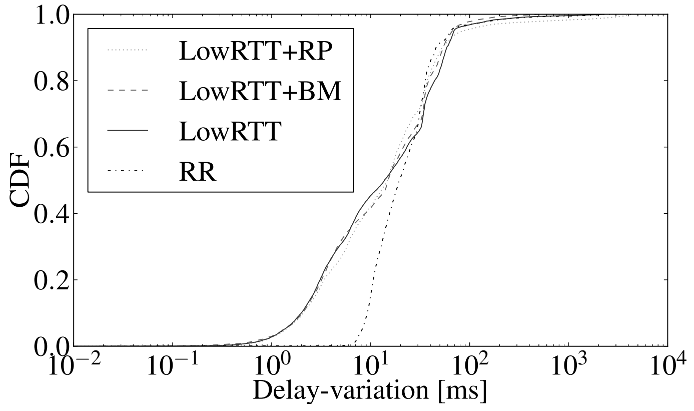
We also evaluate the delay-jitter when sending at unlimited rate within the `NorNet` testbed. In this case, both WiFi and 3G are fully utilized and bufferbloat might happen on the 3G path. We observe that the LowRTT+BM scheduler effectively reduces the bufferbloat and keep the application delay lower compared to other schedulers.

4.4 Conclusion

In this chapter, we have analyzed the issues that Multipath TCP encounters when being used in heterogeneous environments. We proposed heuristics that might help to overcome these issues. Our first approach is reactive, handling receive-window limitations after they occur by opportunistically retransmitting



(a) 500 Kbps



(b) 1875 Kbps

Figure 4.22: `NorNet`: 3G and WLAN with unbounded (16 MB) buffers

and trying to mitigate future limitations by penalizing subflows. An analysis by using the experimental design approach allowed to fine-tune the algorithm.

The second approach looks at heterogeneous environments from the scheduler's point-of-view. We proposed and implemented a modular scheduler selection framework that allows Multipath TCP to change the way data is multiplexed across the different TCP subflows. We used this framework to experimentally evaluate schedulers in a wide variety of environments in both emulated and real-world experiments. In these environments, we could quantify the performance of different schedulers, and scheduler extensions, with respect to goodput as well as application delay-jitter.

A bad scheduling decision triggers two effects: First, head-of-line blocking if the scheduler sends data across a high-RTT subflow. Second, receive-window limitation, which prevents the subflows from being fully utilized. We have shown that a simple strategy to preferentially schedule data on the subflow with lowest RTT (LowRTT) helps to reduce the application delay-jitter compared to a simple round-robin (RR) scheduler.

Multipath scheduling should ideally be done in a way that the data is received in-order. This minimizes head-of-line blocking and receive-window limitations as applications are able to continuously read data out of the receive queue. However, it is not trivial to design such a scheduler with rough estimations on capacity and RTTs of the paths, maintained by the kernel. Up to today, the reactive approach with retransmission and penalization seems to provide reasonably good results. Nevertheless, improvements in the scheduler might be promising to improve the performance of Multipath TCP.

Chapter 5

Multipath TCP at scale

The Linux Kernel implementation of Multipath TCP, started by Sébastien Barré during his PhD thesis [Bar11], has been developed over the last years. It is recognized as the reference implementation of Multipath TCP. Being the most feature-rich and complete implementation of RFC 6824, it is slowly being used by more and more commercial deployments. This chapter provides an overview of the main building blocks that allow Multipath TCP to be used in the Linux Kernel. This chapter is based on release v0.88 of the Multipath TCP implementation[1].

During the design process of the Linux Kernel implementation several goals influenced its design. These goals were motivated by the fact that Multipath TCP should be designed in such a way that it could be relatively acceptable for an upstream submission to the official Linux Kernel. Our design-goals include:

1. **Minimize performance impact on regular TCP** - a Multipath TCP implementation that reduces the performance of regular TCP is not suitable for a general-purpose operating system like the Linux Kernel.

2. **Reasonable complexity inside TCP** - Multipath TCP is tightly mangled within the TCP stack. To reduce the maintenance overhead, the complexity should be kept at a reasonable level.

3. **Achieve high performance for Multipath TCP** - finally, the implementation should achieve high performance for Multipath TCP.

This chapter details how our implementation transforms the Linux TCP/IP stack such that Multipath TCP can be used. We justify the design decisions with respect to the above presented goals and show that our implementation achieves a high performance.

[1] `http://multipath-tcp.org/pmwiki.php?n=Main.Release88`

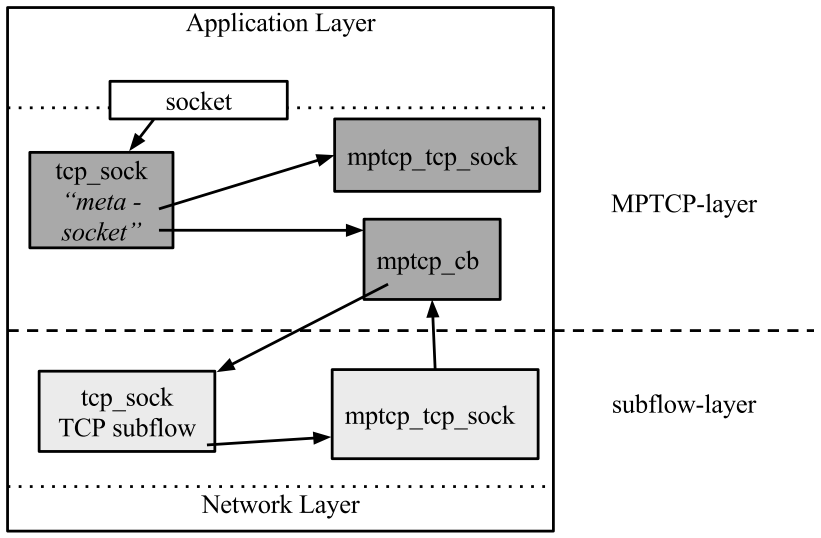


Figure 5.1: The different structures are connected through pointers (arrows in the figure). Creating additional structures allows to avoid increasing the memory footprint of regular TCP.

5.1 Data structures

As has already been explained in Chapter 2, the Linux Kernel implementation of Multipath TCP is split in two layers. The aggregating MPTCP-layer and the individual subflows at the subflow-layer. Within the implementation we call the MPTCP-layer the *meta-socket* which represents the state-machine of each Multipath TCP connection. The state of a full Multipath TCP connection is composed of several structures linked to each other by pointer references (as can be seen in Figure 5.1). It is roughly divided in two parts. First, the structures that are relative to the MPTCP-layer (in dark gray) and second, the structure that belong to the subflows (in light gray).

The Linux Kernel maintains the state of a TCP connection in a structure called *tcp_sock*. Multipath TCP requires to store additional information for each TCP subflow. To avoid increasing the memory-footprint of *tcp_sock* (and thus of all TCP connections - even those that do not use Multipath TCP), the structure *mptcp_tcp_sock* has been added. Each Multipath TCP subflow, as well as the meta-socket, contains a pointer to the additional allocated *mptcp_tcp_sock*. It contains information such as the address-id that belongs to this subflow (useful when a REMOVE_ADDR is received to decide which subflow needs to be removed).

The MPTCP-layer also implements a complete TCP state machine. Thus, the same *tcp_sock*, together with the *mptcp_tcp_sock*, is used to store this state. Additionally, general information for the entire connection must be maintained. Again, to avoid increasing the memory footprint of regular TCP, we created the structure *mptcp_cb*, representing the Multipath TCP control buffer. It holds information like pointers to TCP subflows and the list of addresses advertised by the peer.

5.2 Connection setup

During the connection setup phase, the end-hosts detect whether the peer supports Multipath TCP and exchange the crypto-material to compute the initial data sequence number as well as the token and keys necessary for the establishment of the new subflows. From an implementation point-of-view, the goal is to create the necessary structures to allow a Multipath TCP connection to exist. An implementer might take the following approach.

He might allocate all the data structures necessary for Multipath TCP right at the beginning. The advantage of this approach is that, if the peer supports Multipath TCP, all the structures are already in place. If however, at the reception of the SYN/ACK the host realizes that the peer does not support Multipath TCP, these structures are unnecessary and must be freed again. In a world, where Multipath TCP is ubiquitously available, this choice would probably be the most favored. However, today rather the opposite is the case. Most connections will still be using regular TCP. A general-purpose operating system should not assume that chances are high that a Multipath TCP connection will be established. In this case, this first choice would bring an excessive performance penalty for the common TCP connections.

Thus, in the Linux kernel implementation we chose to optimize for the common case and sticked to an alternative implementation. Our implementation allocates the Multipath TCP specific structures only after we detect that the peer supports Multipath TCP. This approach is explained in the following.

5.2.1 Client side

On the client side, an application creates a socket with the `socket()` system-call and connects to a server with `connect()`. We chose to only create a regular TCP socket upon the application's call to `socket()`. When the sending of the initial SYN is being triggered by the call to `connect()` (the call graph is shown in Figure 5.2), the client generates a random key and inserts it inside the MP_CAPABLE option (`mptcp_set_key_sk()`). Up to here, the client's only additional work compared to a regular TCP connection is the generation of the key (and verification of the uniqueness of the token). To ensure that the same key

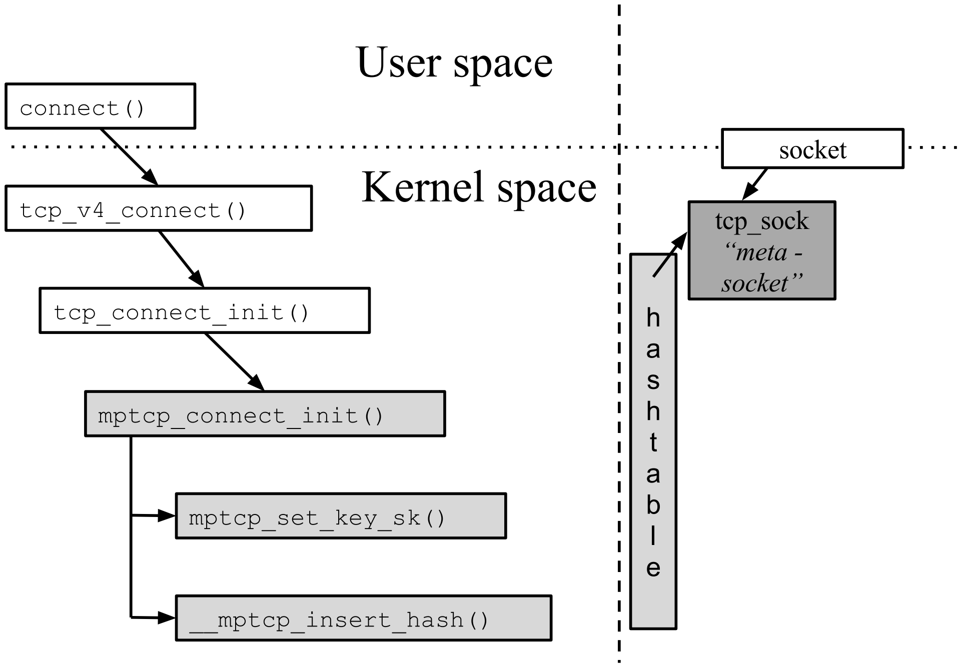


Figure 5.2: Upon a `connect()`, the client generates a random key but does not allocate additional data structures.

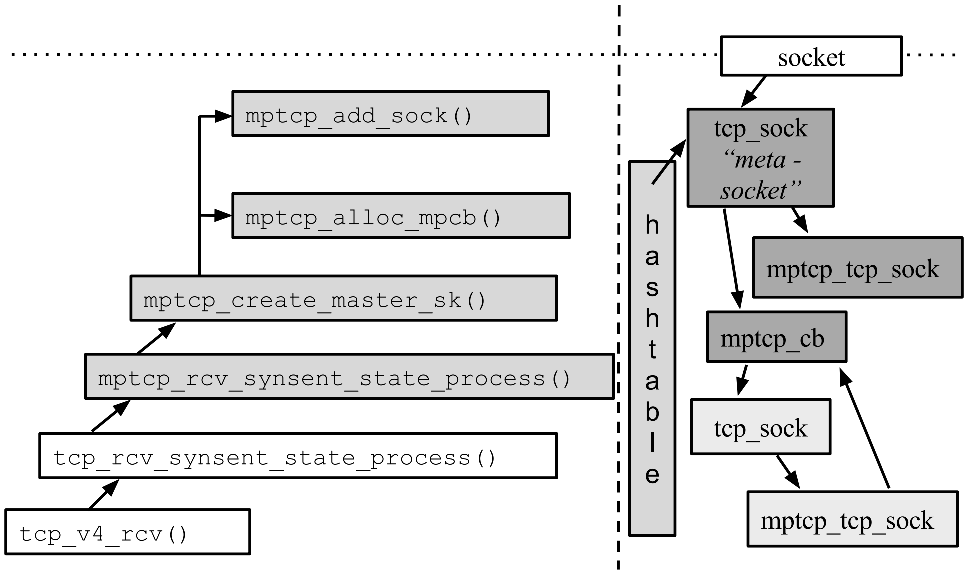


Figure 5.3: During the processing of the SYN/ACK containing an MP_CAPABLE option, the required data structures are allocated and linked together.

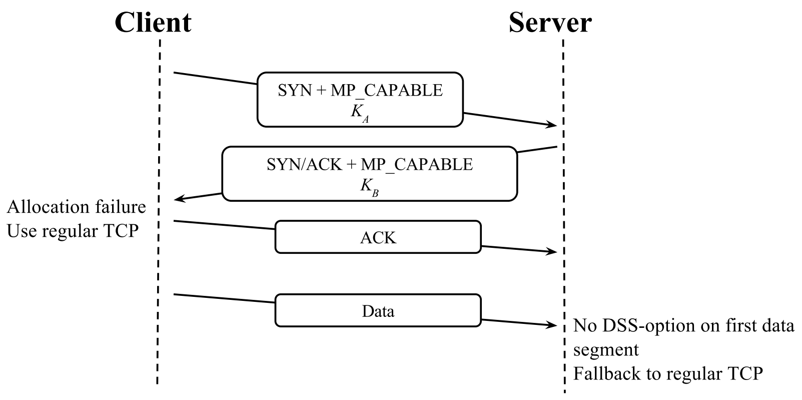


Figure 5.4: Upon a failed memory allocation on the client side, the connection falls seamlessly back to regular TCP

is not used twice for different connections, a reference to the *tcp_sock* is stored inside a hashtable, whose keys are the tokens of the Multipath TCP connections (`__mptcp_insert_hash()`). This hashtable is used to verify the uniqueness of the tokens. No other additional allocation is performed by the TCP stack at this stage.

The server replies with a SYN/ACK, including (or not) an MP_CAPABLE option. Upon receiving this reply, the client knows whether the server supports Multipath TCP. It can now allocate the data structures that are required for Multipath TCP. Two important points need to be considered for this procedure whose call-graph is shown in Figure 5.3.

It is known that under memory-pressure an allocation might fail. As the reception of the SYN/ACK is done within a soft-interrupt context, this allocation cannot sleep to wait for the available memory. For example, if the memory allocation of the *mptcp_tcp_sock* fails, Multipath TCP cannot be used for this connection. It will revert back to a regular TCP connection. Here, the middlebox support of Multipath TCP comes in handy, as shown in Figure 5.4. Because, the server actually thinks that Multipath TCP has been successfully negotiated (it replied with an MP_CAPABLE in the SYN/ACK), the seamless fallback to regular TCP described in Section 2.3.2 will make the server realize that indeed regular TCP is being used.

In case the server replies with a SYN/ACK without MP_CAPABLE option, the client is not affected by much additional work. It only needs to remove the *tcp_sock* from the above mentioned hashtable so that the key used in the MP_CAPABLE can be used again for subsequent Multipath TCP connection establishments.

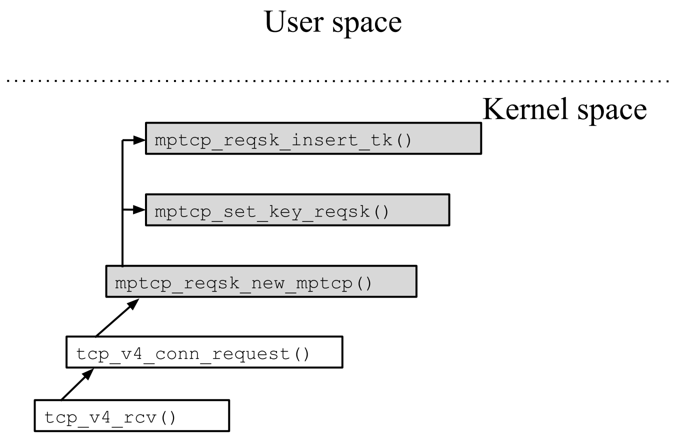


Figure 5.5: A slightly larger *request_sock* is created to store the random key and add the *request_sock* in the hashtable to avoid token collisions.

5.2.2 Server side

On the server side, the connection establishment can be split in two major steps. The first step happens during the reception of the SYN including the MP_CAPABLE option and the second step is triggered by the reception of the final ACK of the 3-way handshake.

The Linux kernel TCP stack does not create a full-fledged socket upon reception of a SYN. Instead, it creates a lightweight so-called *request_sock* that stores the minimal information necessary for a stateful handling of the 3-way handshake. To support a correct handling of Multipath TCP, only a small change had to be done in this process (call graph is shown in Figure 5.5). First, the allocated *request_sock* is slightly larger to accommodate the generated random key and some other state variables specific to Multipath TCP. It must be noted that this larger *request_sock* is only allocated if the SYN contains the MP_CAPABLE option. This allows for legacy TCP connections to not be impacted by the increased memory footprint and the resulting increase of cache misses. During the generation of the random key, the uniqueness of the corresponding token must be verified. Furthermore, to avoid collisions, a similar hashtable is used as on the client side and the *request_sock* is added to this one.

Upon reception of the final third ack of the 3-way handshake all the structures of Figure 5.1 are created. The TCP stack creates new sockets as a clone from the listener socket, inheriting many of the settings (e.g., socket options). Our implementation of Multipath TCP again tries to reduce the impact on the legacy TCP stack, thus lets the TCP stack create the socket, which will be used as the

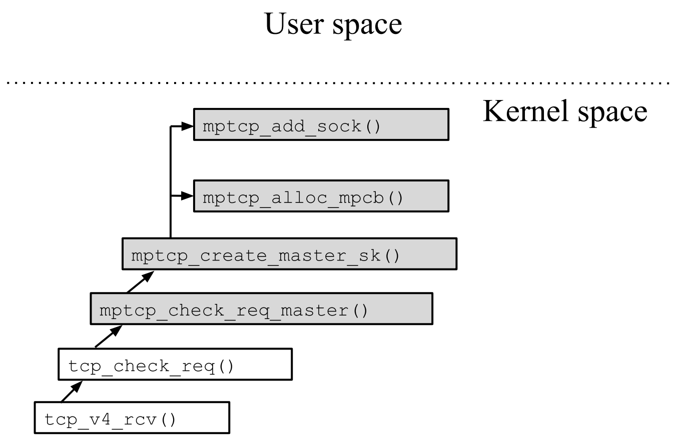


Figure 5.6: Similar to the client side, `mptcp_create_master_sk()` is in charge of creating the subflow and initializing all the necessary state.

meta-socket. Later in the stack, a call to `mptcp_create_master_sk()` is in charge of creating the subflow's *tcp_sock* and linking it to the *mptcp_cb* through a call to `mptcp_add_sock()` as shown in Figure 5.6.

We use a different type of *request_sock* structure to handle SYN's that contain the MP_CAPABLE option. This means that the release of the request socket's memory must be changed. Additionally, the request socket must be removed from the hashtable. The Linux TCP stack handles the request socket through several callbacks, defined in the structure *request_sock_ops*. To achieve our specific "treatment" of the request socket we redefined a new such structure (shown in Listing 5.1). Redefining the `destructor`-callback allows a very clean way to remove the request socket from the hashtable when its memory gets released.

Listing 5.1: Redefining a function pointer allows to nicely integrate Multipath TCP in the legacy TCP stack

```
struct request_sock_ops mptcp_request_sock_ops __read_mostly = {
        .family           =       PF_INET,
        .obj_size         =       sizeof(struct mptcp_request_sock),
        .rtx_syn_ack      =       tcp_v4_rtx_synack,
        .send_ack         =       tcp_v4_reqsk_send_ack,
        .destructor       =       mptcp_v4_reqsk_destructor,
        .send_reset       =       tcp_v4_send_reset,
        .syn_ack_timeout  =       tcp_syn_ack_timeout,
};
```

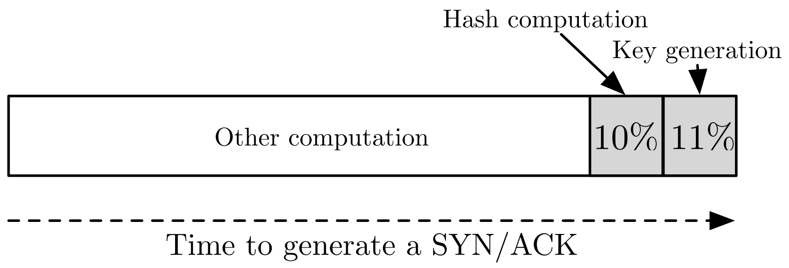


Figure 5.7: More than 20% of the time to generate a SYN/ACK with MP_CAPABLE is spent on the key and token generations while establishing the initial subflow.

5.2.3 Evaluation

The initial handshake involves the generation of the key, token and IDSN and the exchange of the keys. This has an impact on the computational overhead and security of Multipath TCP. The token is a hash of the key and must be locally unique. A consequence of using a hash is that the token may collide with an already existing connection.

After each key-generation the server must verify if the resulting token is locally unique by computing the hash of the key. If it is not unique, a new random key must be generated and again verified for the token's uniqueness. This increases the computational overhead, which grows with the number of existing Multipath TCP connections.

We measured the time to generate a SYN/ACK with MP_CAPABLE on Xeon-class servers with an early version of our Linux Kernel implementation of Multipath TCP [RPB+12]. In order to identify the time consumed by the random number generation and hashing, we replaced the corresponding code by dummy functions. The gain observed when replacing the code represents the cost of the corresponding code. Our evaluation shows that the random number generation and hashing of the key introduces an important overhead. More than 20% of the time taken to generate a SYN/ACK is spent in these two actions (Figure 5.7).

Further, we evaluate the impact of verifying the token's uniqueness. This impact increases as more and more Multipath TCP connections are established on the server, as it must look through many tokens to verify the uniqueness of the generated key.

Our measurement sends a SYN to the server (using the same hardware as the above measurement), including an MP_CAPABLE option. The server replies with a SYN/ACK including the MP_CAPABLE with its choice for the connection's key. This key is guaranteed to be unique among all Multipath TCP connections established on the server. We send 20000 of these SYN segments to the server and

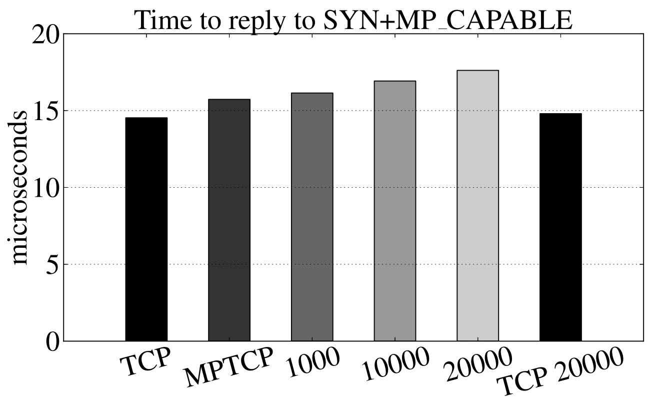


Figure 5.8: Multipath TCP increases the time required to reply to a SYN + MP_CAPABLE because it needs to verify the uniqueness of the token. The more connections are pre-established, the longer this process takes.

collect with `tcpdump` traces on the server. To avoid increased CPU load due to tcpdump storing on the hard drive, we save the trace on a ramdisk. This packet capture allows us to compute the average delay between the reception of the SYN and the transmission of the SYN/ACK. We compare regular TCP with Multipath TCP without preloading the server with connections. Then, we preload the server with 1000, 10000 and 20000 connections. This allows us to measure the impact of the uniqueness verification of the token.

Figure 5.8 shows the mean among the 20000 SYN/ACK delays. It can be seen that even without preloaded connections, Multipath TCP uses a slightly higher delay. This is solely due to the generation of the random key, as well as the hash computation to generate the token. This cost however does not increase if more connections are preloaded. As the number of preloaded connections increases, verifying the uniqueness of the token takes more time. During this verification, the server needs to lookup the hashtable, searching for token collisions. However, collisions in the hashtable make this lookup far from $O(1)$. A solution would be to simply increase the size of the hashtable. This increase comes together with a cost in terms of memory consumption. Nevertheless, even with our relatively small hashtable of only 1024 entries, we are able to keep the overhead with 20000 preloaded connections below 5 microseconds.

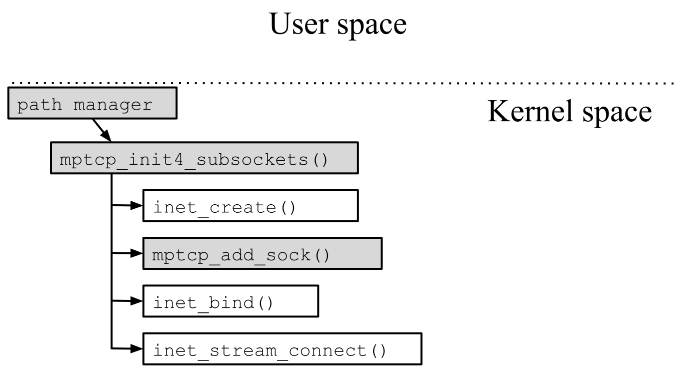


Figure 5.9: The path manager invokes `mptcp_init4_subsockets()`, in charge of mimicking an application that creates a new legacy TCP socket.

5.3 Subflow handling

Once the initial subflow has been correctly established and some data has been exchanged, additional subflows can be created. Our implementation only allows the client to trigger the creation of new subflows. If we would allow also the server to create new subflows, it may happen that multiple subflows are created across the same pair of IP addresses, as client and server may decide to initiate a subflow at the same time. Having multiple subflows across the same pair of IP addresses is not always a desirable situation. Additionally, clients are typically behind NAT devices, which actually do not allow the server from actively opening a new subflow to the client. The creation of additional subflows is explained in this section from the client's and the server's point-of-view.

5.3.1 Client side

A Multipath TCP implementation needs a heuristic that decides when to create a new subflow. This heuristic can be influenced by different factors. It might try to establish a full mesh across all available IP addresses, or just create multiple subflows across the same pair of IP addresses to benefit from ECMP within the network. One might imagine even more specific heuristics, based on the properties of the device's interfaces. To accommodate these different use-cases we implemented a generic and modular path-manager interface. This path manager is invoked by the Multipath TCP implementation through callbacks that are registered at the creation of a connection. It allows to act upon events like the successful establishment of the initial subflow or the reception of a remote address through the ADD_ADDR option. The path manager can then trigger the creation of a new subflow through a call to `mptcp_init4_subsockets()`, detailed in the following.

The creation of a new subflow on the client side involves multiple steps, similar to the creation of a legacy TCP connection by a client-side application, shown in Figure 5.9. First, a *tcp_sock* is created and bound to a specific interface by the calls to `inet_create()` and `inet_bind()`. The sending of the SYN is then triggered by the call to `inet_stream_connect()`. The TCP stack will then simply include the MP_JOIN inside the outgoing SYN together with the token and a random number.

To allow the TCP stack to retrieve the token to add to the MP_JOIN option, our implementation creates the *mptcp_tcp_sock* structure prior to the sending of the SYN (through a call to `mptcp_add_sock()`). This is in contrast with the establishment of the initial subflow. The argument of reducing the overhead for legacy TCP connections does not count anymore for additional subflows of an existing Multipath TCP connection. The new subflow will not fallback to regular TCP if the SYN/ACK does not contain an MP_JOIN option. It will rather be destroyed by the Multipath TCP stack. Further, if we would only allocate the structures upon reception of the SYN/ACK we would add the risk that a memory allocation failure requires to discard the SYN/ACK. A cost we accepted for the initial subflow for the sake of reducing the impact on legacy TCP - but a cost that is unnecessary for the new purely Multipath TCP-related subflows.

When the server replies with a SYN/ACK, including the MP_JOIN option, the client must verify the correctness of the HMAC. In case it is incorrect, it replies with a RST (as specified by RFC6824) and closes the new subflow.

The 3-way handshake of new subflows differs in Multipath TCP compared to regular TCP in such a way that it requires a reliable transmission of the third ACK to ensure the correct delivery of the HMAC included in this acknowledgment. This implies that a substantial change must be done to the implementation, as we need a retransmission timer for this third ACK. The regular retransmission timer cannot be used as the ACK does not consume a byte in the subflow's sequence number space. Our implementation uses an additional retransmission timer, in charge of retransmitting the third ACK until the server replies with an acknowledgment to signal the reception and validity of the third ACK and its HMAC. Further, any transmission of data must be prevented on this subflow as long as the final fourth acknowledgment has not been received by the client. We achieve this by adding a `pre_established`-flag to the *mptcp_tcp_sock* that is being verified when the Multipath TCP scheduler decides to transmit data on a subflow. As long as this flag is set to 1, the scheduler will not attempt to transmit data on this subflow.

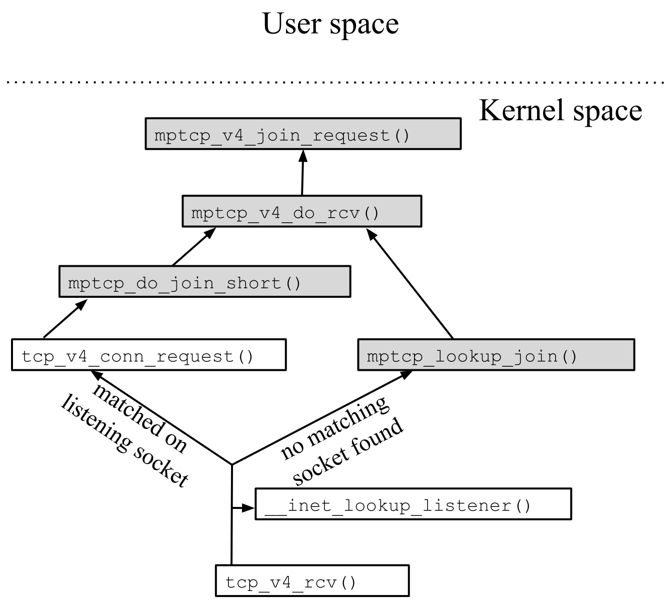


Figure 5.10: If the SYN + MP_JOIN matches on a listening socket, this code-path is being reused.

5.3.2 Server side

On the server side, the incoming SYN must be linked to the Multipath TCP connection based on the token included in the MP_JOIN option. An incoming SYN + MP_JOIN may have a different destination port than the one used in the initial subflow or use different IP addresses. A Multipath TCP implementation must thus check whether each and every incoming SYN contains an MP_JOIN option and whether the indicated token corresponds to an existing connection. In this case, the processing of the SYN must be done in such a way that it does not trigger the creation of a new legacy TCP connection but rather that this new subflow is linked to the existing Multipath TCP connection that matches the indicated token. Thus, an MP_JOIN option inside the TCP option space has priority when it comes to matching the SYN to a listening socket.

To redirect the SYN processing to the Multipath TCP connection, a naive implementation would parse the TCP options of every incoming SYN segment, looking for an MP_JOIN option. Only after this "pre-parsing" the control flow would be redirected to the legacy TCP stack to allow matching on a listening socket. It is obvious that this presents a performance penalty for a generic operating system, whose majority of the incoming SYN segments probably do not contain the MP_JOIN option.

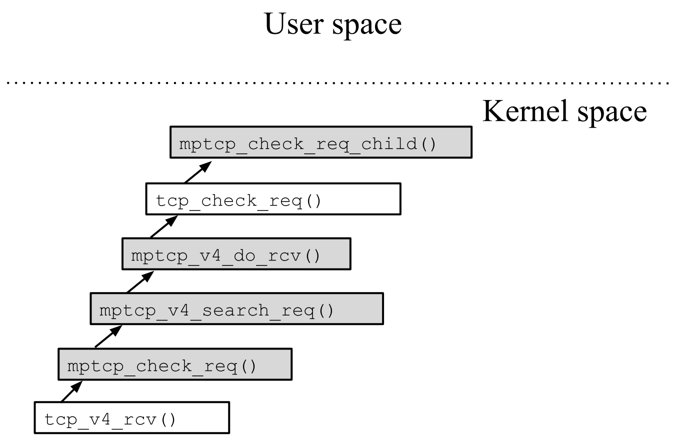


Figure 5.11: For every incoming ACK we need to verify whether a corresponding request socket exists.

Our implementation handles this by implementing the control flow shown in Figure 5.10. First, the incoming SYN may have a destination port corresponding to an existing listening socket. In this case, the legacy TCP stack will match the SYN on this socket through the call to `__inet_lookup_listener()`. Later on, in `tcp_v4_conn_request()`, the TCP option space is being parsed. At this point, the stack realizes that indeed an MP_JOIN option is included in the SYN, enforcing us to redirect the SYN handling to the Multipath TCP connection. `mptcp_do_join_short()` will lookup in the hashtable the connection that is linked to the indicated token and let it create a request sock and emit a SYN/ACK. Figure 5.10 also shows another call-trace in which no listening socket was found for the 5-tuple of the SYN. In this case we are still forced to parse the TCP options separately in `mptcp_lookup_join()` to search the MP_JOIN and lookup the Multipath TCP connection in the hashtable.

The server then creates in `mptcp_v4_join_request()` our enhanced request socket that stores additional state information, already described in Section 5.2.2. This request socket must be stored as part of the Multipath TCP connection but also inside a hashtable that allows it to be looked up based on the 5-tuple. This is necessary due to several reasons. During the 3-way handshake, the third acknowledgment triggers the creation of the TCP socket on the server side (as explained in Section 5.2.2). The host finds the request socket that corresponds to the incoming ACK by matching it on the listening socket, which stores this request socket in a list. But, during the establishment of a new subflow, there is no listening socket holding the request socket. This means that for every incoming acknowledgment that does not match on an existing estab-

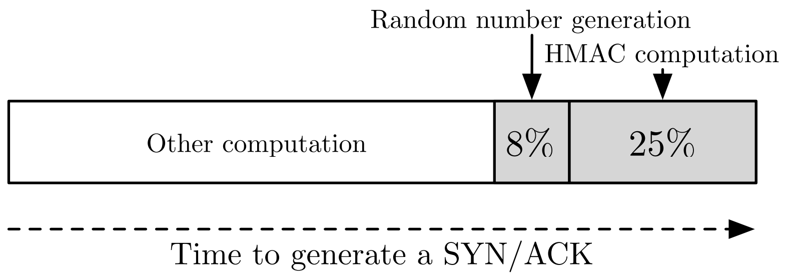


Figure 5.12: Upon the establishment of additional subflows, the HMAC accounts for 25% of the time to generate a SYN/ACK with MP_JOIN

lished socket we need to verify if there exists a pending request socket, waiting for the final acknowledgment of the MP_JOIN exchange. The current implementation indeed takes this brute-force approach, shown in Figure 5.11. A call to `mptcp_check_req()` looks up the 5-tuple of an incoming ACK (through a call to `mptcp_v4_search_req()`) inside our hashtable to see if there exists a pending request socket. This lookup is done for all ACK segments that did not match on an existing, established socket. If a request socket has been found, the calls are redirected to `tcp_check_req()`, which will create the new TCP socket structures, finalizing this by a call to `mptcp_check_req_child()` that finishes the initialization and link the new subflow to the Multipath TCP connection through a call to `mptcp_add_sock()`.

It must be said that this lookup in the hashtable for every incoming ACK is rather suboptimal. A malicious node could flood our stack with ACK segments and thus impose an increased CPU utilization. An optimization is possible to improve the performance of the lookup inside the hashtable. RCU lists [McK07] should be used instead of regular lists as they allow for a lock-less lookup, reducing the impact of taking a spin-lock at the server. A similar approach as for the SYN reception (explained above) might be difficult to achieve, because we cannot rely on the fact that an MP_JOIN option is included inside the third ACK. For example a RST segment still needs to be processed correctly, as it might be necessary to destroy the request socket.

5.3.3 Evaluation

The additional subflows require the computation of the HMAC in order to authenticate the hosts. Furthermore, the token is used to identify the Multipath TCP connection. What is the impact on the CPU resources and the security of the additional subflows?

When receiving a SYN with MP_JOIN, the server has to generate a random

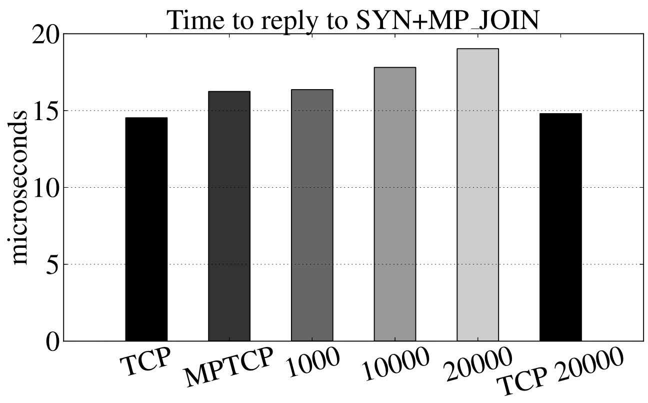


Figure 5.13: Looking up the token in the hashtables is a cost that increases with the number of preloaded Multipath TCP connections.

number and calculate an HMAC. As already shown in Figure 5.7, generating a random number takes a significant time. The HMAC calculation consumes even more CPU cycles. In order to compute one HMAC, an implementation must perform four SHA-1 computations on 512 bit blocks. As can be seen in Figure 5.12, calculating the HMAC accounts for 25% of the time spent to generate a SYN/ACK.

Further, the token included inside the SYN + MP_JOIN identifies the connection. The server must lookup the token hashtable to find the Multipath TCP connection that corresponds to the token to allow generating the HMAC included inside the SYN/ACK. As in Section 5.2.3 we measure the time it takes to generate a SYN/ACK with a different number of preloaded connections. We observe again an increase of the response-time, shown in Figure 5.13. One might think, that this increase is because upon receiving the MP_JOIN, the host must lookup the meta-socket in the token hashtable. However, in our specific setup, this lookup does not increase the response-time. This is because the connection is added to the top of the collision-list in the hashtable. Looking up this token can thus be done in $O(1)$. The increased response-time is rather due to the more complex code-path taken when receiving an MP_JOIN. To reduce the complexity inside the TCP stack we reuse some functions of the regular TCP stack (as has been shown in Figure 5.10). This codepath involves multiple lookups for established sockets with a five-tuple corresponding to the incoming SYN + MP_JOIN. As more connections are pre-established, this lookup takes more time and thus increases the response-time.

5.4 Data exchange

The previous sections explained how the subflows are setup and initialized so that the hosts are ready to send and receive data. This section sheds some light on the process of sending and receiving data.

5.4.1 Data transmission

The Linux Kernel uses a packet-based TCP implementation. An application sending data through one of the system calls (e.g., `send()`, `sendpage()`,...) provides to the kernel a buffer of continuous data to transmit. The kernel's TCP stack will split this data into individual segments, whose size is controlled by the MSS of the connection. These segments are created upon *skb*'s that are control structures containing pointers to the data of each segment and additional information necessary to allow linking the skb into a queue and store layer-specific information. This layer-specific information is for example (at the TCP layer) the sequence numbers associated to this segment. The TCP sequence numbers are stored in a structure called *tcp_skb_cb* (shown in Listing 5.2), part of the skb. A part from the sequence numbers, it also stores information about when the segment has been sent, if it has been acknowledged by a SACK-block and some IP-specific info inside *inet_skb_parm*. When transmitting data, the TCP stack will populate the `seq` and `end_seq` fields with the corresponding information and queue the skb inside the send-queue. Storing the sequence numbers allows for easy processing for example when receiving cumulative acknowledgments (to know whether the segment can be freed) and is used when writing the sequence numbers inside the TCP header before pushing the segment on the interface.

Multipath TCP has its layered architecture, where the subflows are separated from our meta-socket. An application transmitting data will push the bytes onto the meta socket, which splits the segments in MSS-sized segments[2]. Our implementation uses the same *tcp_skb_cb* structure to store the data sequence numbers and adds the skb's inside the send-queue of the meta socket. Upon reception of data acknowledgments, these skb's will be removed from the send-queue and their memory is freed up.

These segments, queued up in the meta send-queue, will be pushed on the individual subflows. The Multipath TCP scheduler, explained in Chapter 4, decides on which subflow to transmit the data. This transition from meta send-queue to subflow send-queue involves several steps, in the function `mptcp_write_xmit()`, shown in Figure 5.14. First, we check whether the segment fits inside the receive window advertised by the peer through a call to `tcp_snd_wnd_test()`.

[2]The MSS may be different among the subflows. We take the largest MSS among all subflows.

Listing 5.2: TCP control information part of the skb

```
struct tcp_skb_cb {
        union {
                struct inet_skb_parm    h4;
#if IS_ENABLED(CONFIG_IPV6)
                struct inet6_skb_parm   h6;
#endif
        } header;       /* For incoming frames          */
        __u32           seq;            /* Starting sequence number     */
        __u32           end_seq;        /* SEQ + FIN + SYN + datalen    */
        __u32           when;           /* used to compute rtt's        */
        __u8            tcp_flags;      /* TCP header flags. (tcp[13])  */

        __u8            sacked;         /* State flags for SACK/FACK.   */
#define TCPCB_SACKED_ACKED      0x01    /* SKB ACK'd by a SACK block    */
#define TCPCB_SACKED_RETRANS    0x02    /* SKB retransmitted            */
#define TCPCB_LOST              0x04    /* SKB is lost                  */
#define TCPCB_TAGBITS           0x07    /* All tag bits                 */
#define TCPCB_EVER_RETRANS      0x80    /* Ever retransmitted frame     */
#define TCPCB_RETRANS           (TCPCB_SACKED_RETRANS|TCPCB_EVER_RETRANS)

        __u8            ip_dsfield;     /* IPv4 tos or IPv6 dsfield     */
        /* 1 byte hole */
        __u32           ack_seq;        /* Sequence number ACK'd        */
};
```

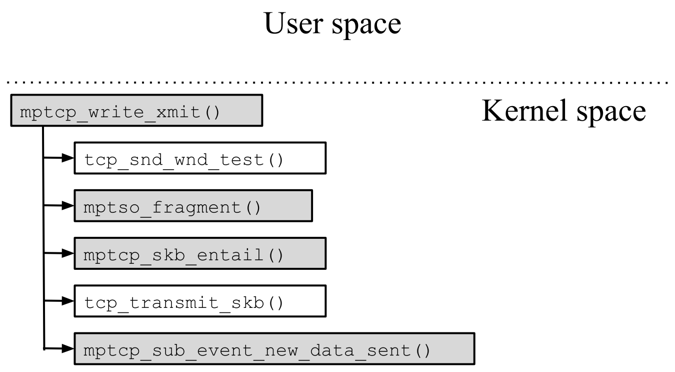


Figure 5.14: A simplified view of the different steps involved when transmitting data in Multipath TCP

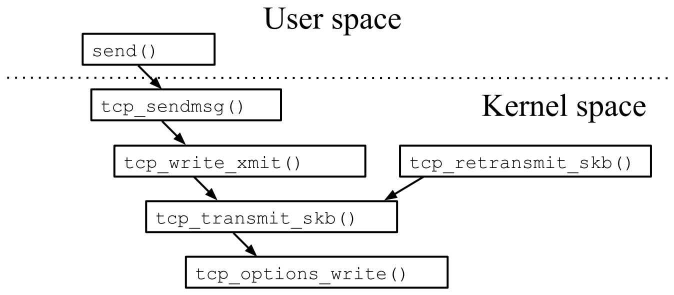


Figure 5.15: The TCP options are written in the header anew at every transmission.

As the segment in the meta send-queue may be larger than the MSS allowed on the chosen subflow (e.g., because we use the largest MSS among all subflows, or simply because the MTU of the interface changed in the middle of the connection) we fragment the segment to an appropriate size by a call to `mptso_fragment()`. The call to `mptcp_skb_entail()` does all the magic of queuing the segment on the subflow's send-queue. We cannot simply leave the segment on the meta send-queue as the subflow's retransmission timers need to ensure a coherent byte stream. The internal workings of `mptcp_skb_entail()` are explained in a follow-up paragraph as it is very specific to how the TCP stack handles the TCP options. Finally, `mptcp_write_xmit()` makes a call to `tcp_transmit_skb()` to push the segment one layer further, on the IP-layer where it will end up on the NIC interface. `mptcp_write_xmit()` makes sure that the retransmission timer of the subflow is set accordingly and that further state variables are correctly set within the subflow with a call to `mptcp_sub_event_new_data_sent()`.

When a segment is being pushed on a subflow, the sequence number space changes to the per-subflow sequence numbers. The values stored inside the *tcp_skb_cb* are thus being changed. However, we still need to write the data sequence number inside the TCP option space. To understand the complexity of this, let's look at how the regular TCP stack handles TCP options. Figure 5.15 shows the call graph of the data transmission. The segments are pushed on the interface by the call to `tcp_transmit_skb()`, which will also write the TCP options inside the TCP header. This means, that every time a segment is pushed on an interface, its TCP options are written anew inside the header. E.g., when a segment is sent due to the application's call to `send()`, its TCP options will be written for the first time inside the header. If it is going to be retransmitted due to a timeout (coming from `tcp_retransmit_skb()`) its options will again be

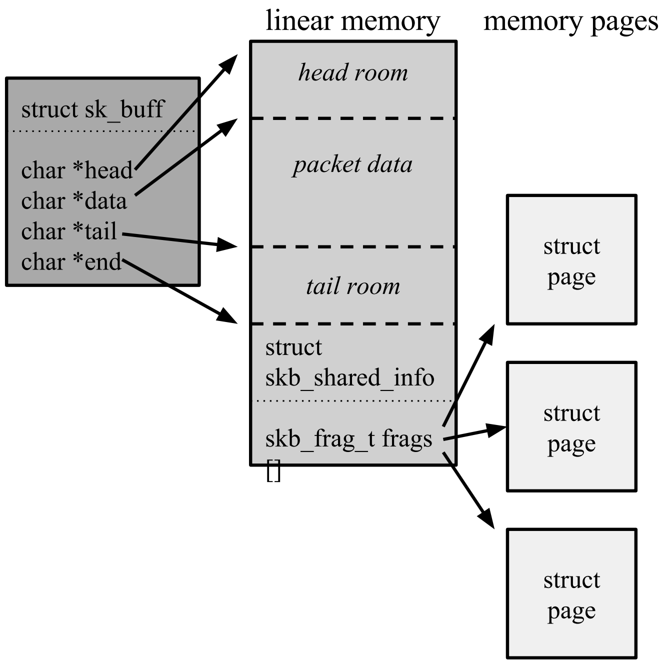


Figure 5.16: The skb is divided in different parts. The control structure (*sk_buff*), the linear data and the paged memory for the packet's payload.

written inside the header. For Multipath TCP this implies that the DSS option (although for a specific segment it does not change) would need to be written every time inside the TCP header. We thus would need to store as part of the skb the data sequence number. A solution might be to extend *tcp_skb_cb* with additional fields to store the required information. However, this would mean to increase the size of an skb - impacting the memory footprint of **all** networking protocols being used inside the stack. A modification, unacceptable for a generic Linux Kernel as it is critical for the performance to keep the size of an skb as small as possible.

Our implementation in Multipath TCP v0.88 handles this in the following way. We actually benefit from the fact that the DSS option does not change across multiple transmissions of the same segment. We immediately write the DSS option inside the TCP header space when moving a segment from the meta send-queue to the subflow send-queue. This seems like a straight-forward approach to handle the above mentioned problem. But - as usual - the devil is in the details and handling the DSS option in the described way introduced new problems that needed to be taken care of. To understand these problems (and the implemented solution), more details about the structure of an skb must be provided.

An skb is divided into different parts, shown in Figure 5.16. The control structure *sk_buff* maintains the different informations about the skb, like list-pointers, as well as *tcp_skb_cb* and pointers to the payload of the segment.There are several such pointers, all shown in Figure 5.16. When creating the skb upon a `send`-call, it is not yet known to the stack how much header space it should reserve for the TCP, IP and link-layer header. Thus, it reserves the maximum that might be used. This reserved space is the region between the `head` and `data` pointers. The stack starts filling the payload of the application starting from `data` until it reaches the `end`-pointer. Following the `end`-pointer is a reserved memory region that holds pointers to paged memory regions so that the payload can be put inside the memory pages[3].

When the TCP stack writes the TCP header (together with the TCP options) on top of the payload, it moves the `data` pointer upwards for the required number of bytes. As suggested above, Multipath TCP could write before the call to `tcp_options_write()` the DSS option on top of the payload. Multipath TCP would then need to move the `data` pointer upwards to still allow the TCP stack to write the remaining options on top of it. This seems feasible but many corner cases will then encounter problems and would need special handling. The `len` field of the skb would need to take the DSS option into account. However, this same field is also used to compare the length of the payload with the MSS. The DSS option will thus make the segment look larger than it actually is. We would need to check all accesses to `skb->len` and modify them to ignore (or not - depending on the use-case) the size of the DSS option.

We decided that it is easier to simply add the DSS option, without modifying the `data`-pointer nor the `len`-field of the skb. There is just one problem to handle in this case, and in terms of implementation complexity, it is much more localized and requires a smaller impact across the whole stack compared to the previous solution. The problem is that calls to `pskb_copy()`, in charge of copying an skb with its linear memory, will not copy the DSS option as they only copy from `data` to `tail`. This is easily solved, by handling the calls (only two of them inside the current TCP stack) to this function in a special way, forcing it to take into account the DSS option.

The above described problem of writing the DSS option inside the TCP option space has gone through multiple iterations over the history of the Linux Kernel implementation of Multipath TCP. We are far from the ideal solution and as of this writing another approach is being envisioned to reduce the complexity and overhead of the Multipath TCP implementation[4]. This latter solution seems to be the most clean one and is being adopted for the next version of Multipath TCP.

[3]This allows for efficient use of memory-mapping through the `splice()` system call.

[4]`https://listes-2.sipr.ucl.ac.be/sympa/arc/mptcp-dev/2014-06/msg00047.html`

5.4.2 Data reception

The receiving host processes the incoming segments on the individual subflows as if they were regular TCP. Segments received out-of-order, e.g. due to a packet loss, are appended to the out-of-order queue of the subflow. Cumulative acknowledgments are sent for the in-order segments, allowing the sender to advance the subflow sequence number space. As soon as in-order data is received, the subflow passes the control to the Multipath TCP layer to reconstruct the bytestream.

Passing the control to the Multipath TCP layer is done through the callback `sk_data_ready`. In regular TCP this callback is used to wakeup the application and signal that data is waiting in the receive-queue of the TCP socket. For Multipath TCP this callback has simply been redefined, allowing a clean layer separation between the TCP subflows and the Multipath TCP layer. The callback is redefined to the function `mptcp_data_ready()`, which is in charge of extracting the segments from the subflow's receive-queue and push it into the meta-socket's receive/out-of-order queue.

The Multipath TCP layer iterates over the in-order segments of the TCP subflow and as a first step does some preliminary checks on the segments in the function `mptcp_prevalidate_skb()`. As already mentioned in Section 2.3.3, if a host receives data at the beginning of the connection without a DSS option, it should fallback to regular TCP or destroy the subflow with a RST. The function `mptcp_prevalidate_skb()` is in charge of verifying this condition.

Next, the DSS mapping of the segments sitting at the head of the subflow's receive-queue is extracted in function `mptcp_detect_mapping()`. During the parsing of the options, we store inside *tcp_skb_cb* the offset of the position of the DSS option in the TCP header. This is because we do not have the space in *tcp_skb_cb* to store the full information of the mapping. Within `mptcp_detect_mapping()` the DSS option is read out of the TCP header. If the Multipath TCP connection is in infinite mapping-mode due to a fallback to regular TCP, the mapping is inferred from the next expected data sequence number[5]. Finally, we need to transform a potential 32 bit data sequence number in the DSS option to the 64 bit counterpart. This can be done as we know where we are currently situated in the 64 bit sequence number space. It may of course be the case that the stack receives a very old segment (more than 2^{32} bytes ago) and thus we might infer the incorrect 64 bit data sequence number. In this case, the checks on the DSS checksum will fail (as it covers the 64 bit sequence number) and no corruption of data will happen.

Before the data can finally be pushed to the meta socket, the stack must take

[5]In infinite mapping, Multipath TCP guarantees that the data sequence numbers are in-order on the subflow.

care of split and/or coalesced segments. A NIC on the sender side that supports TCP Segmentation Offload (TSO) will split segments, so that the DSS mapping effectively covers multiple smaller TCP segments. A subsequent segment coalescing middlebox might then coalesce again some of these. Creating segments where the mapping only covers half of the TCP segment, meaning that multiple mappings are associated to a single segment. These segments must again be split by the Multipath TCP stack, so that the data of each segment belongs to one single mapping. The function `mptcp_validate_mapping()` takes care of this splitting.

Finally, a mapping of the subflow's receive-queue head has been detected. This mapping may be spread across multiple segments, but each of these is split in such a way that they entirely belong to the same mapping. `mptcp_queue_skb()` is now in charge of pushing these segments to the meta-socket receive-queue or out-of-order queue. But first, the DSS checksum is validated by a call to `mptcp_verif_dss_csum()`. If this one is valid, the stack verifies that the data sequence number indeed falls within the range defined by the advertised receive-window. The processing finishes by moving the segments out of the subflow's receive-queue into the meta socket's out-of-order queue (in case of reordering) or to the receive-queue (in case of in-order reception).

To allow the stack to process fewer individual segments, continuous regions in the data sequence space of the receive-queue and the out-of-order queue are merged together in one single skb (an optimization already applied to the legacy TCP stack of the Linux kernel). This allows to process the queues with fewer iterations and reduces the memory overhead, as every additional skb consumes additional memory for the *sk_buff* control structure and the TCP/IP headers.

5.4.3 HTTP performance

From the latency results in Section 5.2.3, we can see that on a LAN, a Multipath TCP connection starts slightly behind the equivalent TCP connection. A small amount of bandwidth and CPU cycles are also consumed to establish additional subflows. HTTP is notorious for generating many short connections. How long does an HTTP connection using Multipath TCP need to be for these startup costs to be outweighed by Multipath TCP's ability to use extra paths?

We directly connected a client and server via two gigabit links. For our tests we use `apachebench` [6], a benchmarking software developed by the Apache foundation that allows us to simulate a large number of clients interacting with an HTTP server. We configured `apachebench` to emulate 100 clients that generate 100000 requests for files of different sizes on a server. The server was running

[6]http://httpd.apache.org/docs/2.0/programs/ab.html

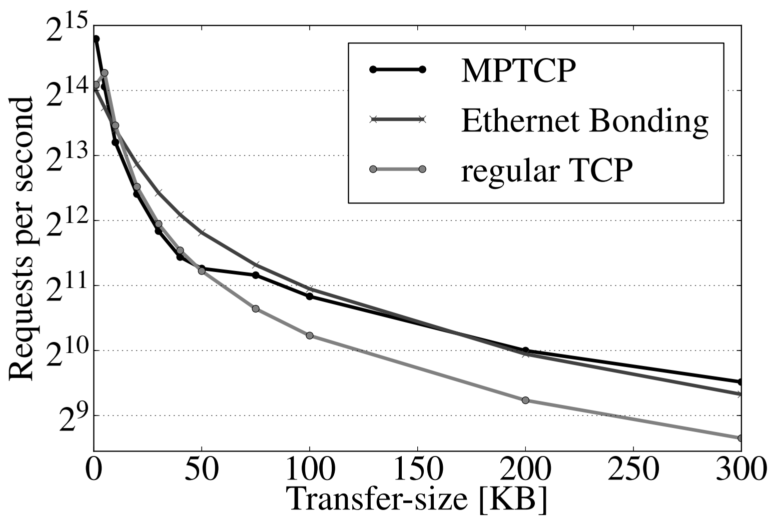


Figure 5.17: Apache-benchmark with 100 clients - As the file size grows, Multipath TCP's benefit becomes more apparent as it outweighs the cost of the CPU.

Multipath TCP Linux v0.86 and used `apache` version 2.2.16 with the default configuration.

We tested regular TCP that uses a single link, TCP with Ethernet bonding using both interfaces and finally Multipath TCP using one subflow per gigabit link. Ethernet bonding uses an Equal Cost Multipath (ECMP) approach to send flows across different interfaces in a round-robin manner. Ethernet bonding is only possible with a custom configuration of the end-host and the switch connected to this end-host. Figure 5.17 shows the number of requests per second served in all three configurations. We expect Multipath TCP to be significantly better than regular TCP, and indeed this shows up in experiments: when the file sizes are larger than 100 KB Multipath TCP doubles the number of requests served.

With files that are shorter than 30 KB, Multipath TCP decreases the performance compared to regular TCP. This is mainly due to the overhead of establishing and releasing a second subflow compared to the transmission time of a single file. These small flows take only a few RTTs and terminate while still in slow start.

TCP with link-bonding performs very well especially when file sizes are small: the round-robin technique used by the Linux implementation manages to spread the load evenly, utilizing all the available capacity. Multipath TCP has a slight advantage over TCP with link-bonding only when file sizes are greater than 150 KB in our experiment.

With larger files, there is a higher probability that link-bonding ends up congesting one of its two links, and some flows will be slower to finish. Flows on the faster link will finish quickly generating new requests, half of which will be allocated to the already congested link. This generates more congestion on an already congested link, with the effect that one link is highly congested while one is underutilized; the links will flip between the congested and underutilized states quasi randomly.

This evaluation shows that the transmission and reception of data through Multipath TCP can compete in terms of performance with regular TCP, although all over our implementation we needed to make choices that are suboptimal for Multipath TCP in order to comply with the restrictions of a generic operating system design.

5.5 Connection closure

Closing a Multipath TCP connection involves the handling of different state-machines. Each subflow has to go through its own TCP-specific state-machine, as well as the Multipath TCP layer together with the meta socket. The former is triggered through the exchange of FIN flags in the TCP header, while the latter is controlled through the DATA_FIN inside the DSS option. Due to this, no particular implementation needed to be done for the subflow's state transitions. For the state transitions of the meta socket, additional processing has been added to handle the reception and transmission of the DATA_FIN.

5.5.1 Avoiding time-wait state

Figure 5.18 shows a very particular issue within Multipath TCP. Many high-performance applications try to avoid Time-Wait state by deferring the closure of the connection until the peer has sent a FIN. That way, the client on the left of Figure 5.18 does a passive closure of the connection, transitioning from Close-Wait to Last-ACK and finally freeing the resources after reception of the ACK of the FIN. An application running on top of a Multipath TCP enabled Linux kernel might also use this approach. The difference here is that the `close()` of the connection (Step 1 in Figure 5.18) only triggers the sending of a DATA_FIN. Nothing guarantees that the kernel is ready to combine the DATA_FIN with a subflow-FIN[7]. The reception of the DATA_FIN will make the application trigger the closure of the Multipath TCP connection (step 2), trying to avoid Time-Wait state with this late closure. This time, the kernel might decide to combine the DATA_FIN with a subflow-FIN. This decision will be fatal, as the subflow's state machine will

[7]As specified in RFC6824, this combination can only be done if no other outstanding data is awaiting acknowledgment on other subflows.

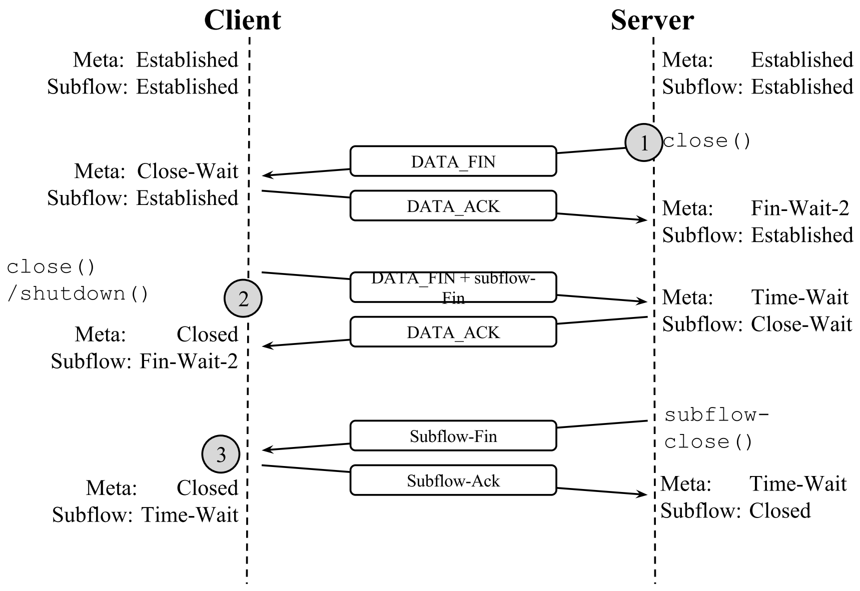


Figure 5.18: Although the application tries to avoid Time-Wait state by doing a "late" `close()` (step 2) of the connection, the subflow may still end up in Time-Wait state if the peer chose to not combine its FIN with the DATA_FIN (step 1) during its active closure.

not transition from Close-Wait to Last-Ack, but rather go through Fin-Wait-2 into Time-Wait state. The Time-Wait state will consume resources on the host for at least 2 MSL (Maximum Segment Lifetime). Thus, a smart application, that tries to avoid Time-Wait state by doing late closure of the connection actually ends up with one of its subflows in Time-Wait state. A high-performance Multipath TCP kernel implementation should honor the desire of the application to do passive closure of the connection and successfully avoid Time-Wait state - even on the subflows.

The solution to this problem lies in an optimistic assumption that a host doing active-closure of a Multipath TCP connection by sending a DATA_FIN will soon also send a FIN on all its subflows. Thus, the passive closer of the connection can simply wait for the peer to send exactly this FIN - enforcing passive closure even on the subflows. Of course, to avoid consuming resources indefinitely, a timer must limit the time our implementation waits for the FIN.

5.5.2 Resource freeing

The different structures that have been allocated during the creation of the Multipath TCP connection must be freed by the kernel during the connection closure. This operation can be split in two parts. The subflows and the meta socket.

Subflows are not visible to the application and must thus be closed by the kernel. The closure of a TCP socket is achieved through a call to `tcp_close()`. This function assumes to be running in user-context, meaning that the call usually comes directly from the application. However, the application does not have access to the subflows and thus cannot close them. Additionally, it is not up to the application to decide when to close a subflow. It's rather the kernel that has the knowledge of when a closure is appropriate (e.g., when a REMOVE_ADDR option has been received). To execute `tcp_close()` within user-context we create a work-queue when the closure of the subflow is scheduled.

All over the TCP stack, we have to access the meta socket to handle different parts of a Multipath TCP connection. This implies, that the memory of the meta socket cannot be freed until all TCP subflows have been closed. A simple way to achieve this is to "abuse" the reference counter of the meta socket. In legacy TCP, a reference counter is used on a socket which is being increased for example when a timer is launched to prevent the kernel from freeing the socket while the timer is still scheduled. Multipath TCP increases this reference counter for every subflow that is added to the connection. That way, as long as subflows remain, the memory of the meta socket will not be freed.

Additional structures have been created while establishing the Multipath TCP connection, shown in Figure 5.1. The *mptcp_tcp_sock* of each socket will be freed as soon as the corresponding *tcp_sock*'s memory is liberated. This is simply achieved through the callback `sk_destruct`, which has been redefined to `mptcp_sock_destruct()`, which is in charge of freeing the Multipath TCP specific structures.

5.6 Squeezing the performance

The TCP stack of the Linux Kernel has gone through many years of performance improvements. Sometimes, new APIs were offered to the applications to allow a more efficient use of the Kernel's stack, sometimes the hardware offered new capabilities that the stack could use to its benefits.

The following explains how we changed the Multipath TCP implementation to use three of these performance improvements to achieve up to 51.8 Gbps performance for a single Multipath TCP connection.

5.6.1 Hardware offloading

Handling many small segments in the TCP stack of the operating system imposes a significant cost in terms of CPU consumption. Every segment imposes an overhead on the OS. The more data is included inside a segment, the better the overhead/payload-ratio will be. TCP Segmentation Offload (TSO) tries to improve this ratio, by allowing the stack to create segments that are longer than one MSS. These large segments are then sent to the NIC, which is in charge of splitting them in MSS-sized segments. That way, the stack can process fewer individual segments for a large transmitted payload.

At the receiver-side the same overhead/payload-ratio should be minimized. Generic Receive Offload (GRO) allows the receiver to group consecutive incoming segments into one larger segment, before passing it to the TCP/IP stack. This can be done either by the NIC, or by the operating system. Segments, that are consecutive in the TCP sequence number space, and have the same TCP options can thus be merged together to form one single larger segment.

Multipath TCP can also benefit from TSO and GRO. At the sender side, using large segments allows spending less CPU cycles for a given amount of bytes. The TSO implementation splits the segments in smaller pieces, copying the DSS option on each segment. At the receiver side, the Multipath TCP stack merges these segments back together to reconstruct the mapping. A Multipath TCP sender, supporting TSO also allows the use of GRO at the receiver because the DSS option for the split segments will be the same - allowing GRO to merge the small segments in a larger one.

5.6.2 Flow-to-core affinity

Recent years have shown a trend in the architecture of today's microprocessor. The performance increase nowadays does not come anymore from an increase in the clock speed, but rather by increasing the number of cores of the CPU and thus increasing the parallelism of the execution. This parallelism needs to be correctly handled by the applications running on this system. The TCP stack of the kernel is also influenced by the increasing number of CPU cores on the end-hosts.

Incoming packets of a TCP connection could get spread across the CPU cores to increase the parallelism of the packet-processing. Although the packet handling inside the TCP-stack cannot be parallelized (as the shared memory - the socket - must be protected from concurrent accesses), the processing prior to the TCP stack (e.g., IP-layer and the other lower layers) could benefit from the parallelism. However, the gain of this parallelism does not outweigh the cost. The cost consists in the increased number of cache misses and lock contention when the segments are being aggregated in the TCP layer. Lock contention and cache misses are the biggest performance bottleneck in modern architectures as the CPU speed is considerably higher than the memory speed [JNW10].

The TCP stack and the NICs handle this by enforcing flow-to-core affinity. Indeed, a NIC will send the hardware-interrupt upon packet-reception always on the same CPU core for all segments belonging to the same 5-tuple. Further, the operating system will then reschedule this hardware-interrupt in a soft-interrupt to process the incoming packet. An extension to the operating system allows to schedule this software interrupt on the same CPU core as the application is running. This process, called Receive-Flow-Steering (RFS) [Edg10] requires the kernel to track on which CPU the application is running. This is achieved by a simple function-call upon the `send()`/`recv()`/... system calls. This call registers the 5-tuple of TCP connections in a hashtable such that the soft-interrupt of incoming segments can be scheduled on the appropriate CPU.

With Multipath TCP multiple TCP subflows are being used to transmit and receive the data. These segments might be handled by a different CPU, as neither the NIC nor the RFS-implementation is aware of Multipath TCP. As the subflows are being sent on the different CPU cores, a considerable amount of cache misses and lock contention will limit Multipath TCP's performance. An ideal implementation would send the hardware interrupts as well as the soft-interrupts each on the same CPU core to reduce the amount of cache misses and lock contention. Scheduling the hardware interrupts of all TCP subflows belonging to a Multipath TCP connection on the same CPU core would require that the NIC understands Multipath TCP and recognizes which subflow belongs to which Multipath TCP connection. We do not have the capabilities to modify the hardware of the NIC to take Multipath TCP into account. However, with RFS - an in-kernel implementation - we are able to improve the situation.

As previously mentioned, RFS tracks on which CPU core the application is running and registers the 5-tuple of the corresponding connection in a hashtable to schedule the soft-interrupt on the specified CPU core. Extending RFS to support Multipath TCP is relatively easy. In fact, RFS must simply register the 5-tuples of all the TCP subflows belonging to the same Multipath TCP connection in the hashtable in such a way that the segments of all these subflows will be scheduled on the same CPU core.

5.6.3 Zero-copy support

An application transmitting data puts this data inside a memory region, and passes this pointer via the `send()`-system call to the kernel. The kernel needs to copy this memory region from the user-space to the kernel-space. This is because after the `send()` system call, the user-space may overwrite the memory region, although the segment has not even left the host. This copying from user-space to kernel-space consumes a significant amount of CPU cycles, as many cache-misses may happen during this operation.

To avoid this memory copying process, zero-copy support has been added to the kernel. System-calls like `sendfile()` and `splice()` use this feature. It takes benefit of the use-case where the user-space does not even process the data it is transmitting from one file-descriptor to another. This may for example be the case when copying two files. The user actually does not care about the content of the file. The same principle applies to web- or file-servers. They use the `sendfile()` system call as the content of a static webpage does not need to be parsed by the user space.

The system calls end up in the kernel, where the function `do_tcp_sendpages()` is simply moving memory pages from one file-descriptor to another. On the Multipath TCP side of the file descriptors, skb's are being created. As the memory-pages may be scattered around the entire memory, scatter/gather must be supported by the NIC to allow using zero-copy support. The TCP-stack verifies the NIC's scatter/gather capabilities before using `sendfile()`. In case the NIC does not support it, it falls back to regular behavior, copying the data from the pages to the kernel memory.

For Multipath TCP we must make sure that scatter/gather is supported on all NICs that are used among the subflows belonging to this Multipath TCP connection. If there is a NIC that does not support it, the Multipath TCP connection cannot benefit from zero-copy support.

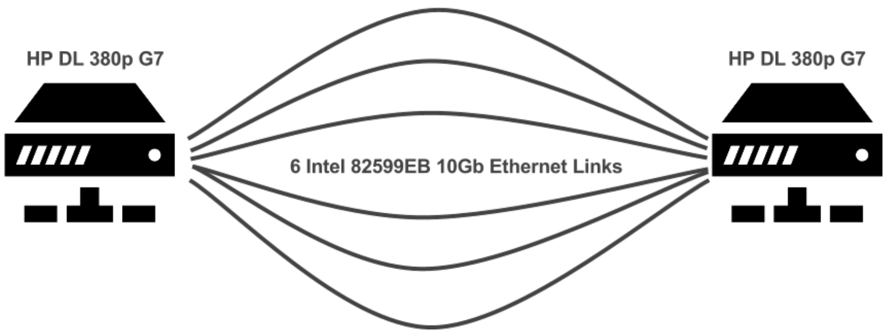


Figure 5.19: We interconnect our servers with 6 10 Gig interfaces back-to-back.

5.6.4 Pushing Multipath TCP to the limits

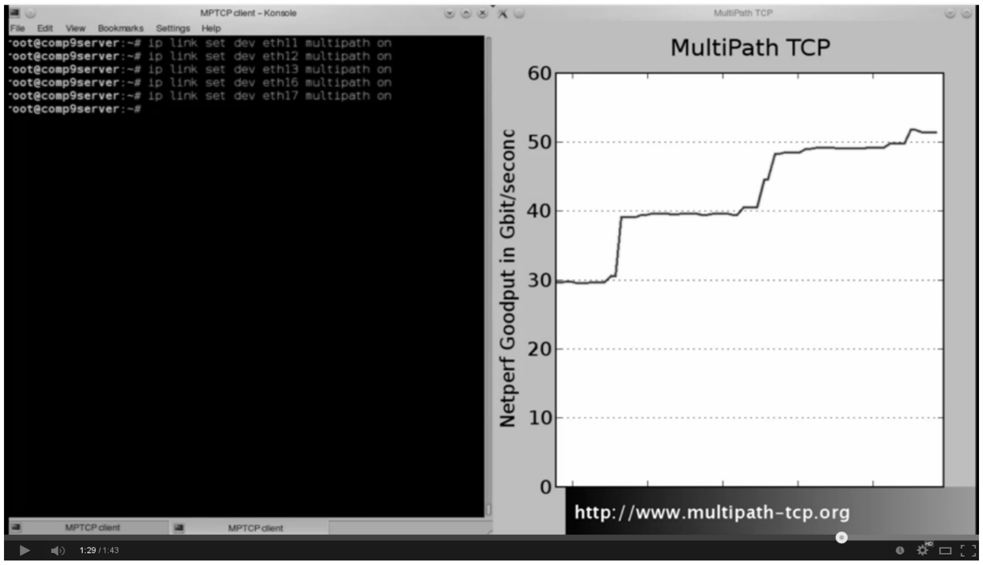
With the Linux kernel implementation of Multipath TCP moving out of its "prototype" status over the last years, we are curious what performance this implementation actually can achieve when being offered high-end servers and enough capacity between these servers. For this trial we use HP DL 380p G7 servers, each equipped with 3 dual-port 10Gig NICs. The machines were interconnected back-to-back, as shown in Figure 5.19.

We used a slightly customized implementation of the Multipath TCP Linux Kernel. We modified the kernel to only create one single subflow for each interface[8]. Apart from this minor modification, the kernel is a standard Multipath TCP implementation. It includes all hardware offloading extensions, flow-to-core affinity as well as zero-copy support, as described earlier in this section. Further, we optimized the kernel configuration for the specific hardware and disabled unwanted features from the stack (like `iptables`).

We also optimized the run-time configuration of the system. We hand-tuned the interrupt coalescing on the server to improve its segment coalescing when performing generic receive offloading. Increasing the interrupt-coalescing allows for more segments to be received by the NIC, before triggering the hardware interrupt. This increases the chances of having consecutive segments that are candidates for the generic receive coalescing feature as described in Section 5.6.1

We also pinned the application to a specific CPU, to prevent the OS's scheduler from moving the application from one CPU core to another, introducing cache misses. Further, we instructed the NIC to send all its hardware interrupts for incoming segments to the same CPU core (different than the one where the application is running). This allows to make the pre-processing of the segments (like GRO) happen on another CPU, freeing CPU cycles for the Multipath TCP stack

[8]Multipath TCP by default tries to establish a full mesh across all available interfaces - not required for our purposes


MPTCP client – Konsole
File Edit View Bookmarks Settings Help
MultiPath TCP
MultiPath TCP
Netperf Goodput in Gbit/seconc
60
50
40
30
20
10
0
http://www.multipath-tcp.org
MPTCP client
MPTCP client
1:29 / 1:43

```
oot@comp9server:~# ip link set dev eth11 multipath on
oot@comp9server:~# ip link set dev eth12 multipath on
oot@comp9server:~# ip link set dev eth13 multipath on
oot@comp9server:~# ip link set dev eth16 multipath on
oot@comp9server:~# ip link set dev eth17 multipath on
oot@comp9server:~#
```

Figure 5.20: Our Linux kernel implementation of Multipath TCP is able to scale up to 51.8 Gbps for a single Multipath TCP connection. `https://www.youtube.com/watch?v=VMdPI9Cfi9k`

and the application. Finally, we significantly increased the memory limitations of the socket-configuration to avoid being limited by the receive window.

All these tunings allowed us to scale the Multipath TCP kernel up to a goodput of 51.8 Gbps for a single Multipath TCP connection, transmitting data across these two machines, using 6 10 Gig interfaces. A video (screenshot can be seen in Figure 5.20), available at `https://www.youtube.com/watch?v=VMdPI9Cfi9k`, shows how Multipath TCP behaves when each of these 10 Gig interfaces is added one after the other to the connection.

5.7 Conclusion

This chapter has shown the high-level design of the Linux kernel implementation of Multipath TCP. It has been shown why certain choices have been made and what their underlying motivation is. Many of these decisions have been made in order to reduce the performance overhead for regular TCP. This kind of design decision would not be necessary in a "Multipath TCP"-only world. However, we are still far away (and may never reach this state) from such a world and thus must rather optimize for the common-case which nowadays is still regular TCP, with Multipath TCP being the exception.

Nevertheless, it has been shown that our design - even when respecting all these restrictions - is still able to achieve a high performance of up to 51.8 Gbps. It remains to be seen how the implementation will evolve in the future and whether Multipath TCP will be accepted in an upstream submission to the standard Linux kernel.

Chapter 6

Retrospecting on Multipath TCP

The previous chapters have discussed and proposed improvements to different parts of Multipath TCP. The specification has been explained in detail in Chapter 2. The *Experimental Design* approach allows to evaluate Multipath TCP and has shown that our reactive approach to receive-window limitations brings good performance in heterogeneous environments. Chapter 5 explains the implementation of Multipath TCP and how the code has been designed to be scalable. Multipath TCP is being commercially deployed [Bon13] and many researchers are using and improving it. This shows that Multipath TCP, as it is specified today, works reasonably well for most use-cases.

However, one might still ask how could we design Multipath TCP today, taking into account the lessons learned during the last four years? More specifically, we address the following questions. Does the specification allow for sufficient flexibility to accommodate new use-cases in the future? Does the security elements of Multipath TCP satisfy the requirements other use-cases might demand?

This chapter takes a step back and looks at how one might design Multipath TCP by taking into account the experience gained during the last years.

6.1 Establishing connections

The initial handshake of a Multipath TCP connection serves different goals, as described in Section 2.1.1. Among these are the exchange of the tokens, the computation of the IDSN and the derivation of the keys for subflow authentication. We have shown in Chapter 5 that both, the key generation and the verification for its uniqueness, add a cost in terms of performance.

In this section we consider how Multipath TCP could be redesigned if we ignore for a moment the security requirements. This means that during the initial handshake, no keys need to be negotiated. The security aspect of Multipath TCP will then be considered in the next section.

We suggest a *low-overhead Multipath TCP* whose goal is to reduce the overhead in terms of CPU cycles during the handshake of the initial subflow as well as for the additional subflows. We achieve this by removing the security from Multipath TCP. The target environment for this flavor of Multipath TCP might be closed environments like data centers, where one can be sure that no attackers are present. Another use-case would be where the application-layer protocol already provides sufficient security such that hijacking-attacks of the Multipath TCP connection are not possible. Transport Layer Security (TLS) provides such kind of protection against session hijacking.

The following details how our proposed low-overhead handshake allows to establish the initial as well as the subsequent subflows.

6.1.1 Initial Handshake

The information that needs to be negotiated during the initial handshake are the token and the IDSN. The negotiation must take place during the establishment of the initial subflow, inside the MP_CAPABLE options. In order to differentiate our proposed low-overhead version from Multipath TCP as defined in RFC 6824, the version field of the MP_CAPABLE is set to 1 during the handshake.

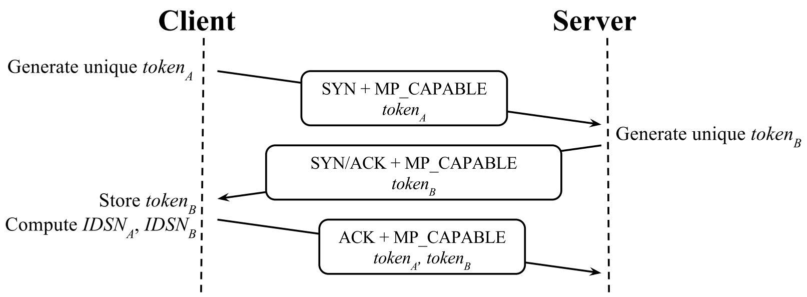
In order to negotiate the tokens during the handshake, each host generates a locally unique token (Figure 6.1(a)). The tokens are echoed back in the third ACK of the handshake to support stateless servers [Edd06].

The generation of an IDSN can be done thanks to the concatenation of the tokens. The IDSN for packets going from the client to the server ($IDSN_A$) is given by concatenating $token_B$ and $token_A$. The reverse approach is used for the packets sent by the server ($IDSN_B$).:

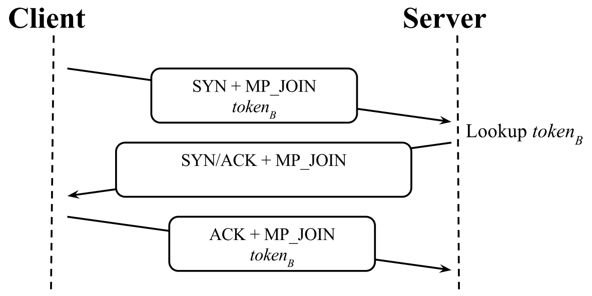
$$IDSN_A = token_B || token_A$$

$$IDSN_B = token_A || token_B$$

Using a concatenation of the tokens weakens the security of Multipath TCP against data-injection attacks compared to the hash of the keys used by RFC 6824. The hash of the keys is more secure, because the keys will never be exchanged in clear-text after the 3-way handshake of the initial subflow. In low-overhead Multipath TCP, the tokens are revealed during the establishment of additional subflows, giving the attacker an indication of the initial data-sequence number. However, as we consider the high-performance version of Multipath TCP to be used in secure environments (e.g., data centers), this tradeoff should be acceptable.


Client
Server
Generate unique $token_A$
SYN + MP_CAPABLE $token_A$
Generate unique $token_B$
SYN/ACK + MP_CAPABLE $token_B$
Store $token_B$
Compute $IDSN_A$, $IDSN_B$
ACK + MP_CAPABLE $token_A$, $token_B$

(a) Initial subflow establishment.



(b) Additional subflows establishment.

Figure 6.1: Low-overhead Multipath TCP: A very simple mechanism to start a Multipath TCP connection and create new subflows.

6.1.2 Additional subflows

As we do not intend to protect against hijacking attacks, the authentication of new subflows is not necessary. Our proposal only needs a way to identify the Multipath TCP connection to which the subflow must be linked.

To establish a new subflow, the client sends the SYN together with the MP_JOIN option (Figure 6.1(b)) that contains the token of the server ($token_B$). The token allows the server to identify the connection associated to this new subflow – similar to standard Multipath TCP. Including the token in the third ACK allows to handle the MP_JOIN in a stateless manner (see below).

Compared to the analysis of Section 5.2.3 it is obvious that the most costly parts of the handshake for additional subflows have been removed. There is no need to generate a random number or compute an HMAC. Furthermore, the MP_JOIN option consumes fewer bytes in the TCP header – beneficial for future extensions to TCP.

6.1.3 Token generation

In Section 5.2.3 we identified the issue of the token generation and the possibility of token collisions. We mitigate this in our proposal, as shown hereafter.

The token can be the result of applying a 32-bit block cipher (e.g., RC5 [Riv95]) on a counter together with a local secret: $token = RC5(counter \otimes secret)$. Although, we may have token-collisions (wrap-around of the counter), a collision is very rare as the old token has to survive 2^{32} other connections. Additionally, the output looks random (due to RC5) and is hard to guess for an attacker, thanks to the local secret.

6.1.4 Stateless server

Handling the establishment of additional subflows in a stateless manner is beneficial as an attacker cannot consume memory resources on the server by sending out SYN segments with the MP_JOIN option. However, allowing stateless handling of new subflows implies that the third ACK has to carry all the information to allow the server to identify the Multipath TCP connection, and that this third ACK is indeed a reply to a SYN/ACK issued by the server.

To support stateless handling of SYN's with MP_JOIN, the server has to perform the following upon the reception of a SYN:

- Check if there exists a Multipath TCP connection corresponding to the token inside the MP_JOIN.
- Reply with a SYN/ACK

When receiving the third ACK (sent reliably as in today's standardized Multipath TCP [FRHB13]), the server verifies that indeed it has generated a SYN/ACK (like regular TCP's SYN-cookie mechanism [Edd06]). The third ACK also contains an MP_JOIN with the token, allowing the server to identify the Multipath TCP connection this new subflow is joining.

6.1.5 Evaluation

We implemented the proposed low-overhead Multipath TCP on top of our Linux Kernel implementation [RPB+12]. Our evaluation is twofold. First, we measure the time taken to generate a SYN/ACK, and analyze where the performance gain comes from. Second, we evaluate the benefits of our proposal on the number of HTTP-requests a server can handle.

Our testbeds contain two types of machines. The first testbed uses two Xeon-class servers, interconnected over 10Gig Ethernet interfaces. These machines use

2 dual-core Intel Xeon 2.53Ghz processors and 8GB of RAM. The second testbed uses a *Raspberry Pi* as the server [1]. A *Raspberry Pi* is a low-end machine with a 100Mb Ethernet interface and an ARM (700 BogoMIPS) processor.

Generating a SYN/ACK

It is important to distinguish two cases when evaluating the time to generate a SYN/ACK. Indeed, as described in Section 2.1.1, the establishment of the initial subflow (MP_CAPABLE) involves different steps and computations than the establishment of additional subflows (MP_JOIN).

To evaluate the cost of our solution, we establish Multipath TCP connections and measure the time a server takes to reply to a SYN/ACK. To achieve best performance, this time must be small. The measurement setup and procedure is similar to the one used in Section 5.2.3.

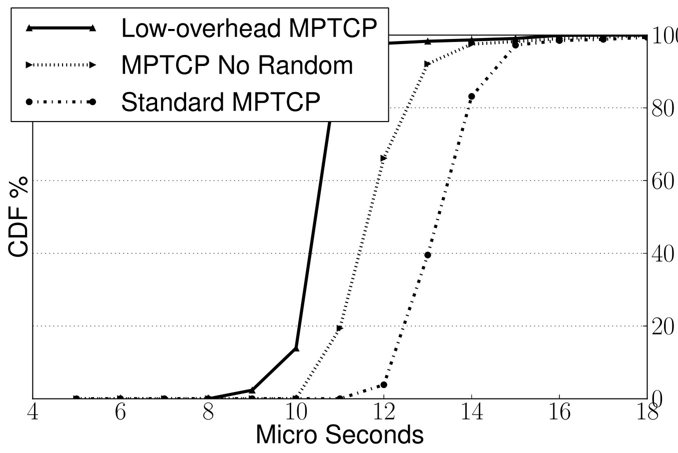
MP_CAPABLE Figure 6.2 shows the time taken to reply to a SYN for the initial subflow. The Xeon-class servers (Figure 6.2(a)) take between 12 and 14 µs with standard Multipath TCP. Our proposal to use a low-overhead handshake improves this by 2 to 3 µs. On a *Raspberry Pi* (Figure 6.2(b)), this improvement is even more apparent, as 80% of the SYN's experience a gain of 40 µs. Overall, we see a 20% reduction of the time needed to generate a SYN/ACK regardless of the server's hardware.

This improvement comes from the time it takes to generate a random 64-bit key. Indeed, today's standard Multipath TCP performs two computations to reply to a SYN with the MP_CAPABLE option: *(i)* Generate a random 64-bit key; *(ii)* Compute a SHA-1 on this key in order to verify the token's uniqueness. We observe that generating the 64-bit random key accounts for half of the gain. Replacing the random key generation by a simpler hash (e.g. using MD5) on the 5-tuple and a local secret greatly improves the performance. *MPTCP No Random* in Figure 6.2 shows the improvement of standard Multipath TCP by using such a more efficient random number generator.

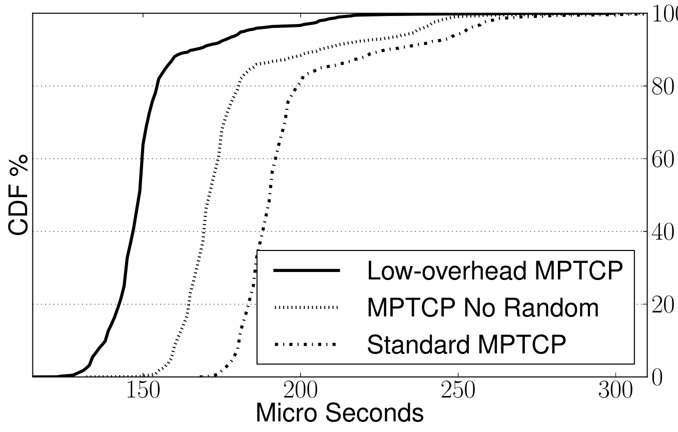
To improve the overall performance, we therefore recommend to replace all random number generations in Multipath TCP with a hash of the 5-tuple and a local secret. This complies with current recommendations for generating random numbers provided by RFC 4086 [ESC05].

MP_JOIN Establishing an additional subflow involves more complex computation than for the initial subflow such as an HMAC calculation and the generation of a random number. *MPTCP No Random* from Figure 6.3 depicts the effect

[1]http://www.raspberrypi.org/

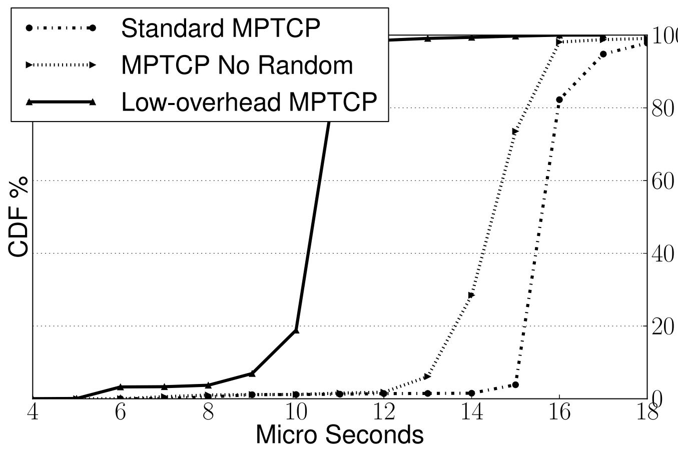


(a) Xeon Class Server

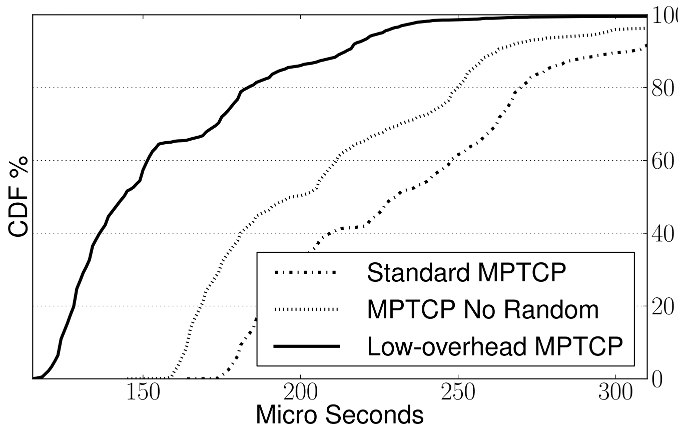


(b) Raspberry Pi

Figure 6.2: Our implementation of low-overhead Multipath TCP decreases the time to generate a SYN/ACK with MP_CAPABLE by more than 20%.



(a) Xeon Class Server



(b) Raspberry Pi

Figure 6.3: The cost of the HMAC becomes apparent when measuring the time to reply to a SYN with MP_JOIN.

of replacing the random number generation of standard Multipath TCP by the above proposed hash computation. The main difference between standard Multipath TCP and the low-overhead proposal is the HMAC computation. The difference between *MPTCP No Random* and low-overhead Multipath TCP in Figure 6.3 shows the impact of generating an HMAC on the time taken to generate a SYN/ACK. Overall, the gain is around 33% for the Xeon-class server as well as on the Raspberry-Pi.

Impact on real traffic

We also measured the performance of network-heavy applications with respect to each solution. Indeed, changing the Multipath TCP handshake also impacts the application's performance as it influences the time taken to generate the SYN, receiving the SYN/ACK and sending the third ACK, and finally verifying the HMAC in the third ACK.

We use `apachebenchmark`[2], a benchmarking software developed by the Apache foundation, that allows us to simulate a large number of HTTP-clients sending requests to a server. We measure the number of HTTP-requests the server is able to handle. We use an HTTP reply of 1KB in order to identify the impact of the Multipath TCP handshakes.

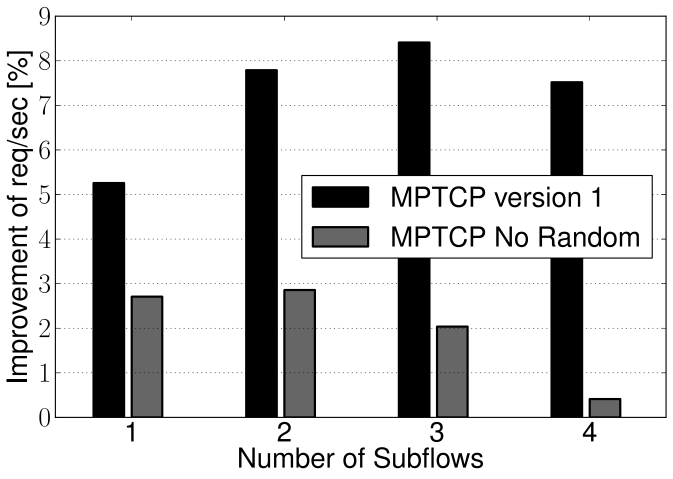
We simulated 250 clients with `apachebenchmark`, talking to the Xeon-class server over a 10Gig interface. We instructed Multipath TCP to create between 1 and 4 subflows. The results are presented in Figure 6.4. Our proposed Multipath TCP variant improves the performance by up to 10% compared to standard Multipath TCP. Replacing the random number generation by the previously described hash generation improves the performance if the number of subflows is not too high. Indeed, when too many subflows are being created, the generation of the HMAC becomes the bottleneck in today's Multipath TCP.

We have performed a similar evaluation by replacing the Xeon-class HTTP server with a Raspberry Pi. We down-scaled the number of parallel clients to 10 as the Raspberry Pi is not able to handle a large number of parallel HTTP requests. The Raspberry Pi shows similar results as on the Xeon-class server, with an improvement of up to 8% in terms of number of requests per second it can serve.

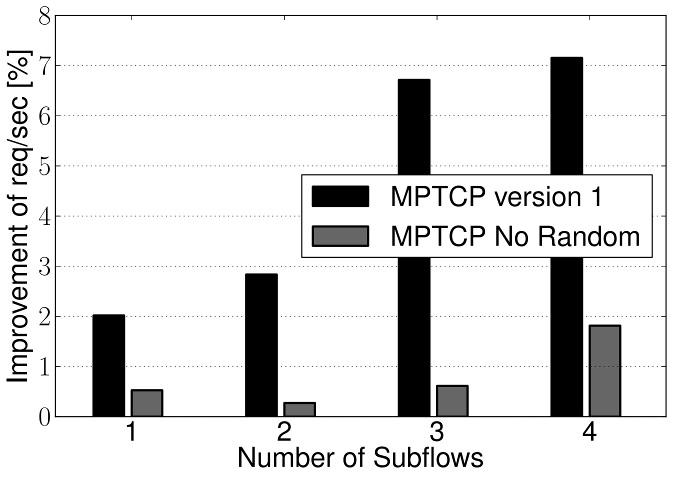
6.2 Securing Multipath TCP

The previous section explored how we could redesign the handshake of the initial subflow in order to reduce the computational overhead that comes with the

[2] http://httpd.apache.org/docs/2.2/programs/ab.html



(a) Xeon Class Server



(b) Raspberry Pi

Figure 6.4: The low-overhead variant of Multipath TCP increases the number of requests per second by up to 10% on Xeon-class servers. A similar improvement can be observed on a Raspberry Pi.

key generation. One motivation for this might be that upper layer protocols like TLS already provide protection against hijacking/traffic injection attacks.

However, even Multipath TCP as specified in RFC 6824 [FRHB13] has some weaknesses with respect to security and its vulnerability with respect to Denial-of-Service attacks.

6.2.1 Residual threats in Multipath TCP

The security of Multipath TCP is based on the clear-text exchange of a pair of keys during the handshake of the initial subflow. Further, the exchange of HMACs allows the end-hosts to authenticate each other during the establishment of additional subflows. The following analyses some of the residual threats that are present in the specification of Multipath TCP [BPG+14].

Eavesdropper in the initial handshake

An attacker, which observed the 3-way handshake of the initial subflow, can move away from the initial path and establish a subflow with the original end-hosts. It can authenticate itself since it has knowledge of the keys - observed during the 3-way handshake of the initial subflow. This subflow then allows the attacker to fully hijack the connection. This vulnerability was readily identified at the moment of the design of the Multipath TCP security solution and the threat was considered acceptable.

DoS attack on MP_JOIN

Upon reception of a SYN + MP_JOIN message of the 3-way handshake for additional subflows, the server needs to create state. This, because the token is used to identify the Multipath TCP connection this new subflow belongs to. As this token is not resent in the third ACK of the handshake, the server must remember that the 5-tuple of the SYN belongs to the specified Multipath TCP connection.

Assume that there exists a Multipath TCP connection between host A and host B, with tokens T_A and T_B. An attacker, sending a SYN + MP_JOIN to host B, with the valid token T_B, will trigger the creation of state on host B. The number of these half-open connections that a host can store per Multipath TCP connection is limited. This limitation is implementation dependent. The attacker can simply exhaust this limit by sending multiple SYN + MP_JOIN with different 5-tuples. The (possibly forged) source address of the attack packets will typically correspond to an address that is not in use, or else the SYN/ACK sent by Host B would elicit a RST from the impersonated node, thus removing the corresponding state at Host B.

This effectively prevents host A from sending any more SYN + MP_JOIN to host B, as the number of acceptable half-open connections per Multipath TCP connection on host B has been exhausted.

The attacker needs to know the token T_B in order to perform the described attack. This can be achieved if it is a partial on-time eavesdropper, observing the 3-way handshake of the establishment of an additional subflow between host A and host B. If the attacker is never on-path, it has to guess the 32-bit token.

SYN flooding amplification

SYN flooding attacks [Wes07] use SYN messages to exhaust the server's resources and prevent new TCP connections. A common mitigation is the use of SYN cookies [Wes07] that allow the stateless processing of the initial SYN message.

With Multipath TCP, the initial SYN can be processed in a stateless fashion using the aforementioned SYN cookies. However, as we described in the previous section, as currently specified, the SYN + MP_JOIN messages are not processed in a stateless manner. This opens a new attack vector. The attacker can now open a Multipath TCP connection by sending a regular SYN and creating the associated state but then sending as many SYN + MP_JOIN messages as supported by the server with different source address and source port combinations, consuming server's resources without having to create state in the attacker. This is an amplification attack, where the cost on the attacker side is only the cost of the state associated with the initial SYN while the cost on the server side is the state for the initial SYN plus all the state associated to all the following SYN + MP_JOIN.

6.2.2 Leveraging user-space security

It has been shown that an eavesdropping attacker can easily hijack the Multipath TCP connection as the keys are sent in clear during the 3-way handshake of the initial subflow.

However, some application-layer protocols like TLS or SSH already negotiate a shared key between the end-points. In this section we reconsider the design-decision to negotiate keys within Multipath TCP. We do this by setting us in the context of an application-layer protocol that already negotiates a secret between the end-points. Our solution then relies on this secret to authenticate the new subflows to each other.

The following explains how we transform Multipath TCP to achieve the above goal.

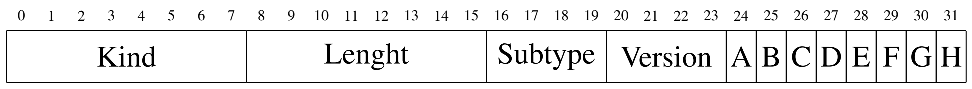


Figure 6.5: Header of the MP_CAPABLE option

Connection initiation

The handshake of the initial subflow is a small variation to the handshake of RFC 6824 and also integrates nicely with the handshake proposed in Section 6.1. The header of the MP_CAPABLE option of these two Multipath TCP-versions has the format, shown in Figure 6.5.

We propose to use the B bit in this option to indicate whether the host that sent the MP_CAPABLE option will use an application supplied key to authenticate the additional subflows or not. When the B bit is set, it indicates that the authentication key is supplied by the application. If the B bit has not been set in both directions, the authentication mechanism is used as defined by the Multipath TCP version ([FRHB13] or [PB12]).

In Multipath TCP version 0, even if the B bit is set, the end-hosts still have to generate a key that fulfills the requirements as defined in Multipath TCP version 0. This is necessary to handle the case where the client supports the B bit, but the server not yet. A more in-depth analysis of the deployment scenario is provided below.

By using the same handshake as in Section 6.1, the proposed handshake can also benefit from the lower overhead for generating the token and thus the faster establishment of the initial subflow.

Starting a new subflow

The handshake for the establishment of a new subflow is similar to the one specified in [FRHB13]. There are two important differences. First, the HMAC is computed by using the keys provided by the application. Second, the token and the client's random number are included inside the third ACK to allow stateless operation of the passive opener of an additional subflow.

In order to allow the $token_B$ and R_A inside the third ACK, the $HMAC_A$ must also be a truncated version of the 160-bit HMAC-SHA1. Thus, $HMAC_A$ is the truncated (leftmost 128 bits) of the HMAC as shown in Figure 6.6.

As the third ACK includes the token and the random nonce, the MP_JOIN message format of the third ACK is as show in Figure 6.7. The length of the MP_JOIN option in the third ACK is 28 bytes. There remains enough space to insert the timestamp option in the third ACK.

The semantics of the backup-bit "B" and the Address ID are the same as

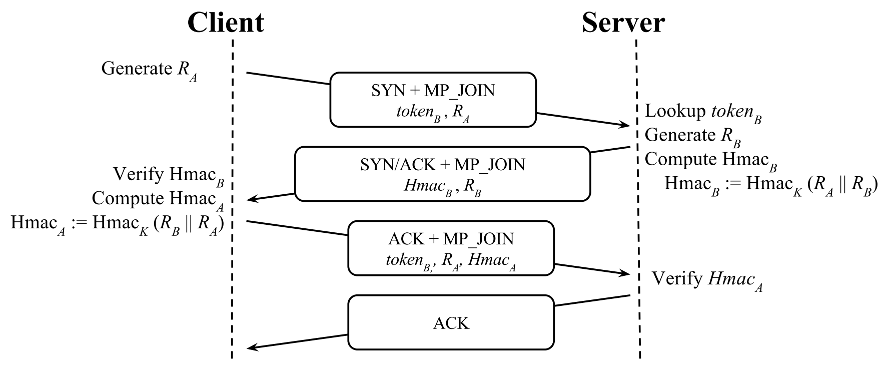


Figure 6.6: Handshake of a new subflow.

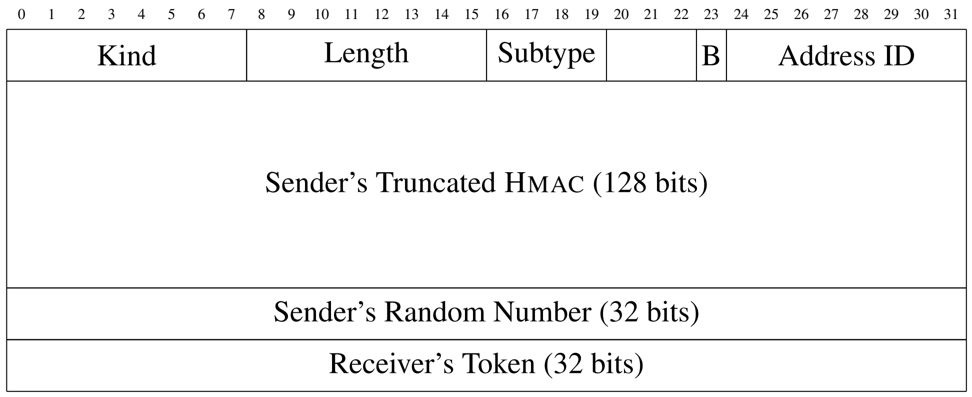


Figure 6.7: Format of the MP_JOIN option

in [FRHB13].

Multipath TCP API

The proposed mechanism requires an interaction between the application and the Multipath TCP layer. This can be achieved by the means of socket options. We define two socket options:

MPTCP_ENABLE_APP_KEY : This socket option tells the socket layer that an application supplied key will be used to secure the establishment of additional subflows. This socket option must be used before establishing the initial subflow, or before starting to listen on a socket to accept new connections. When this socket option is used, the MP_CAPABLE option is sent with the "B"-bit set to 1.

MPTCP_KEY : This socket option allows the application to provide a key to the Multipath TCP layer. Both end-points must use this socket option in order to allow the Multipath TCP-layer to create new subflows. It is up to the application to negotiate the key between the end-points. E.g., in the case of TLS, a derivate of the master secret can be used, by following the key derivation recommended in RFC 5705 [Res10].

Evaluation

Leveraging the user-space keys to the Multipath TCP layer may have implications in terms of performance as well bring deployment problems. This section sheds some light on these issues.

Deployment This proposed mechanism assumes that the application uses new socket-options to provide the key to the Multipath TCP-layer. Thus, the first requirement for deploying this Multipath TCP handshake is that the TLS-layer has been modified. There may of course be scenarios, where the client is supporting the proposed solution, but the server not. Thus, the client sends out the MP_CAPABLE with the B bit set, but the server replies without enabling the B bit. Upon reception of the SYN/ACK, it is up to the client's policy to react. It can either continue with the negotiated version of Multipath TCP but without using the key from the application or fallback to regular TCP.

The applications will have to pass the shared key to the Multipath TCP-layer by the means of a socket-option. It may be that the client's application has already done the call to the socket-option but not the server's application. The server will receive a SYN with the MP_JOIN option, without knowing the key. In that case the server should silently drop the SYN. The TCP retransmission mechanism on the client-side will retransmit the SYN after the expiration of the initial RTO (after 1 second). And the server's application will have eventually set the key via the socket-option.

Performance We implemented the proposed solution in order to measure how long it takes to establish a second subflow, compared to standard Multipath TCP. We decided to implement it with TLS. However, it is also possible to implement it with SSH, or any other application-level protocol that negotiates a shared secret. In order to provide a derivate of TLS's master secret to the Multipath TCP layer we extended the OpenSSL[3] library.

Today's Multipath TCP opens a second subflow after 2 RTTs. The first RTT is dedicated to the 3-way handshake of the initial subflow, the second RTT is spent

[3]http://openssl.org/

while waiting for a first ACK of the first chunk of data over the initial subflow (described in [RPB+12]). With external keys, an additional subflow can only be established after 3 RTTs. The first RTT is still dedicated to the 3-way handshake. The second and third RTTs are necessary to allow TLS to negotiate the master secret.

Type	Delay (Data Center)	Delay (Internet-like)	RTTs
Standard MPTCP	352 ± 2 µs	40.372 ± 0.002 ms	2
MPTCP ext. keys	1910 ± 13 µs	61.934 ± 0.02 ms	3

Table 6.1: Using the external keys delays the establishment of an additional subflow by one RTT in the realistic *Internet-like*-scenario.

We evaluate this in our testbed in two scenarios. First, without adding any additional delay (e.g., data-center like). Second, by emulating a 20 ms delay between the hosts. The results are presented in Table 6.1. Without additional delay, the creation of the second subflow is delayed 5-6 times more than with standard Multipath TCP. This is due to the delay it takes for the application to perform the crypto computation and provide the shared secret to Multipath TCP via the `MPTCP_KEY` socket option. The influence of the RTT is negligible in this scenario.

With an RTT of 20 ms, it becomes apparent, that standard Multipath TCP needs two RTTs in order to start establishing the second subflow, while using the external keys increases this delay to three RTTs.

We argue that this additional delay is not of a big concern. In *Internet-like* scenarios, the second subflow is delayed by only one RTT. Moreover, as TLS is being used, no data will be sent unless the shared secrets have been exchanged by TLS. The second subflow will be established as soon as TLS starts sending the application's data and thus the user can fully benefit from Multipath TCP.

6.3 Long-term extensibility

The previous sections introduced a low-overhead Multipath TCP handshake and another extension that allows to do the authentication of new subflows by using keys derived by the application. It becomes apparent that Multipath TCP requires more flexibility when signaling its control information to allow a long-term extensibility. Especially the limitation of 40 bytes within the TCP option-space is a major concern for future extensions to Multipath TCP and TCP in general. Further, it has been shown that there are some problems that require a different way of exchanging the control information within Multipath TCP. For example, the ADD_ADDR option has been proven to be a security risk, as it allows off-path attackers to hijack a Multipath TCP connection [BPG+14]. A solution to this requires to add additional security to the address-advertisement, requiring even more bytes in the TCP option space. Further, it has been shown that the loss of a REMOVE_ADDR can cause performance problems [PDD+12]. A reliable transmission of the REMOVE_ADDR option would be beneficial for Multipath TCP.

All these problems have their origin in the design-choice of using the TCP option space for the signalling of Multipath TCP's control information. In this section, we show that another design can be envisioned. Instead of using only TCP options to exchange control information, we show that it is possible to define a control stream in parallel with the data stream. This new stream can be used to exchange the control information over the established subflows. It allows two Multipath TCP hosts to reliably exchange control information without being restricted by the limited TCP option space. A similar suggestion has been done early at the design-time of Multipath TCP with MCTCP [Sch12]. However, this proposal was vulnerable to deadlocks, because the DATA_ACK was considered to be part of the ”control stream”. Our proposal uses the control stream solely to exchange control information that is not directly linked to the progression of the data stream.

Together with the control stream, we propose to modify the Multipath TCP-handshake so that no crypto information is exchanged inside the TCP options and basically use the low-overhead handshake presented in Section 6.1. We suggest to use the control stream instead to negotiate the crypto information.

A new sequence number space

In contrast to SCTP [Ste07], TCP and Multipath TCP [FRHB13] only support one data stream. SCTP uses chunks to allow the communicating hosts to exchange control information of almost unlimited size. As explained earlier, having a control stream in Multipath TCP would enable a reliable delivery of the control information without strict length limitations.

This section defines a control stream that allows to exchange Multipath TCP control information of arbitrary length besides the regular data stream. The control stream holds data in a TLV-format and thus any type of data can be exchanged within it. Further, the control stream provides a reliable and in-order delivery of the control data.

The control stream is sent inside the payload of TCP segments. This ensures a reliable delivery of the TLVs exchanged in the control stream. Further, a separate control-sequence number space is defined for the control stream to ensure in-order delivery of the control stream. The Initial Control stream Sequence Number (ICSN) is the same as the IDSN in the respective directions. A DSS-mapping is

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Kind	Lenght	Subtype	(reserved)	S	F	m	M	a	A
Control ACK (4 or 8 octets, depending on flags)									
Control sequence number (4 or 8 octets, depending on flags)									
Subflow Sequence Number (4 octets)									
Control-Level Length (2 octets			Checksum (2 octets)						

Figure 6.8: The S bit of the 'reserved' field is set to 1 when sending on the control stream.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Type	Length
Value (Length - 4)	

Figure 6.9: The TLV option format

used within the TCP option space to signal the control stream sequence numbers as well as control stream acknowledgements. This DSS-mapping option is the same as the one defined in [FRHB13]. To differentiate the control stream from the data stream, we use the last bit of the 'reserved' field of the Multipath TCP DSS option. We call this bit the Stream (S) bit. When the DSS option is used to map regular data, this bit is set to 0 as currently recommended in RFC 6824. When the DSS option is used to map a TLV on the control stream, it is set to 1 (see Figure 6.8).

The control information exchanged in the control stream is encoded by using a TLV format, where the type and length are 16-bit values. This allows for maximum extensibility and to use very long data within the control stream. The format of the proposed TLV option is shown in Figure 6.9.

Window considerations Multipath TCP uses the receive window to do flow control at the receiver. The Multipath TCP receive window is used at the data sequence level, however any segment sent on a subflow must obey to the last window-announcement received on this particular subflow with respect to the subflow-level sequence number.

The control stream is no different with respect to this last point. The subflow-sequence numbers used for control stream data must fit within the window announced over this specific subflow. However, to avoid issues of receive-window handling at the control stream sequence number level, a host may never have more than one unacknowledged TLV in-flight. This effectively limits the amount of memory required to support the control-stream down to 64 KB (the maximum size of a TLV-field).

TCP uses the congestion window to limit the amount of unacknowledged in-flight data within a TCP connection. The control stream must also obey to this limitation on each TCP subflow. As the control stream uses regular TCP sequence numbers, the congestion-window limitations apply too.

Use-cases

The control stream can be used to negotiate the crypto material to authenticate new subflows. Thus, the handshake of the initial subflow does not need anymore to exchange teh 64-bit keys in plaintext. The control stream actually enables the use of our low-overhead variant of Multipath TCP, presented in Section 6.1.

How this exchange of the crypto material within the control stream is achieved remains to be specified and depends on the different use-cases. For example, a Diffie-Hellman key exchange could be done over the control stream.

Further, other use cases can benefit from the control stream. E.g., the address advertisement could be done via this solution. In RFC6824, the address-signaling is achieved through the ADD_ADDR and REMOVE_ADDR options. These options are sent within the TCP option space and thus do not benefit from reliable delivery. However, it has been shown that such a reliable delivery of the REMOVE_ADDR would be beneficial for Multipath TCP [PDD+12]. Further, security concerns rose concerning the ADD_ADDR option (as explained in [BPG+14]). Using the control stream to signal the addition or removal of addresses allows to make these options reliable and provides the space to add any kind of cryptographic material to enhance their security.

6.4 Conclusion

In this chapter we took a step back and reflected on the issues as well as the possible room for improvement that have been identified during the last years of Multipath TCP's usage and implementation experience. We have shown that the current Multipath TCP handshake introduces an overhead in terms of performance, while not even providing sufficient security to protect against on-path passive attackers. An alternative, lightweight handshake would improve the performance, while leveraging the security from application-layer protocols would bring the required security to Multipath TCP.

Going even further, we considered in this chapter how Multipath TCP might look like if we fundamentally change the way the control information is exchanged. The control stream would allow for a reliable and flexible way to exchange the necessary control information, while not being limited by the scarce amount of TCP option space. Our proposed control stream allows to exchange arbitrary information and opens the door for a more flexible way to exchange crypto material and control the way subflows are used. This control stream is a more disruptive change in Multipath TCP and it remains to be seen whether it will be deemed to be useful in the future.

Chapter 7

Conclusion

The ever-growing amount of traffic that transits the Internet, as well as the ultimate goal to deliver an even better experience to its users, demands for more and more improvements on how the data is exchanged between the end hosts. Multipath TCP tries to improve this by using the available multiple paths in the most efficient way. Users benefit from Multipath TCP, because it is able to pool the resources among these paths, effectively increasing the goodput of the transmission. It also allows for a better resilience against link failures as traffic can be moved from a failed path to another. However, efficiently using these multiple paths remains a challenge for Multipath TCP.

1. It must be usable across today's Internet with all its middleboxes and firewalls. This constraint had a significant impact on the design of Multipath TCP.

2. Multipath TCP must be able to use the different paths in an optimal way. The congestion control schemes developed for Multipath TCP [WRGH11, KGP+12, CXF12] manage to balance the load across the different paths. Heterogeneous environments bring another dimension to Multipath TCP, with new problems that demand specific heuristics on how to transmit the data.

3. To the applications Multipath TCP should be as usable as regular TCP, which implies that it must provide the same reliable and in-order delivery service.

4. The security of Multipath TCP is handled in the protocol by exchanging keys during the handshake of the initial subflow. These keys serve as a shared secret, being used to authenticate during the establishment of additional TCP subflows.

Bringing Multipath TCP to a wide-spread deployment demands that it works well in a large range of environments and use-cases. This thesis contributed to this goal, by improving Multipath TCP in different aspects. The basis of this thesis lies in the usage and improvement of the Linux kernel implementation. Indeed, a protocol can only be deployed if a scalable implementation allows end-users to actually use it. We improved Multipath TCP by evaluating it in a wide range of environments using experimental design – an approach seldom used in networking research. We designed heuristics so that it works well in heterogeneous environments, providing the best possible user-experience. We implemented Multipath TCP in such a way that it fits nicely within the Linux TCP stack, while still being scalable and achieving high performance. Finally, we took a step back and reflected on how Multipath TCP could have been designed while taking into account the lessons learned during the last years. In detail, the main contributions of this thesis were:

In Chapter 3 we introduced *"Experimental Design"*, the planned approach to experimentation that provides a statistical ground to support claims about the performance of a protocol. We explained how this approach can be applied to the evaluation of transport-layer protocols like Multipath TCP. We listed a set of influencing factors that have an impact on the performance with respect to the different Multipath TCP goals. For these goals we established metrics that allow to quantify how well each goal is met. Finally, we designed the experiment by using space-filling across the space of influencing factors. This allows to measure how the protocol behaves within the bounds of this space. Applying this process to Multipath TCP allowed us to discover its performance with respect to its resource-pooling capabilities, its load balancing as well as the delay perceived by the application.

As mentioned above, heterogeneous environments bring a new set of problems to Multipath TCP. Chapter 4 shed some light on these problems and how they influence the user-experience. We explained that heterogeneous environments provoke receive-window limitations and head-of-line blocking. Receive-window limitations prevent the end host from fully using the capacity of its subflows, effectively decreasing the goodput of the data stream. Head-of-line blocking prevents the stack from delivering the data to the application in a continuous way, creating bursts of data. Such bursts are bad for applications relying on a low application delay, like video streaming applications. We established a reactive approach to handle receive-window limitations. The reactive approach retransmits segments who cause the limitation on low-delay subflows. Further, the subflow causing these limitations is penalized by reducing its congestion window. We showed that this approach is very promising and we fine-tuned it by using the experimental design approach. Another way of solving these issues would be to schedule the data across the subflows in such a way that they cannot occur. We opened the door for such a scheduler by implementing a modular scheduler framework that allows researchers to easily experiment with different schedulers.

Chapter 5 describes the implementation of Multipath TCP in the Linux Kernel. We outlined the goals that were the basis for the design of the implementation. Throughout the chapter we showed how these goals affected the design decisions. The implementation tries to minimize the performance impact on regular TCP. Sometimes, this constraint required us to take decisions that are not optimal from a performance point-of-view for Multipath TCP. Nevertheless, we were able to show that Multipath TCP still achieves very good performance. It is able to quickly respond to incoming connection requests, transmit data at a high speed – even when using benchmarking tools that put a high pressure on the implementation. Finally, we showed that Multipath TCP is able to achieve up to 51.8 Gbps of goodput. Such a record has never been established before for the transmission of one single data stream.

With several years of experience in designing, implementing, testing and deploying Multipath TCP, we take a step back and looked at three key-components of Multipath TCP and whether they could have been designed differently. Chapter 6 first analyses the handshake and how its key-exchange affects the performance of the Linux Kernel implementation. It suggests and alternative, lightweight handshake which sacrifices security for the benefit of improved performance. Following this in Chapter 6, we consider how Multipath TCP could benefit of the security provided by other layers. Particularly we explore how to leverage the shared secrets exchanged in Transport Layer Security (TLS) into Multipath TCP in order to authenticate new subflows. Finally, we reflect on a more disruptive change to the way Multipath TCP exchanges control information. The original design-decision to exchange it in the TCP option space has shown to often be the limiting factor in Multipath TCP. We explore a new control stream that allows to reliably exchange arbitrary-length control information. This would open the door for a more flexible way of controlling Multipath TCP's behavior.

Open problems

This thesis does one step into improving Multipath TCP so that multipath communication can become the rule on the future Internet. However, there still remain some open research challenges as described hereafter.

The introduction of experimental design to the evaluation of Multipath TCP shows the benefits of such an approach to improve a protocol. We believe that it can advance future research as it provides a tool to validate the overall performance of a protocol. A more in-depth analysis of the performance of Multipath TCP still remains to be done, e.g., considering a wider range of applications, as well as introducing competing traffic to the environment being used during the evaluation. Further, different metrics could be established to quantify the performance of Multipath TCP with respect to other aspects than the ones considered in this thesis.

The scheduling of data across the different subflows opens up a whole new dimension. Scheduling might reduce head-of-line blocking in heterogeneous environments, enabling streaming applications to reduce their buffer requirements. It remains unclear, how such a scheduling should take into account the feedback in terms of round-trip-time and bandwidth-delay-product estimation to achieve high performance.

Finally, the longer term evolution of the Multipath TCP protocol specification will probably require a stronger security than the one provided by RFC 6824. This thesis has suggested some possible ways to go. The control-stream might be the addition to Multipath TCP that provides the required flexibility. Future research might also come up with a completely different strategy that allows to provide a secure communication with Multipath TCP.

Bibliography

[ACO+06] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding Traceroute Anomalies with Paris Traceroute. In *ACM IMC*, 2006.

[AF99] M. Allman and A. Falk. On the Effective Evaluation of TCP. *ACM SIGCOMM Computer Communication Review*, 29(5):59–70, 1999.

[AFLV08] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. 38(4):63–74, 2008.

[All12] M. Allman. Comments on Bufferbloat. *ACM SIGCOMM Computer Communication Review*, 43(1):30–37, 2012.

[And12] Ö. Andersson. *Experiment! Planning, Implementing and Interpreting*. Wiley, 2012.

[ATRK10] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock. Host-to-Host Congestion Control for TCP. *IEEE Communications Surveys & Tutorials*, 12(3):304–342, 2010.

[BA02] E. Blanton and M. Allman. On Making TCP More Robust to Packet Reordering. *ACM SIGCOMM Computer Communication Review*, 32(1):20–30, 2002.

[Bar11] S. Barre. *Implementation and assessment of Modern Host-based Multipath Solutions*. PhD thesis, UCLouvain, 2011.

[BD87] G. Box and N. R. Draper. *Empirical Model Building and Response Surfaces*. John Wiley & Sons, 1987.

[Bon13] O. Bonaventure. Apple seems to also believe in Multipath TCP. `http://perso.uclouvain.be/olivier.bonaventure/blog/html/2013/09/18/mptcp.html`, 2013.

[BP95] L. Brakmo and L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[BPB11] S. Barre, C. Paasch, and O. Bonaventure. Multipath TCP: From Theory to Practice. In *IFIP Networking*, 2011.

[BPG$^+$14] M. Bagnulo, C. Paasch, F. Gont, O. Bonaventure, and C. Raiciu. Analysis of MPTCP residual threats and possible fixes. IETF Internet-Draft draft-ietf-mptcp-attacks-00, January 2014.

[CLG$^+$13] Y-C Chen, Y-S Lim, R. Gibbens, E. Nahum, R. Khalili, and D. Towsley. A Measurement-based Study of Multipath TCP Performance over Wireless Networks. In *ACM SIGCOMM IMC*, 2013.

[CXF12] Y. Cao, M. Xu, and X. Fu. Delay-based Congestion Control for Multipath TCP. In *IEEE ICNP*, 2012.

[DBRT10] T. Dreibholz, M. Becke, E.P. Rathgeb, and M. Tuxen. On the Use of Concurrent Multipath Transfer over Asymmetric Paths. In *IEEE Globecom*, 2010.

[DH98] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6). IETF RFC 2460, December 1998.

[DRC$^+$10] N. Dukkipati, T. Refice, Y Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An Argument for Increasing TCP's Initial Congestion Window. *SIGCOMM Computer Communication Review*, 40(3):26–33, July 2010.

[DSTR10] T. Dreibholz, R. Seggelmann, M. Tüxen, and E.P. Rathgeb. Transmission Scheduling Optimizations for Concurrent Multipath Transfer. In *PFLDNeT*, 2010.

[Edd06] W. Eddy. Defenses Against TCP SYN Flooding Attacks. *The Internet Protocol Journal*, 9(4):2–16, 2006.

[Edg10] J. Edge. Receive flow steering. `http://lwn.net/Articles/382428/`, 2010.

[ESC05] D. Easlake, J. Schiller, and S. Crocker. Randomness Requirements for Security. IETF RFC 4086, June 2005.

[F$^+$49] R. A. Fisher et al. *The Design of Experiments*. Number 5th ed. Oliver and Boyd, London and Edinburgh, 1949.

[Fou09] Linux Foundation. Bonding. `http://www.linuxfoundation.org/collaborate/workgroups/networking/bonding`, 2009.

[FRHB13] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. IETF RFC 6824, January 2013.

[GHJ+09] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. 39(4):51–62, 2009.

[GN11] J. Gettys and K. Nichols. Bufferbloat: Dark Buffers in the Internet. *ACM Queue*, 9(11):40–54, 2011.

[HDP+13] B. Hesmans, F. Duchene, C. Paasch, G. Detal, and O. Bonaventure. Are TCP Extensions Middlebox-proof? In *CoNEXT Workshop Hot-Middlebox*, 2013.

[HHJ+12] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible Network Experiments using Container-based Emulation. In *ACM CoNext*, 2012.

[HNR+11] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *ACM IMC*, 2011.

[HRX08] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[IAS06] J. Iyengar, P. Amer, and R. Stewart. Concurrent Multipath Transfer using SCTP Multihoming over Independent End-to-End Paths. *IEEE/ACM Transactions on Networking*, 14(5):951–964, 2006.

[IEE00] IEEE. Link Aggregation. `http://www.ieee802.org/3/ad/index.html`, 2000.

[Jac88] V. Jacobson. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, 18(4):314–329, 1988.

[JBB92] V. Jacobson, B. Braden, and D. Borman. TCP Extensions for High Performance. IETF RFC 1323, May 1992.

[JNW10] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[Kas11] D. Kaspar. *Multipath Aggregation of Heterogeneous Access Networks*. PhD thesis, University of Oslo, 2011.

[KBE+14] A. Kvalbein, D. Baltrūnas, K. Evensen, J. Xiang, A. Elmokashfi, and S. Ferlin. The Nornet Edge Platform for Mobile Broadband Measurements. *Computer Networks*, 61:88–101, 2014.

[KGP+12] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J-Y. Le Boudec. MPTCP is not Pareto-Optimal: Performance Issues and a Possible Solution. In *ACM CoNEXT*, 2012.

[KGPLB13] R. Khalili, N. Gast, M. Popovic, and J.-Y. Le Boudec. MPTCP Is Not Pareto-Optimal: Performance Issues and a Possible Solution. *IEEE/ACM Transactions on Networking*, 21(5):1651–1665, 2013.

[KLC05] S. Kleijnen, J.and Sanchez, T. Lucas, and T. Cioppa. State-of-the-art Review: a User's Guide to the Brave new World of Designing Simulation Experiments. *INFORMS Journal on Computing*, 17(3):263–289, 2005.

[KSG+09] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *ACM IMC*. ACM, 2009.

[LIJM+11] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. Internet Inter-Domain Traffic. *ACM SIGCOMM Computer Communication Review*, 41(4):75–86, 2011.

[McK07] P. McKenney. What is RCU, Fundamentally? `http://lwn.net/Articles/262464/`, 2007.

[MM95] M. Morris and T. Mitchell. Exploratory Designs for Computational Experiments. *Journal of Statistical Planning and Inference*, 43(3):381–402, 1995.

[MM09] R. Myers and D. Montgomery. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Wiley.com, 2009.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. IETF RFC 2018, October 1996.

[NB09] E. Nordmark and M. Bagnulo. Shim6: Level 3 Multihoming Shim Protocol for IPv6. IETF RFC 5533, June 2009.

[NR12] H. X. Nguyen and M. Roughan. Rigorous Statistical Analysis of Internet Loss Measurements. *IEEE/ACM Transactions on Networking*, 38(1):734–745, 2012.

[PB12] C. Paasch and O. Bonaventure. MultiPath TCP Low Overhead. IETF Internet-Draft draft-paasch-mptcp-lowoverhead-00, October 2012.

[PCVB13] C. Pelsser, L. Cittadini, S. Vissicchio, and R. Bush. From Paris to Tokyo: On the Suitability of Ping to Measure Latency. In *ACM IMC*, 2013.

[PDD+12] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure. Exploring Mobile/WiFi Handover with Multipath TCP. In *ACM SIGCOMM workshop CellNet*, pages 31–36, 2012.

[Per97] C. Perkins. Mobile IP. *IEEE Communications Magazine*, 35(5):84–99, 1997.

[PJA11] C. Perkins, D. Johnson, and J. Arkko. Mobility Support in IPv6. IETF RFC 6275, July 2011.

[PKB13] C. Paasch, R. Khalili, and O. Bonaventure. On the Benefits of Applying Experimental Design to Improve Multipath TCP. In *ACM CoNEXT*, 2013.

[Pos81a] J. Postel. INTERNET PROTOCOL. IETF RFC 791, September 1981.

[Pos81b] J. Postel. Transmission Control Protocol. IETF RFC 793, September 1981.

[RBP+11] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM 2011*, 2011.

[Res10] E. Rescorla. Keying Material Exporters for Transport Layer Security (TLS). IETF RFC 5705, March 2010.

[RHW11] C. Raiciu, M. Handley, and D. Wischik. Coupled Congestion Control for Multipath Transport Protocols. IETF RFC 6356, October 2011.

[Riv95] R. Rivest. The RC5 Encryption Algorithm. In *Fast Software Encryption*, 1995.

[RNBH11] C. Raiciu, D. Niculescu, M. Bagnulo, and M. Handley. Opportunistic Mobility with Multipath TCP. In *ACM MobiArch 2011*, 2011.

[RPB+12] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *USENIX NSDI*, 2012.

[SBL+13] G. Sarwar, R. Boreli, E. Lochin, A. Mifdaoui, and G. Smith. Mitigating Receiver's Buffer Blocking by Delay Aware Packet Scheduling in Multipath Data Transfer. In *IEEE WAINA*, 2013.

[SCBS12] J. Santiago, M. Claeys-Bruno, and M. Sergent. Construction of Space-Filling Designs using WSP Algorithm for High Dimensional Spaces. *Chemometrics and Intelligent Laboratory Systems*, 113, 2012.

[Sch12] M. Scharf. Multi-Connection TCP (MCTCP) Transport. IETF Internet-Draft draft-scharf-mptcp-mctcp-01, July 2012.

[SGTG+12] A. Singh, C. Goerg, A. Timm-Giel, M. Scharf, and T.-R. Banniza. Performance Comparison of Scheduling Algorithms for Multipath Transfer. In *IEEE GLOBECOM*, 2012.

[SH07] L. Stewart and J. Healy. Light-Weight Modular TCP Congestion Control for FreeBSD 7. Technical report, CAIA, Tech. Rep, 2007.

[SHS+12] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Elses Problem: Network Processing as a Cloud Service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[SK06] M. Scharf and S. Kiesel. Head-of-line Blocking in TCP and SCTP: Analysis and Measurements. In *IEEE GLOBECOM*, 2006.

[SMM98] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. *ACM SIGCOMM Computer Communication Review*, 28(4):315–323, 1998.

[SRC84] J. Saltzer, D. Reed, and D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[STA14] Ferlin S., Dreibholz T., and Alay. Tackling the Challenge of Bufferbloat in Multi-Path Transport over Heterogeneous Wireless Networks. In *IEEE/ACM IWQoS*, 2014.

[Ste07] R. Stewart. Stream Control Transmission Protocol. IETF RFC 4960, September 2007.

[STR10] R. Seggelmann, M. Tuxen, and E.P. Rathgeb. Stream Scheduling Considerations for SCTP. In *IEEE SoftCOM*, 2010.

[SWMW89] J. Sacks, W. J. Welch, T. J. Mitchell, and H. P. Wynn. Design and Analysis of Computer Experiments. *JSTOR Statistical science*, 4(4):409–423, 1989.

[SX01] R. Stewart and Q. Xie. *Stream Control Transmission Protocol (SCTP)*. Addison-Wesley Professional, 2001.

[TS13] M. Tuxen and R. Stewart. UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication. IETF RFC 6951, May 2013.

[Wes07] E. Wesley. TCP SYN Flooding Attacks and Common Mitigations. IETF RFC 4987, August 2007.

[WHB08] D. Wischik, M. Handley, and M. Braun. The Resource Pooling Principle. *ACM SIGCOMM Computer Communication Review*, 38(5):47–52, 2008.

[WQX$^+$11] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An Untold Story of Middleboxes in Cellular Networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 374–385. ACM, 2011.

[WRGH11] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *USENIX NSDI*, 2011.

[ZNN$^+$10] B. Zhang, T. Ng, A. Nandi, R. Riedi, P. Druschel, and G. Wang. Measurement-Based Analysis, Modeling, and Synthesis of the Internet Delay Space. *IEEE/ACM Transactions on Networking*, 18(1):85–98, 2010.