How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP

Costin Raiciu[†], Christoph Paasch[‡], Sebastien Barre[‡], Alan Ford,
Michio Honda[◇], Fabien Duchene[‡], Olivier Bonaventure[‡] and Mark Handley[⋆]

[†]Universitatea Politehnica Bucuresti, [‡]Universite Catholique de Louvain
[◇]Keio University, [⋆]University College London

ABSTRACT

Networks have become multipath: mobile devices have multiple radio interfaces, datacenters have redundant paths and multihoming is the norm for big server farms. Meanwhile, TCP is still only single-path.

Is it possible to extend TCP to enable it to support multiple paths for current applications on today's Internet? The answer is positive. We carefully review the constraints—partly due to various types of middleboxes—that influenced the design of Multipath TCP and show how we handled them to achieve its deployability goals.

We report our experience in implementing Multipath TCP in the Linux kernel and we evaluate its performance. Our measurements focus on the algorithms needed to efficiently use paths with different characteristics, notably send and receive buffer tuning and segment reordering. We also compare the performance of our implementation with regular TCP on web servers. Finally, we discuss the lessons learned from designing MPTCP.

1. INTRODUCTION

In today's Internet, servers are often multi-homed to more than one Internet provider, datacenters provide multiple parallel paths between compute nodes, and mobile hosts have multiple radios. Traditionally, it was the role of routing to take advantage of path diversity, but this has limits to responsiveness and scaling. To really gain both robustness and performance advantages, we need transport protocols engineered to utilize multiple paths. Multipath TCP[5] is an attempt to extend the TCP protocol to perform this role. At the time of writing, it is in in the final stage of standardization in the IETF.

Multipath TCP stripes data from a single TCP connection across multiple subflows, each of which may take a different path through the network. A linked congestion control mechanism[23] controls how much data is sent on each subflow, with the goal of explicitly moving traffic off the more congested paths onto the less congested ones. This paper is not about congestion control, but rather it is about the design of the Multipath TCP protocol itself. In principle, extending TCP to use multiple paths is not difficult, and there are a number of obvious ways in which it could be done. Indeed it was first proposed by Christian Huitema in 1995[11]. In practice though, the existence of middleboxes greatly constrains the design choices. The challenge is to make Multipath TCP not only robust to path failures, but also robust to failures in the presence of middleboxes that attempt to optimize single-path TCP flows. No previous extension to the core Internet protocols has needed to consider this issue to nearly the same extent.

In the first half of this paper we examine the design options for multipath TCP, with the aim of understanding both the end-to-end problem and the end-to-middle-to-end constraints. We use the results of a large Internet study to validate these design choices.

Designing MPTCP turned out to be more difficult than expected. For instance, a key question concerns how MPTCP metadata should be encoded — embed it in the TCP payload, or use the more traditional TCP options, with the potential for interesting interactions with middleboxes. In the IETF opinions were divided, with supporters on both sides. In the end, careful analysis revealed that MPTCP needs explicit connection level acknowledgments for flow control; further, these acknowledgments can create deadlocks if encoded in the payload. In reality, there was only one viable choice.

The second half of this paper concerns the host operating system. To be viable, Multipath TCP must be implementable in modern operating systems and must perform well. We examine the practical limitations the OS poses on MPTCP design and operation. This matters: our experiments show that one slow path can significantly degrade the throughput of the whole connection when MPTCP is underbuffered. We propose novel algorithms that increase throughput ten-fold in this case, ensuring MPTCP always matches what TCP would get on the best interface, regardless of buffer size.

It is not our goal to convince the reader that multipath transport protocols in general are a good idea. There has been a wealth of work that motivates the use of multipath transport for robustness[24], the use of linked congestion control across multiple paths for load balancing[23, 14, 4] and the ability of multi-path transport protocols

to find and utilize unused network capacity in redundant topologies[10, 19]. Rather, the main contribution of this paper is the exploration of the design space for MPTCP confined by the many constraints imposed by TCP's original design, today's networks which embed TCP knowledge, and the need to perform well within the limitations imposed by the operating system.

2. GOALS

As many researchers have lamented, changing the behavior of the core Internet protocols is very difficult [7]. An idea may have great merit, but without a clear deployment path whereby the cost/benefit tradeoff for early adopters is positive, widespread adoption is unlikely.

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, the majority of which use TCP for transport. To support such unmodified applications we must work below the sockets API, providing the same service as TCP: byte-oriented, reliable and in-order delivery. In theory we could use different protocols to implement this functionality as long as fallback to TCP is possible when one end does not support multipath. In practice there is no widely deployed signaling mechanism to select between transport protocols, so we have to use options in TCP's SYN exchange to negotiate new functionality.

The goal is for an unmodified application to start (what it believes to be) a TCP connection with the regular API. When both endpoints support MPTCP and multiple paths are available, MPTCP can set up additional subflows and stripe the connection's data across these subflows, sending most data on the least congested paths.

The potential benefits are clear, but there may be costs too. If negotiating MPTCP can cause connections to fail when regular TCP would have succeeded, then deployment is unlikely. The second goal, then, is for MPTCP to work in all scenarios where TCP currently works. If a subflow fails for any reason, the connection must be able to continue as long as another subflow has connectivity.

Third, MPTCP must be able to utilize the network at least as well as regular TCP, but without starving TCP. The congestion control scheme described in [23] meets this requirement, but congestion control is not the only factor that can limit throughput.

Finally MPTCP must be implementable in operating systems without using excessive memory or processing. As we will see, this requires careful consideration of both fast-path processing and overload scenarios.

3. DESIGN

The five main mechanisms in TCP are:

- Connection setup handshake and state machine.
- Reliable transmission & acknowledgment of data.
- Congestion control.
- Flow control.
- Connection teardown handshake and state machine.

The simplest possible way to implement Multipath TCP would be to take segments coming out of the regular stack and "stripe" them across the available paths *somehow*[1]. 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 (our study shows that a third of paths will break such connections). To achieve robust, high performance multipath operation, we need more substantial changes to TCP, touching all the components listed above. Congestion control has been described elsewhere[23] so we will not discuss it further in this paper.

In brief, here is how MPTCP works. MPTCP is negotiated via new TCP options in SYN packets, and the endpoints exchange connection identifiers; these are used later to add new paths—subflows—to an existing connection. Subflows resemble TCP flows on the wire, but they all share a single send and receive buffer at the endpoints. MPTCP uses per subflow sequence numbers to detect losses and drive retransmissions, and connection-level sequence numbers to allow reordering at the receiver. Connection-level acknowledgements are used to implement proper flow control. We discuss the rationale behind these design choices below.

3.1 Connection setup

The TCP three-way handshake serves to synchronize state between the client and server[2]. 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.

MPTCP must use this initial handshake to negotiate multipath capability. An MP_CAPABLE option is sent in the SYN and echoed in the SYN/ACK if the server

[1]such *striping* needs additional mechanisms because both destination-based forwarding and network ECMP try hard not to stripe packets belonging to the same TCP connection

[2]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.

understands MPTCP and wishes to enable it. Although this form of extension has been used many times, the Internet has grown a great number of middleboxes in recent years. Does such a handshake still work?

We performed a large study to test this - complete results are available in [9]. Our code generates specific TCP segments with the aim of testing what really happens on Internet paths. These tests were run from 142 access networks in 24 countries, including a wide mix of cellular providers, WiFi hotspots, home networks, as well as university and corporate networks. Although we cannot claim full coverage, the sample is large enough to provide good evidence for what does and what does not work in today's Internet.

We found that 6% of paths tested remove new options from SYN packets. This rises to 14% for connections to port 80 (HTTP). We did not observe any access networks that actually dropped a SYN with a new option. Most importantly, no path removed options from data packets unless it also removed them from the SYN, so it is possible to test a path using just the SYN exchange.

A separate study[3] probed Internet servers to see if new options in SYN packets caused any problems. Of the Alexa top 10,000 sites, 15 did not respond to a SYN packet containing a new option.

From these experiments we conclude that negotiating MPTCP in the initial handshake is feasible, but with some caveats. There is no real problem if a middlebox removes the MP_CAPABLE option from the SYN: MPTCP simply falls back to regular TCP behavior. However removing it from the SYN/ACK would cause the client to believe MPTCP is not enabled, whereas the server believes it is. This mismatch would be a problem if data packets were encoded differently with MPTCP. The obvious solution is to require the third packet of the handshake (ACK of SYN/ACK) to carry an option indicating that MPTCP was enabled. However this packet may be lost, so MPTCP must require all subsequent data packets to also carry the option until one of them has been acked. If the first non-SYN packet received by the server does not contain an MPTCP option, the server must assume the path is not MPTCP-capable, and drop back to regular TCP behavior.

Finally, if a SYN needs to be retransmitted, it would be a good idea to follow the retransmitted SYN with one that omits the MP_CAPABLE option.

It should be clear from this brief discussion of what should be the simplest part of MPTCP that anyone designing extensions to TCP must no longer think of the mechanisms as concerning only two parties. Rather, the negotiation is two-way *with mediation*, where the packets that arrive are not necessarily those that were sent. This requires a more defensive approach to protocol design than has traditionally been the case.

3.2 Adding subflows

Once two endpoints have negotiated MPTCP, they can open additional subflows. In an ideal world there would be no need to send new SYN packets before sending data on a new subflow - all that would be needed is a way to identify the connection that packets belong to. The strawman design simply sent TCP segments along different paths, and the endpoints used the 5-tuple to identify the proper connection. In practice though, we see that NATs and Firewalls rarely pass data packets that were not preceded by a SYN.

Adding a subflow raises two problems. First, the new subflow needs to be associated with an existing MPTCP flow. The classical five-tuple cannot be used as a connection identifier, as it does not survive NATs. Second, MPTCP must be robust to an attacker that attempts to add his own subflow to an existing MPTCP connection.

When the first MPTCP subflow is established, the client and the server insert 64-bit random keys in the MP_CAPABLE option. These will be used to verify the authenticity of new subflows.

To open a new subflow, MPTCP performs a new SYN exchange using the additional addresses or ports it wishes to use. Another TCP option, MP_JOIN is added to the SYN and SYN/ACKs. This option carries a MAC of the keys from the original subflow; this prevents blind spoofing of MP_JOIN packets from an adversary who wishes to hijack an existing connection. MP_JOIN also contains a connection identifier derived as a hash of the recipient's key [5]; this is used to match the new subflow to an existing connection.

If the client is multi-homed, then it can easily initiate new subflows from any additional IP addresses it owns. However, if only the server is multi-homed, the wide prevalence of NATs makes it unlikely that a new SYN it sends will be received by a client. The solution is for the MPTCP 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.

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. But the Internet itself is so frequently asymmetric that we need two distinct ways, one implicit and one explicit, to indicate the existence of additional addresses.

3.3 Reliable multipath delivery

In a world without middleboxes, MPTCP could simply stripe data across the multiple subflows, with the sequence numbers in the TCP headers indicating the sequence number of the data in the connection in the normal TCP way. Our measurements show that this is infeasible in today's Internet:

- We observed that 10% of access networks rewrite TCP initial sequence numbers (18% on port 80). Some of this re-writing is by proxies that remove new options; a new subflow will fail on these paths. But many that rewrite do pass new options - these appear to be firewalls that attempt to increase TCP initial sequence number randomization. As a result, MPTCP cannot assume the sequence number space on a new subflow is the same as that on the original subflow.
- Striping sequence numbers across two paths leaves gaps in the sequence space seen on any single path. We found that 5% of paths (11% on port 80) do not pass on data after a hole - most of these seem to be proxies that block new options on SYNs and so don't present a problem as MPTCP is never enabled on these paths. But a few do not appear to be proxies, and so would stall MPTCP. Perhaps worse, 26% of paths (33% on port 80) do not correctly pass on an ACK for data the middlebox has not observed - either the ACK is dropped or it is "corrected".

Given the nature of today's Internet, it appears extremely unwise to stripe a single TCP sequence space across more than one path. The only viable solution is to use a separate contiguous sequence space for each MPTCP subflow. For this to work, we must also send information mapping bytes from each subflow into the overall data sequence space, as sent by the application. We shall return to the question of how to encode such mappings after first discussing flow control and acknowledgments, as the three are intimately related.

3.3.1 Flow control

TCP's receive window indicates the number of bytes beyond the sequence number from the acknowledgment field that the receiver can buffer. The sender is not permitted to send more than this amount of additional data.

Multipath TCP also needs to implement flow control, although packets now arrive over multiple subflows. If we inherit TCP's interpretation of receive window, this would imply an MPTCP receiver maintains a pool of buffering per subflow, with receive window indicating per-subflow buffer occupancy. Unfortunately such an interpretation can lead to a deadlock scenario:

1. The next packet that needs to be passed to the application was sent on subflow 1, but was lost.
2. In the meantime subflow 2 continues delivering data, and fills its receive window.
3. Subflow 1 fails silently.
4. The missing data needs to be re-sent on subflow 2, but there is no space left in the receive window, resulting in a deadlock.

The receiver could solve this problem by re-allocating subflow 1's unused buffer to subflow 2, but it can only do this by rescinding the advertised window on subflow 1. Besides, the receiver does not know which subflow the next packet will be sent on. The situation is made even worse because a TCP proxy[3] on the path may hold data for subflow 2, so even if the receiver opens its window, there is no guarantee that the first data to arrive is the retransmitted missing packet.

The correct solution is to generalize TCP's receive window semantics to MPTCP. For each connection a single receive buffer pool should be shared between all subflows. The receive window then indicates the maximum *data* sequence number that can be sent rather than the maximum subflow sequence number. As a packet resent on a different subflow always occupies the same data sequence space, no such deadlock can occur.

The problem for an MPTCP sender is that to calculate the highest data sequence number that can be sent, the receive window needs to be added to the highest data sequence number acknowledged. However the ACK field in the TCP header of an MPTCP subflow must, by necessity, indicate only *subflow* sequence numbers. Does MPTCP need to add an extra *data acknowledgment* field for the receive window to be interpreted correctly?

3.3.2 Acknowledgments

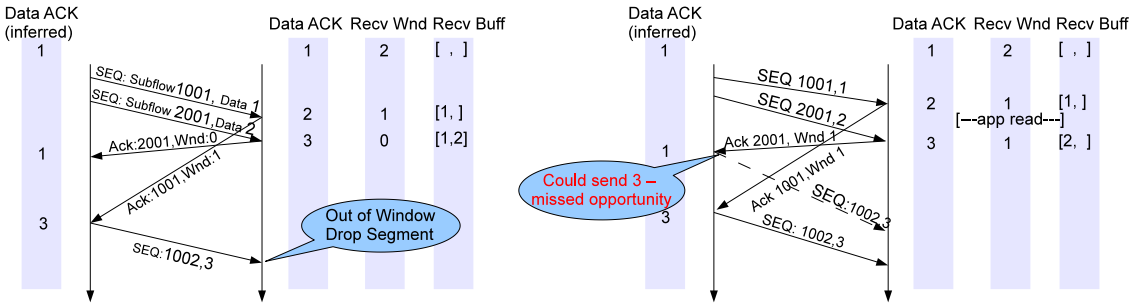
To correctly deduce a cumulative data acknowledgment from the subflow ACK fields, an MPTCP sender might keep a scoreboard of which data sequence numbers were sent on each subflow. However, the inferred value of the cumulative data ACK does not step in precisely the same way that an explicit cumulative data ACK would. Consider the sequence shown in Fig.1(a)[4]:

1. Data sequence no. 1 is sent on subflow 1 with subflow sequence number 1001.
2. Receiver sends ACK for 1001 on subflow 1.
3. Data sequence no. 2 is sent on subflow 2 with subflow sequence number 2001.
4. Receiver sends ACK for 2001 on subflow 2.
5. ACK for 2001 arrives (the RTT on subflow 2 is shorter).
6. ACK for 1001 arrives at sender.

The receiver expected the ACK for 1001 to be an implicit data ACK for 1, and the ACK for 2001 to be an implicit ACK for 2. However, as the ACK for 2001 does not implicitly acknowledge both 1 and 2, the sender's inferred data ACK is still 0 after step 5. Only after step 6 does the inferred data ACK become 2.

This sort of reordering is inevitable with multipath given the different RTTs of the different paths, and it would not by itself be a problem, except that the receiver needs to code the receive window field relative to the implicit data ACK. Suppose the receive buffer were only

[3]Most will prevent MPTCP being negotiated, but a few do not.
[4]The example uses packet sequence numbers for clarity, but MPTCP actually uses byte sequence numbers just like TCP



(a) Drops due to incorrect inference

(b) Stalls due to incorrect inference

Figure 1: Problems with inferring the cumulative data ACK from subflow ACK

two packets, and the application is slow to read. In the ACK for 1001, the receiver closes the receive window to one packet. In the ACK for 2001 the receiver closes the receive window completely, as there is no space remaining. When the ACK for 1001 is finally received, the inferred cumulative data ACK is now 2; the sender adds the receive window of one to this, and concludes incorrectly that the receiver has sufficient buffer space for one more packet. Fig. 1(b) shows a similar situation where reordering causes sending opportunities to be missed.

To avoid such scenarios MPTCP must carry an explicit data acknowledgment field, which gives the left edge of the receive window.

3.3.3 Encoding

We have seen that in the forward path we need to encode a mapping of subflow bytes into the data sequence space, and in the reverse path we need to encode cumulative data acknowledgments. There are two viable choices for encoding this additional data:

- Send the additional data in TCP options.
- Carry the additional data within the TCP payload, using a chunked or escaped encoding to separate control data from payload data.

For the forward path we have not found any compelling arguments either way, but the reverse path is a different matter. Consider a hypothetical encoding that divides the payload into chunks where each chunk has a TLV header. A data acknowledgment can then be embedded into the payload using its own chunk type. Under most circumstances this works fine. However, unlike TCP's pure ACK, anything embedded in the payload must be treated as data. In particular:

- It must be subject to flow control because the receiver must buffer data to decode the TLV encoding.
- If lost, it must be retransmitted consistently, so that middleboxes can track sequence state correctly[5]

[5]In our observations, the usual TCP proxies re-asserted the original content when sent a "retransmission" with different data. We also found one path that did this without exhibiting any other proxy behavior - this is symptomatic of a traffic normalizer[8] - and one on port 80 that reset the connection.

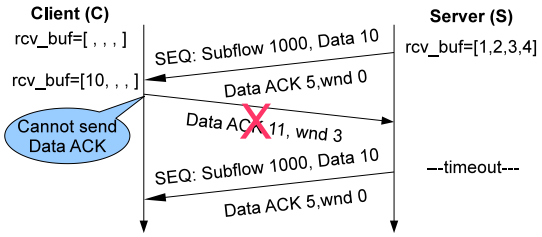


Figure 2: Flow Control on the path from C to S inadvertently stops the data flow from S to C

- If packets before it are lost, it might be necessary to wait for retransmissions before the data can be parsed - causing head-of-line blocking.

Flow control presents the most obvious problem for the chunked payload encoding. Figure 2 provides an example. Client C is pipelining requests to server S; meanwhile S's app is busy sending the large response to the first request so it isn't yet ready to read the subsequent requests. At this point, S's receive buffer fills up.

S sends segment 10, C receives it and wants to send the DATA ACK, but cannot: flow control imposed by S's receive window stops him. Because no DATA ACKs are received from C, S cannot free his send buffer, so this fills up and blocks the sending application on S. S's application will only read when it has finished sending data to C, but it cannot do so because its send buffer is full. The send buffer can only empty when S receives the DATA ACK from C, but C cannot send this until S's application reads. This is a classic deadlock cycle.

As no DATA ACK is received, S will eventually time out the data it sent to C and will retransmit it; after many retransmits the whole connection will time out.

It has been suggested that this can be avoided if DATA ACKs are simply excluded from flow control. Unfortunately any middlebox that buffers data can foil this; it is

unaware the DATA ACK is special because it looks just like any other TCP payload.

When the return path is lossy, decoding DATA ACKs will be delayed until retransmissions arrive - this will effectively trigger flow control on the forward path, reducing performance. In effect, this would break MPTCP's goal of doing "no worse" than TCP on the best path.

Our conclusion is that DATA ACKs cannot be safely encoded in the payload. The only real alternative is to encode them in TCP options which (on a pure ACK packet) are not subject to flow control.

3.3.4 Data sequence mappings

If MPTCP must use options to encode DATA ACKs, it is simplest to also encode the mapping from subflow sequence numbers to data sequence numbers in a TCP option. This is the *data sequence mapping* or DSM.

At first we thought that the DSM option simply needed to carry the data sequence number corresponding to the start of the MPTCP segment. Unfortunately middleboxes and "smart" NICs make this far from simple.

Middleboxes that resegment data would cause a problem. [6] TCP Segmentation Offload (TSO) hardware in the NIC also resegments data and is commonly used to improve performance. The basic idea is that the OS sends large segments and the NIC resegments them to match the receiver's MSS. What does TSO do with TCP options? We tested 12 NICs supporting TSO from four different vendors. All of them copy a TCP option sent by the OS on a large segment into all the split segments.

If MPTCP's DSM option only listed the data sequence number, TSO would copy the same DSM to more than one segment, breaking the mapping. Instead the DSM option must say precisely which subflow bytes map to which data sequence numbers. But this is further complicated by middleboxes that rewrite sequence numbers; these are commonplace — 10% of paths. Instead, the DSM option must map the offset from the subflow's initial sequence number to the data sequence number, as the offset is unaffected by sequence number rewriting. The option must also contain the length of the mapping. This is robust - as long as the option is received, it does not greatly matter which packet carries it, so duplicate mappings caused by TSO are not a problem.

3.3.5 Send buffer management

The sender will free segments from the connection-level send queue only when they are acknowledged by a DATA ACK. Even if a segment is ACKed at the subflow level, its data is kept in memory until we receive a DATA ACK. If a DATA ACK does not arrive, a timer fires and the sender retransmits that data. This allows the receiver to ACK all segments correctly received at the subflow level, which in turn allows the sender to correctly infer path properties. This separation of functionality also allows the receiver to drop data that is in-window at the subflow level but out-of-window at the connection level.

Further, if a middlebox coalesces packets, TCP's limited option space means it can only keep one of the data sequence mapping options on the coalesced segment. The receiver will get a bigger segment where some of the bytes have no mapping. The packet will be acknowledged at the subflow-level, and only the bytes with the mapping will be acknowledged at the data level. This causes the sender to retransmit the missing bytes, allowing the MPTCP connection to make progress.

3.3.6 Content-modifying middleboxes

Many NAT devices include application-level gateway functionality for protocols such as FTP: IP addresses and ports in the FTP control channel are re-written to correct for the address changes imposed by the NAT.

Multipath TCP and such content-modifying middleboxes have the potential to interact badly. In particular, due to FTP's ASCII encoding, re-writing an IP address in the payload can necessitate changing the length of the payload. Subsequent sequence and ACK numbers are then fixed up by the middlebox so they are consistent from the point of view of the end systems.

Such length changes break the DSM option mapping - subflow bytes can be mapped to the wrong place in the data stream. They also break every other mapping mechanism we considered, including chunked payloads. There is no easy way to handle such middleboxes.

After much debate, we concluded that MPTCP must include a checksum in the DSM mapping so such content changes can be detected. MPTCP rejects a modified segment and triggers a fallback process: if any other subflows exists, MPTCP terminates the subflow on which the modification occurred; if no other subflow exists, MPTCP drops back to regular TCP behavior for the remainder of the connection, allowing the middlebox to perform rewriting as it wishes.

Calculating a checksum over the data is comparatively expensive, and we did not wish to slow down MPTCP just to catch such rare corner cases. MPTCP therefore uses the same 16-bit ones complement checksum used in the TCP header. This allows the checksum over the payload to be calculated only once. The payload checksum is added to a checksum of an MPTCP pseudo header covering the DSM mapping values and then inserted into the DSM option. The same payload checksum is added to the checksum of the TCP pseudo-header and then used in the TCP checksum field.

With this mechanism a software implementation incurs little additional cost from calculating the MPTCP

[6] We did not observe any that would both permit MPTCP and resegment, though.

checksum. Unfortunately, modern NICs frequently perform checksum offload. If the TCP stack uses the NIC to calculate checksums, with MPTCP it will still need to calculate the MPTCP checksum in software, negating the benefits of checksum offload. There is little we can do about this, other than to note that future NICs will likely perform MPTCP checksum offload too, if MPTCP is widely deployed. In the meantime, MPTCP allows checksums to be disabled for high performance environments such as data-centers where there is no chance of encountering such an application-level gateway.

The fallback-to-TCP process, triggered by a checksum failure, can also be triggered in other circumstances. For example, if a routing change moves an MPTCP subflow to a path where a middlebox removes DSM options, this also triggers the fallback procedure.

3.4 Connection and subflow teardown

TCP has two ways to indicate connection shutdown: FIN for normal shutdown and RST for errors such as when one end no longer has state. With MPTCP, we need to distinguish subflow teardown from connection teardown. With RST, the choice is clear: it must only terminate the subflow, or an error on a single subflow would cause the whole connection to fail.

Normal shutdown is slightly more subtle. TCP FINs occupy sequence space; the FIN/FIN-ACK/ACK handshake and the cumulative nature of TCP's acknowledgments ensure that not only all data has been received, but also both endpoints know the connection is closed and know who needs to hold TIMEWAIT state.

How then should a FIN on an MPTCP subflow be interpreted? Does it mean that the sending host has no more data to send, or only that no more data will be sent on this subflow? Another way to phrase this is to ask whether a FIN on a subflow occupies data sequence space, or just subflow sequence space?

Consider first what would happen if a FIN occupied data sequence space. This could be achieved by extending the length of the DSM mapping in a packet to cover the FIN. Mapping the FIN into the data sequence space in this way tells the receiver what the data sequence number of the last byte of the connection is, and hence whether any more data is expected from other subflows.

Suppose that some data had been transmitted on subflow A just before the last data and FIN were sent on subflow B. If the receiver is really unlucky, subflow A may fail (perhaps due to mobility) before the last data arrives. When the sender times out this data, it will wish to re-send it on subflow B, but it has already sent a FIN on this subflow. Sending data after the FIN is sure to confuse middleboxes and firewalls that tore down state when they observed the FIN. This problem might be avoided by delaying sending the FIN until all outstanding data has been DATA ACKed, but this adds an unnecessary RTT to all connections during which the receiving application doesn't know if more data will arrive.

Much simpler is for a FIN to have the more limited "no more data on this subflow" semantics, and this is what MPTCP does. An explicit DATA FIN, carried in a TCP option, indicates the end of the data sequence space and can be sent immediately when the application closes the socket. To be safe, either the sender waits for the DATA ACK of the DATA FIN before sending a FIN on each subflow, or it sends DATA FIN on all subflows together with a FIN.

MPTCP's FIN semantics also allow subflows to be closed cleanly while allowing the connection to continue on other subflows. Finally, to support mobility, MPTCP provides a REMOVE_ADDR message, allowing one subflow to indicate that other subflows using the specified address are closed. This is necessary to cleanly cope with mobility when a host loses the ability to send from an address and so cannot send a subflow FIN.

4. IMPLEMENTATION ISSUES

To validate the design of MPTCP and understand its impact on real applications, we added full support for MPTCP to version 2.6.38 of the Linux kernel. This is a major modification to TCP: our patch to the Linux kernel, available from http://mptcp.info.ucl.ac.be, is about 10,400 lines of code . The software architecture is described in detail in [1]. To our knowledge, this is the first full kernel implementation of MPTCP.

We will focus on three of the more recent important improvements to the MPTCP implementation. We first briefly describe the algorithms that have been included in our MPTCP implementation to deal with middleboxes. Then we explain how to reduce the MPTCP memory usage. Finally we show how an MPTCP receiver is able to handle out-of-order data efficiently.

4.1 Supporting middleboxes

As we have seen, middleboxes constrain the design of MPTCP in many ways. To verify whether our design and its implementation are robust to middleboxes, we implemented Click elements [15] that model the various operations performed by middleboxes, namely:

- NAT
- Sequence number rewriting
- Removing TCP options
- Payload modification
- Segment splitting
- Segment coalescing
- Pro-active acking

The simplest middleboxes are NATs and those that rewrite sequence numbers: beyond implementing the basic MPTCP design, no special code is required to support them. Middleboxes may also remove TCP options. If a middlebox removes the MP_CAPABLE option from

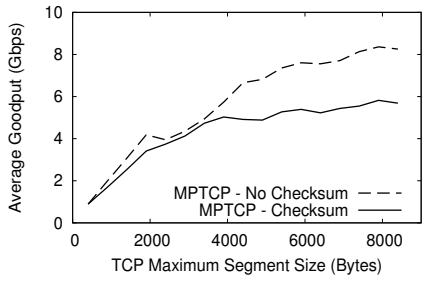


Figure 3: Impact of enabling or disabling DSM checksums in 10G environments.

the SYN or SYN/ACK, MPTCP is not used for the connection. If a middlebox removes the MPTCP option from non-SYN segments, our implementation falls back to regular TCP and continues the data transfer.

We also considered the impact of middleboxes that split or coalesce segments. NICs that support TCP Segmentation Offload (TSO) are an example of the former and traffic normalizers [8] are an example of the latter. Our implementation supports both. However, coalescing middleboxes cause a performance degration due to the loss of data sequence mappings that force the sender to retransmit data. In reality though, we have not observed any middleboxes that coalesce segments with unknown options.

Application-level gateways[22] are the most difficult middleboxes to support; they modify the payload and adjust TCP sequence numbers to compensate. MPTCP uses the DSM checksum to detect these. If we detect a DSM-checksum failure on only one subflow, that subflow is reset and the transfer continues on another subflow. If the middlebox affects all subflows, our implementation falls back to regular TCP.

Unfortunately, calculating checksums may affect performance. To evaluate this impact, we used Xeon class servers attached to 10 Gbps Ethernet interfaces. Figure 3 shows the MPTCP goodput as a function of MSS. Checksum offloading is not yet supported in our code, so the per-packet checksums are computed in software. With the default Ethernet MSS, the performance is limited by per-packet costs such as interrupt processing.

As the MSS increases, the fixed per-packet costs have less impact and goodput increases. When DSM checksums are switched off, our implementation uses the NIC to offload checksum calculations at the sender and receiver. When DSM checksums are enabled, the sender must calculate the DSM checksum in software and the receiver must check it. With jumbo frames, these checksums reduce throughput by 30%.

4.2 Minimizing memory usage

TCP and MPTCP provide in-order, reliable delivery of data to the application. The network can reorder packets or lose them, so the receiver must buffer out-of-order packets before sending a cumulative ACK and passing them to the application. Consequently, the sender also allocates a similar sized pool of memory to hold in flight segments until they are acknowledged.

How big must the receive buffer be for TCP to work well? In the absence of loss, a bandwidth-delay product (BDP) of buffering is needed to avoid flow control. If, after a packet loss, we want the sender to be able to keep sending packets while in fast retransmit we need an extra BDP of receive buffer.

For MPTCP the story is a bit different. Assuming there are no losses, and no special scheduling at the sender, the receive buffer must be at least $\sum x_i RTT_{max}$ where x_i is the throughput of subflow i and RTT_{max} is the highest RTT of all the subflows. This allows all paths to keep sending while waiting for an early packet to be delivered on the slowest path. If we want to allow all paths to keep sending while any path is fast retransmitting, the buffer must be doubled: $2\sum x_i RTT_{max}$.

We first observe that, fundamentally, memory requirements for MPTCP 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 KB. MPTCP running on the same two paths will need 375 KB — nearly four times the sum of the path BDPs.

We used our Linux implementation to test this issue. Fig. 4(a) shows the throughput achieved as a function of receive-window for TCP and MPTCP running over an emulated 8Mbps WiFi-like path (base RTT 20ms, 80ms buffer) and an emulated 2Mbps 3G path (base RTT 150ms, 2s buffer).

MPTCP will send a new packet on the lowest delay link that has space in its congestion window. When there is very little receive buffer, MPTCP sends all packets over WiFi, matching regular TCP. With a bigger buffer, additional packets are put on 3G and overall throughput drops. Somewhat surprisingly, even 370KB 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 needed to avoid flow control increases.

We see that a megabyte of receive-buffer (and send-buffer) are needed for a single connection over 3G and WiFi. This is a problem, and may prevent MPTCP from being used on busy servers and memory-scarce mobile phones. TCP over WiFi even outperforms MPTCP over both WiFi and 3G when the receive buffer is less than 400KB, removing any incentive to deploy MPTCP.

In the rest of this section we outline a series of mechanisms to be implemented at the sender that allow MPTCP make the most of the memory it has available. Solutions

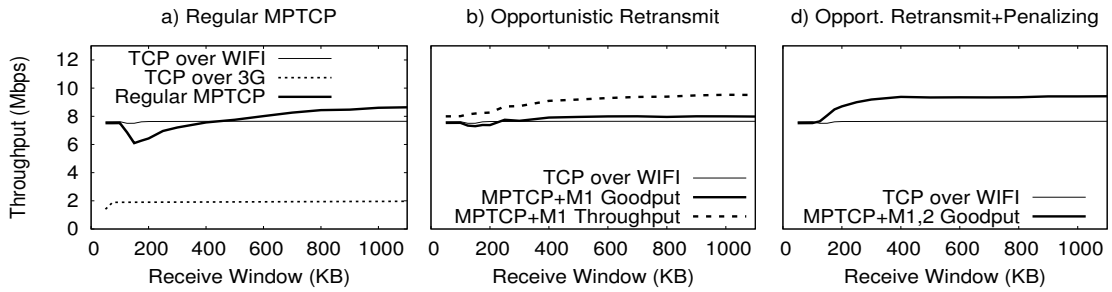


Figure 4: Receive buffer impact on throughput

have to be adaptive: as more receive buffer becomes available, MPTCP should use up more of the capacity it has available. This way, if the OS is prepared to spend the memory it will achieve higher throughput; if not, it will receive the same as TCP.

Mechanism 1: Opportunistic retransmission. When a subflow has sufficient 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 data unacknowledged data sent on the slow 3G subflow.

The motivation is that this allows the fast path to send 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 Fig. 4(b): MPTCP throughput is almost always as good as TCP over WiFi, and mostly it is better.

Unfortunately opportunistic retransmission is rather wasteful of capacity when underbuffered, as it unnecessarily pushes 2Mbps traffic over 3G; this accounts for the difference between goodput and throughput in Fig.4(b).

Mechanism 2: Penalizing slow subflows. 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 RTT on the subflow that is holding up the advancement of the window. To do this, MPTCP can reduce that subflow's window; in our tests we halved the congestion window and set the slowstart threshold to the reduced window size. To avoid repeatedly penalizing the same flow, only one reduction is applied per subflow round-trip time.

Penalizing and opportunistic retransmission work well together, as seen in Fig. 4(c): MPTCP always outperforms or a least matches TCP over WiFi.

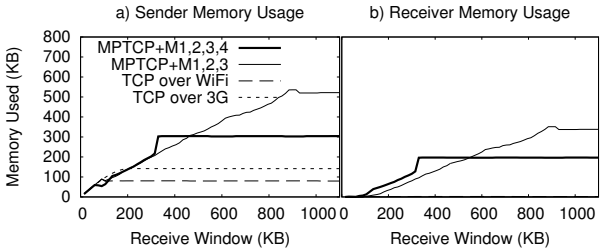


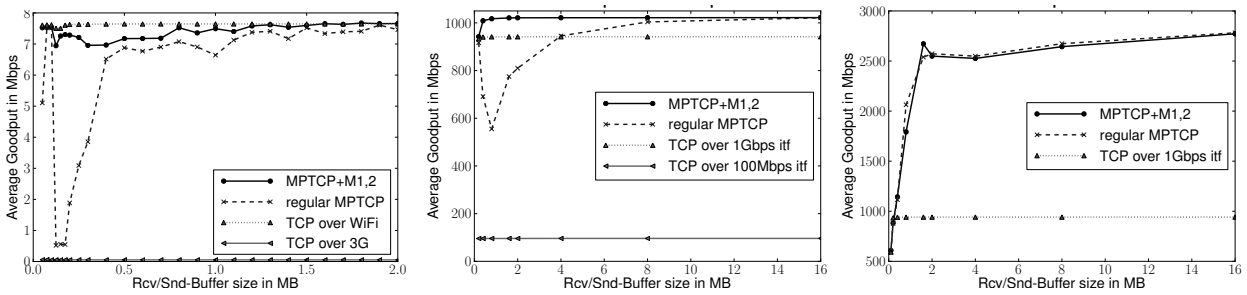
Figure 5: Receive buffer impact on memory use

Mechanisms 3 & 4: Buffer autotuning with capping. Taken together, mechanisms 1 & 2 allow an MPTCP sender to effectively utilize whatever receive buffer the receiver makes available. However, modern TCP implementations don't just blindly allocate large buffers from the outset - they adaptively increase the buffer size as they discover more buffer is needed. We've implemented both send and receive buffer tuning, done using the MPTCP buffer size formula above. In our experiments, we set the maximum send and receive buffers with the usual sysctls, but it is autotuning that automatically increases the actual buffer.

With TCP it is generally safe to configure large maximum send and receive buffer sizes; autotuning ensures they won't be used unless they are really needed. With MPTCP, however, if one of the subflows is on a path with excessive network buffering, as is common with 3G providers, autotuning will measure a large value for RTT_{max} and ramp up the receive buffer size unnecessarily. Mechanisms 1 & 2 only kick in once the receive buffer has grown and then been filled.

To see the effect of buffer autotuning, in Fig. 5 we use *htsim* to simulate the average memory consumption as a function of configured maximum receive buffer for WiFi and 3G. Memory consumption at the sender is lowest for TCP over WiFi, where the BDP is lowest. TCP over 3G has higher consumption, and MPTCP uses up to 500KB when the configured receive-buffer permits it.

This is more than MPTCP really needs; most of the time it is unnecessary to fill the large buffers on the 3G link. To avoid this effect, we might *cap* the congestion window when the amount of data buffered is above one



(a) WiFi and a lossy 3G path (b) 1Gbps and 100Mbps links (c) Three 1Gbps links

Figure 6: The Receive-buffer optimizations significantly improve goodput with small buffers

BDP. This is easy to implement: measure the base RTT of the subflow's path by taking the min of all RTT measurements, and cap $cwnd$ when the smoothed RTT is double the base RTT. In this simulation, capping halves MPTCP's memory usage when the configured receive buffer is large. Our Linux implementation does not yet support capping, but FreeBSD's regular TCP implementation has supported this since 2002, enabled via the net.inet.tcp.inflight.enable sysctl[6].

Despite the large advertised receive window, actual memory consumption at the receiver is small for single path TCP on both 3G and WiFi so long as losses are rare and the receiving application reads as soon as data is available. The same is not true for MPTCP: the receiver will spend at least two thirds of the memory the sender spends, due to reordering induced by the use of multiple paths. The effect is pronounced in the example we chose, where the difference between WiFi and 3G RTTs is seven-fold. For equal delay paths, MPTCP's receiver memory consumption is also close to zero.

4.2.1 Further evaluation

To evaluate our algorithms, we used both our *htsim* simulator and our Linux kernel implementation. We used simulation to test the viability of our proposals and their sensitivity to a wide range of path properties. The sensitivity analysis showed that the algorithms are robust and work well in a wide range of scenarios. We also tested MPTCP competing with single path TCP flows and found that MPTCP does get the same throughput as TCP on the best path or strictly better in the vast majority of cases. MPTCP does underperform TCP by 20%-30% when the best subflow experiences frequent timeouts; however, this is not caused by the receive-buffer algorithms, but by MPTCP's congestion controller overestimating the throughput of subflows that experience loss rates of greater than 10%.

To illustrate more clearly the impact of mechanisms 1 and 2, 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 6(a) shows throughput achieved using our Linux implementation on an emulated WiFi path (8Mbps, 20ms RTT, 80ms buffer) and an emulated very slow 3G link (50Kbps, 150ms RTT, 2s buffer). As the link is so slow, the loss rate will be high on the 3G path, and the large network buffer means that retransmission over 3G takes a long time. With receive buffer sizes of less than 400KB, whenever a loss happens on 3G, regular MPTCP ends up flow-controlled, unable to send on the fast WiFi path. MPTCP plus mechanisms 1 and 2 is able to avoid this being 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 quickly. With receive buffer sizes around 200KB, these mechanisms increase MPTCP throughput tenfold.

Next, we use two hosts connected by one gigabit and one 100Mb/s link to emulate inter-datacenter transfers with asymmetric links. Fig. 6(b) shows that MPTCP+M1,2 is able to utilize both links using only 250KB of memory, while regular MPTCP underperforms TCP over the 1Gbps interface until the receive buffer is at least 2MB.

When the hosts are connected via symmetric links—we used three such links in Figure 6(c)—both regular MPTCP and MPTCP+M1,2 perform equally well, regardless of the receive buffer size. This is because in this scenario, when underbuffered, 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 MPTCP 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 an application that sends 8 KByte blocks of data and timestamps each block's transmission and reception. This allows us to measure the variation of the end-to-end delay as seen by the application.

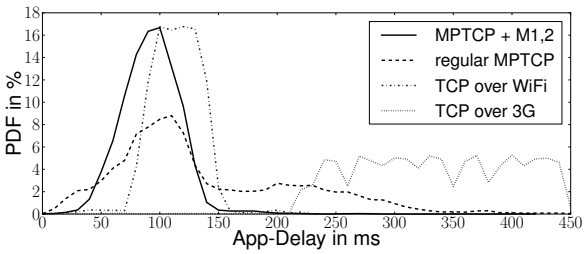


Figure 7: Application level latency for 3G/WiFi case

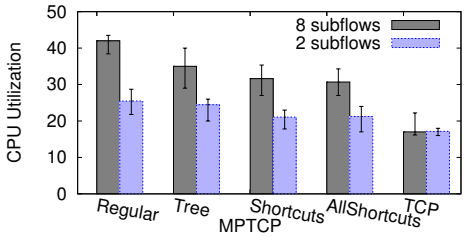


Figure 8: Effect of ofo receive algorithms on load

Figure 7 shows the probability density function of the application-delay with a buffer-size of 200KB running over 3G and WiFi. Mechanisms 1 and 2 do a good job of avoiding the larger latencies seen with regular MPTCP. Somewhat counter intuitively, the latency of TCP over WiFi is actually greater than MPTCP+M1,2. The reason for this is that 200KB is more buffering than TCP needs over this path, so the data spends much of the time waiting in the send buffer. MPTCP'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 MPTCP.

4.3 Coping with reordering

Most TCP implementations support Van Jacobson's fast path processing. The receiver assumes that data is received in-order and TCP quickly places the data received in-sequence in its receive buffer, either at the end of the in-order receive-queue (which the app can read) or at the end of the out-of-order queue. The latter happens when a packet was lost and we are waiting for the retransmission. In the rare case when a segment is received out of order, TCP scans the out-of-order queue to find the exact location of the received data.

With MPTCP the situation is completely different: while subflow sequence numbers are received in-order, data sequence numbers are often out-of-order forcing receivers to scan the large out-of-order queue. An obvious fix is to use a binary tree to reduce the out-of-order queue lookup time. This adds complexity to the code, and still takes logarithmic time to place a packet.

To obtain a simple, constant-time receive algorithm we leverage the way packets are sent: when a subflow is ready to send data, segments with contiguous data sequence numbers (a batch) will be allocated by the connection and sent on this subflow, as allowed by the subflow's congestion window. Each subflow, then, will receive in-order at the data level as long as the batch size is large. The receiver augments each subflow's data structures with a pointer to the connection-level out-of-order queue where it expects the next segment of that subflow to arrive. If the pointer is wrong, we revert to scanning the whole out-of-order queue.

This gives big benefits; the shortcuts work for 80% of the received packets. However, when the batch size is very small this optimization might not be enough. It is also not enough when we are receive-window limited and our retransmission mechanism kicks in.

For the 20% of the cases where the shortcut is not working, the receiver has to iterate over all packets in the out-of-order queue to insert the received packet. To avoid this behaviour, we modify the lookup mechanism as follows. First, the out-of-order queue groups in-sequence segments into batches. Then, we iterate over these batches instead of iterating over all the segments. As there are significantly less batches than packets in the out-of-order queue, the lookup process will be much faster.

We evaluate these algorithms by considering a client directly connected to a server by using two 1 Gbps links. The client starts a long download and we measure the receiver's CPU load. With more subflows the number of out-of-sequence segments that need to be processed increases; for clarity, we only present results with 2 subflows, a lower bound to utilize the links, and 8 subflows beyond which results are similar.

Figure 8 compares CPU load for the different receive algorithms. TCP (with 2 and 8 connections) is used as a benchmark. The *Tree* algorithm reduces CPU utilization, but *Shortcuts* and its improvement *AllShortcuts* help much more. When 8 subflows are used, CPU utilization drops from 42% to 30%, and when 2 subflows are used it drops from 25% to 20%.

5. MPTCP PERFORMANCE

The two main motivations to deploy MPTCP today are wireless networks where MPTCP could enable hosts to use both WiFi and 3G networks [20, 18] and datacenters where MPTCP allows servers to better exploit the load-balanced paths [19]. We experimentally evaluate the performance of our MPTCP implementation in these two environments.

5.1 MPTCP over WiFi and 3G

In the previous section, we used emulated networks to improve the algorithms used in our MPTCP implementation. Here, we use MPTCP over a 3G network offered

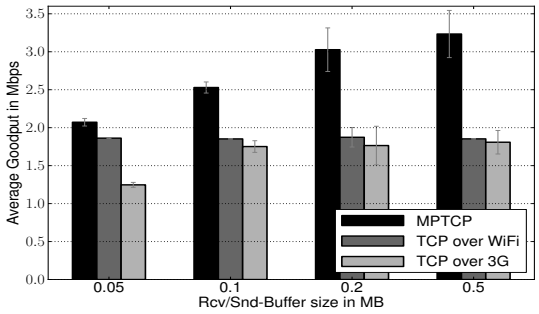


Figure 9: MPTCP used over real 3G and WiFi

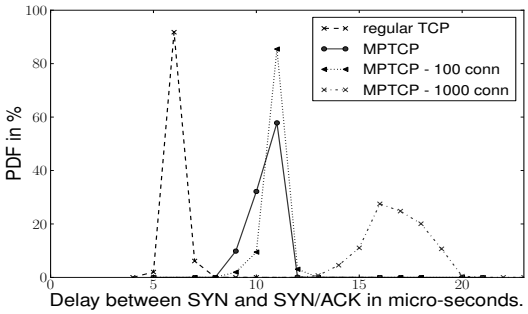


Figure 10: Connection establishment latency

by a commercial provider in Belgium; TCP's maximum throughput on this network is 2Mbps. Our MPTCP implementation correctly works over this 3G network despite its installed middleboxes. We also used a WiFi access point that was capped at 2 Mbps. This capping was implemented on the access point and would represent the bandwidth offered on a shared public WiFi network such as BT's FON. Figure 9 shows the average goodput achieved by TCP and MPTCP in function of the receive/send buffer sizes. Regular TCP achieves roughly the same goodput with both 3G and WiFi except when the buffer size is small where the larger round-trip-time penalizes the performance over 3G. MPTCP gets most of the available bandwidth when the buffer reaches 200KB, and it never underperforms TCP.

Our measurements show that MPTCP is able to utilize both the 3G and WiFi networks when the buffer is large enough. With a 500 KBytes buffer, MPTCP achieves almost the double of the goodput of regular TCP. With a 100 KBytes buffer, reaches a goodput that is 25% larger than regular TCP over WiFi or 3G.

5.2 Connection setup latency

During MPTCP connection setup the client and server generate a random key and verify that its hash is unique among all established connections (see Section 3.2). These keys are used later to verify the addition of new subflows. How does this affect connection setup latency?

The measurements use Xeon X5355 servers connected via Gigabit ethernet. Fig. 10 shows a PDF of the delay between receiving a SYN and sending the SYN/ACK, measured at the server. For regular TCP, 91% of the 20,000 connection setup attempts are processed in 6μs. Each connection is closed before the next attempt is made.

Setting up the first subflow of an MPTCP connection takes the server between 10 and 11μs if it has no established connections. The extra latency is mainly because MPTCP must hash the received key, generate the server key and verify that its hash is unique. If the server has established MPTCP connections, the verification of hash uniqueness is more expensive — Fig. 10 shows how the latency increases when the server already has 100 and 1000 established MPTCP connections.

This additional latency, although small compared to a LAN RTT, could be significantly reduced by maintaining a pool of precomputed keys.

5.3 HTTP performance

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

We directly connected a client and server via two gigabit links. For our tests we use `apachebench` [7], 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 (requests are closed-loop). The server was running MPTCP Linux and used `apache` version 2.2.16 with the default configuration.

We tested regular TCP that uses a single link, TCP with link-bonding using both interfaces and finally MPTCP. Fig. 11 shows the number of requests per second served in all three configurations. We expect MPTCP to be significantly better than regular TCP, and indeed this shows up in experiments: when the file sizes are larger than 100 KBytes MPTCP doubles the number of requests served. With files that are shorter than 30 KBytes, MPTCP 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 slowstart.

TCP with link-bonding performs very well especially when file sizes are small: the round-robin technique used

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

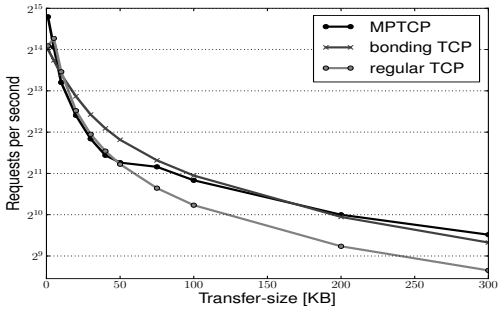


Figure 11: Apache-benchmark with 100 clients

by the Linux implementation manages to spread the load evenly, utilizing all the available capacity. MPTCP has a slight advantage over TCP with link-bonding only when file sizes are greater than 150KB 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. We ran experiments with varying levels of congestion and found that MPTCP can serve 25% more requests than link bonding in such cases.

6. RELATED WORK

There has been a good deal of work on building multipath transport protocols[11, 24, 16, 10, 13, 4, 21, 5]. Most of this work aims to leave applications unchanged and focuses on the protocol mechanisms needed to implement multipath transmission. Key goals are robustness to long term path failures and to short term variations in conditions on the paths.

Huitema's Internet Draft [11] proposes using PCB identification to replace ports as demultiplexing points at the end-hosts; our connection tokens are similar in spirit. The proposal stripes segments over many addresses, using a single sequence number across all subflows.

Both MTCP [24] and M/TCP [21] use a single sequence number together with a scoreboard at the sender that allows maintaining congestion state and performing retransmissions per path. MTCP uses a single return path for ACKs which decreases its robustness; also it has been designed to run over an overlay network (RON), reducing its deployability and efficiency.

pTCP [10] is one of the most complete proposals to date. The SYN exchange signals the addresses that will be used in the multipath connection, and this set is fixed - pTCP does not support mobility. Congestion control and retransmissions are performed per subflow. At connection level there are global send and receive buffers; a data sequence number and acknowledgment helps deal with reordered data at the connection level. This is inserted in a pTCP header that follows the TCP header.

RMTP [16] is a rate-based protocol targeted for mobile hosts that uses packet-pairs to estimate available bandwidth on each path and supports both reliable an unreliable delivery. RMTP does not offer the same service as TCP, and requires app changes.

The Stream Control Transmission Protocol (SCTP) has been designed with multihoming in mind to support telephony signaling applications. SCTP's multihoming support was initially only capable of recovering from failures. However, several authors have extended it to support load-sharing [2, 13]. SCTP-CMT [12] uses a single sequence number across all paths and keeps a scoreboard and other information to have accurate per-path congestion windows and to drive retransmissions.

Our protocol design has drawn on all this literature, and has been further guided by our experimental study of middleboxes. In light of this study, none of the existing approaches are deployable as most use single subflow sequence numbers which will be dropped. pTCP does not use subflow sequence numbers but it is unclear how its additional headers should be encoded. Further, pTCP will not cope with resegmenting or content-changing middleboxes.

On the OS side, none of the previous works on TCP have addressed the practical problems of getting multipath transport working in reality. Most use simulation analysis, and do not consider receive-buffer issues. SCTP-CMT has been implemented in the FreeBSD kernel, but its performance has not been evaluated in detail.

A technique that was previously proposed to reduce the size of the receive-buffer is to use sender-side scheduling to get the packets "in-order" at the receiver (see e.g. [17]). Unfortunately, this solution is brittle: any packet losses or just variations in RTT will disrupt the ordering, causing the receiver to buffer just as much data. Further, the sender still has to buffer as much data as before.

7. LESSONS LEARNED

In today's Internet, the three-way-handshake involves not only the two communicating hosts, but also all the middleboxes on the path. Verifying the presence of a particular TCP option in a SYN+ACK is not sufficient to ensure that a TCP extension can be safely used. As shown in [9], some middleboxes pass TCP options that they don't understand. This is safe for TCP options that are purely informative (e.g. RFC1323 timestamps) but causes problems with other options such as those that redefine the semantics of TCP header fields. For example, the large window extension in RFC1323 changes

the semantics of the window field of the TCP header and extends it beyond 16 bits. Nearly 20 years after the publication of RFC1323, there are still stateful firewalls that do not understand this option in SYNs but block data packets that are sent in the RFC1323 extended window. A TCP extension that changes the semantics of parts of the packet header must include mechanisms to cope with middleboxes that do not understand the new semantics.

In an end-to-end Internet, all the information carried inside TCP packets is immutable. Today this is no longer true: the entire TCP header and the payload must be considered as mutable fields. If a TCP extension needs to rely on a particular field, it must check its value in a way that cannot be circumvented by middleboxes that do not understand this extension. The DSM checksum is an example of a solution to deal with these problems.

Most importantly, deployable TCP extensions must necessarily include techniques that enable them to fallback to regular TCP when something wrong happens. If a middlebox interferes badly with a TCP extension, the problem must be detected and the extension automatically disabled to preserve the data transfer. A TCP extension will only be deployed if it guarantees that it will transfer data correctly (and hopefully better) in all the cases where a regular TCP is able to transfer data.

8. CONCLUSIONS

TCP was designed when the Internet strictly obeyed the end-to-end principle and each host had a single IP address. Single-homing is disappearing and a growing fraction of hosts have multiple interfaces/addresses. In this paper we evaluated whether TCP can be extended to efficiently support such hosts.

We explored whether it was possible to design Multipath TCP in a way that is still deployable in today's Internet. The answer is positive, but any major change to TCP must take into account the various types of middleboxes that have proliferated. In fact, they influenced most of the design choices in Multipath TCP besides the congestion control. Our experiments show that MPTCP safely operates through all the middleboxes we've identified in our previous study [9].

From an implementation viewpoint, we proposed new algorithms to solve practical but important problems such as sharing a limited receive buffer between multiple flows on a smartphone, or optimizing the MPTCP receive code. Experiments show that our techniques are effective, making MPTCP ready for adoption.

This work highlights once again the fact that hidden middleboxes increase the complexity of the Internet, making evolution difficult. We should revisit the Internet architecture to recognize explicitly their role. The big challenge, however, is to build a solution that is deployable in today's Internet.

Acknowledgments

We would like to thank all the members of the Trilogy project and of the MPTCP IETF working-group who have helped us to shape the MPTCP protocol. Xinxing Hu wrote the first version of the Click elements that model middlebox behaviours.

This work has been supported partially by Google through a University Research project and by FP7 projects ECODE and CHANGE. Christoph Paasch has partially been supported by Nokia Research.

9. REFERENCES

[1] S. Barré. *Implementation and Assessment of Modern Host-based Multipath Solutions*. PhD thesis, Université catholique de Louvain, November 2011.

[2] M. Becke et al. Load sharing for the stream control transmission protocol (sctp). Internet draft, draft-tuexen-tsvwg-sctp-multipath-02.txt, work in progress, July 2011.

[3] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh. The case for ubiquitous transport-level encryption. In *USENIX Security'10*, pages 26–26, Berkeley, CA, USA, 2010. USENIX Association.

[4] Y. Dong, D. Wang, N. Pissinou, and J. Wang. Multi-path load balancing in transport layer. In *EuroNGI Conference*, 2007.

[5] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP extensions for multipath operation with multiple addresses, Jan 2012. IETF draft (work in progress).

[6] FreeBSD Project. tcp – internet transmission control protocol. *FreeBSD Kernel Interfaces Manual*.

[7] M. Handley. Why the internet only just works. *BT Technology Journal*, 24:119–129, 2006.

[8] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, pages 9–9, 2001.

[9] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *IMC 2011, 11th Internet Measurement Conference*, Nov. 2011.

[10] H.-Y. Hsieh and R. Sivakumar. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *Proc. MobiCom '02*, pages 83–94, New York, NY, USA, 2002. ACM.

[11] C. Huitema. Multi-homed TCP. Internet draft, IETF, 1995.

[12] J. Iyengar, P. Amer, and R. Stewart. Performance implications of a bounded receive buffer in concurrent multipath transfer. *Computer Communications*, 30(4), February 2007.

[13] J. R. Iyengar, P. D. Amer, and R. Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Trans. Netw.*, 14(5):951–964, 2006.

[14] P. Key, L. Massoulié, and D. Towsley. Path selection and multipath congestion control. In *Proc. IEEE Infocom*, May 2007.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18:263–297, August 2000.

[16] L. Magalhaes and R. Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. *ICNP*, page 0165, 2001.

[17] F. Mirani, M. Kherraz, and N. Boukhatem. Forward prediction scheduling: Implementation and performance evaluation. In *ICT 2011*, pages 321 –326, may 2011.

[18] C. Pluntke, L. Eggert, and N. Kiukkonen. Saving mobile device energy with Multipath TCP. In *MobiArch*, 2011.

[19] C. Raiciu, S. Barré, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with Multipath TCP. In *Proc ACM Sigcomm*, 2011.

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

[21] K. Rojviboonchai and H. Aida. An evaluation of multi-path transmission control protocol (M/TCP) with robust acknowledgement schemes. *IEICE Trans. Communications*, 2004.

[22] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) terminology and considerations. RFC 2663, August 1999.

[23] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI'11*, Berkeley, CA, USA, 2011. USENIX Association.

[24] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proc USENIX '04*, 2004.