MultiPath TCP: From Theory to Practice

Sébastien Barré, Christoph Paasch, and Olivier Bonaventure [*] [**]

ICTEAM, Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium
{firstname.lastname}@uclouvain.be

Abstract. The IETF is developing a new transport layer solution, MultiPath TCP (MPTCP), which allows to efficiently exploit several Internet paths between a pair of hosts, while presenting a single TCP connection to the application layer. From an implementation viewpoint, multiplexing flows at the transport layer raises several challenges. We first explain how this major TCP extension affects the Linux TCP/IP stack when considering the establishment of TCP connections and the transmission and reception of data over multiple paths. Then, based on our implementation of MultiPath TCP in the Linux kernel, we explain how such an implementation can be optimized to achieve high performance and report measurements showing the performance of receive buffer tuning and coupled congestion control.

Keywords: TCP, multipath, implementation, measurements

1 Introduction

The Internet is changing. When TCP/IP was designed, hosts had a single interface and only routers were equipped with several physical interfaces. Today, most hosts have more than one interface and the proliferation of smart-phones equipped with both 3G and WiFi will bring a growing number of multihomed hosts on the Internet. End-users often expect that using multihomed hosts will increase both redundancy and performance. Unfortunately, in practice this is not always the case as more than 95% of the total Internet traffic is still driven by TCP and TCP binds each connection to a single interface. This implies that TCP by itself is not capable of efficiently and transparently using the interfaces available on a multihomed host.

The multihoming problem has received a lot of attention in the research community and within the IETF during the last years. Network layer solutions

[*] This work is supported by the European Commission via the 7th Framework Programme Integrated Project TRILOGY

such as shim6 [15] or the Host Identity Protocol (HIP) [14] have been proposed and implemented. However, they remain experimental and it is unlikely that they will be widely deployed. Transport layer solutions have also been developed, first as extensions to TCP [13, 7, 18, 23]. However, to our knowledge these extensions have never been implemented nor deployed. The Stream Control Transmission Protocol (SCTP) [19] protocol was designed with multihoming in mind and supports fail-over. Several SCTP extensions [8, 12] enable hosts to use multiple paths at the same time. Although implemented in several operating systems [8], SCTP is still not widely used besides specific applications. The main drawbacks of SCTP on the global Internet are first that application developers need to change their application to use SCTP. Second, various types of middle-boxes such as NATs or firewalls do not understand SCTP and block all SCTP packets.

During the last two years, the MPTCP working group of the IETF has been developing multipath extensions to TCP [6] that enable hosts to use several paths possibly through multiple interfaces, to carry the packets that belong to a single connection. This is probably the most ambitious extension to TCP to be standardized within the IETF. As for all Internet protocols, its success will not only depend on the protocol specification but also on the availability of a reference implementation that can be used by real applications.

In this paper we explain the challenges in implementing MPTCP in the Linux kernel and evaluate its performance based on lab measurements. This is the first implementation of this major TCP extension in an operating system kernel.

The paper is organized as follows. In section 2, we briefly describe MPTCP and compare it to regular TCP. Then we describe the architecture of the implementation. In section 4, we describe a set of general problems that must be solved by any protocol wanting to simultaneously use several paths, together with the chosen solutions. Next, we measure the performance of the implementation in different scenarios to show how the implementation-choices taken do influence the results.

2 MultiPath TCP

MultiPath TCP [6] is different from existing TCP extensions like the large windows, timestamps or selective acknowledgement extensions. These older extensions defined new options that slightly change the reaction of hosts when they receive segments. MultiPath TCP allows a pair of hosts to use several paths to exchange the segments that carry the data from a single connection.

To understand the operation of MultiPath TCP, let us consider a very simple case where a client having two addresses, A.1 and A.2 establishes a connection with a single homed server. The client first sends a SYN segment from address A.1 that contains the MP_CAPABLE option [6]. This option indicates that the client supports MultiPath TCP and contains a token that identifies the MultiPath TCP connection. The server replies with a SYN+ACK segment containing the same option and its own token. The client concludes the three-way handshake. Once the TCP connection has been established, the client can advertise its other addresses by sending TCP segments with the `ADD_ADDR` option. It can also open a second TCP connection, called a subflow, that will be linked to the first one by sending a `SYN` segment with the `MP_JOIN` option that contains the token sent by the server in the initial subflow. The server replies by sending a `SYN+ACK` segment with the `MP_JOIN` option containing the token chosen by the client in the initial subflow and the client terminates the three-way handshake. These two subflows are linked together inside a single MultiPath TCP connection and both can be used to send data. Subflows may fail and be added during the lifetime of a MultiPath TCP connection. Additional details about the establishment of MultiPath TCP connections and subflows may be found in [6].

The data produced by the client and the server can be sent over any of the subflows that compose a MultiPath TCP connection, and if a subflow fails, data may need to be retransmitted over another subflow. For this, MultiPath TCP relies on two principles. First, each subflow is equivalent to a normal TCP connection with its own 32-bits sequence numbering space. This is important to allow MultiPath TCP to traverse complex middle-boxes like transparent proxies or traffic normalizers. Second, MultiPath TCP maintains a 64-bits data sequence numbering space. Two TCP options use these data sequence numbers : `DSN_MAP` and `DSN_ACK`. When a host sends a TCP segment over one subflow, it indicates inside the segment, by using the `DSN_MAP` option, the mapping between the 64-bits data sequence number and the 32-bits sequence number used by the subflow. Thanks to this mapping, the receiving host can reorder the data received, possibly out-of-sequence over the different subflows. In MultiPath TCP, a received segment is acknowledged at two different levels. First, the TCP cumulative or selective acknowledgements are used to acknowledge the reception of the segments on each subflow. Second, the `DSN_ACK` option is returned by the receiving host to provide cumulative acknowledgements at the data sequence level. When a segment is lost, the receiver detects the gap in the received 32-bits sequence number and traditional TCP retransmission mechanisms are triggered to recover from the loss. When a subflow fails, MultiPath TCP detects the failure and retransmits the unacknowledged data over another subflow that is still active.

Another important difference between MultiPath TCP and regular TCP is the congestion control scheme. MultiPath TCP cannot use the standard TCP control scheme without being unfair to normal TCP flows. Consider two hosts sharing a single bottleneck link. If both hosts use regular TCP and open one TCP connection, they should achieve almost the same throughput. If one host opens several subflows for a single MultiPath TCP connection that all pass through the bottleneck link, it should not be able to use more than its fair share of the link. This is achieved by the coupled congestion control scheme that is discussed in details in [17, 22]. The standard TCP congestion control [1] increases and decreases the congestion window and slow-start threshold upon reception of acknowledgments and detection of losses. The coupled congestion control scheme also relies on a congestion window, but it is updated according to the following principle:

- For each non-duplicate ack on subflow i, increase the congestion window of the subflow i by $min(\alpha/cwnd_{tot}, 1/cwnd_i)$ (where $cwnd_{tot}$ is the total congestion window of all the subflows and $\alpha = cwnd_{tot}\frac{\max_i(\frac{cwnd_i * mss_i^2}{RTT_i^2})}{(\sum_i \frac{cwnd_i * mss_i}{RTT_i})^2}$).
- Upon detection of a loss on subflow i, decrease the subflow congestion window by $cwnd_i/2$.

3 Linux MultiPath TCP architecture

Our Multipath architecture for the Linux kernel is based on an important rethinking of the building blocks that make the current Linux TCP stack (the current Linux patch has more than 12000 lines[1]). Of major importance is the separation between *user context* and *interrupt context.* A piece of kernel code runs in *user context* if it is performing a task for a particular application, and can be interrupted by the OS scheduler. All system calls (like `send, rcv`) are run in user context. Being under control of the scheduler, user context activities are fair to other applications. On the other hand, *software interrupts* are run with higher priority than any process, and follow an event (e.g. timer expiry, packet reception). They can be interrupted only by *hardware interrupts*, and must thus be as short as possible. Unfortunately the handling of incoming packets happens in a software interrupt, and the TCP reordering is normally done there as well, because it is needed to acknowledge as fast as possible. To allow spending less time in software interrupts, and hence gaining in overall system responsiveness, Van Jacobson proposed to force reordering in the user context when the application is waiting in a `receive` system call [10]. Instead of reordering itself, the interrupt then puts the segments in a so-called *prequeue*, and wakes up the process, that performs reordering and sends acknowledgements. More details on the Linux networking architecture can be found in [20].

As shown in figure 1, our proposed architecture for multipath is made of three elements. The first element is the **master subsocket**. If MPTCP is not supported by the peer, only that element is used and regular TCP is executed. The master subsocket is a standard socket structure that provides the interface between the application and the kernel for TCP communication. The second building block is the **Multi-path control bock (mpcb)**. This building block supervises the multiple flows used in a given connection, it runs the *decision algorithm* for starting or stopping subflows (based on local and remote information), the *scheduling algorithm* for feeding new application data to a particular subflow, and the *reordering algorithm* that allows providing an in-order stream of incoming bytes to the application. Note that reordering at the subflow-level is performed in the subflows themselves, this service being implemented already as part of the underlying TCP. The third building block is the **slave subsocket**. Slave subsockets are not visible to the application. They are initiated, managed and closed by the multipath control block. They otherwise share functionality

[1] Available at `http://inl.info.ucl.ac.be/mptcp/`

with the master subsocket. The master subsocket and the slave subsockets together form a pool of subflows that the mpcb can use for scheduling outgoing data, and reordering incoming data. Should the master subsocket fail, it keeps its role of contact point between the application and the kernel, but simply it stops being used as an eligible subflow by the MPTCP scheduler, until it recovers.

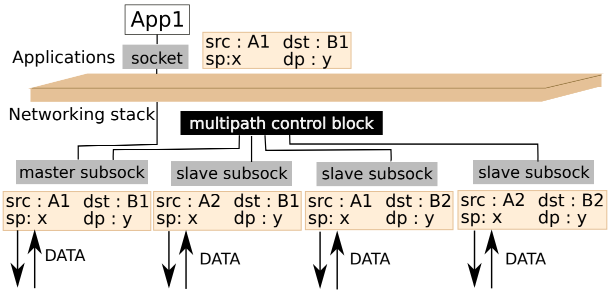

Fig. 1. Overview of the multipath architecture

Connection establishment: The initiator of a TCP connection uses the initial subflow to learn from the peer its set of addresses, as defined in [6]. It then combines those addresses to try establishing subflows on every known path to the peer. The case of the receiver is more tricky: We want to be able to create new subsockets without relying on the usual *listen*. The reason is that the key to retrieve a listener is no longer the usual TCP 5-tuple, but instead a token attached in a TCP option. We solve this by attaching an accept queue to the multipath control block. The `mpcb` is found thanks to a token-based hash table lookup, and a new half-open socket is appended in the accept queue. Only when the new subsocket becomes established is it added to the list of active subsockets.

Scheduling and sending data: If more than one subflow is in *established* state, MPTCP must decide on which subflow to send data. The current policy implemented in the scheduler tries to fill all subflows, as described later in this paper. However, the scheduler is modular and other policies like preferring one interface over the other could be implemented in the future.

The scheduler must deal with the granularity of the allocations, that is, the number of contiguous bytes that are sent over the same subflow before deciding to select another subflow. This granularity influences the performance of a MultiPath TCP implementation with high bandwidth flows. The optimal use of all subflows would be an argument in favor of small allocation units. On the other hand, to spare CPU cycles and memory accesses, one would tend to allocate in large units (because that would result in fewer calls to the scheduler, and would facilitate segmentation offload). In this paper, we favor an optimal allocation of segments, deferring a full study of the performance trade-offs for another paper.

We have examined two major design options for the scheduler. In the first design, whenever an application performs a `sendmsg()` system call or equivalent, the scheduler is invoked and data is immediately pushed to a specific subflow. This implies per-subflow send buffers, and has the advantage of preformatting segments (with correct MSS and sequence numbers) so that they can be sent very quickly each time an acknowledgement opens up more space in the congestion window. However, there may be several hundreds of milliseconds between the time when a segment is enqueued on a subflow, and its actual transmission on the wire. The path properties may change during that time and what was a good choice when running the scheduler may reveal a very wrong choice when the data is put on the wire. Even worse is the case of a failing subflow, because a full buffer of segments needs to be moved to another subflow when this happens.

These major drawbacks lead us to a second design, that solves those problems. In the current implementation, the data pushed by the applications is not scheduled anymore, but instead stored in a connection-level send buffer. Subflows pull data from the shared send buffer whenever they receive an acknowledgement. This design is similar to the one proposed for pTCP in [7], but pTCP has only been simulated and not implemented. This effectively solves the problem of fluctuating path properties and allows to run the scheduler *when the segment is sent on the wire.*

Another problem that needs to be considered by an implementation is that different subflows may use different MSS. In this case, a byte-based connection-level send buffer would probably be needed. However, this would be inefficient in the Linux kernel that is optimized to handle queues of packets. To solve this problem, our implementation uses the minimum MSS over all subflows to send segments. This is slightly less efficient from an overhead viewpoint if one interface uses a much smaller MSS than the other, but in practice this is rarely the case.

To evaluate the impact of using the same MSS over all subflows, we performed a simple test by contacting the 10000 most popular web servers according to the Alexa top-10000 list. We sent a `SYN` segment advertising an MSS value of 1460, 4096 or 9148 bytes to each of these servers and analyzed the returned MSS. 97% of these servers returned an MSS of 1380 bytes without any influence of the MSS option that we included in our `SYN` segment. Most of the other web servers returned an MSS that was very close to 1380 bytes. Thus, using the same MSS for all subflows appears reasonable on today's Internet.

Receiving data: Data reception is performed in two steps. The first step is to receive data at the subflow level, and reorder it according to the 32-bits subflow sequence numbers. The second step is to reorder the data at the connection level by using the data sequence numbers, and finally deliver it to the application. As in regular TCP, each subflow maintains a `COPIED_SEQ` and a `RCV.NXT` [16] pointer, resp. to track the next byte to give to the upper layer (now the upper layer becomes the mpcb) and the next expected subflow sequence number. Likewise, the multipath control block maintains a connection level `COPIED_SEQ` and a `RCV.NXT` pointer, resp. to track the next byte to deliver to the application and the next expected data sequence number that is used when returning a DATA_ACK option.

To store incoming data until the application asks for it, we use a single connection-level receive queue. All subflow-receive queues are always empty, because as soon as a segment becomes in order at the subflow-level, it is enqueued in the connection-level receive queue, or out-of-order queue. The problem of charging the application for this additional processing time (to ensure fairness with other running processes) can be solved by using Van Jacobson's prequeues [10], just like regular TCP does. Enabled by default in current kernels, that option is even more important for MPTCP, because MPTCP involves more processing, especially with regards to reordering, as we will show in section 5.

4 Reaching full path utilization

One obvious goal of MPTCP is to be able to consider all available paths as a shared resource, just behaving as the sum of the individual resources [21]. However, from an implementation viewpoint, reaching that goal requires to deal with several new constraints. We first study the implications of MPTCP on the receive buffers and then the coupled congestion control scheme.

4.1 Receive buffer constraints

To optimize the size of the receive buffer, Fisk et al. have proposed [4] to dynamically tune the TCP receive buffer to ensure that twice the Bandwidth-Delay Product (BDP) of the path can be stored in the buffers of the receiver. That value allows supporting reordering by the network (this requires one BDP), as well as fast retransmissions (one other BDP). This buffer tuning [4] is integrated in the Linux kernel.

In the context of a multipath transfer, additional constraints appear. The problem, described in details in [9] for SCTP multipath transfer, is that a fast path can be blocked due to the receive buffer becoming full. In single-path TCP the receive buffer can become full only if the application stops consuming data. In MultiPath TCP, it can become full as well if some data coming over a path with a high delay is needed for connection-level reordering, before to be able to deliver the bytes to the application. In the worst case, each path i is continuously transmitting at a BW_i rate, while the slowest path (that is, the path with highest RTT) is fast retransmitting. To accommodate for such a situation, the receive buffer must be dimensioned as [7, 5]: $rbuf = 2 * \sum_{i \in subflows} BW_i * RTT_{max}$

Our implementation uses the technique proposed in [4] to estimate the RTT at the receiver by using the TCP timestamps, and computes the contribution of each path to the overall receive buffer. Whenever any of the contributions changes, the global receive buffer limit is updated in the mpcb. In practice, this dynamic tuning may reach the maximum allowed receive buffer configured on the system. This should be used as a hint to indicate that a subflow is underperforming and disable the slowest path.

4.2 Implementing the Coupled Congestion Control

When implementing in the Linux kernel the coupled congestion control described in section 2, several issues need to be solved. The main problem is that the Linux kernel does not support floating point numbers. This implies that increasing the congestion window by $1/cwnd_i$ is not possible, because $1/cwnd_i$ is not an integer. Our implementation counts the number of acknowledged packets and maintains this information inside a subflow-variable ($cwnd\_cnt_i$), as it is already done for other congestion control algorithms in the Linux kernel like New-Reno. Then, the congestion window is increased by one as soon as $cwnd\_cnt_i > tot_{cwnd}/\alpha$ [17].

The coupled congestion control involves the calculation of the α-factor described in section 2. Our implementation computes α as shown in equation 1. As our implementation uses the same mss for all the subflows, the formula does not need the mss anymore. rtt_{max} and $cwnd_{max}$ are precalculated from the numerator, but rtt_{max} has been put into the denominator to reduce the number of divisions in the formula. As the Linux kernel only handles fixed-point calculations, we need to scale the divisions, to reduce the error due to integer-underflow ($scale_{num}$ and $scale_{den}$). Later, the resulting scaling factor of $scale_{num}/scale_{den}^2$ has to be taken into account.

$$\alpha = cwnd_{tot} \frac{cwnd_{max} * scale_{num}}{(\sum_i \frac{rtt_{max} * cwnd_i * scale_{den}}{rtt_i})^2} \tag{1}$$

5 Evaluation

To evaluate the performance of our implementation, we performed lab measurements in the HEN testbed at University College London (`http://hen.cs.ucl.ac.uk/`). We used two different scenarios. The first scenario is used to evaluate the coupled congestion control scheme while the second analyses the factors that influence the performance of MultiPath TCP.

5.1 Measuring the Coupled Congestion Control

The first scenario is the *congestion testbed* shown in figure 2. In this scenario, we used four Linux workstations. The two workstations on the left use Intel® Xeon CPUs (2.66GHz, 8GB RAM) and Intel® 82571EB Gigabit Ethernet Controllers. They are both connected with one 1 Gbps link to a similar workstation that is configured as a router. The router is connected with a 100 Mbps link to a server. The upper workstation in figure 2 uses a standard TCP implementation while the bottom host uses our MultiPath TCP implementation.

Detailed simulations analyzed by Wischik et. al in [22] show that the coupled congestion control fulfills its goals. In this section we illustrate that our Linux kernel implementation of the coupled congestion control achieves the desired effects, even if it is facing additional complexity due to fixed-point calculations compared to the user-space implementation used in [22]. In the congestion

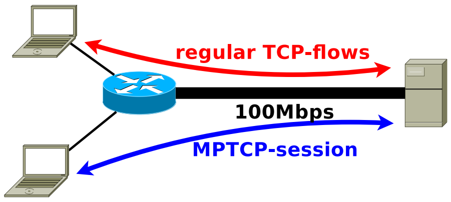

Fig. 2. Congestion testbed

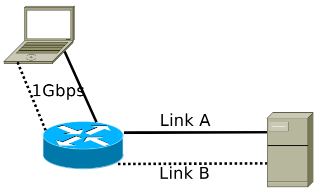

Fig. 3. Performance testbed

testbed shown in figure 2, the coupled congestion control should be fair to TCP. This means that an MPTCP connection should allow other TCP sessions to take over the bandwidth on a shared bottleneck. Furthermore, an MPTCP connection that uses several subflows should not slow down regular TCP connections. To measure the fairness of MPTCP, we use the bandwidth measurement software `iperf`[2] to establish sessions between the hosts on the left and the server on the right part of the figure. We vary the number of regular TCP connections from the upper host and the number of MPTCP subflows used by the bottom host. We ran the iperf sessions for a duration of 10 minutes, to allow the TCP-fairness over the bottleneck link to converge [11]. Each measurement runs five times and we report the average throughput.

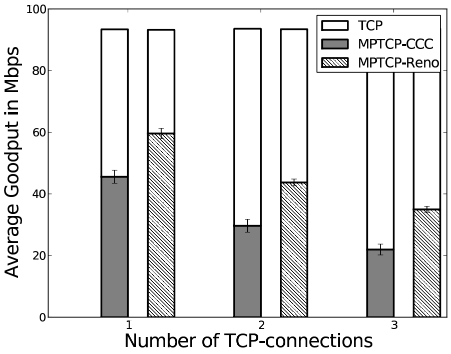

Fig. 4. MultiPath TCP with coupled congestion control behaves like a single TCP connection over a shared bottleneck with respect to regular TCP.

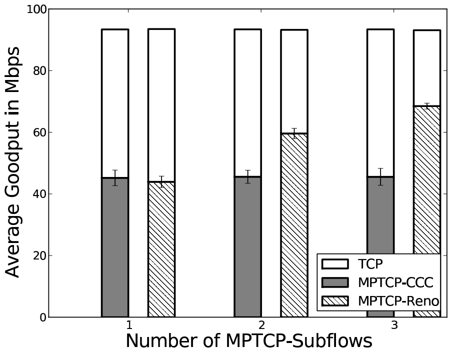

Fig. 5. With coupled congestion control on the subflows, MultiPath TCP is not unfair to TCP when increasing the number of subflows.

Thanks to the flexibility of the Linux congestion control implementation, we perform tests by using the standard Reno congestion control scheme with regular TCP and either Reno or the coupled congestion control scheme with MPTCP.

[2] http://sourceforge.net/projects/iperf/

The measurements show that MPTCP with the coupled congestion control is fair to regular TCP. When an MPTCP connection with two subflows is sharing a bottleneck link with a TCP connection, the coupled congestion control behaves as if the MPTCP session was just one single TCP connection. However, when Reno congestion control is used on the subflows, MPTCP gets more bandwidth because in that case two TCP subflows are really competing against one regular TCP flow (Fig. 4).

The second scenario that we evaluate with the coupled congestion control is the impact of the number of MPTCP subflows on the throughput achieved by a single TCP connection over the shared bottleneck. We perform measurements by using one regular TCP connection running the Reno congestion control scheme and one MPTCP connection using one, two or three subflows. The MPTCP connection uses either the Reno congestion control scheme or the coupled congestion control scheme. Figure 5 shows first that when there is a single MPTCP subflow, both Reno and the coupled congestion control scheme are fair as there is no difference between the regular TCP and the MPTCP connection. When there are two subflows in the MPTCP connection, figure 5 shows that Reno favors the MPTCP connection over the regular single-flow TCP connection. The unfairness of the Reno congestion control scheme is even more important when the MPTCP connection is composed of three subflows. In contrast, the measurements indicate that the coupled congestion control provides the same fairness when the MPTCP connection is composed of one, two or three subflows.

5.2 MPTCP performance

Our second scenario, depicted in figure 3, allows us to evaluate the factors that influence the performance of our MultiPath TCP implementation. It is composed of three workstations. Two of them act as source and destination while the third one serves as a router. The source and destination are equipped with AMD Opteron$^{\text{TM}}$ Processor 248 single-core 2.2 GHz CPUs, 2GB RAM and two Intel® 82546GB Gigabit Ethernet controllers. The links and the router are configured to ensure that the router does not cross-route traffic, i.e. the packets that arrive from the solid-shaped link in figure 3 are always forwarded over the solid-shaped link to reach the destination. This implies that the network has two completely disjoint paths between the source and the destination. We configure the bandwidth on `Link A` and `Link B` by changing their Ethernet configuration.

As explained in section 4.1, one performance issue that affects MultiPath TCP is that MultiPath TCP may require large receive buffers when subflows have different delays. To evaluate this impact, we configured `Link A` and `Link B` with a bandwidth of 100 Mbps. Figure 6 Shows the impact of the maximum receive buffer on the performance achieved by MPTCP with different delays. For these measurements, we use two MPTCP subflows, one is routed over `Link A` while the second is routed over `Link B`. The router is configured to insert an additional delay of 0, 10, 100 and 500 milliseconds on `Link B`. No delay is inserted on `Link A`. This allows us to consider an asymmetric scenario where the two subflows are routed over very different paths. Such asymmetric delays

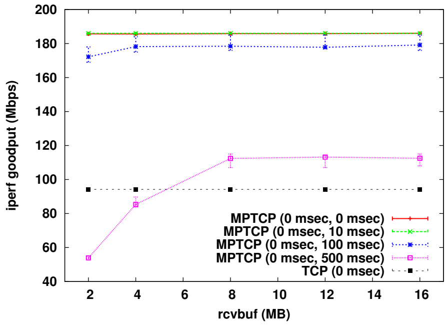

Fig. 6. Impact of the maximum receive buffer size

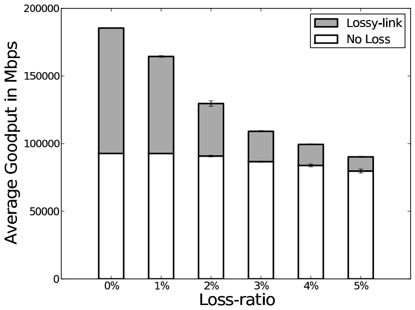

Fig. 7. Impact of the packet loss ratio

force the MPTCP receiver to store many packets in its receive buffer to be able to deliver all the received data in sequence. As a reference, we show in figure 6 the throughput achieved by `iperf` with a regular TCP connection over `Link A`. When the two subflows have the same delay, they are able to saturate the two 100 Mbps links with a receive buffer of 2 MBytes or more. When the delay difference is of only 10 millisecond, the goodput measured by `iperf` is not affected. With a difference of 100 milliseconds in delay between the two subflows, there is a small performance decrease. The performance is slightly better with 4 MBytes which is the default maximum receive buffer in the Linux kernel. When the delay difference reaches 500 milliseconds, an extreme case that would probably involve satellite links in the current Internet, the goodput achieved by MultiPath TCP is much more affected. This is mainly because the router drops packets and these losses cause timeout expirations and force MPTCP to slowly increases its congestion window due to the large round-trip-time.

A second factor that affects the performance is the loss ratio. To evaluate whether losses on one subflow can affect the performance of the other subflow due to head-of-line blocking in the receive buffer, we configured the router to drop a percentage of the packets on `Link B` but no packets on `Link A` and set the delays of these links to 0 milliseconds. Fig. 7 shows the impact of the packet loss ratio on the performance achieved by the MPTCP connection. The figure shows the two subflows that compose the connection. The subflow shown in white passes through `Link A` while the subflow shown in gray passes through `Link B`. When there are no losses, the MPTCP connection is able to saturate the two 100 Mbps links. As expected, the gray subflow that passes through `Link B` is affected by the packet losses and its goodput decreases with the packet loss ratio. However, the goodput of the other subflow remains stable with packet loss ratios of 1, 2 or 3 %. It is only when the packet loss ratio reaches 4% or 5% that the goodput of the white subflow decreases slightly.

The last factor that we analyze is the impact of the Maximum Segment Size (MSS) on the achieved goodput. TCP implementors know that a large MSS enables higher goodput [2] since it reduces the number of interrupts that needs

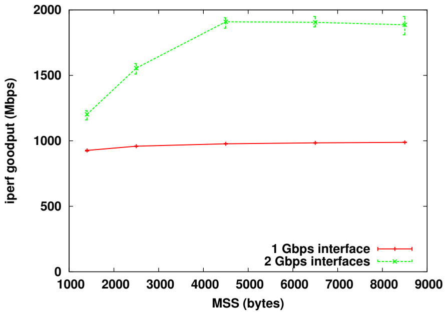

Fig. 8. Impact of the MSS on the performance

to be processed. To evaluate the impact of the MSS, we do not introduce delays nor losses on the router and use either one or two Gigabit Ethernet interfaces on the source and the destination. Figure 8 shows the goodput in function of the MSS size. With a standard 1400 bytes Ethernet MSS, MPTCP can fill one 1Gbps link, and partly use a second one. It is able to saturate two Gigabit Ethernet links with an MSS of 4500 bytes. Note that we do not use TSO (TCP Segmentation Offload) [3] for these measurements. With TSO the segment size handled by the system could have grown virtually to 64KB and allowed it to reach the same goodput with a lower CPU utilization. TSO support will be added later in our MPTCP implementation. Finally, we have looked at the CPU consumption in the system. Given that the MPTCP scheduler runs for every transmitted segment, we were afraid that increasing the number of subflows (hence the cost of running the scheduler) could significantly impact the performance. *This is not true*. We have run MPTCP connections containing 1 to 8 subflows on a shared 1 Gbps bottleneck with various MSS. Increasing the number of concurrent subflows from 1 to 8 has no significant impact on the overall system charge. We also found that the server is globally more busy than the sender, which can be explained by the cost of reordering the received segments. MPTCP currently uses the same algorithm as TCP to reorder segments, but while TCP typically reorders only a few segments at a time, MPTCP may need to handle hundreds of them. A better reordering algorithm could probably improve the receiver performance. Regarding the repartition of the load between the software interrupts and the user context, we found that the majority of the processing was done in the software interrupt context in the receiver: around 50% of the CPU time is spent in soft interrupt, 8% in the user context with a 1400 bytes MSS and a single Gigabit Ethernet link. This can be explained by the fact that prequeues are not in use. The sender spends around 17% in soft interrupt and 25% in user context under the same conditions. Although more work is performed in user context, which is good, there is still some amount of work performed in soft interrupt because the majority of the segments are sent when an incoming acknowledgement opens more space in the congestion or sending window. This event happens in interrupt context.

6 Conclusion and future work

MultiPath TCP is a major extension to TCP that is being developed within the IETF. Its success will depend on the availability of a reference implementation. From an implementation viewpoint, MultiPath TCP raises several important challenges. We have analyzed several of them based on our experience in implementing the first MultiPath TCP implementation in the Linux kernel. In particular, we have shown how such an implementation can be structured and discussed how buffer management must be adapted due to the utilization of multiple subflows. We have analyzed the performance of our implementation in the HEN testbed and shown that the coupled congestion control scheme is more fair than the standard TCP congestion control scheme. We have also evaluated the impact of the delay on the receive buffers and the throughput and showed that losses on one subflow had limited impact on the performance of another subflow from the same MultiPath TCP connection. Finally, we have shown that our implementation was able to efficiently utilize Gigabit Ethernet links when using large packets.

Our implementation in the Linux kernel is a first step towards the adoption and the deployment of MultiPath TCP. There are however several research and implementation issues that remain open. Firstly, the MultiPath TCP protocol is not yet finalized and for example the security issues are still being developed. Secondly, MultiPath TCP currently uses the standard TCP retransmission mechanisms on each subflow while multipath-aware retransmission mechanisms could probably improve the performance in some cases. Thirdly, our current implementation uses all available subflows while better performance would probably be possible by adding and removing subflows based on their measured performance. Fourthly, the MultiPath TCP implementation should better interact with the network interfaces to benefit from the TCP segment offload mechanisms that some of them include. Finally, the performance of MultiPath TCP in the global Internet and its interactions with real middle-boxes should be evaluated. We expect that our publicly available implementation will allow other researchers to also contribute to this work.

Acknowledgements

We thank the members of the Trilogy project for their help and in particular Costin Raiciu for his help with the coupled congestion control, and Adam Greenhalgh for maintaining the HEN-Testbed and giving us the ability to execute the performance measurements.

References

1. Allman, M., Paxson, V., Blanton, E.: TCP Congestion Control. RFC 5681 (Draft Standard) (Sep 2009), http://www.ietf.org/rfc/rfc5681.txt
2. Borman, D.A.: Implementing TCP/IP on a cray computer. SIGCOMM Comput. Commun. Rev. 19, 11–15 (April 1989), http://doi.acm.org/10.1145/378444.378446

3. Currid, A.: TCP Offload to the Rescue. Queue 2, 58–65 (May 2004), `http://doi.acm.org/10.1145/1005062.1005069`
4. Fisk, M., chun Feng, W.: Dynamic right-sizing in TCP. In: Proceedings of the Los Alamos Computer Science Institute Symposium. pp. 01–5460 (2001)
5. Ford, A., Raiciu, C., Barré, S., Iyengar, J.: Architectural Guidelines for Multipath TCP Development (December 2010), internet draft, draft-ietf-mptcp-architecture-03.txt, Work in progress
6. Ford, A., Raiciu, C., Handley, M.: TCP Extensions for Multipath Operation with Multiple Addresses (October 2010), internet draft, draft-ietf-mptcp-multiaddressed-02.txt, Work in progress
7. Hsieh, H.Y., Sivakumar, R.: pTCP: An End-to-End Transport Layer Protocol for Striped Connections. In: ICNP. pp. 24–33. IEEE Computer Society (2002)
8. Iyengar, J., Amer, P.D., Stewart, R.R.: Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. IEEE/ACM Trans. Netw. 14(5), 951–964 (2006)
9. Iyengar, J., Amer, P., Stewart, R.: Receive buffer blocking in concurrent multipath transfer. In: IEEE Global Telecommunications Conference, 2005. GLOBECOM'05. p. 6 (2005)
10. Jacobson, V.: Re: query about tcp header on tcp-ip (Sep 1993), available at ftp://ftp.ee.lbl.gov/email/vanj.93sep07.txt
11. Li, Y.T., Leith, D., Shorten, R.N.: Experimental evaluation of TCP Protocols for High-Speed Networks. IEEE/ACM Trans. Netw. 15, 1109–1122 (October 2007), `http://dx.doi.org/10.1109/TNET.2007.896240`
12. Liao, J., Wang, J., Zhu, X.: cmpSCTP: An extension of SCTP to support concurrent multi-path transfer. Communications (2008)
13. Magalhaes, L., Kravets, R.: Transport Level Mechanisms for Bandwidth Aggregation on Mobile Hosts. In: ICNP. pp. 165–171. IEEE Computer Society (2001)
14. Moskowitz, R., Nikander, P.: Host Identity Protocol (HIP) Architecture. RFC 4423 (May 2006), `http://www.ietf.org/rfc/rfc4423.txt`
15. Nordmark, E., Bagnulo, M.: Shim6: Level 3 Multihoming Shim Protocol for IPv6. RFC 5533 (Jun 2009), `http://www.ietf.org/rfc/rfc5533.txt`
16. Postel, J.: Transmission Control Protocol. RFC 793 (Standard) (Sep 1981), updated by RFCs 1122, 3168
17. Raiciu, C., Handley, M., Wischik, D.: Coupled Multipath-Aware Congestion Control. Internet draft (work in progress), Internet Engineering Task Force (July 2010), `http://tools.ietf.org/html/draft-ietf-mptcp-congestion-00`
18. Rojviboonchai, K., Osuga, T., Aida, H.: R-M/TCP: Protocol for Reliable Multi-Path Transport over the Internet. In: AINA. pp. 801–806. IEEE Computer Society (2005)
19. Stewart, R.: Stream Control Transmission Protocol. RFC 4960 (Sep 2007)
20. Wehrle, K., Pahlke, F., Ritter, H., Muller, D., Bechler, M.: The Linux networking architecture: design and implementation of network protocols in the Linux kernel. Prentice Hall (2004)
21. Wischik, D., Handley, M., Braun, M.B.: The Resource Pooling Principle. SIGCOMM Comput. Commun. Rev. 38(5), 47–52 (2008)
22. Wischik, D., Raiciu, C., Greenhalgh, A., Handley, M.: Design, implementation and evaluation of congestion control for multipath TCP (April 2011), USENIX NSDI
23. Zhang, M., Lai, J., Krishnamurthy, A.: A transport layer approach for improving end-to-end performance and robustness using redundant paths. In: USENIX 2004. pp. 99–112 (2004)