HTMT: High-Throughput Multipath Tunnelling for Asymmetric Paths

Richard Sailer, Jörg Hähner
Organic Computing Group
University of Augsburg, Augsburg, Germany
Email: richard.willi.sailer@student.uni-augsburg.de, joerg.haehner@informatik.uni-augsburg.de

***Abstract*—Multipath Tunnelling (MT) is an effective approach to increase the reliability and performance of network connections. In contrast to Multipath TCP (MPTCP) or SCTP, it also works for UDP traffic and its deployment is easy. It requires neither full access to all uplink endpoints nor a multipath protocol implementation in every involved host. Despite its potential, research on MT is sparse. Most known prototypes use either round robin or MPTCP schedulers that are not able to fully utilise the potential of MT.**

This paper proposes HTMT, a novel packet scheduling algorithm designated for MT that achieves high throughput for reliable traffic on asymmetric paths. Using the tunnel transport protocol's path estimation, it is able to dynamically adapt to changing subtunnel characteristics. Additionally flow awareness allows HTMT to associate packets to a flow and only send it on subtunnels where it won't overtake any of its predecessors. This avoids packet reordering and its detrimental effect on TCP throughput, without the latency and performance costs of a re-reordering buffer at the tunnel exit.

In multiple testbed experiments, we compare HTMT with several schedulers for MT and MPTCP, including LowRTT (current MPTCP default), OTIAS and AFMT. In terms of throughput, HTMT offers similar performance for symmetric paths but outperforms all competitors on asymmetric paths.

***Index Terms*—internet architecture, multipath, cross layer optimisation, reliability**

I. Introduction

Network paths can be unreliable or have low bandwidth [1], [2]. Redundancy can be a solution for these issues. Redundancy has been used to provide reliability and higher throughput in many fields of computer engineering, e.g. databases, storage or power supply, but seldomly for network paths. There are several proposed concepts for redundancy: Load balancing [3], Multipath TCP [4] and Multipath Tunnelling [5].

Load balancing can be used on the client side. At connection establishment, it selects one of multiple available paths. If a path goes down, all flows routed through this path are cut. In this way it uses several paths, but reliability is not improved.

Multipath TCP (MPTCP) grants more reliability, but needs every client and every server to have direct full access to all network uplinks (Figure 1) [6]. This complicates wiring. Additionally, it needs all clients to implement MPTCP, a complex protocol. It also does not solve the problem for UDP flows. So while MPTCP performs better than load balancing, it still leaves several issues unaddressed.

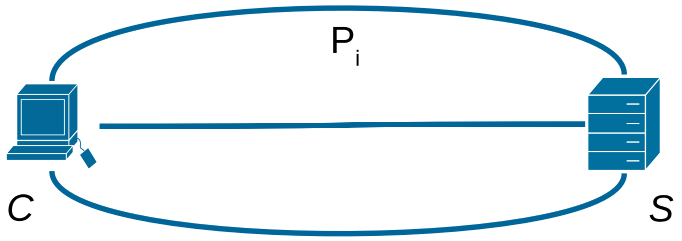


Fig. 1. Multipath TCP Network Topology. For every packet sent from S to C, S decides (schedules) the path P_i to use. For this, S and C need an implementation of MPTCP and direct access to all the paths.

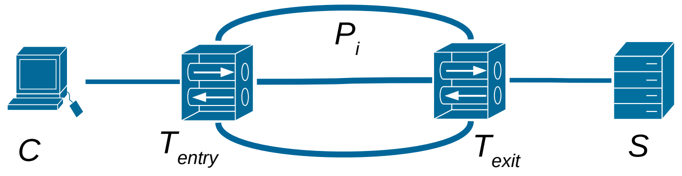


Fig. 2. Multipath Tunnelling Network Topology. A Packet p sent from a Client C is encapsulated at the tunnel entry T_{entry} and sent via one of the paths P_i to the tunnel exit T_{exit}. There, it is decapsulated and sent to the destination server S.

Multipath Tunnelling addresses these issues. As is visible in Figure 2, only the tunnel endpoints T_{entry} and T_{exit} need to see and understand the subtunnels. The clients and servers don't know they're connected by multiple paths, they need no additional wiring or implementation of a new network protocol. Since all flows between the two networks are tunnelled, this also works for UDP and can provide additional confidentiality or a VPN.

Our novel contributions are:

- We introduce HTMT, a flow-aware packet scheduling algorithm that avoids packet reordering and gains higher throughput than existing MPTCP and MT algorithms, especially on asymmetric paths.
- We evaluate HTMT and multiple other schedulers for various bandwidth and latency configurations with two or three subtunnels and multiple payload flows. We show the distribution of throughputs and latencies and find that HTMT outperforms all others on asymmetric paths while it performs comparable on symmetric ones.

II. Background

A *flow* is the set of all sent packets in a transport layer association (both directions, forward and reverse) [1].

We define a subtunnel as one of the multiple packet tunnels in a multipath tunnelling (MT) system.

Packet reordering describes the arrival of packets in a flow in an other order as they were sent at their source [7]. TCP is sensitive to such packet reordering, as it interprets it as an indicator of a congestion event and reacts with spurious packet retransmission and a throughput reduction, often to one half [8]. Multipath tunnelling, with simple non-flow-aware scheduling algorithms often induce packet reordering [5].

TCP uses a sliding window (congestion window, *CWND*) algorithm to adjust its send rate to the path's capacity [9]. Often after receiving a cumulative ACK, send space in the CWND for several packets is freed. We call this free space *free slots* or s_{free}. Many TCP implementations send several packets as a *burst* or *flowlet* [3]. TCP and DCCP track a smoothed round trip time (*SRTT*) for its path. In multipath routing context, *packet scheduling* refers to the task of choosing an output queue for every packet from an input queue [10].

DCCP [11], the Datagram Congestion Control Protocol, is a transport layer protocol that can be seen as a middle ground between UDP and TCP. Like UDP, it sends datagrams instead of a byte stream and does not provide any reliability. But like TCP, it does congestion control (*CC*). This makes DCCP a good fit for Multipath tunnelling, although conventional single path tunnelling approaches use UDP as transport protocol [1]. CC information from the transport layer proves to be very helpful for multipath scheduling decisions, but we don't want the head-of-line blocking, reliability and byte stream orientation of TCP, therefore we and [12] chose DCCP as our subtunnel protocol.

III. Related Work

LowRTT [10] is a simple scheduler currently used as default in the Linux Kernel. When there is a new packet to schedule and at least one subflow with free slots (free subflow), it is invoked and puts the packets into the free subflow with the lowest SRTT.

OTIAS [13] schedules every packet p to the fastest available subflow, similar to SRTT. But it does not only send packets when there is free space in the CWND, but may also enqueue them into the subflow's send buffer. For every subflow, OTIAS uses $T_{delivery}$, the time until p would arrive at the peer using SRTT, CWND and the number of packets in-flight and in the send buffer.

AFMT [14] uses TCP for the subtunnels and does no reordering at T_{exit}. The packet scheduling algorithm is flow-aware and adaptive. Flow awareness means sending packets of one flow in a way that respects their ordering. Unfortunately, the time estimations used don't take the time a packet needs to wait in the send buffer into account. In our approach we avoid buffering altogether by only scheduling when there is $s_{free} > 0$, which keeps estimations accurate.

[12] uses DCCP subtunnels and re-reordering for unreliable multimedia traffic. For packet scheduling it uses LowRTT or OTIAS (see Section III). Therefore, this approach is adaptive but not flow-aware.

IV. HTMT

Conceptually for every packet HTMT first determines the *applicable subtunnels* (subtunnels on which the packet won't overtake any of its predecessors) (flow awareness), and then picks the currently best of them (adaptivity). To find the applicable subtunnels, a central flow table is used. For every flow id, it contains the last subtunnel used by this flow and a send time stamp. Initially, HTMT obtains the flow-id of p ($p.flow\_id$) from the operating system. Every operating system that supports network address translation (NAT) needs to track flows and can provide flow-ids. Therefore, there's no overhead for flow identification and tracking. For Linux, this is possible with the *conntrack* module.

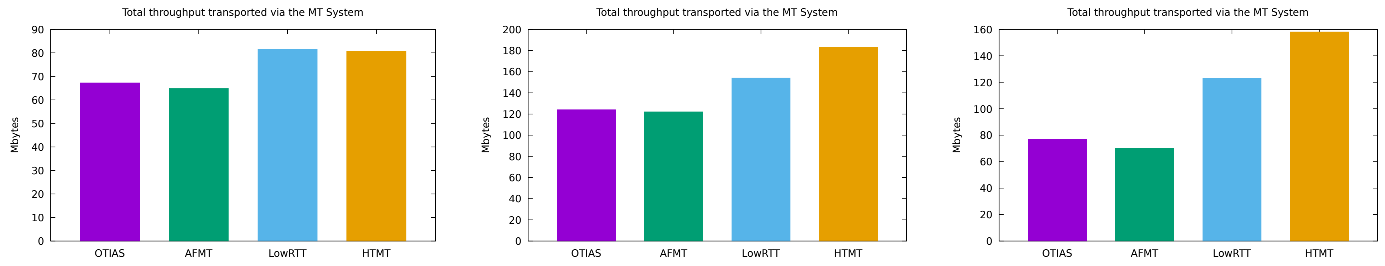
Next, we look up the flow id in the flow table . If it exists, we calculate δ, the time that has passed since the last packet of our flow was sent. With $s_i.SRTT$, the SRTT of a subtunnel i, it is possible to predict when p will arrive at T_{exit}, namely in $s_i.SRTT$ time from now and $s_i.SRTT + \delta$ time from when p_{last} was sent. Comparatively, $s_{last}.SRTT$ gives the arrival time of p_{last} from when it was sent. Therefore, if $s_i.SRTT+\delta$ is larger than $s_{last}.SRTT$, p will arrive after p_{last}, and we can add s_i to the list of applicable subtunnels .

After acquiring the list of applicable subtunnels, HTMT selects the best of them (s_{opt}) (see below). Then, the flow table is updated with the new values of s_{opt} and the current time t_{now} and p is finally sent via s_{opt} . If $p.flow\_id$ is not found in the flow table, i.e. it starts a new flow, HTMT directly calls the adaptive selection process with all available subtunnels. If no applicable subtunnels are found HTMT waits until at least one subtunnel opens up.

Algorithm 1: HTMT: Adaptivity

```
1 $s_{opt} \leftarrow s_i$ with maximal $s_i.weighted\_s_{free}$
    where
2         $s_i.weighted\_s_{free} = log(s_i.s_{free})/s_i.SRTT$
```

Algorithm 1 illustrates how HTMT's final subtunnel selection aims adaptively for low latency and high throughput. Line 1 iterates over the subtunnels and calculates the *weighted free slots* for each, the subtunnel with the highest one is selected. This is equivalent to choosing the subtunnel with the most free capacity avoiding unnecessary congestion. However, we also want to provide low latency if possible, therefore we scale s_{free} by dividing it by the subtunnel's SRTT. Our first experiments showed that using both factors linearly gives the bandwidth (s_{free}) too much impact, since its values vary within a significantly larger range than those of the SRTT. With scaling s_{free} logaritmically, we achieve competitive throughput and latency.



(a) Path 1: 50ms, 8 Mbit/s - Path 2: 50ms, 8 Mbit/s (b) Path 1: 50ms, 16 Mbit/s - Path 2: 70ms, 8 Mbit/s (c) Path 1: 50ms, 8 Mbit/s - Path 2: 70ms, 16 Mbit/s

Fig. 3. Total aggregated Goodput the four different packet schedulers achieve with five Linux TCP Cubic flows as payload. 70s bulk traffic, evaluated for different path configurations.

V. Evaluation

The topology of our network testbed is comparable to Figure 2 but with two subtunnels, and router nodes between the tunnel gates. All nodes are i7-2600 desktop computers with 8GB RAM connected via Gigabit Ethernet. As operating system, we use Debian Bullseye with Linux 5.6.14-1. All network interfaces use pfifo_fast as queueing discipline and *tc netem* on the routers to emulate latency and bandwidth limitation.

We created a user space MT implementation with plugable packet schedulers called *einsfroest*. It opens a DCCP socket for every subtunnel to be used and gets the DCCP internal congestion information via *getsockopt()* with the option DCCP_SOCKOPT_CCID_TX_INFO. We had to implement this getsockopt variant as two small kernel commits. One of them is already accepted upstream. Our test traffic is created and measured with *iperf*, running the iperf client on T_{entry} and the server on T_{exit}.

Figure 3 illustrates the total throughput sums the four different packet schedulers could achieve in 70 seconds for 5 TCP Cubic bulk flows. Subfigure 3a displays the results for a symmetric path configuration, both paths have the same latency and bandwidth. Here HTMT performs comparably to the state-of-the-art LowRTT with roughly 80 Mbyte. While both OTIAS and AFMT perform worse with about 65 Mbyte, this is likely due to the number of lost packets because the DCCP send buffers overflew at some point, causing TCP to throttle as also visible in Retransmissions reported by iperf.

Subfigures 3b and 3c display asymmetric path configurations. One where the low latency path has more bandwidth (3b) and one where the high latency path has more (3c). In both cases HTMT performs noticeably better than all others: For 3b it's 180 Mbyte vs. LowRTT's 155 Mbyte for 3c it is roughly 160 Mbyte vs. LowRTT's 125 Mbyte. This is very likely the positive effect of flow awareness and an adaptivity scheme that takes not only SRTT but also the path's estimated capacity into account. AFMT performs considerably worse although it's flow aware and adaptive presumably because of packet loss due to send buffer overflows, which makes it comparable to OTIAS.

It is worth noting that throughout all experiments the latency the payload flows experienced was lower average for HTMT and the highest for AFMT . Regarding throughput fairness all schedulers treated their flows comparably fair.

VI. Conclusion

In this paper we presented HTMT a Multipath Tunnelling system for asymmetric paths relying on DCCP subtunnels. In several experiments we evaluated throughput and latency for different symmetric and asymmetric subtunnel paths, regarding throughput and latency. In all asymmetric cases HTMT outperforms all comparison candidate packet schedulers regarding throughput. Among them are LowRTT, the current standard of MPTCP, but also OTIAS and AFMT.

References

[1] A. S. Tanenbaum, *Computer networks, 4-th edition*, 2003.

[2] K. Dominikn, "Multipath aggregation of heterogeneous access networks," Ph.D. dissertation, PhD Thesis, University of Oslo, 2011.

[3] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic Load Balancing Without Packet Reordering," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 2, pp. 51–62, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1232919.1232925

[4] C. Paasch and O. Bonaventure, "Multipath tcp," *Queue*, vol. 12, no. 2, p. 40, 2014.

[5] M. Bednarek, G. Barrenetxea, M. Kühlewind, and B. Trammell, "Multipath bonding at layer 3." in *ANRW*, 2016, pp. 7–12.

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

[7] A. Jayasumana, N. Piratla, T. Banka, A. Bare, and R. Whitner, "Improved Packet Reordering Metrics," RFC 5236, June 2008.

[8] S. Bohacek, J. P. Hespanha, J. Lee, C. Lim, and K. Obraczka, "A new TCP for persistent packet reordering," *IEEE/ACM Transactions on Networking (TON)*, vol. 14, no. 2, pp. 369–382, 2006.

[9] J. Postel, "Transmission Control Protocol," *RFC 793*, 1981.

[10] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure, "Experimental evaluation of multipath tcp schedulers," in *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*. ACM, 2014, pp. 27–32.

[11] E. Kohler, M. Handley, and S. Floyd, "Datagram congestion control protocol (DCCP)," *RFC 4340*, 2006.

[12] M. Amend, E. Bogenfeld, M. Cvjetkovic, V. Rakocevic, M. Pieska, A. Kassler, and A. Brunstrom, "A framework for multiaccess support for unreliable internet traffic using multipath dccp," in *2019 IEEE 44th Conference on Local Computer Networks (LCN)*, 2019, pp. 316–323.

[13] F. Yang, Q. Wang, and P. D. Amer, "Out-of-order transmission for in-order arrival scheduling for multipath tcp," in *2014 28th International Conference on Advanced Information Networking and Applications Workshops*. IEEE, 2014, pp. 749–752.

[14] R. Sailer and J. Hähner, "An adaptive flow-aware packet scheduling algorithm for multipath tunnelling," in *2019 IEEE 44th Conference on Local Computer Networks (LCN)*. IEEE, 2019, pp. 109–112.