Understanding MPTCP in Multi-WAN Routers: Measurements and System Design

Kariem Fahmi
Trinity College Dublin
kfahmi@tcd.ie

Douglas Leith
Trinity College Dublin
doug.leith@tcd.ie

Stepan Kucera
Nokia
stepan.kucera@nokia.com

Holger Claussen
Tyndall National Institute
holger.claussen@tyndall.ie

***Abstract*—MPTCP is used in Multi-WAN Routers to aggregate multiple WAN/Internet connections using two architectural variants: proxying and tunneling. The proxy variant creates one MPTCP connection for each TCP connection, resulting in a large number of parallel uncoordinated MPTCP connections, which leads to underutilizating the available capacity, suboptimal scheduling, and increased loss rate.The tunnel variant encapsulates TCP over MPTCP, stacking two reliability layers, which leads to large number of spurious retransmissions, an issue known as TCP meltdown. We propose a new solution, BOOST, that eliminates the problems with both variants by multiplexing TCP connections over a single persistent multi-path connection. BOOST also takes a hybrid approach to multi-path scheduling combining load balancing and scheduling by transmitting short flows across a single path, avoiding HoL blocking, and opportunistically transmitting long flows across multiple path, utilizing left-over capacity. Evaluations show that BOOST provides better throughput, lower losses, and retransmissions compared to MPTCP**

I. Introduction

Multi-WAN Routers (MWR) provide WAN/Internet connectivity by combining two or more backhaul links (e.g. 2 LTE connections) for improved capacity and reliability. They are commonly used in public transportation services, such as trains, buses, ships and even airplanes to provide seamless Internet connectivity to a large number of passengers using wireless backhauls, which are known to deteriorate under high mobility. For example, providers of train-to-ground (T2G) communication combine LTE/5G from different carriers to improve the capacity for on-board internet access and increase reliability for mission-critical applications such as communication based train control (CBTC) or closed circuit TV (CCTV) [1].

MPTCP is used by some commercial MWRs as a transport layer solution for allowing individual TCP connections from user devices to utilize the capacity of all the available backhauls [2]. An MPTCP MWR needs to either transparently split incoming TCP connections into a TCP and an MPTCP portion, like a proxy, or encapsulate them inside MPTCP, like a tunnel. In both cases, a network-edge proxy-server/tunnel-server is used to translate/decapsulate the traffic back into TCP to be forwarded to the content server. Figures 1 and 5 show the architecture and network topology for the proxy and tunnel variants respectively.

In the proxy variant of the MWR, shown in Figure 1, each user TCP connection is terminated at the MWR and its packets are transmitted using MPTCP to the network-edge proxy. Thus for each TCP connection, a corresponding MPTCP connection is created. Our measurements of a proxy MPTCP MWR used to carry passenger traffic in a train to ground (T2G) communication system show that this results in thousands of concurrent MPTCP connections, each with their independent congestion control on each link plus their own multi-path scheduler. We find that this leads to low capacity utilization and high loss, as will be explained in more detail later.

In the tunnel variant of the MWR, shown in Figure 5, the packets from user TCP connections are encapsulated and transmitted over all the available backhauls using a single persistent MPTCP connection. That is, a single MPTCP connection carries all of the user TCP connections over the backhaul. This eliminates the large number of connections observed with proxy variant but creates new issues. In particular, the tunnel stacks two reliability layers (MPTCP and TCP), each with its own congestion control and retransmission mechanism for lost packets. This can lead to so-called TCP meltdown [3], where a high number of spurious re-transmissions are constantly triggered. Additionally, there is cross-flow Head of Line (HoL) blocking between separate user flows sharing the same MPTCP connection.

A second key issue that is common to both architectural variants is frequent HoL blocking due to the heterogeneity of the wireless links used for multi-path scheduling. This type of blocking occurs when packets arrive out of order, leading to them being placed into a reordering queue until they can be delivered in-order, increasing delay. This delay disproportionately impacts short flows (e.g. web browsing) as they are much more latency-sensitive compared to long flows (e.g. file download). HoL blocking is also more prevalent when scheduling over heterogeneous links with different delay profiles, which are commonly used in MWRs. For example, a T2G MWR tends to utilize multiple LTE links from different carriers, which are likely to have different delay profiles due to different coverage, cross-traffic and core-latency.

Based on our analysis of the MPTCP proxy and tunnel issues in MWR, we propose a new multi-path proxy solution more suited for MWR, called BOOST. This solution eliminates the problems with both the MPTCP proxy and tunnel approaches by multiplexing passenger traffic over a single, persistent and selectively-reliable multi-path connection, allowing UDP as well as TCP to benefit from multi-connectivity. It splits TCP connections, similarly to a proxy, but transmits the traffic over the backhaul using a single BOOST connection, similarly to a tunnel. BOOST is based on QUIC and re-uses many of its streaming features for flow multiplexing, while adding new features for multi-path transmission and selective reliability. BOOST also takes a hybrid approach to multi-path scheduling. Namely, short flows are transmitted across a single link to avoid HoL blocking while longer flows are opportunistically transmitted across multiple paths, utilizing left-over capacity, thereby improving efficiency and robustness. The scheduler also provides prioritization and redundant multi-path scheduling for high priority traffic.

Concretely, we make the following contributions

- We collect traces from a production T2G communication system that utilizes an MPTCP proxy MWR and use this to uncover performance issues with the MPTCP proxy

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) and Nokia Bell Labs, and is co-funded under the European Regional Development Fund under Grant Number 13/RC/2077.

approach. We also highlight issues with the MPTCP tunnel approach in light of previous work and our own trace-driven emulation results.
- Based on the outlined key issues with MPTCP, we propose BOOST, a novel multi-connectivity solution that addresses the issue present in both variants of the MPTCP architecture. We evaluate BOOST performance against both an MPTCP proxy and tunnel, and against single-path TCP in a trace-driven emulation setup based on real LTE link traces and real passenger traffic traces. The results of this evaluation show that BOOST significantly improves average passenger flow throughput and reduces CCTV delay while experiencing much lower loss and out of order delivery.

II. Measurements Collection

We collected 30 hours of packet traces from a production Train-to-Ground (T2G) communication system that uses an MPTCP proxy MWR with two different public LTE connections as a backhaul. The system used a transparent proxy client inside an MWR that is running an MPTCP kernel, and a corresponding proxy server placed in a network-edge server that was also running an MPTCP kernel. The MWR was configured as the default gateway for the on-board WiFi system which provided free internet for all passengers. TCP connections entering the MWR were transparently redirected to the proxy client using the IPTables TProxy option [4]. Packet traces were collected from the network-edge server and the MWR in order to measure capacity for both LTE links as well as capture the profile of passenger traffic for use later in analysis and trace-driven emulation. Additionally, we injected a probe into the MPTCP kernel using the FTrace [5] kernel interface to monitor the evolution of internal MPTCP state variable.

A. Measuring capacity

Since most passenger traffic was downlink, our interest is to measure the downlink capacity. Since the passenger traffic may underflow the capacity of the links due to lack of traffic or the action of the MPTCP congestion control, simply observing the received throughput at the MWR may provide inaccurate capacity estimates. To address this, two parallel UDP iPerf [6] flows, each assigned to one LTE link, were transmitted from the network-edge server towards the MWR. The goal was to ensure that there was a constant bit rate stream of data flowing through both links at all times. The choice of UDP eliminates the possibility of TCP congestion control underflowing the link capacity.

The UDP iperf throughput was limited to 2Mbit/s, which is below the capacity of the typical LTE link, in order to not take too much capacity away from the passenger traffic. However, since 2Mbit/s are likely to be below the LTE capacity, the capacity is estimated by analyzing the iPerf packet trains. iPerf transmits a train of packets periodically, depending on the assigned rate, and injects the same timestamp into all packets belonging to the same train, allowing the receiver to identify packets belonging to the same train. To estimate the capacity, the difference between the arrival time of the first and last packets in each train is observed by the receiver, and the size of the train is divided by the amount of time it took receive it. Since trains are sent periodically, this estimate is used to extrapolate the capacity until the arrival of the next train.

More formally, let c_i represent the capacity estimate between the arrival time of the first packet in train i, and the firs packet in following train. c_i is calculated as follows

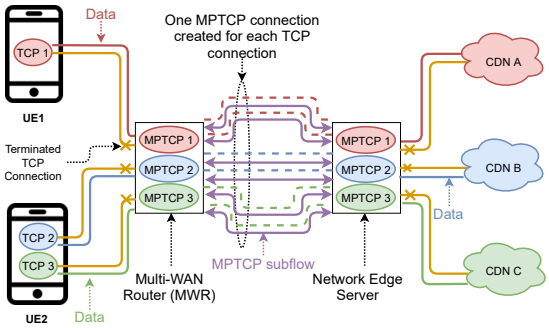
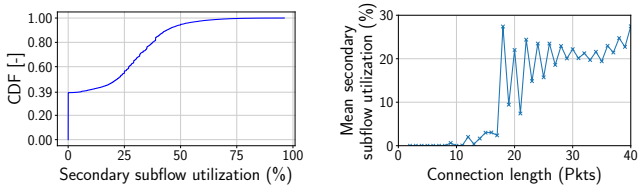


Fig. 1: Proxy based MPTCP MWR



(a) (b)

Fig. 2: (a) shows the CDF the secondary subflow utilization in all connections. (b) shows the utilization ratio vs connection length. (c) shows the CDF of of flow length.

$$c_i = \frac{\sum_{j=0}^{n_i} s_i^j}{t_i^{last} - t_i^{first}} \quad (1)$$

where t_i^{first} and t_i^{last} are the arrival times of the first and last packet of train i respectively, s_j^j is the length of the j th packet in the train, and n_i is the total number of packets in the train. In cases where $t_i^{last} = t_i^{first}$ i.e. the first and last packets in a packet train arrived at the same time, that packet train is ignored for capacity calculation and the previous capacity estimate is re-used.

III. Analysis of the performance of MPTCP Proxy MWR

As briefly discussed in the introduction, an MPTCP proxy MWR variant establishes one MPTCP connection for each TCP connections, as shown in Figure 1. According to measurements of passenger traffic collected in Section II, this results in the creation of thousands of uncoordinated MPTCP connections, most of which are short or on/off (i.e. long-lived with low frequency of data exchange such as HTTP connections with keep-alive). In this section we analyze the performance of the proxy variant in terms of link utilization, multi-path scheduling and congestion control.

A. Link utilization

For the MPTCP default scheduler, MinRTT, to utilize more than one subflow, it first needs to establish the default subflow using the standard TCP 3-way handshake and then establish the secondary subflows by performing a similair handshake over the other available paths, which takes at least one round trip time. Before the second subflow is established, only the default subflow is used for data transmission. Once the secondary subflow is established, the connection will have to fill the

CWND of the subflow with the lowest estimate for smooth round trip time (SRTT) before using other subflows.

Short connections that transmit less than 10 packets will never use the secondary subflow since all their packets will fit in the initial CWND [7] of the default subflow. Also connections that transmit more than 10 packets may still not utilize any secondary subflow as they may transmit multiple CWND worth of data on the default subflow before the secondary subflows are established. Figure 2(b) shows the link utilization ratio versus the flow length. It can be seen that flows with a length smaller than 18 almost never utilized the secondary subflow. Beyond reducing utilization, since the first subflow is always established over the default route configured in the operating system routing table, this will cause the performance of short connections to be highly sensitive to the choice of default route.

On/off connections that periodically transmit small bursts of data, such as HTTP connection with keep-alive, may also never make use of more than one link if they do not have enough data in flight at once to completely fill the CWND of the link with the lowest SRTT. Figures 3(a) and 3(b) show two connections that were transmitting data during the same time period but were only making use of one link each.

Figure 2(a) shows the CDF of secondary subflow utilization based on the measurements from the production T2G system as described in Section II. It shows that roughly 40% of MPTCP connections did not use the secondary subflow.

B. Multi-path Scheduling

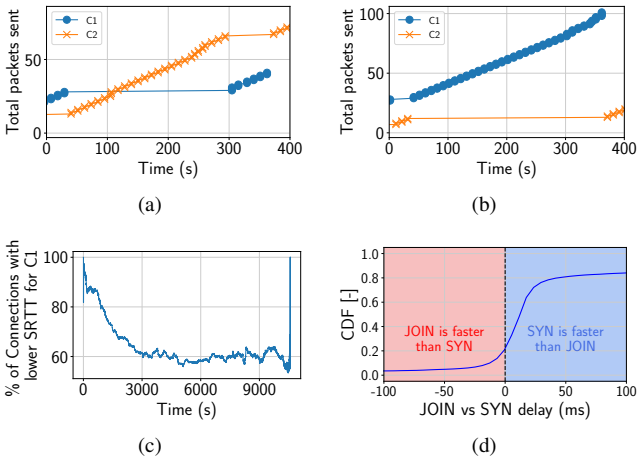


Fig. 3: (a)(b) show the total packets transmitted from both LTE links (C1 and C2) during the same time period from 2 different MPTCP connections. (c) shows the proportion of connections that with a lower SRTT estimate for C1 during a train trip. (d) shows the CDF for the difference in delay between the JOIN packet and the SYN packet in the same connection.

MinRTT relies on data transmission for estimating the SRTT of its subflows which is needed for optimal scheduling. Every time the underlying subflow transmits a packet and receives its acknowledgement, it will calculate the RTT for that packet and use it to update the estimate of the SRTT. In an MPTCP proxy MWR, all connections will perform isolated estimations of the links and will not share information between each other.

On/off connections that do not have enough traffic to probe all links may transmit data on a slower route for a long period of time. Figure 3(a) and 3(b) show the packets transmitted on LTE links from two carriers (C1 and C2) during the same time period by two MPTCP connections. It can be seen that both connections transmitted traffic on opposite links exclusively for over 200 seconds without ever probing the second link to update SRTT value, which will lead to higher delay for at least one of the connections. Figure 3(c) shows the proportion of all MPTCP connections with a lower SRTT value for C1 during a train trip. It can be seen that for the majority of the time, 40% of the MPTCP connections had a lower SRTT estimate for C1, which means they were making mostly the opposite scheduling decisions compared to the remaining 60% of the connections, indicating that a significant amount of connections are affected by this.

New connections will always transmit the SYN packet over the default subflow, which may be slower. Figure 3(d) shows the CDF for the difference in delay between the JOIN packet and the SYN packet transmitted by the same MPTCP connection, which is a reasonable estimate for the difference in delay between the 2 available links at the moment of the start of the connection . It can be seen that for 20% of the connections, it would have been faster to initiate the connection over the secondary subflow, which for short flows can significantly affect completion time.

C. Congestion control

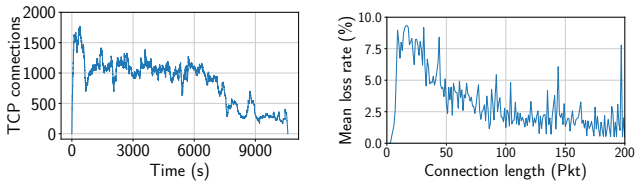


Fig. 4: (a) shows the number of active TCP connections over time during a 3 hour train trip (b) shows the loss rate for different connection length in packets

The congestion control relies on data transmission for estimating the CWND parameter which decides the sending rate. The multi-path scheduler also uses this parameter for deciding how much to transmit on each link. Similair to the SRTT, in an MPTCP proxy MWR, this estimation is also done by each connection independently resulting in a less accuracy. Having an inaccurate estimate for the CWND can cause undershooting of capacity, wasting capacity, or overshooting of capacity, causing losses.

When using loss-based congestion control (e.g. CUBIC [8]), an MPTCP connection will need to experience a loss on that link before lowering the sending rate. Figure 4(a) shows the number of active TCP/MPTCP connections during a train trip. It can be seen that there was around 1000 connections active for most of the trip, each with their own congestion control. This will result in a large number of un-necessary losses. These losses will disproportionately affect short connections that always start with the same value for the CWND potentially resulting in early losses that increase their completion time significantly. Figure 4(b) shows the mean loss rate for different connection lengths. It can be seen that the shorter connections, starting from around 8 packets in length, had significantly higher loss rate and it decreased as the connection length became longer.

D. Issues with an MPTCP Tunnel MWR

MPTCP tunnel MWRs are based on encapsulating TCP, and possibly UDP, connections and transmitting them over a single

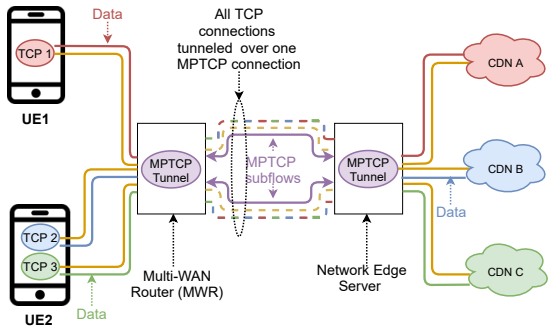


Fig. 5: Tunnel based MPTCP MWR

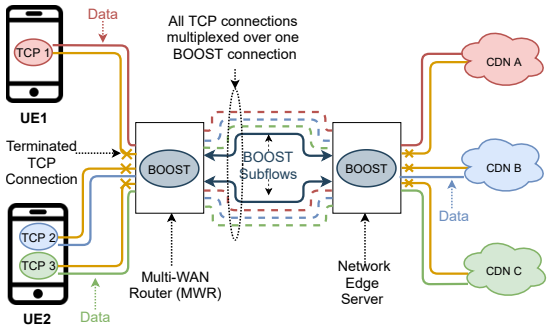


Fig. 6: BOOST MWR

persistent MPTCP connection, as shown in Figure 5. While this approach eliminates the high number of parallel MPTCP connections in the proxy variant, it introduces new performance issues due to stacking multiple layers of reliability and cross-flow HoL blocking. However, since we were not able to acquire measurements from a production communication system that relies on an MPTCP tunnel MWR, as we did for the MPTCP proxy MWR, we discuss the issues in light of previous work and confirm them later in the evaluation Section.

1) Stacking two layers of reliability: When a new TCP connection is created, it will often have a short re-transmission timer than the established MPTCP tunnel connection, as the latter is more adapted to the link properties. When data is lost in the MPTCP tunnel, it will queue up a re-transmission and increase its timeout. However, since the MPTCP tunnel will be blocked during the re-transmission period, the TCP connection does not get any acknowledgments, and it will thus queue up its own re-transmission of the same data. Since the timeout in the tunnel is less than the TCP connection, it may queue up even more re-transmissions causing the transmission of new data to stall completely, and every re-transmissions further compounds this problem. This behavior is referred to as TCP meltdown [3]. In wireless links under mobility, such as in trains, which experience frequent losses due to capacity fluctuations and handovers, frequent TCP meltdowns will occur.

2) Cross-flow HoL blocking: When data is received out of order due to loss or multi-path transmission over heterogeneous/time-varying links, received packets are held in a reordering buffer until they can be delivered in order. This is referred to as HoL blocking. In an MPTCP tunnel MWR, there is a single reordering buffer for all TCP connections being transported. Therefore, packets belonging to one TCP connection will have to wait in the re-ordering buffer needlessly until packets from another connection can be delivered in-order simply because packets for the latter were sent first. This issue is similar to the well-known HTTP head of line blocking problem [9].

IV. BOOST Design and Implementation

The proposed solution, referred to as BOOST, is based on two main design features: multiplexing application flows over a single persistent multi-path connection and hybrid multi-path scheduling. In this section we will discuss both in detail.

A. Persistent multi-path connection with multi-plexing

The first feature of BOOST is that it transparently proxies and multiplexes TCP and UDP flows over a single persistent and selectively-reliable multi-path connection between the MWR and a network-edge server, as shown in Figure 6. When the BOOST client first starts, it establishes a connection with the BOOST server and then establishes all subflows (details of the protocol are discussed in the next section). The BOOST connection has a single multi-path scheduler and a single instance of congestion control for each link. When new TCP flows are routed through the MWR, BOOST terminates each TCP connection and multiplexes the data over the existing multi-path BOOST connection. Similarly, UDP flows are also multiplexed but without enforcing reliability or in-order delivery. Data from individual TCP and UDP flows are tagged using a unique stream identifier to allow the receiver to distinguish between them.

This feature of BOOST addresses the problems in the variants of the MPTCP MWR as follows.

a) Underutilization of capacity: Since all application traffic is handled by one multi-path scheduler, it allows the scheduler to optimally distribute all application traffic across the available links, eliminating the underutilization of capacity by short flows and on/off flows.

b) Suboptimal scheduling and increased loss rate : BOOST maintains a single estimate for the CWND and the SRTT on each link that is generated by data transmitted from all traffic. This allows it to derive a much more accurate estimate of these parameters, leading to lower loss, better capacity utilization and more optimal multi-path scheduling. Additionally, since there is only one single congestion control process, a single loss on a link is sufficient for BOOST to reduce its sending rate, compared to distributed congestion control where each flow has to experience an independent loss.

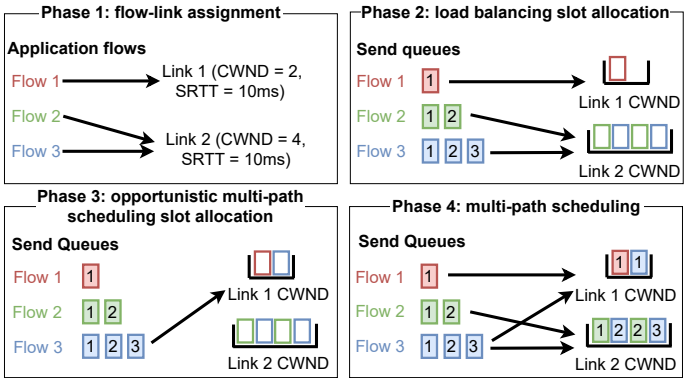
c) Stacking multiple layers of reliability: BOOST functions as a proxy by splitting TCP connections and thus does not stack multiple layers of reliability

d) Cross-flow HoL blocking: BOOST stores a unique stream identifier inside each packet that links to its parent application flow and hence can avoid cross-flow HoL blocking.

B. The BOOST protocol

The BOOST protocol is based on QUIC with modifications to enable multiplexing of TCP and UDP connections, and multi-connectivity.

1) Multi-connectivity: To enable multi-connectivity, we adopt an existing proposal [10] for a multi-path version of QUIC (MPQUIC) that modifies the protocol in two ways. First, the main QUIC packet header includes a new PathID field which identifies which subflow the packet belongs to. Furthermore, the existing ConnectionID field is used to link a path to the original connection. There are no handshakes needed to establish secondary subflows like in MPTCP. Second, acknowledgement (ACK) packets also include the PathID field



(a)

Fig. 7: Hybrid multi-path scheduling example

to identify which path the ACK packets belong to. Finally, the PacketNumber field is specific to each path.

2) Multiplexing: QUIC already includes a multi-streaming functionality, so we leveraged this to enable our multiplexing feature. In the BOOST implementation of QUIC, we modify the use of streams to refer to TCP connections instead of the traditional application streams. Each TCP connection is assigned a unique StreamID by BOOST.

To enable each proxied TCP connection to be forwarded to its correct final destination, BOOST implements a new QUIC extension/frame called SET_FORWARD_ADDRESS. It contains the following information: ConnectionID, StreamID, and the application destination ip, port and Protocol. This frame is transmitted with the first packet sent for an application flow.

3) Toggling reliability and in-order delivery : BOOST does not impose reliable delivery on UDP connections that it is carrying. In order to inform the BOOST server to the reliability of a particular stream, the BOOST client leverages the FIN flag in the first STREAM frame sent for a particular stream to indicate reliability to the BOOST server. An unreliable stream will not wait for in-order delivery before being forwarded. Additionally, neither the BOOST client nor server will re-transmit lost UDP packets, but the congestion control will observe losses and reduce rate appropriately.

C. Hybrid multi-path scheduling

The second key feature of BOOST is the hybrid multi-path scheduling approach which uses multi-path load balancing for short flows in order to minimize delay by preventing HoL blocking and opportunistic multi-path scheduling for longer flows to make use of left-over capacity. BOOST also supports replicating data from selected flows over all available links.

The scheduler is split into 4 phases as follows

1) Phase 1 (Flow-link assignment): The scheduler assigns application flows to available links according to the ratio R_i, which is calculated as follows

$$R_i = \frac{t_i}{\sum_{j=0}^{m} t_j} \quad \text{and} \quad t_i = \frac{CWND_i}{SRTT_i} \tag{2}$$

where $CWND_i$ is the CWND of link i, $SRTT_i$ is the SRTT of link i, and m is the number of available links. The goal is to load balance flows over links proportionate to the ratio of the throughput of the links. Load balancing occurs periodically every γ milliseconds, where γ is a design parameter we choose to be 100ms, and every time a new connection is detected. Applications flows that have not received new data for over one second, are removed from the assignment. Connections configured to be replicated are assigned to all links.

2) Phase 2 (load balancing slot allocation): After assigning a flow to a link, it is inserted into a priority queue ordered by the amount of bytes the connection has transmitted so far. If a flow has not transmitted anything yet, this value is 0. The front flow in the priority queue is then selected and at most one maximum transmission unit (MTU) worth of data, which is typically 1500 bytes, is allocated on the assigned link's send queue for use by this flow. The bytes-transmitted counter for the flow is then incremented by the amount of data allocated on the link and the flow is re-inserted into the priority queue. This process is repeated until all flows assigned to that link have been allocated space equal to the data in their send queue (i.e. they have no more data) or the total allocated data on a link is equal to its available CWND. For connections with replication configured, the assigned links are cycled every time they have a slot allocated.

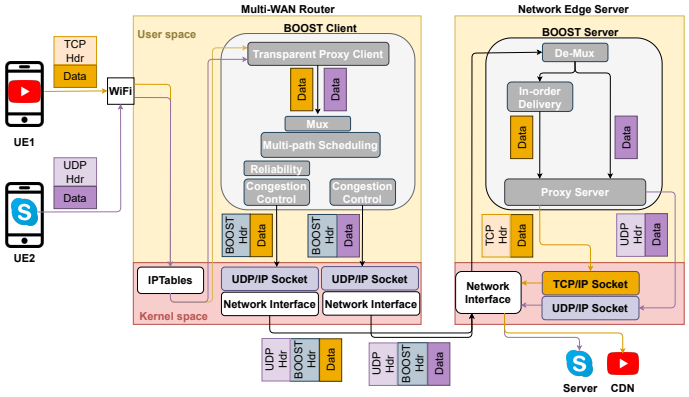
3) Phase 3 (Opportunistic multi-path scheduling slot allocation): This phase of the BOOST scheduler will only trigger if at least one link has space in its CWND and at least one flow has data that has not been allocated space on a link. All eligible application flows are placed in another priority queue that is also ordered by the amount of data transmitted just like before. A multi-path scheduling algorithm will then select between all available links to allocate space for data from the selected connection. For example, the MinRTT scheduler would pick the link with the lowest SRTT amongst the links that have space in their CWND. Afterwards the flow's transmitted data counter is incremented and it is returned to the priority queue. Replicated flows are not selected in this phase.

4) Phase 4: Multi-path scheduling: In this phase packets are taken from the flow send queue and placed on links with allocated space. If phase 3 did not occur (i.e. a connection only has allocated space on one link) then packets are placed on that link in order of their sequence number. If phase 3 did occur, the same multi-path scheduling algorithm as in phase 3 assigns packets from the flow send queue to the allocated space on the links.

5) An example: Fig 7 shows an example of the hybrid multipath scheduling. In phase 1, there are 3 flows to be assigned to two links. Since link 1 and link 2 have the same SRTT but link 2 has double the CWND, two flowss are assigned to link 2, while only one flows is assigned to link 1. In phase 2, each flow is allocated space on the assigned link. Flow 1 is assigned one MTU slot on link 1, while flows 2 and 3 are assigned 2 slots on link 2 each. Note that the flows were alternated in the order of the assignment due to the priority changing after being assigned a slot. In phase 3, the opportunistic multi-path scheduling triggered because flow 3 still had one packet remaining in its send queue and link 1 had a space in its CWND. As such, flow 3 was allocated one slot on link 1. Finally, in phase 4 packets were placed in the allocated spaces by the multi-path scheduling algorithm. In this case, the algorithm placed packet 1 from flow 3 on link 1 and packet 2 from flow 3 on link 2 as both of these slots were likely to be delivered before the last slot on link 2, where it placed the last packet from flow 3.

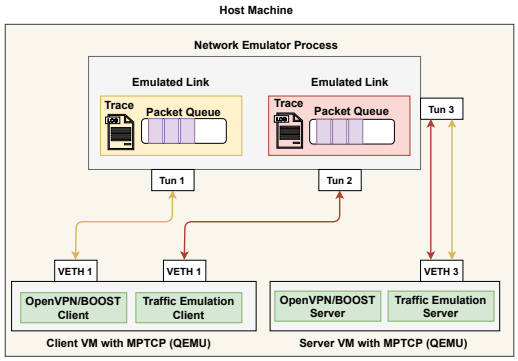
D. Implementation

BOOST is implemented in Linux using C++ as a user-space protocol inside a transparent proxy as shown in Figure 8. This allows it to be easily deployable in an MWR as a user-space application without any modifications needed to the



(a)

Fig. 8: BOOST architecture during uplink transmission



(a)

Fig. 9: Emulation setup

MWR's kernel or operating system. Figure 8 shows the software architecture during uplink transmission of both TCP and UDP traffic. As shown, BOOST implements its own reliability and congestion control on top of UDP sockets similar to QUIC [11].

V. Evaluation

We evaluated BOOST using a trace-driven emulation based on the measurements collected in Section II against both MPTCP variants as well as a single-path TCP proxy over both links.

A. Setup

Two Python scripts, one acting as a client and the other as a server, were used to emulate the passenger side and the content server side respectively. The client script initiated TCP connections with the same frequency as the client traces and transmitted an identifier to the server script to denote which connection in the traces corresponded to the current connection. The server script then transmitted packets over the TCP connection with the same sizes and frequency as the server traces. To understand how the performance of each scheme scales, three versions of the traces were created: one with the same number of connections, one with 3-times the number of connections and one with 5-times the number of connections. The extra connections in each version of the trace were created at the same time as the original and transmitted the same amount of data. In addition to the TCP traffic, the server script also transmitted dummy UDP video traffic to represent a closed-circuit television (CCTV) camera. The CCTV traffic was emulated by transmitting UDP packets at a rate of 1Mbit/s.

The scripts were placed inside two QEMU virtual machines (VM) with a v0.95 MPTCP kernel, as shown in Figure 9. MPTCP was configured with the default send and receive buffer sizes, and used the default MinRTT scheduler. The client VM was given two virtual interfaces to emulate the two LTE links in the measurement setup. Data transmitted from or to these virtual interface was forwarded into a corresponding virtual tunnel interface in a network emulator process running in the host. The network emulator throttled data received by or transmitted to each tunnel interface based on the capacity of the corresponding LTE link, as will be explained in more detail below. Afterwards, the packets were forwarded to the server VM or the client VM.

The client script measured the mean throughput received for all TCP connections during their period of activity, as well as the delay of the CCTV traffic. It detected periods of activity by observing a flag set by the server script to indicate the end of a burst of transmission, which are packets that were sent back to back. This is to eliminate the impact of on/off connection on the mean throughput. The CCTV traffic delay was measured by synchronizing the clocks on both client and server VMs, which was easy since they're hosted on the same machine, and attaching a timestamp inside each transmitted UDP packet by the server script.

An MPTCP kernel probe, based on the FTrace interface, was used in both VMs to calculate the re-transmission rate and the CDF of the out-of-order queue size. The former was calculated by comparing the number of total re-transmission events and the number of transmission events. The latter was calculated by observing enqueuing and dequeuing events into the out-of-order queue. Data on the same performance metrics was also collected for BOOST.

6 different schemes were evaluated: MPTCP C1, MPTCP C2, TCP C1, TCP C2, MPTCP TUN, and BOOST. The first two, MPTCP C1 and C2, represent MPTCP proxy setups with default route set to carrier 1 and carrier 2 respectively. There was no need for the explicit usage of a proxy here since the MPTCP kernel in the client VM transparently converted TCP connections to MPTCP connections. The second two, TCP C1 and TCP C2, are single-path TCP with default route on carrier 1 and carrier 2 respectively. MPTCP TUN represents the MPTCP tunnel setup. To realize this setup, we used an OpenVPN client on the client VM and OpenVPN server on the server VM. It was configured to tunnel traffic over TCP and all traffic was routed through its tunnel interface. We also stripped the MPTCP options from the non-OpenVpn TCP flows in order to ensure there's only a single MPTCP flow between the two OpenVpn processes on each of the VM. BOOST was evaluated in a similar way to the MPTCP TUN by setting up the BOOST client as a transparent proxy client and the BOOST server as a proxy server. To improve CCTV performance, BOOST was configured to replicate the CCTV and prioritize it.

The whole setup ran on a Dell XPS 15 running Ubbuntu 20 with an Intel i7-7700HQ processor and 16GB of RAM. Each QEMU instance was given 2 different cores and 4 GB of RAM.

B. Emulating capacity

The emulator was implemented in C++ as a separate user-space process, which intercepts packets transmitted by both the client and server VMs using IPTables [12]. As shown in Figure 9, data flowing through a particular interface had its capacity emulated to reflect the corresponding LTE link capacity in the traces. Once packets were received by the emulator, the packets would be added to the queue specific to that LTE link.

The queued packet is then delayed sufficiently or dropped to reflect the capacity profile that belonged to that link. A packet would be dropped if the queue size exceeded 1000 packets. If the packet is not dropped, it will be removed from the queue once its delay duration expires and forwarded to the destination VM. The emulator performs a busy wait whenever it has packets in the queue in order to eliminate the need of high resolution timers and the in-accuracy associated with them. The other advantage of this emulator compared to the Linux built-in network emulator NetEm [13] is the ability to switch very efficiently between different rates as the emulation progresses in time without relying on expensive system calls.

1) Calculating packet delay: The emulator calculates delay for each packet based on the current capacity at the time of packet transmission in the emulation and the queuing delay. Using c^i, defined in Section II-A, which is the capacity in the period $[t^i_{first}, t^i_{last}]$, the delay d^j for packet of size s^j and transmitted at time t^j_{send} where $t^j_{send} \in [t^i_{first}, t^i_{last}]$ is computed as follows

$$d^i = \frac{s_i}{c^i} + q_i \tag{3}$$

where q_i is queuing delay resulting caused by buffer buildup. In case where the number of queued packets in the emulator exceeds 1000, the packet will be dropped due to buffer overflow.

VI. Evaluation Results

Figures 10(a)(b)(c) show the CDF of the mean flow throughput for passenger WiFi traffic with the connection multiplier at 1x, 3x and 5x respectively. We can observe that BOOST consistently outperformed all solutions, with the gap widening as the multiplier increased. In the 5x multipler, BOOST had a mean throughput of more than 2.2. Mbit/s, which is around 10 times higher than any of the MPTCP variants. MPTCP C1 performed very similarly to TCP C1 and MPTCP C2 performed very similarly to TCP C2, which implies that the performance was mainly dependent on the choice of default route and that multi-connectivity provided by MPTCP did not improve performance. MPTCP TUN performed by far the worse of all configurations due to the spurious re-transmissions caused by TCP meltdown as discussed in Section III-D1. This becomes clear when observing the high re-transmission ratio in Figure 10(j), where the MPTCP TUN had almost 18% re-transmission rate in the 3x and 5x scenarios. Figure 10(d)(e)(f) shows the out-of-order queue size for all schemes. The hybrid multi-path scheduler allowed BOOST to practically eliminate the need for an out-of-order queue since most TCP connections were load balanced over individual links.

Figure 10 (i) show the CCTV delay for all solutions. We observe that MPTCP TUN had the best CCTV delay, with BOOST having similar delay but higher 95% percentile. The reason for this is that CCTV was sent over UDP, and MPTCP TUN had extremely low TCP throughput due to the TCP meltdown effect, hence freeing up almost all the capacity for the CCTV UDP-based traffic. However, it can be seen that BOOST performed significantly better than all the remaining solutions due to the prioritization and the redundancy.

Overall, BOOST provides better throughput, with lower re-transmissions, and a smaller memory footprint compared to all MPTCP variants for TCP traffic, and improved reliability for UDP traffic.

VII. Related Work

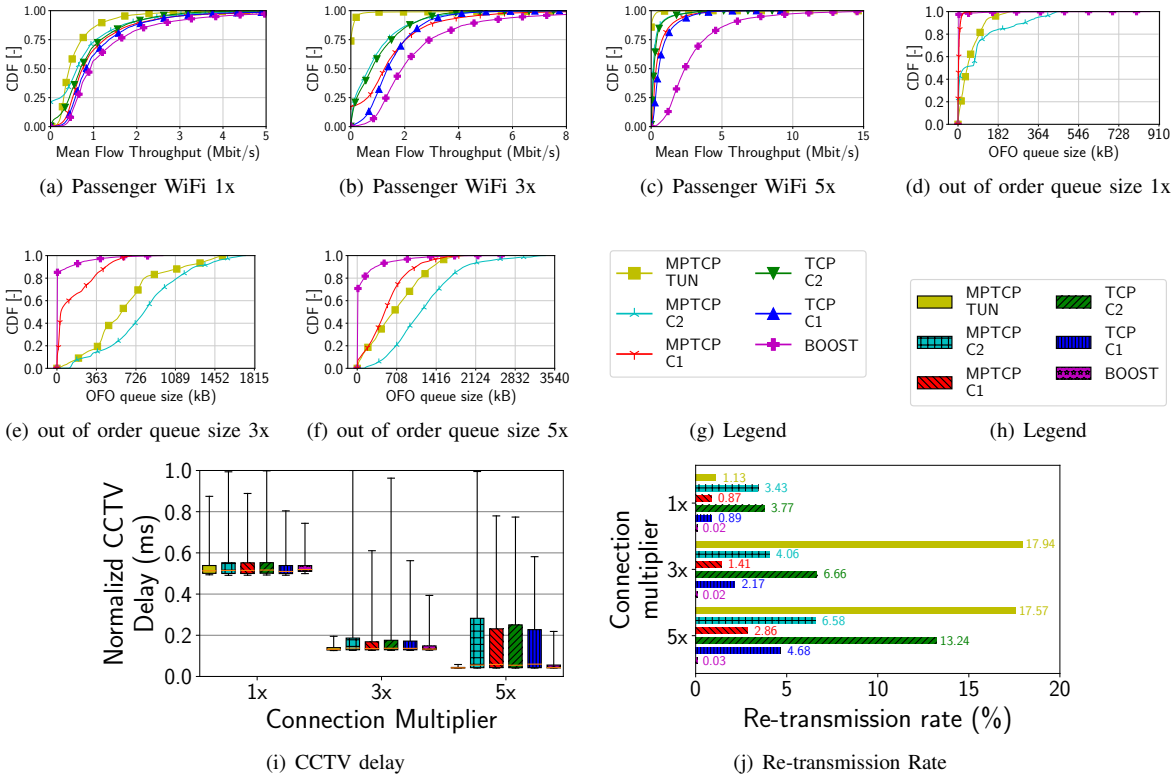
Two related studies have been done into the performance of MPTCP when handling a large number of parallel flows [14] [15]. In [14], authors propose a Software-Defined Networking (SDN) architecture that enables coordinated multi-path routing across different managed networks to address issues with un-coordinated MPTCP scheduling. Specifically, they propose a weighted MPTCP round-robin scheduler that transmits percentages of the traffic per network interface according to a configuration that is communicated through a centralized scheduler. In [15], the authors proposes a client-only approach to load balance TCP connections across multiple network interfaces. They base the validity of this approach on the fact that typical web traffic creates a large number of TCP connections creating ample opportunity for connection-level load balancing to improve network performance.

In the area of optimizing MPTCP performance when dealing with different kinds of flows, authors in [16] make the observation that the MPTCP scheduling approach of one-size-fits-all is not efficient when dealing with smaller flow sizes. They separate flows into Elephant flows, Mice flows and Mosquito. They suggest Elephant flows should be scheduled over all available links using round-robin, mice flows should be scheduled using redundancy, and Mosquito flows should be simply sent on the best available path. They propose a cross-layer solution called Flow Size Aware MPTCP (FSA-MPTCP) where applications signal the kind of flow they are about to transmit to the MPTCP scheduler so it can select the appropriate scheduling strategy.

In [17] authors aim to address the high memory requirement of MPTCP connections created by HoL blocking on IoT devices with limited memory resources. They propose an application-layer scheme that manipulates an underlying MPTCP implementation to allow the application-layer to transmit its objects over different links but ensuring each object is sent over one link only. The application utilizes multiple MPTCP sockets and writes different objects to different sockets. To ensure that the MPTCP implementation does not schedule all the objects across the same link, the application will alternate configuring one link as backup, using MPTCP socket options, for each socket it creates, ensuring that MPTCP will not use the backup link unless the first link goes down.

VIII. Conclusion

In this paper, we outlined the issues in the tunnel and proxy variants of MPTCP MWR. The issues in the proxy variant were confirmed using measurements from a production T2G communication system that relies on an MPTCP proxy MWR. In particular, measurements showed that the production system suffers from underutilization of capacity, sub-optimal multi-path scheduling and high loss. Issues in the tunnel variant were confirmed the using trace-driven emulation based on the same measurements collected from T2G system. To address the issues in both variants, we proposed a new multi-path solution more suited to MWR, called BOOST. This solution eliminates the problems with both the proxy and tunnel approaches by multiplexing TCP and UDP connections over a single persistent multi-path connection. BOOST also takes a novel approach to multi-path scheduling that combines multi-path load balancing and scheduling. In particular, short flows are transmitted across a single link to avoid HoL blocking while longer flows are opportunistically transmitted across multiple paths, utilizing left-over capacity. Evaluations based on trace-driven emulations of BOOST against both variants of MPTCP as well as single-path TCP showed that BOOST provides better throughput, lower loss than all other schemes and consumes less memory than all MPTCP variants. Additionally, BOOST provided improved



(a) Passenger WiFi 1x (b) Passenger WiFi 3x (c) Passenger WiFi 5x (d) out of order queue size 1x

(e) out of order queue size 3x (f) out of order queue size 5x (g) Legend (h) Legend

(i) CCTV delay (j) Re-transmission Rate

Fig. 10: (a)(b)(c) shows the CDF of mean flow throughput for all solutions with connection multipliers 1x, 3x and 5x respectively. (e)(f)(g) shows the CDF for the out-of-order queue size for all solutions with connection multipliers 1x, 3x and 5x respectively. (a) shows the boxplot for the CCTV frame delay for all solutions with connection multipliers 1x, 3x and 5x. (b) shows the re-transmission percentage for all solutions with connection multipliers 1x, 3x and 5x.

latency and reliability for UDP traffic through redundancy and prioritization.

REFERENCES

[1] Train to ground. [Online]. Available: https://www.nokia.com/networks/solutions/train-to-ground/

[2] Multipath TCP solutions for better connectivity. [Online]. Available: https://www.tessares.net/

[3] I. Coonjah, P. C. Catherine, and K. M. S. Soyjaudah, "An investigation of the TCP meltdown problem and proposing raptor codes as a novel to decrease TCP retransmissions in VPN systems," in *Information Systems Design and Intelligent Applications*, ser. Advances in Intelligent Systems and Computing, S. C. Satapathy, V. Bhateja, R. Somanah, X.-S. Yang, and R. Senkerik, Eds. Springer, pp. 337–347.

[4] Transparent proxy support — the linux kernel documentation. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/tproxy.html

[5] ftrace - function tracer. [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[6] iPerf - the TCP, UDP and SCTP network bandwidth measurement tool. [Online]. Available: https://iperf.fr/

[7] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing TCP's initial congestion window," vol. 40, pp. 26–33.

[8] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," vol. 42, pp. 64–74.

[9] How does HTTP/2 solve the head of line blocking (HOL) issue. [Online]. Available: https://community.akamai.com/customers/s/article/How-does-HTTP-2-solve-the-Head-of-Line-blocking-HOL-issuelanguage=en-US

[10] Q. De Coninck and O. Bonaventure, "Multipath QUIC: Design and evaluation," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. Association for Computing Machinery, pp. 160–166. [Online]. Available: https://doi.org/10.1145/3143361.3143370

[11] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The QUIC transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. Association for Computing Machinery, pp. 183–196. [Online]. Available: https://doi.org/10.1145/3098822.3098842

[12] iptables(8) - linux man page. [Online]. Available: https://linux.die.net/man/8/iptables

[13] A. Jurgelionis, J. Laulajainen, M. Hirvonen, and A. I. Wang, "An empirical study of NetEm network emulation functionalities," in *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–6.

[14] T. D. Schepper, J. Struye, E. Zeljković, S. Latré, and J. Famaey, "Software-defined multipath-TCP for smart mobile devices," in *2017 13th International Conference on Network and Service Management (CNSM)*, pp. 1–6.

[15] A. Al-Najjar, S. Teed, J. Indulska, and M. Portmann, "Flow-based load balancing of web traffic using OpenFlow," in *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pp. 1–6.

[16] Y. Xing, J. Han, K. Xue, J. Liu, M. Pan, and P. Hong, "MPTCP meets big data: Customizing transmission strategy for various data flows," vol. 34, no. 4, pp. 35–41.

[17] J. Hwang and J. Yoo, "A memory-efficient transmission scheme for multi-homed internet-of-things (IoT) devices," vol. 20, p. 1436.