MuSher: An Agile Multipath-TCP Scheduler for Dual-Band 802.11ad/ac Wireless LANs

Swetank Kumar Saha*, Shivang Aggarwal*, Rohan Pathak*,
Dimitrios Koutsonikolas*, Joerg Widmer†
*University at Buffalo, The State University of New York, NY, USA †IMDEA Networks Institute, Madrid, Spain
{swetankk,shivanga,rpathak3,dimitrio}@buffalo.edu,joerg.widmer@imdea.org

ABSTRACT

Future WLAN devices will combine both IEEE 802.11ad and 802.11ac interfaces. The former provides multi-Gbps rates but is susceptible to blockage, whereas the latter is slower but offers reliable connectivity. A fundamental challenge is thus how to combine those complementary technologies, to make the most of the advantages they offer. In this work, we explore leveraging Multipath TCP (MPTCP) to use both interfaces simultaneously in order to achieve a higher overall throughput as well as seamlessly switching to a single interface when the other one fails. We find that standard MPTCP often performs sub-optimally and may even yield a throughput much lower than that of single path TCP over the faster of the two interfaces. We analyze the cause of these performance issues in detail and then design *MuSher*, an agile MPTCP scheduler that allows MPTCP to fully utilize the channel resources available to both interfaces. Our evaluation in realistic scenarios shows that *MuSher* provides a throughput improvement of up to 1.5x/2.3x and speeds up the recovery of a traffic stream, after disruption, by a factor of up to 8x/75x, under WLAN/Internet settings respectively, compared to the default MPTCP scheduler.

ACM Reference Format:
Swetank Kumar Saha, Shivang Aggarwal, Rohan Pathak, Dimitrios Koutsonikolas, Joerg Widmer. 2019. MuSher: An Agile Multipath-TCP Scheduler for Dual-Band 802.11ad/ac Wireless LANs. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19), October 21–25, 2019, Los Cabos, Mexico.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3300061.3345435

MobiCom '19, October 21–25, 2019, Los Cabos, Mexico

ACM ISBN 978-1-4503-6169-9/19/10…$15.00
https://doi.org/10.1145/3300061.3345435

1 INTRODUCTION

Millimeter-wave (mmWave) wireless is fast emerging as the prime candidate technology for providing multi-Gbps data rates in future wireless networks. The IEEE 802.11ad standard with its 2 GHz-wide channels provides data rates of up to 6.7 Gbps, a multi-fold increase over legacy WiFi throughput. Multiple 802.11ad-compliant commercial devices (both APs and laptops) have been released over the past few years and the technology is already making its way into smartphones [1, 34].

Nonetheless, communication at mmWave frequencies faces fundamental challenges due to the high propagation and penetration loss and the use of directional transmissions makes links susceptible to disruption by human blockage and client mobility. Even if PHY and MAC layer improvements (e.g., [19, 30, 45, 48]) result in faster beam steering and lower re-connection times in the future, any realistic indoor scenario is expected to contain enough dynamism to cause a large number of re-connection events, which will hurt application performance and result in poor user experience. Further, due to the mmWave channel characteristics, providing full coverage at 60 GHz is extremely difficult and realistic deployments are likely to have some coverage gaps.

In this work, we tackle the challenge of supporting the multi-Gbps throughput provided by the 60 GHz technology while still providing the reliability of legacy WiFi, which is the key for wide-spread adoption of 60 GHz WLANs. Using both 802.11ad and 802.11ac interfaces simultaneously not only offers reliability by providing a fall-back option in case 60 GHz connectivity becomes unavailable but also allows a client to theoretically obtain the sum of the data rates offered by the two technologies. Commercial off-the-shelf (COTS) APs as well as client devices offer tri-band solutions with 2.4, 5, and 60 GHz interfaces [4, 5] and thus such a multi-band approach is feasible with existing hardware.

A critical architectural choice is at which layer of the protocol stack to implement such a solution. We explore Multipath TCP (MPTCP) [17], a transport layer protocol that can use the 802.11ad and 802.11ac interfaces simultaneously to achieve higher throughput when both networks are available

and can seamlessly fall back to 802.11ac in an application-transparent manner when the 802.11ad network becomes unavailable. Although standardized only recently, MPTCP is gaining increasing popularity among smartphone vendors, telecom providers, and startups [2, 3, 6].

MPTCP's design as a transport layer solution decouples it from both the application layer and the IP and MAC layers. Solutions that try to achieve a similar functionality at the MAC layer, such as 802.11ad's Fast Session Transfer (FST) [22], will need to have mechanisms for re-ordering of packets from different interfaces at the receiver in order to provide an in-order data stream and be transparent to higher layers. This would invariably require introducing global sequence numbers at the MAC layer across all interfaces and maintaining a queue to perform re-ordering – an unnecessary duplication of functionality already provided by the transport layer. In fact, a recent work [44] showed that a baseline MAC layer solution utilizing the Linux bonding driver without a mechanism for in-order delivery performs worse than using the 802.11ad interface alone, as excessive re-ordering presents a challenge to TCP. Further, given that FST is part of the 802.11ad specification, any modifications to fix such issues would make it non-standard compliant.

Despite its attractive features, using MPTCP in multi-band WLANs is far from straightforward. A large number of recent studies investigated the performance of MPTCP in scenarios combining WiFi and cellular interfaces [9, 14, 16, 38, 39]) and showed that the protocol performs poorly over heterogeneous paths, due to various interactions among different components of MPTCP. More recent works [25, 40, 44] showed that MPTCP in dual band 5/60 GHz WLANs often yields lower performance than using the 802.11ad interface alone, and the authors in [25, 44] even argue that the two radios should never be used simultaneously.

In contrast, to the best of our knowledge, our work is the first to show that the use of MPTCP is not only viable but a promising solution towards dual-band 5/60 GHz WLANs. We begin with an extensive experimental study using COTS APs and laptops to understand the causes of the observed performance and uncover the pitfalls of the current MPTCP implementation in this new setting. Our study reveals that MPTCP can achieve near optimal throughput under baseline, static scenarios. However, realistic dynamic environments, e.g., with contention in the 5 GHz band or blockage of the 802.11ad link, are extremely challenging for the current MPTCP architecture and result in severe performance degradation. We then design and implement ***MuSher***, a novel MPTCP scheduler that addresses the root-cause of the performance degradation, allowing MPTCP to perform near-optimally under a wide variety of dynamic use cases.

In summary, our work makes the following contributions:

(1) We develop a comprehensive set of tools to instrument MPTCP components, like the queues and scheduler, that help us study the protocol's performance and understand the root-cause of various performance issues. We have made these tools publicly available[1] for others to further improve MPTCP in new settings.
(2) We conduct an extensive measurement study[2] to understand MPTCP performance in dual-band 802.11ad/ac WLANs with COTS devices under realistic settings. We find that, in contrast to previous works that strongly discourage the simultaneous use of the two interfaces, MPTCP yields near optimal throughput in static scenarios but faces a number of challenges in dynamic environments.
(3) We design *MuSher*, a novel MPTCP scheduler that addresses all the identified challenges via throughput-ratio based scheduling and a number of additional mechanisms.
(4) We implement *MuSher* in the Linux kernel[3] and evaluate it in realistic WLAN and Internet settings. We show that it achieves up to a factor of **1.5x**/**2.3x** throughput improvement and reduces recovery time from link failures by up to **8x**/**75x** in local WLAN/Internet settings compared to the default MPTCP scheduler.

2 MPTCP BACKGROUND

Fig. 1 highlights the main components of the MPTCP architecture. On the sender side, an application is exposed to a single TCP socket and outgoing segments generated by the application are placed in the *send-queue*. This queue is at the MPTCP or meta-level, on top of the subflow-level queues. The *Packet Scheduler* reads segments from this queue and assigns them to one of the available subflows. Schedulers are implemented as Loadable Kernel Modules (LKMs) and use different criteria to select the most suitable subflow for each segment. The default *minRTT* scheduler in the Linux implementation of MPTCP chooses the subflow with the smallest round-trip time (RTT) among the subflows that have free space available in their congestion-window (`cwnd`). Apart from the send-queue, a separate, higher priority *reinject-queue* is maintained for segments that need to be retransmitted.

The rate at which segments are sent out over the individual subflows is controlled by the *Congestion Control* algorithm through the use of `cwnd`, similar to single path TCP (SPTCP). MPTCP allows for both *decoupled* and *coupled* variants of congestion control. The decoupled variant runs an independent instance of the default Linux congestion control

[1] https://github.com/swetanksaha/mptcp-tools
[2] Data available at: https://buffalo.box.com/v/mobicom19-musher-data
[3] https://github.com/swetanksaha/mptcp-musher

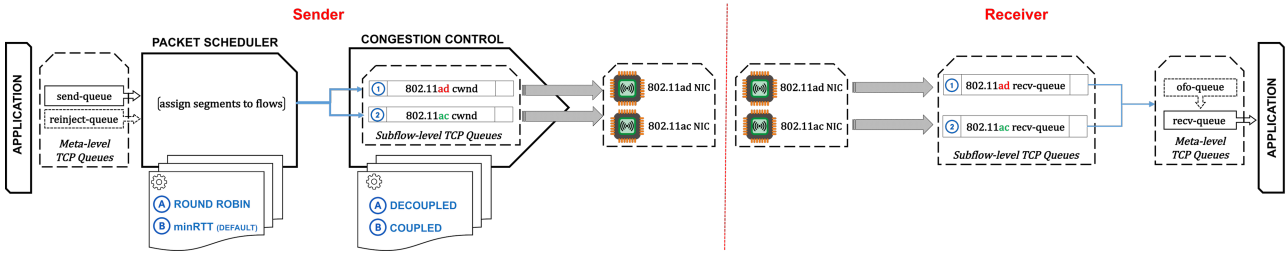


Figure 1: MPTCP Architecture (Sender and Receiver)

algorithm (typically *Cubic*) on each of the subflows while the coupled variants link the increase of the cwnd among the subflows. Use of coupled congestion control [23, 33, 37] is preferred over its counterpart as it maintains fairness with other competing flows running over a bottleneck link [37].

On the receiver side, the segments first arrive at the subflow-level receive queues and are then delivered in-order (at the subflow-level, but not necessarily globally) to a common receive buffer (*recv-queue*) at the MPTCP/meta-level. Segments arriving out-of-order at the meta-level are put in an out-of-order queue (*ofo-queue*) that is shared among all the subflows of an MPTCP connection. The space left in the shared buffer is advertised to the sender as the receive window (`recv_win`).

3 EXPERIMENTAL METHODOLOGY

3.1 Devices

Our setup consists of a Netgear Nighthawk X10 WiFi router and an Acer Travelmate P446-M laptop.

60 GHz (802.11ad) The router has the QCA9008-SBD1 module housing the Qualcomm QCA9500 chipset, which supports all the single-carrier 802.11ad data rates (from 385 Mbps up to 4.6 Gbps). The laptop has the client-version of the module, QCA9008-TBD1, which includes the 802.11ac, 802.11ad, and BT chipsets. It uses the open source wil6210 wireless driver to interface with the chipset. Both the router and the laptop use a 32-element phased antenna array. The 60 GHz radios on both devices use their default rate adaptation and beamforming algorithms.

WiFi (802.11ac) While the router supports up to 4 MIMO spatial streams, our client only supports 2, resulting in a link configuration of 2x2 MIMO, 80 MHz bandwidth, short guard interval, and default rate adaptation.

Traffic Generation and Maximum Goodput A high-end desktop is connected to the router through a 10G LAN SFP+ interface to generate/receive TCP traffic. While this setup would in theory allow us to achieve the maximum 802.11ad rate of 4.6 Gbps, we found that in practice the maximum goodput on the router is limited to 1.6-1.65 Gbps and 500-550 Mbps, with 802.11ad and 802.11ac, respectively.

3.2 MPTCP

We use MPTCP version v0.94 and make all our modifications and instrumentation on top of its code base. We make use of the *fullmesh* Path Manager, which establishes a subflow for each interface combination between the sender and receiver. Our client device is dual-homed with an 802.11ad and 802.11ac interface and the server is single-homed with a 10G Ethernet interface, hence a total of 2 subflows are created. Unless stated otherwise, we use the *default minRTT* scheduler. Finally, we use the default *Lia* coupled congestion control, which also achieves the best performance (Table 1), apart from the first measurement study where we evaluate all the available congestion control algorithms.

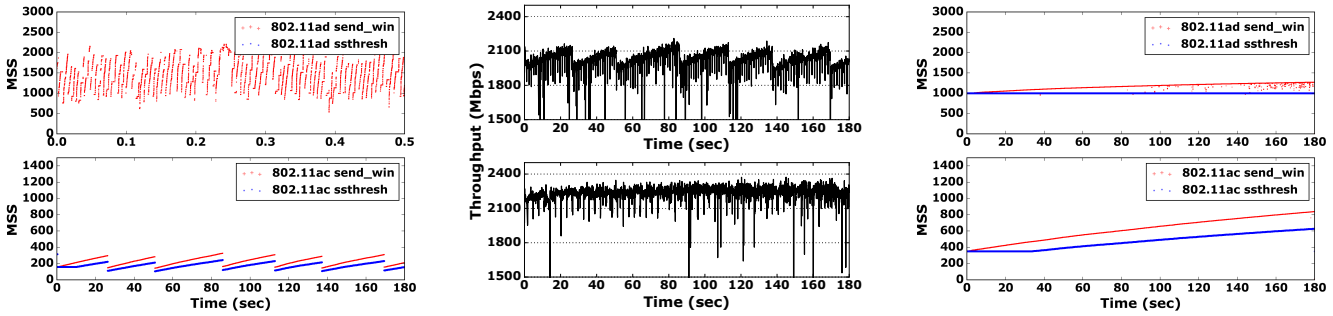
4 MPTCP PERFORMANCE & PITFALLS

In this section, we study MPTCP over dual-band 802.11ad/ac links under a wide range of scenarios using the analysis tools described in Appendix A. The goal is not only to characterize the performance but to understand and analyze the root causes of the observed behavior.

4.1 MPTCP Memory Optimizations

The Linux implementation of MPTCP (since v0.89) includes two complementary optimizations (labeled as Mechanisms 1 and 2 in [38], where they were first introduced) to reduce memory usage. While Mechanism 1 performs opportunistic re-injection of data from one subflow to the other if a flow is `recv_win` limited, Mechanism 2 halves the cwnd and sets the slow start threshold to the reduced window size for the subflow holding up the advancement of the MPTCP connection window.

Although the authors in [38] show improvements with these mechanisms for a scenario involving WiFi and 3G interfaces, our measurements reveal a significant impairment due to these optimizations. Fig. 2a shows the send-window (`send_win=min(cwnd,recv_win)`) and slow-start threshold (`ssthresh`) of the 802.11ad and 802.11ac subflows of an MPTCP connection lasting 180 s. For the 802.11ad subflow (zoomed in on the first 0.5 s), we observe a zig-zag pattern for the `send_win` which is being repeatedly halved



(a) Optimizations enabled (Top: 802.11ad; Bottom: 802.11ac)

(b) MPTCP throughput with optimizations enabled (top) and disabled (bottom)

(c) Optimizations disabled (Top: 802.11ad; Bottom: 802.11ac)

Figure 2: Memory usage optimizations

(due to Mechanism 2) causing the subflow to be stuck in the slow start phase. In fact, the subflow never enters the congestion avoidance phase as it never experiences a loss due to the premature cutting of `cwnd`. The 802.11ac subflow does exit slow start but its `send_win` and `ssthresh` are halved repeatedly over the connection lifetime, even though there is no loss event (triple duplicate ACK or timeout).

Note that these optimizations are applied here as a result of flow penalization that accompanies a forced re-transmission initiated when the `SOCK_NOSPACE` flag is set on the meta-socket on the sender side indicating that it is full. MPTCP treats being send-buffer-limited as a trigger to engage Mechanism 2 as a preemptive step to becoming *recv_win* limited in the future. These optimization induced cuts result in performance degradation and variance over time. Fig. 2b shows the MPTCP throughput of a given 802.11ad/ac link with and without the optimizations. With optimizations enabled (top plot), the mean throughput over 180 s is 2011 Mbps, whereas, with optimizations disabled (bottom plot), the throughput is improved by 216 Mbps to 2227 Mbps and, more importantly, is much more stable over time.

Fig. 2c shows the subflow `cwnd` and `ssthresh` with both mechanisms turned off. Both subflows are able to exit slow start and do not experience any cuts to their windows. Moreover, disabling the optimizations does not result in throttling of the MPTCP flow due to `recv_win` limitations at any time during the 180 s. In light of this finding, we disable both optimizations for the rest of the measurements.

4.2 Baseline Performance

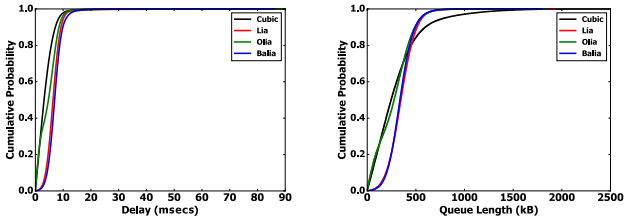
We first establish a baseline for MPTCP performance under static scenarios. We primarily look at how close MPTCP throughput is to the sum of throughputs of the two single path flows (when each of the two interfaces is used alone).

4.2.1 Congestion Control Algorithms. We experiment with four congestion control algorithms available in the Linux implementation – *Cubic* (decoupled), *Lia* [37], *Olia* [23], and *Balia* [33] – under backlogged traffic. Table 1 lists MPTCP throughput along with throughput over each interface when engaged separately for comparison. For each of the four algorithms, *MPTCP can achieve throughput very close to the expected sum* (96%-99%). This is in sharp contrast to several previous works [9, 14, 16, 38, 39] that have shown MPTCP to perform poorly when used with interfaces of heterogeneous data rates, albeit in the context of WiFi+3G/LTE, and more importantly to recent works [25, 44] arguing that 802.11ad and 802.11ac interfaces should not be used simultaneously. Note that *the sum throughput achieved by MPTCP is substantially higher than the throughput over any of the two interfaces alone*. E.g., compared to MUST [44], a MAC layer solution that only uses the 802.11ad interface and switches to 802.11ac in case of blockage, the use of MPTCP would result in a throughput boost of 31%-36%. We also verified that MPTCP can sustain the provided application data rates under non-backlogged traffic.

Table 1: MPTCP congestion control algorithms

	802.11ad only	802.11ac only	MPTCP	Expected Sum	% Sum Achieved
Cubic	1649 ± 74	591 ± 23	2167 ± 162	2240	96.74
Lia	1631 ± 89	596 ± 25	2227 ± 95	2228	99.99
Balia	1638 ± 99	595 ± 22	2230 ± 78	2233	99.83
Olia	1649 ± 121	585 ± 18	2192 ± 112	2235	98.05

Delay. We use the time spent by packets in the MPTCP meta-level *ofo-queue* as a measure of application-perceived delay. This is a much more realistic measure for studying the MPTCP-induced delay due to packet re-ordering required at the meta-level, as it isolates the extra delay a receiver experiences due to MPTCP from the delay of the individual subflows which occurs even if we were to use SPTCP for each subflow. Additionally, the queue length is also a measure of the amount of re-ordering induced by MPTCP. Fig. 3a and 3b plot the *ofo-queue* delay and queue length for each of the four congestion control algorithms. The maximum delay and queue length are upper-bounded by 10 ms and 1.5 MB,

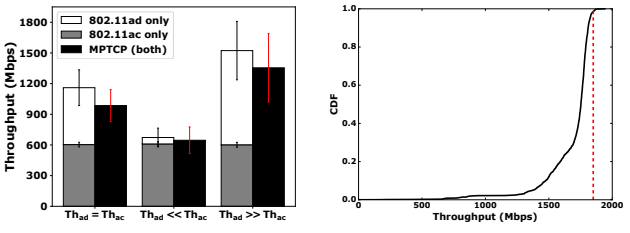


(a) Delay of packets in *ofo-queue* (b) Length of *ofo-queue*

Figure 3: MPTCP impact on delay and queue length

respectively. In the median case, MPTCP adds only a 5 ms delay over SPTCP. Further, since the queue length in the median case is well below 500 kB, using MPTCP does not impose extra memory requirements on the system while providing significant throughput gains.

4.2.2 Suboptimal Links. In §4.2.1, we considered optimal links that can individually support the highest throughput possible with our hardware. We now repeat the measurements over suboptimal links. We first examine a scenario involving an optimal 802.11ac link and a suboptimal 802.11ad link. We vary the quality of the 802.11ad link by changing the client-AP distance and consider three cases based on the relationship between the 802.11ad throughput (Th_{ad}) and the 802.11ac (Th_{ac}) throughput. Fig. 4a shows for each of the three cases the throughput of the two interfaces when they are used individually and when they are used together with MPTCP. We observe that the MPTCP throughput is greater than 90% of the expected sum for the $\mathsf{Th}_{ad} \gg \mathsf{Th}_{ac}$ and $\mathsf{Th}_{ad} \ll \mathsf{Th}_{ac}$ cases and 85% of the expected sum for the $\mathsf{Th}_{ad} = \mathsf{Th}_{ac}$ case. Further, in each of the three cases, the MPTCP throughput is higher than the best of the two single path throughputs (Goal 1 in RFC 6356 [37]). Compared to MUST, this translates to a throughput improvement ranging from 44% (when $\mathsf{Th}_{ad} \gg \mathsf{Th}_{ac}$) to 5x (when $\mathsf{Th}_{ad} \ll \mathsf{Th}_{ac}$).



(a) Sub-optimal 802.11ad link (b) Sub-optimal 802.11ac link

Figure 4: MPTCP throughput with suboptimal links

We then examine a scenario involving an optimal 802.11ad link and a suboptimal 802.11ac link. Since it is not possible to drop the quality of the 802.11ac link by moving the client away from the AP (as that would also result in a steep drop of the 802.11ad link quality), we instead fix the 802.11ac channel width to 20 MHz. Fig. 4b presents a CDF of the MPTCP throughputs collected over several runs of 60 s at 10 randomly selected locations. MPTCP throughput is close to the expected sum (vertical line) most of the time and only for around 17% of the cases is it worse than the faster of the two interfaces (802.11ad/1600 Mbps).

4.2.3 Field Evaluation. We finally perform a field test of MPTCP in three realistic indoor locations – conference room, lab, and lobby – in an academic building. We consider 5 links of varying quality at each of these locations. While the 802.11ac link SNR at the different locations in a given room is similar, the 802.11ad links are affected more by distance and furniture and experience different SNRs. Fig. 5 shows the single path and MPTCP throughputs for the 15 links considered in the trial. Under almost all link conditions, MPTCP throughput is very close to the expected sum. In the only scenario where MPTCP does not provide throughput close to the expected sum (lab, link L5), its average throughput is as high as that of the faster of the two interfaces (in this case 802.11ac), again satisfying Goal 1 in [37] and outperforming MUST, while also adding reliability by allowing for smooth switch-over, if needed.

4.3 Understanding MPTCP Performance

Our measurements in §4.2 clearly demonstrate that MPTCP can provide substantial performance improvement in a wide variety of link quality and environment scenarios, challenging the generally accepted consensus that MPTCP should not be used with heterogeneous interfaces. Thus, a root-cause analysis is needed to answer why MPTCP works well with the specific scenario involving 802.11ad and 802.11ac interfaces. Since our observations in §4.2.1 already indicate that congestion control does not have an impact on the MPTCP throughput, we turn our attention to another key MPTCP component: the **packet-scheduler**, responsible for the distribution of application traffic among the subflows.

4.3.1 minRTT *Packet Scheduler.* We use the tools described in Appendix A to understand the packet assignment dynamics of the *minRTT* scheduler. We looked at the scheduler decisions over the connection lifetime and found that it consistently assigned ~77% of the packets to the 802.11ad subflow and the remaining 23% to 802.11ac. Further investigation of the scheduler decision reasons interestingly showed that for the majority of the time, `cwnd` was full for one of the subflows, thereby forcing the selection of the other subflow. While one might expect the *minRTT* scheduler to primarily make decisions based on the comparison of the RTT values of the two subflows, under backlogged traffic the decisions are essentially controlled by how and when the space opens up in a subflow's `cwnd`. Our results confirm a similar finding

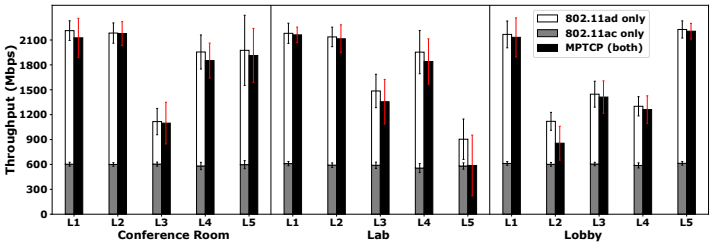


Figure 5: Field trial

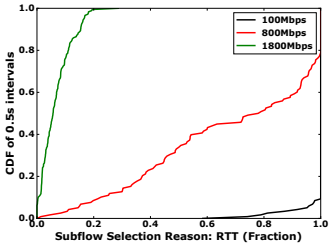


Figure 6: Subflow selection reason

reported in [31], where the authors remark that, under saturated congestion windows, the scheduling decision becomes ACK-clocked.

On the other hand, under non-backlogged traffic, RTT becomes the deciding factor. Fig. 6 plots the CDF of the fraction of time during which the scheduler makes a decision based on RTT in each 0.5 s interval for source application rates of 400, 1200, and 1800 Mbps. For the high source rate of 1800 Mbps, which is close to the overall combined channel capacity of ~2100 Mbps, cwnd occupancy is the deciding factor for the majority of the 0.5 s intervals. When dropping the source rate to 1200 Mbps, a significant portion of scheduler decisions are based on subflow RTT values. Finally, with a low source rate of 400 Mbps, almost all packet assignment decisions are made based on RTT values.

4.3.2 Impact of Packet Scheduling Decisions. Given that, in case of backlogged traffic, ACK-clocked scheduling decisions result in a certain packet distribution between the two subflows, we now investigate in more detail how the traffic distribution between the subflows impacts MPTCP performance. To this end, we design an MPTCP scheduler *FixedRatio* that performs packet assignment based on a user-defined ratio (see appendix A).

Fig. 7a plots the MPTCP throughput against the number of packets assigned to the 802.11ad subflow ($\texttt{Pkts}_{ad}$) out of every 100 packets. In each case, the remaining packets (out of 100) are assigned to the 802.11ac subflow ($\texttt{Pkts}_{ac}$=100-$\texttt{Pkts}_{ad}$). Maximum throughput of ~2.1 Gbps is achieved with $\texttt{Pkts}_{ad}$=77 and performance worsens as we move away from this value with the worst throughput being as low as 400 Mbps ($\texttt{Pkts}_{ad}$=5).

We found that the stark difference in performance with different assignment ratios is a result of the degree to which packets arrive *out-of-order* in the end-to-end MPTCP flow due to the specific distribution of traffic among the subflows. A higher number of out-of-order packets can cause packets to be buffered in the receiver's *ofo-queue* and in extreme cases can even result in throttling of the sender because of limited space in the receiver's buffer. In fact, in Fig. 7b, which plots the CDF of the delay experienced by the data bytes in the *ofo-queue*, we observe that the $\texttt{Pkts}_{ad}$=77 value indeed yields the lowest delay. In general, the $\texttt{Pkts}_{ad}$ values that result in high delay are the ones that result in lower throughput and vice-versa.

We also plot the CDF of the *ofo-queue* occupancy under different $\texttt{Pkts}_{ad}$ values in Fig. 7c. One might also expect to see smaller queue lengths (indicating less out-of-ordering) for packet assignments corresponding to $\texttt{Pkts}_{ad}$ values that yield higher throughputs. However, under extreme $\texttt{Pkts}_{ad}$ values (e.g., 5, 95), the traffic distribution is so skewed towards one of the subflows that almost all the packets flow through one of the interfaces, thereby significantly reducing reordering. As a result, extreme $\texttt{Pkts}_{ad}$ values (5, 15, 25, 85, 95), although sub-optimal throughput-wise, have queue lengths smaller than the $\texttt{Pkts}_{ad}$=77 case. Excluding the extremes, other $\texttt{Pkts}_{ad}$ values show a general trend of having larger queues in conjunction with lower throughput.

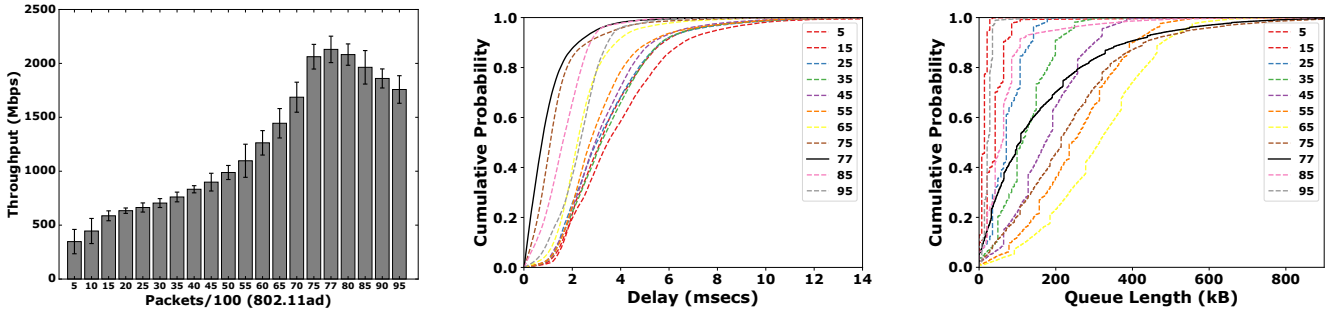
Throughput-optimal ratio. The reason for $\texttt{Pkts}_{ad}$=77 resulting in optimal throughput is that the underlying packet-distribution ratio imposed by this assignment $\texttt{Pkts}_{ratio}$=$\texttt{Pkts}_{ac}$/$\texttt{Pkts}_{ad}$=23/77=0.30 is nearly identical to the ratio of the actual individual throughputs of the two interfaces $\texttt{Tput}_{ratio}$=$\texttt{Tput}_{ac}$/$\texttt{Tput}_{ad}$=500/1600=0.31. Assigning packets with this very specific ratio minimizes the chance of packets arriving out-of-order at the meta-level MPTCP buffers. Note that although in-order delivery of packets within a subflow (intra-subflow) is guaranteed because of SPTCP operation at the subflow level, global in-order delivery among all subflows (inter-flow) needs to be achieved through reordering at the meta-level.

Important Findings:

- Optimal MPTCP performance can be achieved when the packet-assignment ratio is close to the throughput ratio of the two subflows.
- The MPTCP throughput vs. packet assignment ratio curve is unimodal and hence a unique optimal ratio always exists for given subflow throughputs (see appendix B).

4.4 Performance Issues

All the measurements in §4.2 were limited to scenarios where both links remained stable for the duration of the experiment.



(a) Throughput with different ratios (b) Delay in the *ofo-queue* for different $\texttt{Pkts}_{ad}$ (c) Length of the *ofo-queue* for different $\texttt{Pkts}_{ad}$

Figure 7: Impact of packet scheduling decisions

We now look at more challenging scenarios where we relax this assumption and show that the default MPTCP architecture results in sub-optimal performance, often worse than that of SPTCP over the faster subflow.

4.4.1 Varying Channel Conditions. Realistic WLAN scenarios involve cases where link conditions and thereby channel capacity change over time for the two interfaces, e.g., due to contention or mobility. We consider a case where the 802.11ac link experiences contention from nearby competing links. Fig. 8a shows a timeline of the per-flow throughput of a 180 s MPTCP session. We start with a static link where 802.11ad and 802.11ac are at their maximum throughputs and we introduce contention with 300 Mbps TCP cross-traffic at the 30^{th} s for 30 s. The throughput of the 802.11ac subflow drops by 300 Mbps to ~250 Mbps, as expected. Surprisingly, the 802.11ad subflow is also affected negatively during the contention period with its throughput dropping below 1200 Mbps and exhibiting much more variability than in the preceding interval. In fact, the MPTCP throughput during the contention period averages to ~1450(=1200+250) Mbps, which is less than even that of 802.11ad operating alone (1650 Mbps). Note that 802.11ad channel capacity is unchanged as the contention exists only on the 802.11ac link.

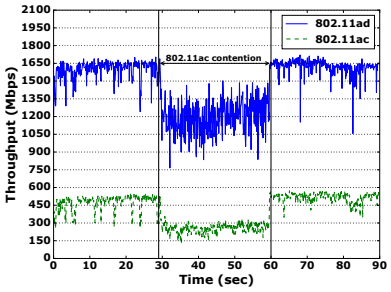
A look at Fig. 8b, which plots the TCP congestion control parameters for the two subflows, explains the unexpected performance drop in 802.11ad. During the contention period, the receiver advertised buffer space (`recv_win`) reduces significantly. Remember that the `recv_win` is maintained at the meta-level and, although advertised on both subflows, is actually shared among them. In this particular case, the sum of `cwnd` values of the two interfaces of 850 (=350+500) MSS exceeds the available receiver buffer space (which varies between 500 and 1000 MSS) several times during the contention period. Under such a scenario, the meta-level global sequence numbers cannot advance, even though `cwnd` allows for it, since the meta-level buffers at the receiver are full, resulting in reduction of throughput on both interfaces. We further confirmed this finding by instrumenting the MPTCP sender to log events where it was unable to send data packets due to being `recv_win` limited.

We observe similar effects when 802.11ad link capacity is varied under different scenarios such as increase/decrease in distance between the AP and the laptop or partial link blockage by humans.

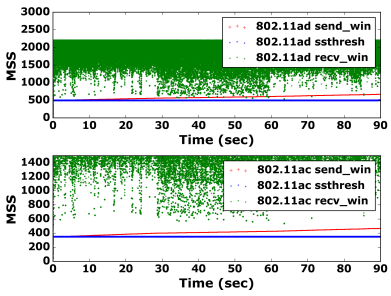
4.4.2 Network Scans. For all the results presented in §4.2 we had disabled the periodic channel scans, which are typically initiated by the network-manager or similar user-space utilities, to avoid biasing our throughput measurements. However, disabling periodic channel scans is problematic in any real scenario as it prevents the client from finding APs with a better link quality or performing efficient handovers.

In order to observe any potential impact of network scans on performance, we start an 802.11ac scan during an MPTCP session. Fig. 9a shows the throughput of the 802.11ad and 802.11ac subflows over 60 s and the scan initiated at the 30 s mark. The 802.11ac throughput is cut down severely during the scan period that lasts for around 6 s. This is expected as the radio is unable to transmit regular data frames during this period. Surprisingly, we observe that the 802.11ad flow is also impacted negatively during this period, even though the scan takes place in the 5 GHz band.

Looking at the `cwnd` values of both subflows during the 802.11ac scan, we find that they are not affected. However, we observe a 6x increase (Fig. 9b) in the amount of data held in the *ofo-queue* at the receiver end. During the scan period, the packet scheduler, which is not aware of the sudden reduction in 802.11ac channel capacity, keeps assigning packets to the 802.11ac subflow even though the interface cannot transmit them immediately. This is problematic as the receiver's packet stream now has *gaps* (missing in-sequence packets). These gaps prevent the receiver from delivering packets to the application until the missing packets arrive or are re-transmitted over the 802.11ad interface. Note that the MPTCP receiver is responsible for re-ordering the packets at the meta-level before delivering them to the application. MPTCP performance drops can be observed with 802.11ad

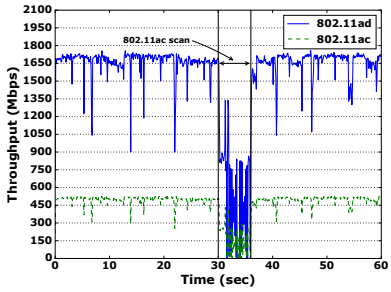


(a) Throughput timeline during contention

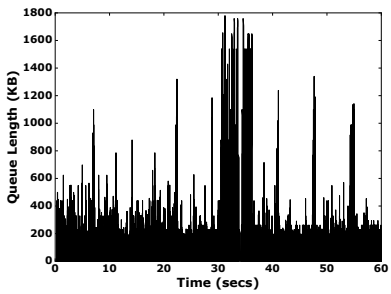


(b) *send_win*, *recv_win*, & *ssthresh*.

Figure 8: 802.11ac contention

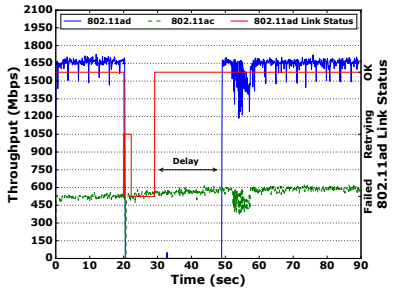


(a) Throughput timeline

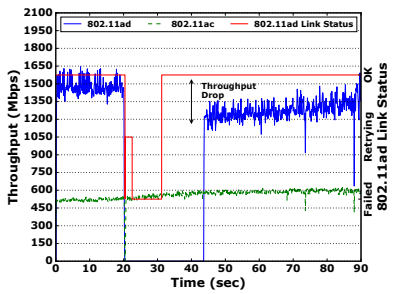


(b) Timeline: *ofo-queue* length

Figure 9: Network scan



(a) Delay in resumption of 802.11ad subflow



(b) Throughput drop after re-connection

Figure 10: 802.11ad blockage

scans as well but due the much shorter duration of the scan their impact is less pronounced.

4.4.3 802.11ad Blockage. In case of a blockage event, MPTCP should be able to switch-over as quickly as possible to using only the 802.11ac interface, without disruption to the application [44]. Additionally, once the 802.11ad link is restored, MPTCP should ideally resume using both interfaces with as little delay as possible. To study how MPTCP reacts to sudden loss of the 802.11ad link, we block the 802.11ad link by hand causing the link to break. We then remove the blockage and allow the device to re-associate with the AP.

Switch-over. Fig. 10a shows a timeline of subflow throughputs along with link status `Failed/OK/Retrying` as reported by the 802.11ad driver. A status of `OK` indicates that the client has successfully associated with the AP and the link can support data transfer. The blockage is introduced at 20 s and the link fails after further 2 s. Once the blockage is removed, connection at the MAC layer is restored at the 30^{th} second. During the entire period of 802.11ad disconnection, MPTCP maintains the 802.11ac subflow throughput without any disruption to the end-to-end connection seen by the application. MPTCP, owing to its design, provides a completely *seamless* switch-over to 802.11ac.

Restoring 802.11ad throughput. In Fig. 10a, although the 802.11ad link is restored at the 30^{th} second, MPTCP does not resume traffic on the 802.11ad subflow for another ~20 s until the 49^{th} s. We repeated this experiment multiple times and found that this extra delay in traffic resumption varied from 6 s to as much as 60 s. For comparison, we repeated the same experiment with a UDP flow over the 802.11ad interface and found that it resumed as soon the driver reported `OK` status. On further investigation, we discovered that interaction between the MPTCP scheduler and TCP congestion-control of the 802.11ad subflow is responsible for the extra delay. In a timeout-based loss event (because of blockage), TCP congestion-control sets the `pf` flag on the socket, indicating it to be *potentially failed*. The MPTCP scheduler treats subflows with the `pf` flag set as being unavailable and does not schedule any packets on them. TCP congestion-control, on the other hand, is waiting for an ACK to unset the `pf` flag and enter the `TCP_CA_RECOVERY` state that can restore the `cwnd` to the value before the loss event. Since no packets are being directed to the 802.11ad subflow, only a subflow-level re-transmission of the 802.11ad subflow can trigger the transmission of an ACK on the receiver side. However, multiple timeout-based losses during the blockage period can lead to excessively high retransmission timeouts, and hence long delays before an ACK is received after reconnection.

Resuming to non-optimal throughput. We also observed cases where the 802.11ad subflow, on resumption, starts with a `cwnd` and `ssthresh` that are half of their pre-loss values. Fig. 10b shows a sample timeline where the 802.11ad flow resumes to 1350 Mbps instead of 1650 Mbps. This behavior depends on the exact specifics of the TCP congestion-control state at the time it enters the recovery state. Nonetheless, it is observed quite often and has a non-negligible impact on throughput.

Important Findings:

- The default MPTCP scheduler performs sub-optimally under varying channel conditions and is unable to fully utilize the available capacities of both interfaces.
- Network scanning during an active MPTCP session on one of the interfaces can severely degrade performance of the other interface.
- In the event of 802.11ad blockage, MPTCP can seamlessly switch over to 802.11ac but has issues resuming traffic on the 802.11ad interface once the 802.11ad connectivity is restored.

5 *MUSHER*: SYSTEM DESIGN & IMPLEMENTATION

In this section, we introduce ***MuSher***, an agile **Mu**ltipath-TCP **Sch**edul**er**, aimed at improving MPTCP performance in dual-band 802.11ad/ac WLANs under diverse scenarios. We present the design of *MuSher* and the different mechanisms it employs to address all the performance issues identified in §4.4. Although such mechanisms can be used on most platforms with an MPTCP implementation available, we chose Linux for our reference implementation. To allow for easy deployment, *MuSher* is implemented entirely as an MPTCP scheduler. Given that MPTCP schedulers are modular components, implemented as LKMs that can be loaded/unloaded without requiring kernel reboot, such a design allows for *MuSher* to be used without requiring any changes to the MPTCP source code tree. Note that, although *MuSher* addresses challenges related to the underlying wireless technologies, it does *not* rely on any specific hints from the wireless interfaces or the device drivers managing them. We made these architecture choices to specifically prevent *MuSher* from being tied to any specific hardware/platform.

We first present *MuSher*'s solution to MPTCP's sub-optimal performance under varying link conditions (§4.4.1) by distributing traffic among the subflows in a throughput-optimal way, and then discuss two other key components:
(1) a SCAN component that improves MPTCP performance by mitigating the negative impact of network scanning (§4.4.2) through careful management of the subflows.
(2) a BLOCKAGE component that helps MPTCP to quickly recover to the optimal throughput after an 802.11ad blockage event (§4.4.3) by addressing the interaction between MPTCP scheduling and subflow-level congestion control.

5.1 Reacting to time-varying links

Our findings in §4.3.2 and §4.4.1 indicate that the underlying reason for the drop in throughput of the other subflow, when channel conditions change on one subflow, is that meta-level receive buffers are filling up. Assignment of packets in a ratio very different from the subflow throughput ratio results in too many out-of-order arrivals at the receiver, using up buffer space. To address this issue, we leverage the finding of 4.3.2 that there exists a unique MPTCP throughput-optimal ratio that depends on the subflow throughput values. For instance, the reaction to contention on 802.11ac is to set the packet-assignment ratio to match the ratio of the throughputs of 802.11ad and 802.11ac flows, accounting for the drop in 802.11ac throughput due to contention. E.g., under 300 Mbps contention $\texttt{Tput}_{ratio}$=$\texttt{Tput}_{ad}$/$\texttt{Tput}_{ac}$=250/1650=0.15, thus we would set $\texttt{Pkts}_{ad}$=86 (see §4.3.2) resulting in the packet assignment ratio $\texttt{Pkts}_{ratio}$=$\texttt{Pkts}_{ac}$/$\texttt{Pkts}_{ad}$=13/87=0.15.

5.1.1 Implementation. In practice, *MuSher* needs a mechanism to quickly determine the throughput-optimal ratio at runtime. Additionally, we need a light-weight mechanism to automatically trigger the search for an optimal ratio.

Finding the optimal ratio. Since the ratio vs. throughput curve is unimodal (§4.3.2), we can use a simple probing approach to find the maximum of the throughput curve and thus the optimal ratio. Specifically, we begin by probing two ratios adjacent to the current ratio, one slightly lower and one slightly higher, and proceed our search in the direction where we observe higher throughput. We obtain throughput estimates by observing the bytes transmitted given by the `tx_bytes` member of the `struct rtnl_link_stats_64`, and the timestamp of the last transmission stored in the `trans_start` member of `struct netdev_queue`. All of this information is maintained by the Linux kernel for each network interface (`struct net_device`) irrespective of the specific underlying device driver. The function `CallSearchRatio` in Algorithm 1 presents the search procedure more formally. An important parameter is the sampling time τ, which is the time spent at a given ratio to estimate the corresponding throughput. It provides a trade-off between the convergence time of the optimal-ratio search and the accuracy of the throughput estimates. We empirically set the value of τ to 200 ms to achieve the desired balance of convergence time and accuracy. For instance, using a `step_size` of 0.05 for a difference of 0.20 between the optimal and current ratio, the search would take 800 ms.

Note that we investigated several different approaches, including binary/ternary search, to find the optimal ratio. We chose our particular design based on two key observations from our measurements: (i) large changes in throughput induced by large changes in the assignment ratio (as part of binary/ternary search) introduce instability in the network for the flow under consideration and other competing flows, and (ii) large jumps in the packet assignment ratio typically require a larger sampling time to obtain accurate measurements, resulting in an increase in the overall convergence time. Our approach specifically avoids such large jumps and achieves a faster convergence time.

Triggering ratio search. To detect changes in the link capacity of either interface and trigger the search for a new optimal ratio, *MuSher* monitors two events: (i) *decrease* in total MPTCP throughput and (ii) *decrease* in *send-queue* occupancy[4] of any of the two subflows, without a change in throughput. Although (i) can detect decreases in link capacity of any of the two subflows, it cannot detect increases if the packet scheduling ratio keeps any of the two interfaces non-backlogged. Using (ii), we can detect such increases as queues are drained faster when the link capacity of the underlying interface increases. The triggering mechanism is presented formally in the while loop of Algorithm 1. Through extensive experimentation, we set the value of sleep time γ to 100 ms. Relying on an event-based trigger mechanism avoids continuous probing of ratios in search of higher throughput, which can negatively impact performance. Note that even if condition (i) or (ii) falsely trigger a ratio search, it will converge to the optimal ratio.

5.2 Managing Network Scans

MuSher arbitrates the network scan requests generated from the user space and disables the scheduling of packets to the subflow where the request has been made for the duration of the network scan. However, disabling future scheduling alone may not be enough to prevent packets from being held-up in the TCP queues or at any of the lower layer buffers. We thus adopt a two-step approach: (1) Stop the assignment of packets to the subflow about to undertake scanning and (2) Wait for the subflow-level *send-queue* to be emptied out. The scan is triggered once steps (1) and (2) are completed.

Signaling scan operation to the sender. The approach discussed above works well in the uplink case, when the client, whose network interface is performing the scan, is the MPTCP sender. In the downlink case, the client needs to notify the other end of the MPTCP connection to temporary disable all traffic to the subflow associated with the interface about to perform the scan. One option would be to tear-down the corresponding subflow but this destroys all the state information on both ends and would result in additional overhead of re-establishing the subflow once the scan is over. Instead, *MuSher* sends an ACK containing the MPTCP `MP_PRIO` option marking the interface as *backup*. The receipt of this option results in the sender stopping further scheduling of traffic on the subflow on which the ACK was received. Once the scan is complete, the client sends another ACK resetting the subflow back to regular operation.

[4]The occupancy is calculated as the difference of two internal pointers maintained by MPTCP for each subflow: `write_seq`, the highest sequence number written by the application into the send buffer, and `snd_una`, the oldest unacknowledged sequence number.

Algorithm 1 MuSher

```
ω = 200Mbps, β = 100KB, α = 3, λ = 3
γ = 100ms, τ = 200ms, δ = 5
while true do
    curr_tput_diff += (GET_CURRENT_TPUT(cur_ratio, γ) − last_tput)
    curr_buffer_diff += (GET_CURRENT_BUFFER_SIZE() − last_buffer_size)
    if |curr_tput_diff| > ω then
        tput_threshold_cnt += 1
        if tput_threshold_cnt == α then
            cur_ratio = CALLSEARCHRATIO(cur_ratio)
            SET_RATIO(cur_ratio)
    else if |curr_buffer_diff| > β then
        buffer_threshold_cnt += 1
        if buffer_threshold_cnt == λ then
            cur_ratio = CALLSEARCHRATIO(cur_ratio)
            SET_RATIO(cur_ratio)
    else
        tput_threshold_cnt = 0, buffer_threshold_cnt = 0

function SEARCHRATIO(start, stop, step_size)
    prev_tput = 0
    for ratio = start to stop step step_size do
        SLEEP(τ)
        cur_tput = GET_CURRENT_TPUT(ratio, τ)
        if cur_tput < prev_tput then
            return ratio − step
        prev_tput = cur_tput
    return stop

function CALLSEARCHRATIO(cur_ratio)
    ratio_right = cur_ratio + δ, ratio_left = cur_ratio − δ
    tput_right = GET_CURRENT_TPUT(ratio_right, τ)
    tput_left = GET_CURRENT_TPUT(ratio_left, τ)
    if tput_left > tput_right then
        return SEARCHRATIO(cur_ratio, 0, −δ)
    else if tput_left < tput_right then
        return SEARCHRATIO(cur_ratio, 100, δ)
    else
        return cur_ratio
```

5.3 Accelerating Blockage Recovery

Our experiments in §4.4.3 highlighted two major impairments for MPTCP in case of 802.11ad link blockage. To reduce the delay in resuming traffic over the 802.11ad subflow, *MuSher* resets the `pf` flag to allow for traffic to be scheduled on the 802.11ad subflow. However, this alone is not enough to resume the traffic flow on the 802.11ad interface. When the 802.11ad link is blocked, the subflow-level `cwnd` is cut to 1, with packets in flight also equal to 1. As a result, the scheduler is unable to schedule any new packets on the 802.11ad subflow since the `cwnd` is reported as full. To overcome this, *MuSher* uses the TCP's window recovery mechanism to restore the `cwnd` to the value just before the loss event. Note that TCP already maintains this (pre-loss) value as part of its congestion-control state. Resetting of `cwnd` also addresses the second issue observed in §4.4.3 where the restored value is half of what it was prior to loss.

Detecting interface state. To invoke its quick recovery mechanisms, *MuSher* monitors the 802.11ad interface status

maintained in the `operstate` member of `net_device` struct in the kernel. This struct and its members are available for all network interfaces by default in the kernel and *MuSher* does not need direct access to the underlying hardware-specific device drivers to receive an explicit notification of the 802.11ad interface becoming available again.

Signaling active subflow to the sender. The blockage recovery mechanisms can be initiated locally on the client in the uplink case but need the transmission of an explicit notification to the other end of the MPTCP connection in the downlink case. *MuSher* achieves this by sending a zero-byte `TCP_KEEPALIVE` packet on the 802.11ad subflow. Receipt of this packet on the other side triggers the immediate recovery and resumption of traffic on the subflow.

Note: The solution mechanisms in §5.2 and §5.3 need to be initiated on the client side by *MuSher*. However, in case of downlink-only traffic, the scheduler is not run on the client side at all, and hence, the mechanisms will never be triggered. To address this issue, *MuSher* uses the Linux's `jprobe` functionality to hook on to the `tcp_rcv_established` function that TCP runs every time a data packet is received and processed. With this setup, we are able to register a callback function inside our scheduler to run even in the absence of any outgoing traffic. We then use this callback function to implement the solutions described above. Note that this mechanism does not require any changes to the MPTCP code base or to any parts of the Linux kernel.

6 *MUSHER*: EVALUATION

In this section, we evaluate *MuSher* under a wide variety of scenarios including both stable and varying channel conditions, mobility, and different combinations of link rates and delay settings, and compare its performance against MPTCP's default *minRTT* scheduler.

6.1 Varying Channel Conditions

We evaluate *MuSher* under different channel dynamics in a typical WLAN, involving static and dynamic contention on the 802.11ac channel[5] and client mobility which changes channel conditions for both interfaces.

6.1.1 Static Contention (802.11ac). We begin by evaluating *MuSher* for different levels of contending 802.11ac traffic. We create contention using a separate independent link that has the same 802.11ac hardware configuration as the main link. We start the cross-traffic at the 5^{th} s of our 60 s run.

Fig. 11 shows the ideal MPTCP throughput (sum of 802.11ad and 802.11ac throughput), MPTCP throughput under the default *minRTT* scheduler, and *MuSher* throughput for different levels of contention. In all cases, the default scheduler achieves less than the expected sum and the magnitude of the gap increases with higher contention. For instance, under 100 Mbps of cross-traffic, *minRTT* achieves ~90 Mbps less than the expected whereas contention of 500 Mbps results in *minRTT* throughput of only ~1100 Mbps vs. the expected sum of 1800 Mbps, a deficit of 700 Mbps. On the other hand, *MuSher* is able to detect contention and converge to a throughput-optimal packet assignment under all scenarios, achieving throughput very close to the ideal sum and outperforming *minRTT* by 120-570 Mbps (a **1.5x** gain, in case of 500 Mbps cross traffic).

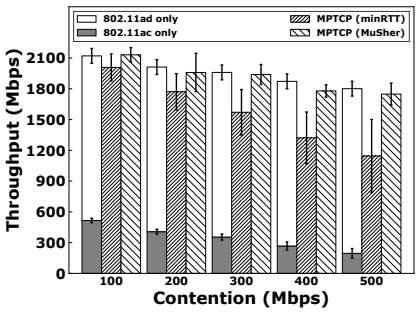


Figure 11: Reacting to 802.11ac contention

6.1.2 Dynamic Contention (802.11ac). We next evaluate how well *MuSher* reacts to changing cross-traffic. We continuously vary contention levels between 300 Mbps and 500 Mbps for a period of 120 s. The actual contention level is selected at random but is kept same across runs for both *minRTT* and *MuSher* for a fair comparison. Further, to study the effectiveness of *MuSher*'s trigger mechanism and convergence time, we consider different frequencies of contention level changes ranging from every 1 s to every 20 s. For each setting, we repeat the resulting 120 s contention timeline several times and present the average. In addition to the default *minRTT* scheduler, we also compare against an optimal oracle scheduler which always performs throughput-optimal assignment of packets between the 802.11ad and 802.11ac flows given the level of contention.

Table 2: Dynamic 802.11ac contention

	minRTT (Gbps)	*MuSher* (Gbps)	Optimal (Gbps)	*MuSher*/Optimal (%)
1 s	1.53	**1.68**	1.78	94.3
5 s	1.42	**1.70**	1.78	95.5
10 s	1.41	**1.68**	1.78	94.3
20 s	1.35	**1.71**	1.78	96.0

Table 2 presents the results for four scenarios ranging from highly dynamic (contention level changes every 1 s) to relatively stable (changes every 20 s). We observe that *MuSher* outperforms *minRTT* in all cases with gains over the default scheduler up to **360** Mbps (20 s case). Even in

[5]The case of contention on the 802.11ad is analogous and hence omitted due to space constraints.

the most challenging scenario where contention changes every 1 s, *MuSher* provides 150 Mbps higher throughput compared to *minRTT*. This improvement can be attributed to continuous adjustment of traffic distribution by *MuSher* to the changing 802.11ac channel capacity whereas *minRTT* either does not adapt (1 s case) or adapts too slowly (20 s case). More importantly, *MuSher* is able to achieve more than **94%** of the optimal throughput possible with a perfect scheduler in all cases thanks to the low overhead of its triggering mechanisms and ratio probing strategy.

6.1.3 Mobility (802.11ad/ac). We finally evaluate how well *MuSher* deals with link capacity changes due to *continuous* mobility. We evaluate three different mobility scenarios, where the client (i) moves away from the AP, (ii) moves towards the AP, and (iii) moves laterally to the AP. In cases (i) & (ii), 802.11ad does not require frequent beam-training as the relative angle between the client and AP does not change. In contrast, case (iii) requires frequent beam training. We perform all measurements in a lobby with furniture and repeat them several times with two different users. For each run, the user continuously moves over a period of 60 s at constant walking speed. We intentionally experiment with the worst case scenarios where mobility is sustained over a long period of time as opposed to intermittent mobility. This helps us obtain a lower bound on the performance of *MuSher* and ensures that we do not violate our original design goal for MPTCP to perform at least as good as SPTCP over the faster of two interfaces.

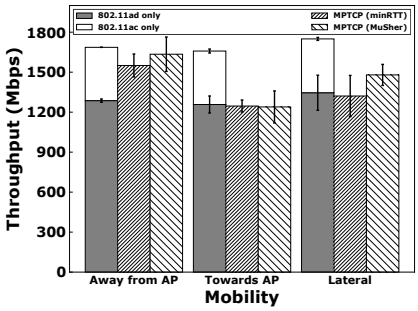


Figure 12: *MuSher*: Performance under mobility

Fig. 12 compares performance of *minRTT*, *MuSher*, and 802.11ad operating alone (faster of the two interfaces). For case (i) & (ii), *MuSher* and *minRTT* provide comparable performance with *MuSher* achieving slightly higher throughput in case (i). In the lateral mobility case, however, *MuSher* outperforms *minRTT* by ~160 Mbps. The lateral case involves more drastic changes in 802.11ad throughput (as indicated by higher std. dev. of 802.11ad alone) which proves challenging for *minRTT* to react. Moreover, *MuSher* always performs equally well or better than 802.11ad alone providing a gain of 101 Mbps in case (iii) and 368 Mbps in case (i). In general, we note that the gains from *MuSher* are lower under device mobility when compared to the dynamic contention scenario. Mobility presents a much more challenging scenario where channel conditions change much faster on both the interfaces simultaneously and in a much more unpredictable fashion compared to the contention case. This accounts for the relatively smaller gains in the mobility case.

6.2 Network Scans

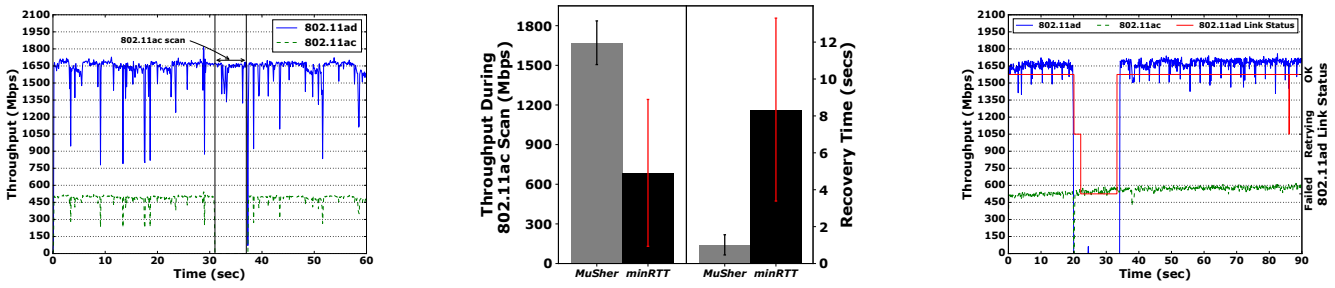
Fig. 13a shows a timeline consisting of an 802.11ac scan but with *MuSher*'s network scan management solution applied during the scan period. We observe (compared to the scan period in Fig. 9a) that the 802.11ad throughput remains unaffected during the scan interval. We repeated the measurements several times with and without the optimization. As can be seen in Fig. 13b (left two bars), the MPTCP throughput for the former shows an average improvement from 700 Mbps to 1650 Mbps (**2.3x** gain).

6.3 802.11ad Blockage

We test our solution in a setup similar to that in §4.4.3. Fig. 13c shows a timeline where blockage is introduced at the 20^{th} s but the connection is already re-established at the 34^{th} s. In contrast to Fig. 10a, where MPTCP resumed traffic on the 802.11ad subflow after a 20 s delay, here MPTCP starts using the 802.11ad interface in less than 1 s after link re-establishment. This is a substantial reduction in delay and in a dynamic environment, where such blockage events might occur frequently, *MuSher*'s gains translate into a significant improvement of user-experience. Fig. 13b (right two bars) shows that *minRTT* on average takes 8 s to recover whereas *MuSher* can resume throughput in 1 s.

6.4 *MuSher* over Internet paths

Until now, we explored *MuSher*'s performance over a network where the combined capacity of the 802.11ad (C_{ad}) and 802.11ac (C_{ac}) wireless interfaces was the bottleneck as the wired path was a 10G link. If *MuSher* runs over the Internet, the bottleneck may well be on the Internet path from the MPTCP server. Additionally, Internet paths have longer RTTs which could affect *MuSher*'s reactive mechanisms. Since we could not find an ISP that could provide us an end-point connection of a link rate of more than few hundred of Mbps, as 1G Ethernet interfaces are typically the norm, we used the Linux `tc` command to control both link rate and delay of the 10G interface to emulate realistic Internet paths. Specifically, we consider three link rates: `100 Mbps`$<C_{ac}<C_{ad}$, C_{ac}`<1 Gbps`$<C_{ad}$ and $C_{ac}<C_{ad}$`<1.8 Gbps`, and three representative RTT values: 10 ms, 30 ms, and 50 ms. Note that the `tc` induced delay is added to the common wired path behind the AP. It affects both the 802.11ad and 802.11ac paths equally and hence does not create any additional RTT asymmetry between the two.



(a) Network Scan (Timeline)

(b) Throughput gain (network scan) and recovery time (802.11ad blockage) comparison

(c) 802.11ad Blockage (Faster recovery)

Figure 13: Managing Network Scans and 802.11ad blockage

6.4.1 Baseline Performance. Table 3 compares the throughput (average of 10 runs) under different combinations of link rates and RTT values.

Table 3: Throttled bandwidth & higher RTTs

Bandwidth	RTT	*minRTT* (Mbps)	*MuSher* (Mbps)
100 Mbps	10 ms	96	**96**
	30 ms	95	**94**
	50 ms	94	**93**
1000 Mbps	10 ms	764	**938**
	30 ms	670	**926**
	50 ms	561	**922**
1800 Mbps	10 ms	1441	**1682**
	30 ms	924	**1659**
	50 ms	711	**1640**

When the wired link rate is capped at 100 Mbps, both *minRTT* and *MuSher* perform similarly and their throughput is close to the available link rate. For the 1 Gbps and 1.8 Gbps case, however, *minRTT* fails to fully use the available link rate, having a utilization of less than 40% in the worst case (1800 Mbps/50 ms). Even in the best case (1000 Mbps/10 ms), the throughput is 25% below the capacity. Furthermore, the performance worsens with increasing delays, indicating that *minRTT* is not a good solution for inter-continental paths with even larger RTTs. In comparison, *MuSher* not only achieves much higher throughput (**2.3x** in the 1800 Mbps/50 ms case) than *minRTT* but is also able to utilize at least 90% of the available link rate under all configurations. We observed that all 10 *minRTT* runs suffer from repeated cuts to the 802.11ad subflow's cwnd whereas *MuSher* runs rarely do. For instance, for the 1800 Mbps/50 ms configuration, *MuSher* had the 802.11ad cwnd cut only in 2 runs. This can be attributed to the fact that *minRTT* always assigns packets to the 802.11ad subflow (as it typically has shorter RTT) and only schedules traffic over the 802.11ac subflow if the 802.11ad send-buffers are full, thereby causing a loss followed by a cwnd reduction. Further, given that Lia uses TCP Reno style cwnd growth function, it takes a long time for the cwnd to recover to a value that is needed to fully utilize the 802.11ad's capacity.

6.4.2 Scan & 802.11ad Blockage. Table 4 compares the effectiveness of *MuSher*'s scan (§5.2) and blockage recovery (§5.3) mechanisms with the default *minRTT* scheduler under three link rate/RTT settings.

Table 4: SCAN & BLOCKAGE performance with throttled bandwidth & higher RTTs

Link rate/ RTT	*SCAN*		*BLOCKAGE*	
	minRTT **(Mbps)**	***MuSher*** **(Mbps)**	***minRTT*** **(s)**	***MuSher*** **(s)**
100 Mbps / 10ms	87 ± 8	**93 ± 1**	5.8 ± 3	**0.13 ± 0.04**
1 Gbps / 30 ms	292 ± 6	**943 ± 0.1**	7.5 ± 3	**0.1 ± 0.04**
1.8 Gbps / 50 ms	319 ± 23	**1655 ± 14**	6.07 ± 3	**0.12 ± 0.04**

Scan. While *minRTT* and *MuSher* perform similarly in the 100 Mbps case, *minRTT* performs extremely poorly in the 1 Gbps and 1.8 Gbps configurations as it is not scan-aware. In contrast, *MuSher* achieves throughput close to the wired link rate in the 1 Gbps cases as it correctly stops scheduling traffic over the 802.11ac subflow and 802.11ad has enough capacity to fully use the available wired link rate. In the 1.8 Gbps case, 802.11ad link alone can only provide 1.65 Gbps which *MuSher* utilizes fully, yielding a **5x** throughput gain compared to *minRTT*.

Blockage. *minRTT* takes at least 5 s in each of the three cases to recover after an 802.11ad blockage event. *MuSher*, on the other hand, has the recovery time upper bounded by 0.17 s and on average reduces it by **an order of magnitude**.

6.5 *MuSher* with Heterogeneous Delays

Since both 802.11ac and 802.11ad are WLAN technologies, we typically do not expect to see large delay heterogeneity between the two interfaces. In contrast, such conditions are often observed in MPTCP over cellular+WiFi scenarios. Nonetheless, for the sake of completeness, we perform additional experiments where we increase the latency for the 802.11ad and 802.11ac paths so that the RTTs are heterogeneous, but leave the bandwidth unchanged. Table 5 shows the performance of *minRTT* and *MuSher* for different combinations of RTT values.

In all cases *MuSher* achieves the sum of 802.11ad (1.6 Gbps) and 802.11ac (600 Mbps) throughputs. We also analyzed the reverse scenario where 802.11ad delays are higher

Table 5: Performance with heterogeneous RTTs

802.11ac RTT	802.11ad RTT	*minRTT* (Gbps)	*MuSher* (Gbps)
5 ms	10 ms	1.76	**2.28**
	30 ms	1.56	**2.25**
	50 ms	1.00	**2.26**

than 802.11ac and observed similar results. Hence, throughput ratio based scheduling can provide optimal throughput even in the case of heterogeneous delays.

Further, to emulate an MPTCP over LTE and WiFi scenario, we repeat the experiment with the 802.11ad interface throttled to 100 Mbps (emulating WiFi) and 802.11ac to 20 Mbps (emulating LTE). The results are presented in Table 6. Even

Table 6: Throttled bandwidth & heterogeneous RTTs

802.11ad RTT	802.11ac RTT	*minRTT* (Mbps)	*MuSher* (Mbps)
5 ms	10 ms	94.9	**108**
	30 ms	94.6	**107**
	50 ms	94.6	**108**

in this case, we observe that *MuSher* achieves close to the sum of the throughputs of the two interfaces. In contrast, the default *minRTT* scheduler not only fails to achieve the sum, but in fact yields lower throughput than that of the fastest interface alone, which agrees with results reported by previous studies analyzing MPTCP performance over 3G/LTE+WiFi.

7 RELATED WORK

MPTCP schedulers. Previously proposed schedulers target WiFi/cellular or Internet scenarios and they can be divided in three classes: (i) schedulers that leverage the difference in the subflow RTTs alone [7, 15, 21, 24, 43], similar to the default scheduler, or in combination with other TCP parameters [8, 26, 27, 41, 46, 47], (ii) schedulers that try to deal with issues caused by heterogeneous paths [15, 24, 41, 43], and (iii) schedulers that improve MPTCP performance for specific application use cases [13, 20], require modifications to applications [18], or are not MPTCP compatible [28, 29]. Our measurements demonstrated that in our target case of dual-band 802.11ac/802.11ad WLANs, the default MPTCP scheduler can work effectively under static scenarios in spite of bandwidth heterogeneity. Consequently, *MuSher* only targets dynamic scenarios that involve a drastic change in the wireless capacity of one of the two paths.

Additionally, previous schedulers targeting WiFi/cellular scenarios take a macroscopic view of the underlying wireless networks by studying how path heterogeneity affects upper layer (transport/application) performance. In contrast, *MuSher* takes a much closer look at the lower layers of the protocol stack, addressing specific challenges associated with 802.11ad/ac, without requiring explicit information from the lower layers. The only other work that addresses a challenge related to the underlying wireless technology is [42], which targets the specific scenario where a mobile client temporarily moves out of the WiFi AP's range and experiences long delays once it comes back in range. The problem is similar to the first of the two problems we report in case of 60 GHz blockage in §4.4.3. Nonetheless, in contrast to *MuSher*, [42] requires cross-layer information from the wireless driver and per-device calibration.

MPTCP performance. A number of works have evaluated different aspects of MPTCP performance under various scenarios [9–12, 14, 16, 28, 32, 36, 38, 39, 42]. All these works consider either Internet paths or scenarios involving WiFi and cellular interfaces, and hence, their findings are very different from the findings of this work. For example, many previous works (e.g., [9, 14, 16, 28, 39] show that heterogeneous paths result in significant performance degradation. In contrast, our measurement study in §4.3.2 shows that MPTCP works well in heterogeneous 802.11ac/802.11ad networks.

Very little work has been done towards leveraging MPTCP in networks involving mmWave links. A few recent works [25, 40, 44] briefly explored the use of MPTCP in dual band 5/60 GHz WLANs and showed that it often results in lower performance than using the 802.11ad interface alone. The work in [35] explored MPTCP performance in 5G cellular networks, over 28 GHz and LTE interfaces, using simulations, and showed that the protocol performs better than SPTCP with uncoupled congestion control but worse with *Balia*. Our work is the first, to our best knowledge, extensive experimental study of the performance of MPTCP over mmWave links, showing that MPTCP works well in static scenarios regardless of the congestion control algorithm.

8 CONCLUSION

In this paper, we explored the use of MPTCP to improve performance and reliability in dual-band 802.11ad/ac WLANs. We showed, in sharp contrast to previous claims, that MPTCP under ideal static conditions can improve throughput compared to using SPTCP over the faster of the two interfaces. However, in dynamic scenarios and for certain network events (channel contention, network-scan, 802.11ad blockage, mobility), MPTCP performs sub-optimally. We then designed, implemented, and evaluated *MuSher*, a novel MPTCP scheduler, to address the underlying causes for performance degradation of MPTCP. Our evaluation in a wide range of scenarios showed that *MuSher* improves MPTCP throughput by up to 2.3x and it can accelerate recovery time from a link failure by up to an order of magnitude, compared to the default *minRTT* scheduler.

ACKNOWLEDGMENTS

We thank our shepherd and the anonymous reviewers for their valuable comments. This work was supported in part by the National Science Foundation grants CNS-1553447 and CNS-1801903, ERC grant CoG 617721 and the Region of Madrid through TAPIR-CM (S2018/TCS-4496).

A ANALYSIS TOOLS

A.1 MPTCP Scheduler Probe

This LKM monitors the MPTCP scheduler and the individual subflows over 35 parameters such as the *cwnd*, packets in flight, which subflow was selected and for what reason, etc. It allows us to monitor and (if needed) log these parameters in real time. The parameters are logged on the sender side every time an ACK is received. It is implemented as a *kretprobe* and it hooks on to the main scheduler function `rtt_get_subflow_from_selectors` (in case of *minRTT*) which is responsible for deciding the subflow to be used for a given segment.

A.2 MPTCP Queue Probe

This LKM monitors the MPTCP *recv-queue* and *ofo-queue*. It logs the size of each queue along with the number of bytes enqueued or dequeued, for every enqueue and dequeue operation. It is implemented as a *kretprobe* and hooks onto the primary MPTCP and TCP functions responsible for processing incoming packets.

A.3 MPTCP *FixedRatio* Scheduler

The *FixedRatio* Scheduler assigns packets to each of the subflows with a fixed user-defined ratio. We use an epoch of 100 segments (each of MSS number of bytes) and specify the ratio in terms of segments to be assigned to a given subflow for each epoch. Any super-sized segments are split at the assignment boundary and the left over segments are taken care of in the next epoch. This ratio can be dynamically changed, during runtime, from the userspace through a *sysctl* variable.

B MPTCP THROUGHPUT VS. PACKET ASSIGNMENT RATIO

MuSher relies on the unimodal nature of the curve presented in Fig. 7a. Our observation regarding this particular nature of the curve comes from the following intuition. To achieve maximum MPTCP (total) throughput, defined as the sum of the maximum throughput of the 802.11ad and 802.11ac interfaces, two conditions need to be met: (i) Both interfaces should always have packets to send (backlogged) and (ii) For a given set of packets, the assignment of the packets should be such that both interfaces finish sending their share of packets at the same time (minimize out-of-order packets at the receiver). Let us assume that at the optimal ratio x packets are assigned to 802.11ad and $100 - x$ to 802.11ac. Given it is the optimal ratio, 802.11ad will run through x packets in the same time as it takes 802.11ac to finish $100 - x$ packets. At any ratio other than the optimal either the 802.11ad interface will finish its set of packets first or 802.11ac will. The 802.11ad interface will be idle in the former case and the 802.11ac interface in the latter case. In both these situations, the required condition (i) for optimal throughput is not met, which translates into non-optimal throughput. Note that if *MuSher* was to assign more packets to the interface that finished first (and is idle), it would be violating the required ratio of assignment resulting in out-of-order packets, thereby not satisfying condition (ii).

REFERENCES

[1] [Online]. *ASUS Republic of Gamers (ROG) Phone.* https://www.asus.com/us/Phone/ROG-Phone/

[2] [Online]. Commercial usage of Multipath TCP. http://blog.multipath-tcp.org/blog/html/2015/12/25/commercial_usage_of_multipath_tcp.htm.

[3] [Online]. Hybrid Access Solution. https://www.tessares.net/solutions/hybrid-access-solution/.

[4] [Online]. Netgear Nighthawk X10 Smart WiFi Router. https://www.netgear.com/landings/ad7200/.

[5] [Online]. TP-Link Talon AD7200 Multi-Band Wi-Fi Router. http://www.tp-link.com/us/products/details/cat-5506_AD7200.html.

[6] [Online]. Use Multipath TCP to create backup connections for iOS. https://support.apple.com/en-us/HT201373.

[7] Sabur Hassan Baidya and Ravi Prakash. 2014. Improving the performance of Multipath TCP over Heterogeneous Paths using Slow Path Adaptation. In *Proc. of IEEE ICC*.

[8] Yuanlong Cao, Qinghua Liu, Guoliang Luo, and Minghe Huang. 2015. Receiver-driven multipath data scheduling strategy for in-order arriving in SCTP-based heterogeneous wireless networks. In *Proc. of IEEE PIMRC*.

[9] Yung-Chih Chen, Yeon sup Lim, Richard J. Gibbens, Erich M. Nahum, Ramin Khalili, and Don Towsley. 2013. A Measurement-based Study of MultiPath TCP Performance over Wireless Networks. In *Proc. of ACM Internet Measurement Conference (IMC)*.

[10] Quentin De Coninck, Matthieu Baerts, Benjamin Hesmans, and Olivier Bonaventure. 2015. Poster: Evaluating Android Applications with Multipath TCP. In *Proc. of ACM MobiCom*.

[11] Quentin De Coninck, Matthieu Baerts, Benjamin Hesmans, and Olivier Bonaventure. 2016. A First Analysis of Multipath TCP on Smartphones. In *Proc. of Passive and Active Measurement Conference (PAM)*.

[12] Quentin De Coninck, Matthieu Baerts, Benjamin Hesmans, and Olivier Bonaventure. 2016. Observing Real Smartphone Applications over Multipath TCP. *IEEE Communications Magazine, Network Testing Series,* 54, 3 (March 2016), 88–93.

[13] Xavier Corbillon, Ramon Aparicio-Pardo, Nicolas Kuhn, GÃĭraldine Texier, and Gwendal Simon. 2016. Cross-layer scheduler for video streaming over MPTCP. In *Proc. of ACM MMSys*.

[14] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. 2014. WiFi, LTE, or Both? Measuring Multi-Homed Wireless Internet Performance. In *Proc. of ACM Internet Measurement Conference (IMC)*.

[15] Simone Ferlin, Ozgu Alay, Olivier Mehani, and Roksana Boreli. 2016. BLEST: Blocking Estimation-based MPTCP Scheduler for Heterogeneous Networks. In *Proc. of IFIP Networking*.

[16] Simone Ferlin, Thomas Dreibholz, and Özgu Alay. 2014. Multi-Path Transport Over Heterogeneous Wireless Networks: Does It Really Pay Off?. In *Proc. of IEEE Global Telecommunications Conference (GLOBECOM)*.

[17] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. 2013. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 1546.

[18] Yihua Ethan Guo, Ashkan Nikravesh, Z. Morley Mao, Feng Qian, and Subhabrata Sen. 2017. Accelerating Multipath Transport Through

Balanced Subflow Completion. In *Proc. of ACM MobiCom*.

[19] Muhammad Kumail Haider and Edward W. Knightly. 2016. Mobility Resilience and Overhead Constrained Adaptation in Directional 60 GHz WLANs: Protocol Design and System Implementation. In *Proc. of ACM MobiHoc*.

[20] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. 2016. MP-DASH: Adaptive Video Streaming Over Preference-Aware Multipath. In *Proc. of ACM CoNEXT*.

[21] Jaehyun Hwang and Joon Yoo. 2015. Packet Scheduling for Multipath TCP. In *Proc. of IEEE ICUFN*.

[22] IEEE 802.11 Working Group. 2012. IEEE 802.11ad, Amendment 3: Enhancements for Very High Throughput in the 60 GHz Band. (2012).

[23] Ramin Khalili, Nicolas Gast, Miroslav Popovic, and Jean-Yves Le Boudec. 2013. MPTCP Is Not Pareto-Optimal: Performance Issues and a Possible Solution. *IEEE/ACM Transactions on Networking* 21, 5 (2013).

[24] Nicolas Kuhn, Emmanuel Lochin, Ahlem Mifdaoui, Golam Sarwar, Olivier Mehani, and Roksana Boreli. 2014. DAPS: Intelligent Delay-Aware Packet Scheduling For Multipath Transport. In *Proc. of IEEE ICC*.

[25] Kien Nguyen, Mirza Golam Kibria, Kentaro Ishizu, and Fumihide Kojima. 2017. Feasibility Study of Providing Backward Compatibility with MPTCP to WiGig/IEEE 802.11ad. In *Proc. of IEEE Vehicular Technology Conference Fall (VTC-Fall)*.

[26] Dan Ni, Kaiping Xue, Peilin Hong, and Sean Shen. 2014. Fine-grained Forward Prediction based Dynamic Packet Scheduling Mechanism for multipath TCP in lossy networks. In *Proc. of ICCCN*.

[27] Dan Ni, Kaiping Xue, Peilin Hong, Hong Zhang, and Hao Lu. 2015. OCPS: Offset Compensation based Packet Scheduling mechanism for multipath TCP. In *Proc. of IEEE ICC*.

[28] Ashkan Nikravesh, Yihua Guo, Feng Qian, Z. Morley Mao, and Subhabrata Sen. 2016. An In-depth Understanding of Multipath TCP on Mobile Devices: Measurement and System Design. In *Proc. of ACM MobiCom*.

[29] Ashkan Nikravesh, Yihua Guo, Xia Zhu Feng Qian, and Z. Morley Mao. 2019. MP-H2: a Client-only Multipath Solution for HTTP/2. In *Proc. of ACM MobiCom*.

[30] Thomas Nitsche, Adriana B. Flores, Edward W. Knightly, and Joerg Widmer. 2015. Steering with Eyes Closed: mm-Wave Beam Steering without In-Band Measurement. In *Proc. of IEEE INFOCOM*.

[31] Christoph Paasch, Simone Ferlin, Özgu Alay, and Olivier Bonaventure. 2014. Experimental evaluation of multipath TCP schedulers. In *Proc. of ACM CSWS*.

[32] Christoph Paasch, Ramin Khalili, and Olivier Bonaventure. 2013. On the Benefits of Applying Experimental Design to Improve Multipath TCP. In *Proc. of ACM CoNEXT*.

[33] Qiuyu Peng, Anwar Walid, Jaehyun Hwang, and Steven H. Low. 2016. Multipath TCP: Analysis, Design, and Implementation. *IEEE/ACM Transactions on Networking* 24, 1 (2016).

[34] Peraso Technologies. 2017. Peraso Launches 802.11ad Reference Design for Smartphone Applications. https://www.prnewswire.com/news-releases/peraso-launches-80211ad-reference-design-for-smartphone-applications-300409463.html.

[35] Michele Polese, Rittwik Jana, and Michele Zorzi. 2017. TCP in 5G mmWave Networks: Link Level Retransmissions and MP-TCP. In *Proc. of IEEE Workshop on 5G New Radio Technologies*.

[36] Feng Qian, Bo Han, Shuai Hao, and Lusheng Ji. 2015. An Anatomy of Mobile Web Performance over Multipath TCP. In *Proc. of ACM CoNEXT*.

[37] Costin Raiciu, Mark Handley, and Damon Wischik. 2011. Coupled Congestion Control for Multipath Transport Protocols. RFC 6356.

[38] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. 2012. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*.

[39] Swetank Kumar Saha, Abhishek Kannan, Geunhyung Lee, Nishant Ravichandran, Parag Kamalakar Medhe, Naved Merchant, and Dimitrios Koutsonikolas. 2017. Multipath TCP in Smartphones: Impact on Performance, Energy, and CPU Utilization. In *Proc. of ACM MobiWac*.

[40] Swetank Kumar Saha, Roshan Shyamsunder, Naveen Muralidhar Prakash, Hany Assasa, Adrian Loch, Dimitrios Koutsonikolas, and Joerg Widmer. 2017. Poster: Can MPTCP Improve Performance for Dual-Band 60 GHz/5 GHz Clients?. In *Proc. of ACM MobiCom*.

[41] Tanya Shreedhar, Nitinder Mohan, Sanji K. Kaul, and Jussi Kangasharju. 2018. QAware: A Cross-Layer Approach to MPTCP Scheduling. In *Proc. of IFIP Networking*.

[42] Yeon sup Lim, Yung-Chih Chen, Erich M. Nahum, Donald F. Towsley, and Kang-Won Lee. 2014. Cross-layer path management in multi-path transport protocol for mobile devices. In *Proc. of IEEE INFOCOM*.

[43] Yeon sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. 2017. ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths. In *Proc. of ACM SIGMETRICS*.

[44] Sanjib Sur, Ioannis Pefkianakis, Xinyu Zhang, and Kyu-Han Kim. 2017. WiFi-Assisted 60 GHz Wireless Networks. In *Proc. of ACM MobiCom*.

[45] Sanjib Sur, Xinyu Zhang, Parameswaran Ramanathan, and Ranveer Chandra. 2016. BeamSpy: Enabling Robust 60 GHz Links Under Blockage. In *Proc. of USENIX NSDI*.

[46] Fan Yang, Paul Amer, and Nasif Ekiz. 2013. A Scheduler for Multipath TCP. In *Proc. of ICCCN*.

[47] Fan Yang, Qi Wang, and Paul Amer. 2014. Out-of-Order Transmission for In-Order Arrival Scheduling for Multipath TCP. In *Proc. of IEEE AINA workshops*.

[48] Anfu Zhou, Xinyu Zhang, and Huadong Ma. 2017. Beam-forecast: Facilitating Mobile 60 GHz Networks via Model-driven Beam Steering. In *Proc. of IEEE INFOCOM*.