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

Shivang Aggarwal, *Student Member, IEEE*, Swetank Kumar Saha, *Member, IEEE*, Imran Khan, Rohan Pathak, Dimitrios Koutsonikolas, *Senior Member, IEEE*, and Joerg Widmer, *Fellow, IEEE*

***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 leverage Multipath TCP (MPTCP) to use both interfaces simultaneously in order to achieve a higher overall throughput as well as seamlessly switch to a single interface when the other one fails. We find that standard MPTCP often performs sub-optimally and can 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* can provide a throughput improvement of 50%/130% under WLAN/Internet settings respectively, compared to the default MPTCP scheduler. It further speeds up the recovery of a traffic stream after disruption by a factor of 8x/75x.**

***Index Terms*— 802.11ad, 802.11ac, multipath TCP.**

I. Introduction

MILLIMETER-WAVE (mmWave) communication is fast emerging as the prime candidate technology to provide 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 have been released over the past few years and the technology is already making its way into smartphones.

Manuscript received 8 February 2021; revised 19 September 2021 and 12 February 2022; accepted 24 February 2022; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Schapira. Date of publication 18 March 2022; date of current version 18 August 2022. This work was supported in part by the National Science Foundation under Grant CNS-1553447 and Grant CNS-1801903 and in part by the Region of Madrid through TAPIR-CM under Grant S2018/TCS-4496. *(Corresponding author: Shivang Aggarwal.)*

Shivang Aggarwal, Imran Khan, and Dimitrios Koutsonikolas are with the Institute for the Wireless Internet of Things, Northeastern University, Boston, MA 02115 USA (e-mail: aggarwal.sh@northeastern.edu; khan.i@northeastern.edu; d.koutsonikolas@northeastern.edu).

Swetank Kumar Saha is with the Department of Computer Science and Engineering, University at Buffalo, The State University of New York, Buffalo, NY 14260 USA (e-mail: swetankk@buffalo.edu).

Rohan Pathak was with the University at Buffalo, The State University of New York, Buffalo, NY 14260 USA. He is now with Microsoft, Redmond, WA 98052 USA (e-mail: ropathak@microsoft.com).

Joerg Widmer is with the IMDEA Networks Institute, Leganés, 28918 Madrid, Spain (e-mail: joerg.widmer@imdea.org).

Digital Object Identifier 10.1109/TNET.2022.3158678

Nonetheless, communication at mmWave frequencies faces fundamental challenges due to the high propagation and penetration loss, and the use of directional transmission makes links susceptible to disruption by human blockage and client mobility. Even if PHY and MAC layer improvements (e.g., [1]–[3]) result in faster beam steering and lower reconnection times in the future, any realistic indoor scenario is expected to contain enough dynamism to cause a substantial number of reconnection 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 of 60 GHz technology while maintaining the reliability of legacy WiFi, which is the key for the 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 and client devices have 2.4, 5, and 60 GHz interfaces and thus, 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) [4], 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 [5]–[7].

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) [8], would need mechanisms for reordering 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 reordering – an unnecessary duplication of functionality already provided by the transport layer. 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. Several recent studies investigated the performance of MPTCP over WiFi and cellular interfaces and showed that the protocol performs poorly over heterogeneous paths, due to various interactions among different components of MPTCP, including packet reordering, scheduling decisions, and congestion control [9]–[13]. Further, MPTCP in dual band 5/60 GHz WLANs often yields lower performance than using the 802.11ad interface alone [14], [15], and researchers have even argued 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 very promising solution for 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 setting. Our study reveals that MPTCP can achieve near optimal throughput under baseline, static scenarios. However, realistic dynamic environments, e.g., with interference 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 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. We make 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]
(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. In contrast to previous works that discourage the simultaneous use of the two interfaces, we find that, MPTCP yields near optimal throughput in static scenarios but faces a number of challenges in dynamic environments.
(3) We thoroughly analyze the impact of packet scheduling decisions on MPTCP performance and show experimentally that maintaining a packet assignment ratio to the two MPTCP subflows equal to the throughput ratio of the two subflows is the key to achieving good performance.
(4) We design *MuSher*, a novel MPTCP scheduler that addresses the aforementioned challenges via throughput-ratio based scheduling and a number of additional mechanisms.
(5) We implement *MuSher* in the Linux kernel[3] using two different approaches to obtain throughput estimates: directly at the TCP layer as well as at the IP later. Our approach does not require access to information from the wireless interfaces or the device drivers.
(6) We perform an extensive evaluation in realistic WLAN and Internet settings. We show that *MuSher* offers large throughput improvements over the default MPTCP scheduler and other state-of-the-art schedulers and reduces significantly the recovery time from link failures. It also works well in scenarios involving interference, mobility, multiple MPTCP flows, paths with heterogeneous delays, and web traffic.

[1]https://github.com/swetanksaha/mptcp-tools
[2]Data available at: http://bit.ly/mptcp-musher-data
[3]https://github.com/swetanksaha/mptcp-musher

II. MPTCP Background

In this section, we provide a brief overview of 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*, 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). 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* congestion control. The decoupled variant runs an independent instance of the default Linux congestion control 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 [16]–[18] is preferred over its counterpart as it maintains fairness with other competing flows running over a bottleneck link [16].

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 temporarily put in an out-of-order queue (*ofo-queue*) that is shared among all the subflows of an MPTCP connection. The remaining space in the shared buffer is advertised to the sender as the receive window (`recv_win`).

III. MPTCP Performance & Pitfalls

In this section, we study the performance of MPTCP over dual-band links for a wide range of scenarios to understand and analyze the root causes of the observed behavior.

A. Methodology

Our setup consists of a Netgear Nighthawk X10 WiFi router and an Acer Travelmate P446-M laptop. Both devices support 802.11ac and 802.11ad. A high-end desktop is connected to

TABLE I
THROUGHPUT COMPARISON (MBPS) OF 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

the router through a 10G LAN SFP+ interface to generate/receive MPTCP traffic. While this setup would in theory allow us to achieve the maximum 802.11ad rate of 4.6 Gbps, 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. A similar observation has been made in other studies using the same hardware, e.g., [14], [19]–[21].

We use MPTCP version v0.94 and make all our modifications 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 and the *default minRTT* scheduler. We first evaluate all the available congestion control algorithms and then use the default *Lia* coupled congestion control for the remainder of the paper. We also disable the two memory optimizations labeled as Mechanisms 1 and 2 in [9], which were introduced in v0.89 to reduce memory usage. Although the authors in [9] show improvements with these mechanisms for a scenario involving WiFi and 3G interfaces, our measurements in [21] reveal significant performance issues in the case of 802.11ac/ad WLANs. A fundamental problem is that the 802.11ad subflow never exits slow start, which leads to unstable (in addition to reduced) performance. We have also confirmed experimentally that these mechanisms do not improve the performance even under dynamic conditions. Previous works [11], [22] have also shown that these two mechanisms result in reduced performance in different heterogeneous environments.

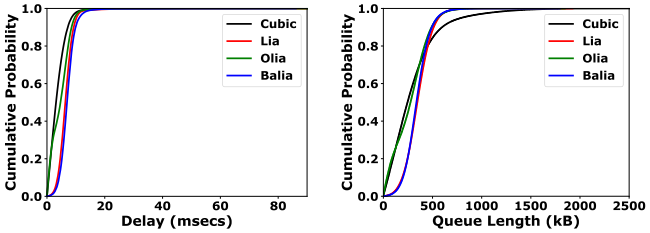
In the rest of the paper, bar graphs show the average values over multiple runs and the error bars show the standard deviations.

B. Baseline Performance

We first establish a baseline for MPTCP performance under static scenarios with ideal channel conditions (where TCP experiences 0% packet loss over each path). 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).

1) Congestion Control Algorithms: We experiment with four congestion control algorithms available in the Linux implementation – *Cubic* (decoupled), *Lia* [16], *Olia* [17], and *Balia* [18] – under backlogged traffic. Table I lists MPTCP throughput along with throughput over each interface when engaged separately.

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]–[13] that have shown MPTCP to perform poorly when used with heterogeneous interfaces, albeit in the context of WiFi+3G/LTE,



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

Fig. 1. MPTCP impact on delay and queue length.

and more importantly to recent works [14], [15] 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*. Compared to MUST [14], a state-of-the-art MAC layer solution that only uses the 802.11ad interface and switches to 802.11ac in case of blockage, MPTCP would result in a throughput boost of 31%-36%. We also verified that it can sustain the provided application data rates under non-backlogged traffic.

Delay: We use the time packets spend in the MPTCP meta-level *ofo-queue* as a measure of application-perceived delay. This isolates the MPTCP-induced delay due to packet reordering across subflows 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 reordering induced by MPTCP. Fig. 1a and 1b plot the *ofo-queue* delay and queue length for each of the four congestion control algorithms. The 99-th percentile of the delay and queue length are upper-bounded by 10 ms and 1.5 MB, 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 any significant memory requirements on the system while providing significant throughput gains.

2) Suboptimal Links: In §III-B.1, we considered optimal links that can individually support the highest throughput possible with our hardware. We now repeat the measurements over suboptimal links. For scenarios 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. We consider three cases based on the relationship between the 802.11ad throughput ($\mathtt{Th}_{ad}$) and the 802.11ac ($\mathtt{Th}_{ac}$) throughput: $\mathtt{Th}_{ad}>\mathtt{Th}_{ac}$, $\mathtt{Th}_{ad}<\mathtt{Th}_{ac}$, and $\mathtt{Th}_{ad}=\mathtt{Th}_{ac}$. We also consider a scenario involving an optimal 802.11ad link and a suboptimal 802.11ac link, denoted as $\mathtt{Th}_{ad}\gg\mathtt{Th}_{ac}$. 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. Note that in all 4 scenarios, the TCP loss rates remain very low over both paths (0.0002%-0.005%), as the link layer does retries and the rate adaptation masks losses from the upper layer of the protocol stack. Fig. 2 shows the single path and MPTCP throughput with each congestion control algorithm for the four scenarios. In addition, in all four scenarios, there is external interference on the WiFi link from the campus WiFi network.

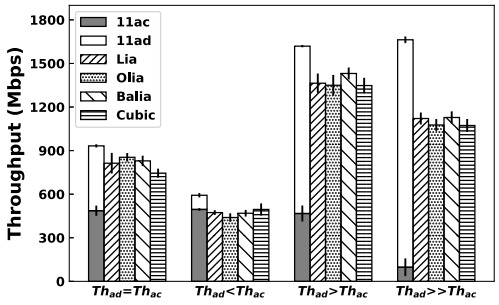


Fig. 2. MPTCP throughput with suboptimal links.

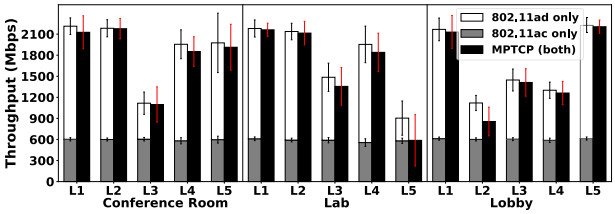


Fig. 3. Field trial.

We observe that, although the MPTCP performance degrades for all congestion control algorithms compared to the case of optimal links in §III-B, it remains satisfactory in three out of four scenarios. In the $\mathrm{Th}_{ad}>\mathrm{Th}_{ac}$ and $\mathrm{Th}_{ad}=\mathrm{Th}_{ac}$ scenarios, MPTCP with various congestion control algorithms achieves 83-88% and 80-92% of the expected sum, respectively. Compared to MUST, this translates to a throughput improvement ranging from 17% (Cubic, $\mathrm{Th}_{ad}>\mathrm{Th}_{ac}$) to 75% (Olia, $\mathrm{Th}_{ad}=\mathrm{Th}_{ac}$). In the $\mathrm{Th}_{ad}<\mathrm{Th}_{ac}$ scenario, MPTCP performs similar to or slightly worse than SPTCP over the faster path. Note that, since there is no blockage involved in this scenario, MUST would still use only 802.11ad, resulting in very low throughput. In contrast, in the $\mathrm{Th}_{ad}\gg\mathrm{Th}_{ac}$, MPTCP achieves only 64-68% of the expected sum and performs worse than SPTCP over the faster path. Note that this is a very challenging scenario, involving highly heterogeneous links and additional time-varying interference further affecting the slower path. We will revisit the issue of interference in §III-D. We also observe that the four congestion control algorithms perform very close to each other in all four scenarios, without a clear winner across all scenarios. Based on this observation, in the rest of the paper we use the default Lia algorithm.

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. 3 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 this is not the case (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 [16] and outperforming MUST, while also adding reliability by allowing for smooth switch-over, if needed.

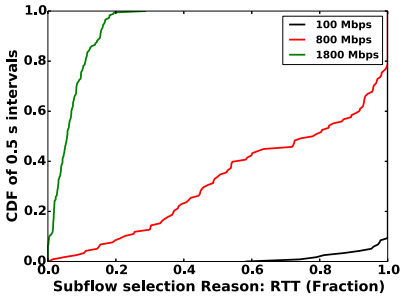


Fig. 4. Subflow selection reason.

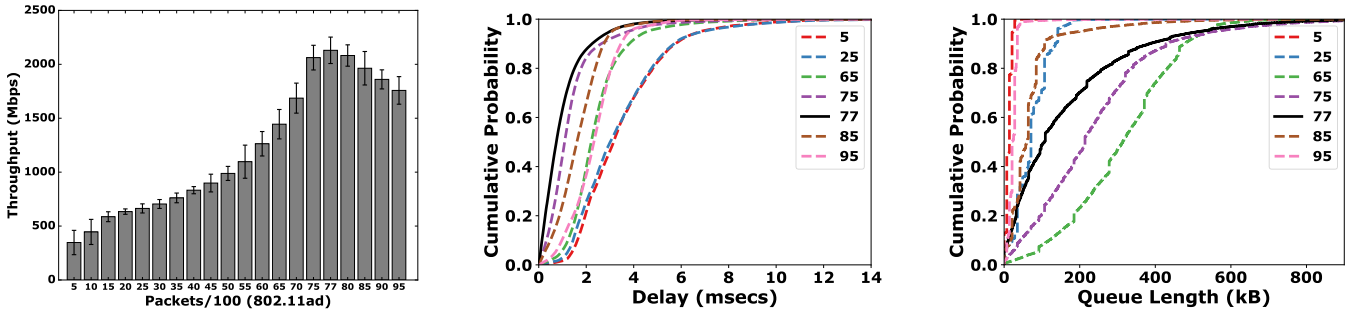
C. Understanding MPTCP Performance

Our measurements in §III-B clearly demonstrate that MPTCP can provide substantial performance improvements in scenarios involving a wide variety of link qualities, 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 §III-B.1-III-B.2 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.

1) minRTT *Packet Scheduler:* We investigate in detail the packet assignment dynamics of the *minRTT* scheduler. Analyzing the scheduler's decisions over the connection lifetime, we find that it consistently assigns ∼77% of the packets to the 802.11ad subflow and the remaining 23% to 802.11ac. Further investigation of the scheduler decision reasons interestingly shows that for the majority of the time, `cwnd` is 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 prior findings that with a saturated congestion window, the scheduling decision becomes ACK-clocked [23].

On the other hand, under non-backlogged traffic, RTT becomes the deciding factor. Fig. 4 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 100, 800, 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 at least 80% of the time in each 0.5 s interval. When the source rate drops to 800 Mbps, a significant portion of scheduler decisions in each 0.5 s interval are based on subflow RTT values. Finally, with a low source rate of 100 Mbps, almost all packet assignment decisions are made based on RTT values.

2) Impact of Packet Scheduling Decisions: Given that in case of backlogged traffic, ACK-clocked scheduling decisions



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

Fig. 5. Impact of packet 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.

Fig. 5a 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-$\texttt{Pkts}_{ad}$). The 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 traffic distribution 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. 5b, 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. 5c. 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:

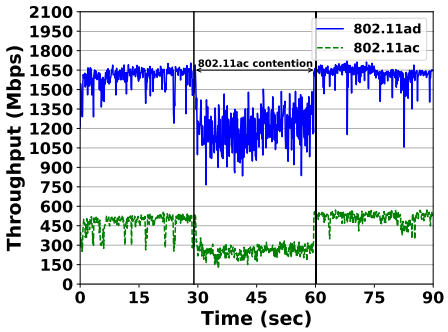
- Optimal MPTCP performance is achieved when the packet-assignment ratio is close to the throughput ratio of the two subflows.
- The MPTCP throughput vs. packet assignment ratio curve is unimodal and hence a unique optimal ratio always exists for given subflow throughputs.

D. Performance Issues

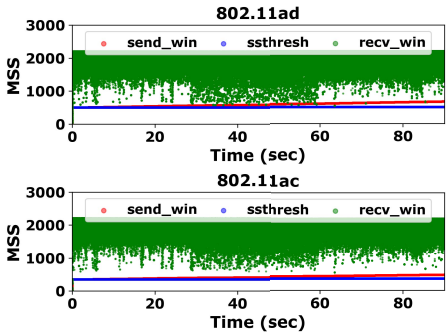
All the measurements in §III-B.1 were limited to scenarios with stable links for the duration of the experiment. In contrast, in §III-B.2, we saw that time-varying external interference can lead to suboptimal MPTCP performance. We now take a closer look at various challenging scenarios, including scenarios with varying channel conditions among others, and examine the causes for the suboptimal performance.

1) Varying Channel Conditions: In realistic WLAN scenarios, link conditions and thereby channel capacity may 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. 6a 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 throughput are at their maximum 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 average throughput during the contention period is $\sim 1450 (= 1200 + 250)$ Mbps, which is even less than 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. 6b, which plots the TCP congestion control parameters for the two subflows, explains the unexpected

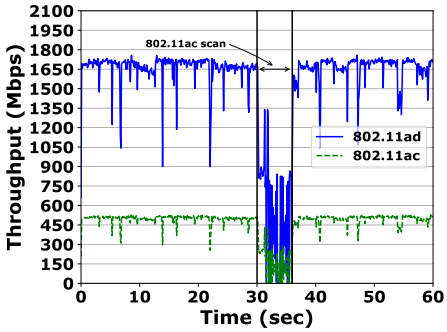


(a) Throughput timeline.

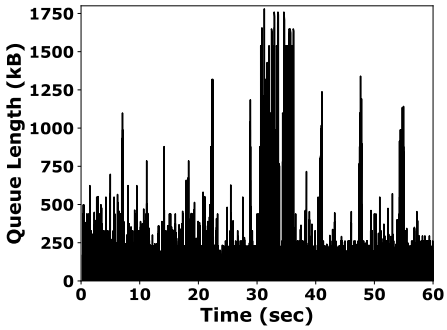


(b) *send_win*, *recv_win*, & *ssthresh* timelines.

Fig. 6. 802.11ac contention.



(a) Throughput timeline.



(b) *ofo-queue* length timeline.

Fig. 7. Network scan.

performance drop in 802.11ad. During the contention period, the receiver advertised buffer space (`recv_win`) reduces significantly. Recall 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 an increase/decrease in the distance between the AP and the laptop or partial link blockage by persons.

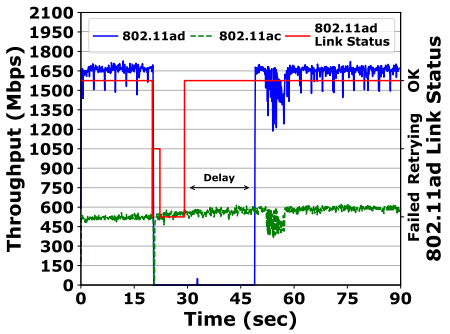
2) Network Scans: For all the results in §III-B we had disabled the periodic channel scans, which are typically initiated by the network-manager or similar user-space utilities, to avoid biasing our throughput measurements. However, disabling periodic channel scans is problematic in any real scenario,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

To study the impact of network scans on performance, we start an 802.11ac scan during an MPTCP session. Fig. 7a shows the throughput of the 802.11ad and 802.11ac subflows over 60 s, with the scan initiated at the 30-th s. The 802.11ac throughput is severely affected 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, the 802.11ad flow is also impacted negatively during this period, even though the scan occurs in the 5 GHz band.

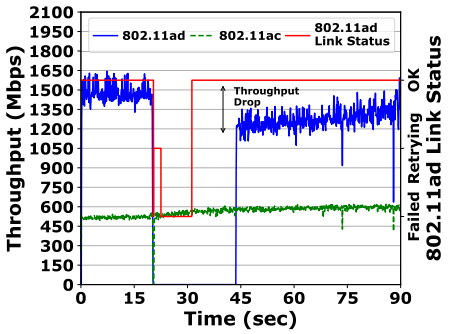
Looking at the `cwnd` values of both subflows during the 802.11ac scan, we find that they are not affected. However, we observe a 6x increase (Fig. 7b) in the amount of data held in the *ofo-queue* at the receiver end. Since the packet scheduler is not aware of the sudden reduction in 802.11ac channel capacity during the scan period, it 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). Since the MPTCP receiver is responsible for reordering the packets at the meta-level, these gaps prevent the receiver from delivering packets to the application until the missing packets arrive or are retransmitted over the 802.11ad interface. MPTCP performance drops can be observed with 802.11ad scans as well but due the much shorter duration of the scan their impact is less pronounced.

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 [14]. 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 antenna by hand, causing the link to break. We then remove the blockage and allow the device to reassociate with the AP.

Switch-over. Fig. 8a 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



(a) Delay in resumption of 802.11ad subflow.



(b) Throughput drop after reconnection.

Fig. 8. 802.11ad blockage.

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. 8b, although the 802.11ad link is restored at the 30^{th} second, MPTCP does not resume traffic on the 802.11ad subflow for another $\sim$20 s until the 49^{th} s. We repeated this experiment multiple times and found that this extra delay until traffic resumes varies 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 resumes as soon the driver reports `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 that it *potentially failed*. The MPTCP scheduler treats subflows with the `pf` flag set as being unavailable and does not schedule any packets for 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 retransmission 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 resumes with a `cwnd` and `ssthresh` that are half of their pre-loss values. Fig. 8b shows an example 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 connectivity is restored.

IV. *MuSher*: System Design & Implementation

In this section, we present the design of ***MuSher***, an agile **Mu**ltipath-TCP **Sche**dule**r** that improves dual-band MPTCP performance by addressing the performance issues identified in §III-D. We chose Linux for our reference implementation, but our mechanisms can be used on most platforms supporting MPTCP. For ease of 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. While *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. These architecture choices prevent *MuSher* from being tied to any specific hardware/platform.

We first present *MuSher*'s solution to the sub-optimal MPTCP performance under varying link conditions (§III-D.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 (§III-D.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 (§III-D.3) by addressing the interaction between MPTCP scheduling and subflow-level congestion control.

A. Reacting to Time-Varying Links

Our findings in §III-C.2 and §III-D.1 indicate that the underlying reason for the drop in throughput of a subflow, when channel conditions worsen for the other subflow, is that meta-level receive buffers are filling up. Assignment of packets with a ratio very different from the ratio of throughputs

Algorithm 1 MuSher

```
ω = 0.15 * init_tput, β = 0.1 * init_buffer_size, α = 3, λ = 3
τ1 = 200 ms, τ2 = 100 ms, δ = 10
//All other variables are initialized to 0
while true do
    curr_tput_dec+ =(last_tput − GET_CURRENT_TPUT cur_ratio, τ2)
    curr_buffer_dec+ =(last_buffer_size − GET_CURRENT_BUFFER_SIZE)
    if curr_tput_dec > ω then
        tput_threshold_cnt+ =1
        if tput_threshold_cnt == α then
            cur_ratio = CALLSEARCHRATIO cur_ratio
            SET_RATIO cur_ratio
            ω = 0.15 * GET_CURRENT_TPUT cur_ratio, τ2
            β = 0.1 * GET_CURRENT_BUFFER_SIZE
            curr_tput_dec = 0, curr_buffer_dec = 0
    else if curr_buffer_dec > β then
        buffer_threshold_cnt+ =1
        if buffer_threshold_cnt == λ then
            cur_ratio = CALLSEARCHRATIO cur_ratio
            SET_RATIO cur_ratio
            ω = 0.15 * GET_CURRENT_TPUT cur_ratio, τ2
            β = 0.1 * GET_CURRENT_BUFFER_SIZE
        curr_tput_dec = 0, curr_buffer_dec = 0
    else
        tput_threshold_cnt = 0, buffer_threshold_cnt = 0

function CALLSEARCHRATIO cur_ratio
    ratio_right = cur_ratio + δ, ratio_left = cur_ratio − δ
    tput_right = GET_CURRENT_TPUT ratio_right, τ1
    tput_left = GET_CURRENT_TPUT ratio_left, τ1
    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

function SEARCHRATIO (start, stop, step_size)
    prev_tput = 0
    for ratio = start to stop step step_size do
        cur_tput = GET_CURRENT_TPUT ratio, τ1
        if cur_tput < prev_tput then
            return SEARCHRATIOFINE ratio − step, prev_tput, step_size/2
        prev_tput = cur_tput
    return stop

function SEARCHRATIOFINE best_ratio, best_tput, step_size
    ratio_right = best_ratio + step_size, ratio_left =
    best_ratio − step_size
    cur_tput = GET_CURRENT_TPUT ratio_right, τ1
    if cur_tput > best_tput then
        return ratio_right
    cur_tput = GET_CURRENT_TPUT ratio_left, τ1
    if cur_tput > best_tput then
        return ratio_left
    return best_ratio
```

achieved over each interface alone results in too many out-of-order arrivals at the receiver, using up buffer space. Thus, the scheduling algorithm in *MuSher* has two design goals: (a) to automatically trigger the search for a new optimal ratio when the capacity of either link changes and (b) to quickly determine the unique throughput-optimal ratio at runtime after triggering a search. In the following, we first describe how we address the two design goals in §IV-A.1 and then we discuss some implementation details in §IV-A.2.

1) Design: The complete algorithm is formally presented in Algorithm 1.

Triggering ratio search. The triggering mechanism is described in the while-loop of Algorithm 1. To detect changes in the link capacity of either interface and trigger the search for a new optimal ratio, *MuSher* continuously monitors the total MPTCP throughput over intervals of τ_2 ms (function `GET_CURRENT_TPUT`) and the *send-queue* occupancy (function `GET_CURRENT_BUFFER_SIZE`) looking for either of the following two events: (i) a *decrease* in total MPTCP throughput by at least 15% (parameter ω in Algorithm 1) and (ii) a *decrease* in *send-queue* occupancy by at least 10% (parameter β in Algorithm 1)[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 current 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. To avoid unnecessarily triggering a search due to fine-scale channel fluctuations that can reduce the stability of the algorithm, we added a hysteresis mechanism; we only trigger a new ratio search when either of the two events is observed for a certain number of consecutive intervals of τ_2 ms (parameters α and λ in Algorithm 1). We found that in most cases, the ratio search does converge to the optimal ratio. We also added a periodic search trigger (not shown in Algorithm 1) to deal with rare cases when the ratio search converges to a suboptimal ratio, e.g., due to the dynamics caused by 802.11ad's CSMA/CA protocol in the case of multiple competing MPTCP flows. We empirically set the periodic search to trigger once every 3 s.

Finding the optimal ratio. We investigated several approaches to find the optimal ratio, including binary/ternary search. We finally chose a 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 throughput estimates, resulting in an increase in the overall convergence time. Our approach, (function `CALLSEARCHRATIO` in Algorithm 1), avoids such large jumps and thus converges more quickly.

Since the ratio vs. throughput curve is unimodal (§III-C.2), we can use a simple probing approach to find the maximum of the throughput curve and thus the optimal ratio. We begin by probing two ratios adjacent to the current ratio, one slightly lower and one slightly higher, and proceed our search at a step δ in the direction where we observe higher throughput. To balance between two conflicting requirements (convergence time, which requires a large step δ, and stability, which requires a small step, as described above), our probing mechanism uses a step size of 10 to quickly discover a ratio close to the optimal ratio (function `SEARCHRATIO`) and then refines the search with a step size of 5 (function `SEARCHRATIOFINE`) to finally discover the optimal ratio.

2) Implementation Details: We now discuss how we choose an appropriate sampling time to obtain throughput estimates and two different mechanisms to estimate the throughput at a given ratio.

Sampling time. An important parameter in Algorithm 1 is the sampling time (τ_1 or τ_2), i.e., the time spent at a given

[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.

ratio to estimate the corresponding throughput. It provides a trade-off between the accuracy of the throughput estimates and (i) the convergence time of the optimal-ratio search (τ_1, in function `CallSearchRatio`) or (ii) the response time to throughput or buffer size changes (τ_2, inside the while-loop). Through extensive experimentation, we set the value of τ_2 to 100 ms. However, we found that it takes longer for the throughput estimates to converge to a stable value after changing the ratio. Hence, we empirically set the value of τ_1 to 200 ms to achieve the desired balance of convergence time and accuracy during a search. For instance, with a `step_size` of 0.1 for a difference of 0.4 between the optimal and current ratio, the search would take 800 ms.

Obtaining throughput estimates. In our earlier work [21], we obtained throughput estimates by monitoring the bytes transmitted by the network interface using `struct rtnl_link_stats_64` and `struct netdev_queue`. However, this method cannot be used when there are multiple MPTCP clients served by a single server, as it does not distinguish which bytes are being transmitted to which client. Hence, in this work, we explore two new methods to derive throughput estimates:

1) **TCP Rate Estimates** are obtained by observing the number of TCP packets acknowledged over a period of time, given by the `delivered` member of `struct tcp_sock`. An object of `struct tcp_sock` is maintained for each TCP connection; hence, we can obtain a separate throughput estimate for each MPTCP client.

2) **IP Rate Estimates** are obtained by leveraging the Linux `iptables` project [24] typically used for packet filtering and accounting by network administrators. Once the accounting is enabled for the relevant set of source and destination IP addresses, we access the `bcnt` member of `struct xt_counters`, which represents the number of bytes transferred from a particular source IP address to a particular destination IP address. `struct xt_counters`, which is encapsulated by `struct xt_table`, contains an entry for each pair of source and destination IP addresses that accounting was enabled for. By monitoring this information over a period of time, we can calculate a throughput estimate.

Note that the `iptables` project is designed to be a userspace tool by default. Hence, there is an additional step required in the userspace of enabling the accounting of bytes transferred for a particular set of IP addresses. This step is necessary for *MuSher* to be able to access the appropriate kernel data structures to calculate the IP rate estimate. In contrast, the TCP rate estimates are calculated using data structures that the Linux kernel maintains by default.

B. Managing Network Scans

MuSher arbitrates the network scan requests generated from the user space and disables the scheduling of packets to the corresponding subflow 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 until the subflow-level *send-queue* is empty. *MuSher* hooks on the scanning related functions present in the `cfg80211` module [25] to receive notifications about scanning requests and delay a scan until after steps (1) and (2) are completed. The `cfg80211` module is a LKM, and thus, a *MuSher* compatible version can be loaded alongside the main *MuSher* module without requiring core kernel changes.

Signaling scan operation to the sender. The above approach 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 MPTCP sender to temporary disable all traffic to the subflow associated with the interface about to perform the scan. To this end, the *MuSher* client sends an ACK containing the MPTCP `MP_PRIO` option marking the interface as *backup*. The receipt of this ACK 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.

C. Accelerating Blockage Recovery

Our experiments in §III-D.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, the `pf` flag that was set upon blockage is reset, which allows for traffic to be scheduled on the 802.11ad subflow again. 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 §III-D.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 needs 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 that the mechanisms in §IV-B and §IV-C need to be initiated on the client side. However, in case of downlink-only

traffic, the scheduler is not run on the client side, and hence, the mechanisms will not 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. This allows to register a callback function inside our scheduler that runs even in the absence of any outgoing traffic. We then use the callback function to implement the solutions described above. This mechanism does not require any changes to the MPTCP code base or to any parts of the Linux kernel.

V. *MuSher*: Evaluation

In this section, we evaluate *MuSher* under a wide variety of scenarios including both stable and dynamic channel conditions, mobility, multiple MPTCP flows, and different combinations of link rates and delay settings. We tested and confirmed that the performance of *MuSher* with both throughput estimation methods is similar in all the evaluation scenarios. For brevity, we only show results with both approaches in scenarios involving multiple MPTCP flows in §V-E. For all other scenarios, we only show results with the TCP rate estimates, due to the additional overhead involved in enabling the IP rate estimates.

A. Ideal Channel Conditions

We begin by comparing the performance of *MuSher* under ideal channel conditions (static clients, no interference) against MPTCP's default *minRTT* scheduler and three other MPTCP schedulers from the literature: BLEST [22], ECF [26], and Peekaboo [27]. BLEST is a state-of-the-art scheduler included in the Linux kernel that tries to prevent head-of-line (HOL) blocking in heterogeneous environments. Similar to our approach, it disables the two memory optimizations introduced in [9] and replaces them with a proactive approach. Instead of penalizing the subflow that causes HOL blocking, it tries to estimate whether a path will cause HOL-blocking and dynamically adapts scheduling to prevent it. ECF is another state-of-the-art scheduler, which takes path heterogeneity into account. It monitors not only subflow RTT estimates, but also their congestion windows and the amount of data queued in the send buffer, and tries to determine whether using a slow path for the injected traffic will cause faster paths to become idle. We used the publicly available implementation from [28]. Peekaboo is a recently proposed learning-based scheduler that leverages an online reinforcement learning mechanism in combination with a stochastic adjustment strategy to adapt to the dynamic characteristics of the paths. We used the authors' publicly available implementation from [27], which integrates Peekaboo with Multipath QUIC (MPQUIC) [29] instead of MPTCP. The results are shown in Fig. 9, where we also include the SPTCP throughput over each interface alone and the theoretical sum for comparison.

Fig. 9 shows that *MuSher* and min RTT perform very well in this static scenario, as expected, both achieving about 97% of the theoretical sum. ECF and BLEST perform suboptimally (90% and 85% of the theoretical sum), as they often use only the faster of the two interfaces (802.11ad). Finally, Peekaboo performs extremely poorly achieving an average throughput of only 474 Mbps (24% of the theoretical sum), in spite of the stable channel conditions in this experiment. Note that in [27], Peekaboo was only evaluated over very low bandwidth (2-50 Mbps) and very long RTT (40-200 ms) scenarios and the parameters of the default implementation are most likely not optimized for the scenarios we consider in this work. In addition, Peekaboo runs over MPQUIC instead of MPTCP, and configuration differences between the two protocol implementations may also contribute to the poor performance. Exploring the potential of learning-based MPTCP schedulers in dual-band 802.11ad/ac WLANs is an interesting avenue for future work. Due to the suboptimal performance of ECF, BLEST, and Peekaboo even under ideal channel conditions, in the following we only compare *MuSher* against minRTT.

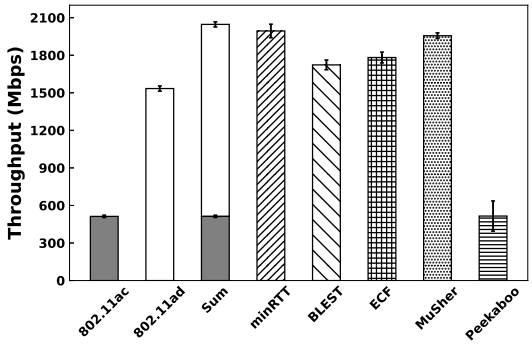


Fig. 9. MPTCP throughput comparison with static links.

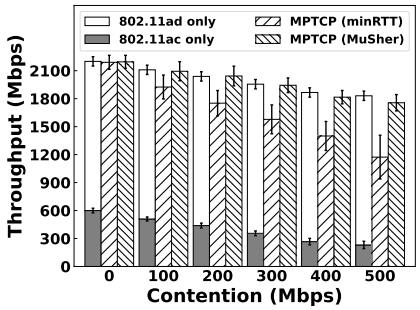


Fig. 10. Performance comparison under 802.11ac contention.

B. Varying Channel Conditions

We evaluate *MuSher* under different channel dynamics in a typical WLAN, involving static and dynamic contention on one of the two links, and client mobility, which changes channel conditions for both interfaces.

1) Static Contention (802.11ac/ad): We evaluate *MuSher* for different levels of contending 802.11ac traffic. We create contention by injecting UDP traffic on a separate 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. 10 shows the ideal MPTCP throughput (sum of 802.11ad and 802.11ac throughput), actual MPTCP throughput under the default *minRTT* scheduler, and *MuSher* throughput for different levels of contention (including the case without contention for reference). In the baseline contention-free scenario, both *minRTT* and *MuSher* perform optimally, as we

TABLE II
DYNAMIC 802.11AC CONTENTION

	minRTT (Gbps)	*MuSher* (Gbps)	Optimal (Gbps)	*MuSher*/Optimal (%)
1 s	1.53 ± 0.11	**1.68 ± 0.07**	1.78	94.3
5 s	1.42 ± 0.13	**1.70 ± 0.06**	1.78	95.5
10 s	1.41 ± 0.1	**1.68 ± 0.09**	1.78	94.3
20 s	1.35 ± 0.13	**1.71 ± 0.07**	1.78	96.0

also saw earlier (Table I). In all cases with contention, the default scheduler achieves less than the expected sum and the gap increases with higher contention. For instance, under 100 Mbps of cross-traffic, *minRTT* achieves ∼170 Mbps less than expected, whereas contention of 500 Mbps results in *minRTT* throughput of only ∼1150 Mbps vs. the expected sum of 1800 Mbps, a deficit of 650 Mbps. In contrast, *MuSher* detects the throughput change and converges to a close to optimal packet assignment ratio under all scenarios, achieving throughput close to the ideal sum and outperforming *minRTT* in the average case by 160-580 Mbps (a 50% gain in case of 500 Mbps cross traffic).

We also experimented with contention on the 802.11ad link. In this case, since we are constraining the faster of the 2 subflows, there is much less reordering at the receiver, thus making it significantly less challenging for MPTCP to handle. Hence, both the *minRTT* scheduler and *MuSher* perform similarly, and achieve close to the expected throughput under different levels of 802.11ad cross-traffic.

2) Dynamic Contention (802.11ac): We next evaluate how well *MuSher* reacts to changing cross-traffic. We vary contention levels between 300 Mbps and 500 Mbps over 120 s. The sequence of contention levels is selected at random but is kept the same across runs for both *minRTT* and *MuSher* for a fair comparison. 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 current level of contention.

Table II presents the results for four scenarios ranging from highly dynamic contention changes (every 1 s) to more stable with changes every 20 s. We observe that *MuSher* outperforms *minRTT* in all cases with average gains over the default scheduler of up to 360 Mbps in the 20 s case. Even in the most challenging scenario where contention changes every 1 s, *MuSher* provides on average 150 Mbps higher throughput than *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 obtained with a perfect scheduler in all cases, thanks to the low overhead of its triggering mechanisms and ratio probing strategy.

3) Mobility (802.11ad/ac): We consider 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 (i) and (ii), 802.11ad does not require frequent beam training as the relative angle between the client and AP does not change, whereas (iii) does require frequent 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 provides a lower bound on the performance of *MuSher*.

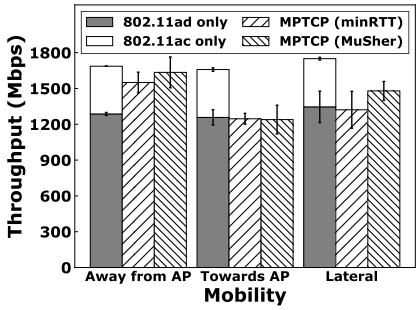


Fig. 11. Performance comparison under mobility.

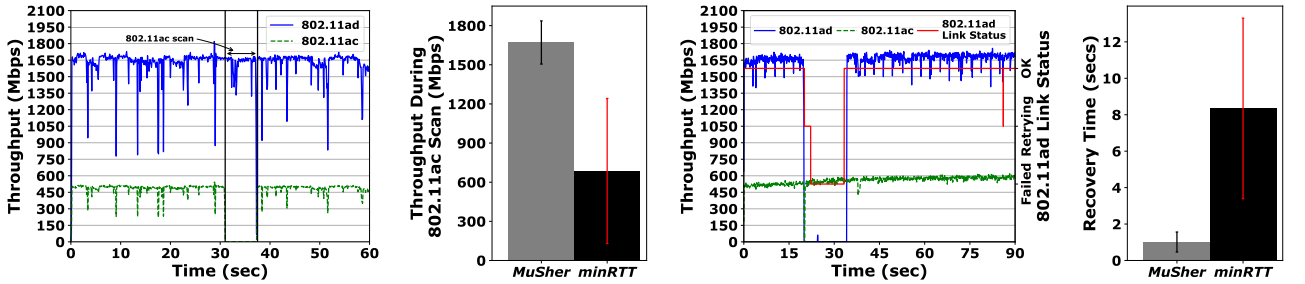
Fig. 11 compares the performance of *minRTT* and *MuSher*. For case (i) and (ii), *MuSher* and *minRTT* provide comparable performance with *MuSher* achieving slightly higher throughput in case (i). In the lateral mobility case, *MuSher* outperforms *minRTT* by ∼160 Mbps on average. The lateral case involves more drastic changes in 802.11ad throughput (as indicated by higher std. dev. of 802.11ad alone) which makes it challenging for *minRTT* to adapt. 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). It thus satisfies our original design goal for MPTCP to perform at least as good as SPTCP over the faster of the two interfaces. Overall, the gains from *MuSher* are lower under device mobility compared to the dynamic contention scenario. Mobility represents a much more challenging scenario, where channel conditions change much faster on both of the interfaces simultaneously and in a much more unpredictable fashion compared to the contention case.

C. Network Scans

Fig. 12a shows a timeline of an 802.11ac scan but with *MuSher*'s scan management solution applied during the scan period. We observe (compared to the scan period in Fig. 7a) 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. 12b, the MPTCP throughput for the former shows an average improvement from 700 Mbps to 1650 Mbps (136% gain).

D. 802.11ad Blockage

We test our solution in a setup similar to that in §III-D.3. Fig. 12c 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. 8c, where MPTCP resumed traffic on the 802.11ad subflow after a 20 s delay, here MPTCP



(a) Network Scan: Throughput timeline. (b) Network Scan: Throughput comparison. (c) Blockage: Throughput timeline. (d) Blockage: Recovery time comparison.

Fig. 12. Managing network scans and 802.11ad blockage.

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. 12d shows that *minRTT* on average takes 8 s to recover whereas *MuSher* can resume throughput in less than 2 s.

E. MuSher *With Multiple MPTCP Flows*

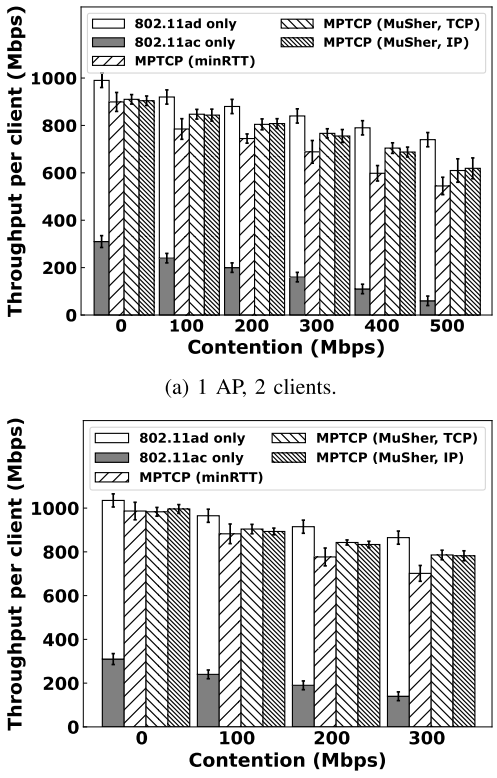
So far, we have investigated the performance of *MuSher* for a single MPTCP flow. In this section, we explore the performance of *MuSher* in two scenarios: 1) two clients connected to the same server via the same AP, and 2) two clients, each connected to a different server via a different AP. As described in §IV, here we evaluate the performance of *MuSher* using both the TCP and the IP rate estimates and compare the performance of the two approaches.

1) Single Access Point, Two Clients: We first look at the baseline performance when both clients are simultaneously running downlink MPTCP sessions (leftmost bars in Fig. 13a). We observe that the dynamics caused by multiple simultaneous flows make it challenging to always converge to the optimal packet assignment ratio. Hence, unlike in the single client scenario, neither the *minRTT* scheduler nor *MuSher* give optimal performance; both versions of *MuSher* and the *minRTT* scheduler achieve 91-92% of the expected throughput.

When we add different levels of contending 802.11ac traffic, we see a sharp drop in performance of the *minRTT* scheduler compared to either version of *MuSher* in Fig. 13a. For all levels of 802.11ac contention, *MuSher* outperforms *minRTT* by 60-100 Mbps on average, yielding up to a 17% improvement. Irrespective of the level of contending traffic and the rate estimates used, *MuSher* can always achieve ∼90% of the expected throughput. Overall, *MuSher* is able to handle dynamic link conditions better than the *minRTT* scheduler and assigns packets in a more optimal way to its subflows.

2) Two Access Points With One Client Each: Fig. 13b shows the performance of *MuSher* and the *minRTT* scheduler under a baseline scenario as well as under different levels of 802.11ac contention. In the baseline scenario, both versions of *MuSher* and the *minRTT* scheduler again perform similarly well, achieving ∼93-94% of the expected throughput.

When 802.11ac contention is introduced, *MuSher* slightly outperforms the *minRTT* scheduler. *MuSher* achieves around



(a) 1 AP, 2 clients.

(b) 2 APs, 2 clients.

Fig. 13. Performance comparison under 802.11 ac contention in scenarios with multiple MPTCP flows.

20-80 Mbps more than *minRTT* on average, a gain of up to 12%. In this scenario, the channel dynamics are even stronger than in the single-AP scenario in §V-E.1, especially on the 802.11ad channel. As was also shown in [30], 802.11ad's CSMA/CA protocol can be highly inefficient, leading to large throughput variations and non-optimal use of the channel by the multiple clients. This makes it particularly challenging to make decisions based on throughput estimates to find the optimal packet assignment ratio. Nonetheless, *MuSher* achieves reasonable performance given the 802.11ad MAC inefficiencies, yielding 85%-90% of the expected throughput under different levels of 802.11ac contention.

In both multiflow scenarios, *MuSher*'s performance is very similar with both rate estimation approaches. Hence, we recommend using TCP rate estimates, which do not incur the setup overhead of the IP-based approach, as explained in §IV.

TABLE III
PERFORMANCE COMPARISON OVER INTERNET PATHS

Bandwidth	RTT	*minRTT* (Mbps)	*MuSher* (Mbps)
100 Mbps	10 ms	96 ± 5	**96 ± 5**
	30 ms	95 ± 5	**94 ± 6**
	50 ms	94 ± 6	**93 ± 6**
1000 Mbps	10 ms	764 ± 142	**938 ± 51**
	30 ms	670 ± 153	**926 ± 125**
	50 ms	561 ± 156	**922 ± 143**
1800 Mbps	10 ms	1441 ± 332	**1682 ± 137**
	30 ms	924 ± 204	**1659 ± 214**
	50 ms	711 ± 180	**1640 ± 277**

F. 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. 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.

1) Baseline Performance: Table III compares the throughput (averaged over 10 runs) for different combinations of link rates and RTT values.

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 average 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 average throughput (130% 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 of 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 a TCP Reno style `cwnd` growth function, it takes a long time for the `cwnd` to recover to a value that allows to fully use 802.11ad's capacity.

TABLE IV
SCAN/BLOCKAGE PERFORMANCE OVER INTERNET PATHS

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**

TABLE V
PERFORMANCE WITH HETEROGENEOUS RTTS

802.11ac RTT	802.11ad RTT	*minRTT* (Gbps)	*MuSher* (Gbps)
5 ms	10 ms	1.76 ± 432	**2.28 ± 231**
	30 ms	1.56 ± 364	**2.25 ± 252**
	50 ms	1.00 ± 198	**2.26 ± 279**

2) Scan and 802.11ad Blockage: Table IV compares the effectiveness of *MuSher*'s scan (§IV-B) and blockage recovery (§IV-C) mechanisms with the default *minRTT* scheduler under three link rate/RTT settings.

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, the 802.11ad link alone can only provide 1.65 Gbps which *MuSher* utilizes fully, yielding a 4.1x average throughput gain compared to *minRTT*.

Blockage. *minRTT* takes at least 5 s on average in each of the three cases to recover after an 802.11ad blockage event, whereas *MuSher*'s average recovery time is *an order of magnitude faster* and is always below 0.17 s.

G. 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 V shows the performance of *minRTT* and *MuSher* for different combinations of RTT values.

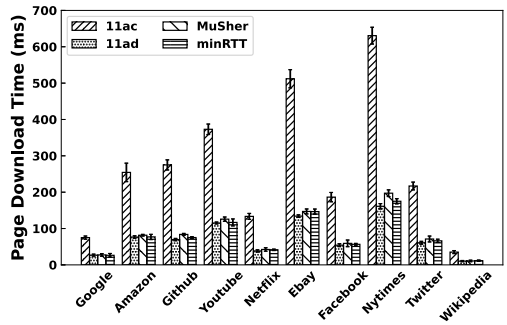
In all cases, the average throughput with *MuSher* is equal to the sum of 802.11ad (1.6 Gbps) and 802.11ac (600 Mbps) throughputs. We also analyzed the reverse scenario where 802.11ad delays are higher than 802.11ac and observed similar results. Hence, throughput ratio based scheduling can provide optimal 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 VI.

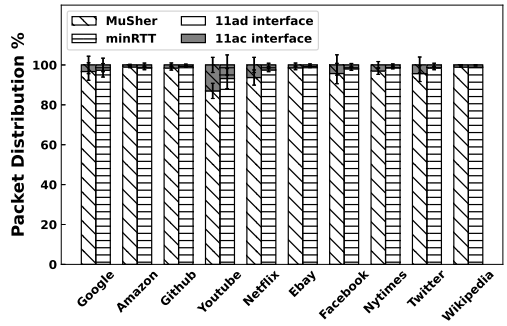
Even in this case, we observe that *MuSher* achieves close to the sum of the throughputs of the two interfaces. In contrast, the default *minRTT* scheduler not only fails to achieve the sum, but in fact yields lower throughput than that of the fastest

TABLE VI
EMULATED LTE-WIFI SCENARIO

WiFi RTT	LTE RTT	*minRTT* (Mbps)	*MuSher* (Mbps)
5 ms	10 ms	94.9 ± 5	**108 ± 7**
	30 ms	94.6 ± 6	**107 ± 6**
	50 ms	94.6 ± 6	**108 ± 7**



(a) Page download time.



(b) Packet distribution.

Fig. 14. Evaluation with web traffic.

interface alone, which agrees with results reported by previous studies analyzing MPTCP performance over 3G/LTE+WiFi.

H. MuSher *With Web Traffic*

In this section, we evaluate *MuSher* with web traffic, which consists of small bursts of downlink traffic. Due to the small size of most webpages, TCP may not always exit the slow start, and hence, the two subflows may not be backlogged. As such, we do not expect to see a performance improvement with *MuSher*. However, we want to ensure that out of order packet arrivals when MPTCP is used do not affect the packet delay and hence the responsiveness, which is more important than throughput for this type of traffic.

Since popular web content servers do not currently support MPTCP, we downloaded the content of 10 popular websites and hosted it on our local server. We then used *wget* on the laptop to download each website and measure the page download time. Fig. 14a compares the page download times with *MuSher*, minRTT, and SPTCP over each of the two interfaces.

In the case of web traffic, which consists of small transfer sizes, SPTCP over 802.11ad achieves the shortest page download times. In other words, the optimal decision in this case is to send all the traffic over the faster of the two interfaces. MPTCP schedulers (both minRTT and *MuSher*) perform slightly worse here, as they schedule some packets over 802.11ac, as shown in Fig. 14b, which plots the packet distribution over the two interfaces with each scheduler. Further, *MuSher* performs slightly worse than minRTT for most websites, as it probes different packet assignment ratios trying to discover the throughput-optimal ratio, which results in more packets being sent over the slower 802.11ac interface compared to minRTT for most websites (Fig. 14b). Overall, we conclude that, even though *MuSher* is not optimized for web traffic, it still achieves satisfactory performance, increasing the page download time by only a few milliseconds.

VI. RELATED WORK

Multipath schedulers. Previously proposed multipath schedulers target WiFi/cellular or Internet scenarios and they can be divided into three classes: (i) MPTCP schedulers that leverage the difference in the subflow RTTs alone [22], [26], [31]–[33], similar to the default scheduler, or in combination with other TCP parameters, such as *ssthresh*, *cwnd*, selective ACKs, or receiver buffer size [34]–[39]. In this work, similar to the results in [23], we show that under backlogged traffic, scheduling decisions become ACK-clocked and subflow RTTs have a negligible impact on packet scheduling. (ii) MPTCP schedulers that try to deal with issues caused by heterogeneous paths [22], [26], [27], [33], [39], [40]. 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. Thus, *MuSher* only targets dynamic scenarios that involve a drastic change in the wireless capacity of one of the two paths. (iii) Schedulers that improve MPTCP performance for specific application use cases [41], [42], require modifications to applications [43], or are inspired by MPTCP but not MPTCP compatible [44]–[46], and hence, are out of scope of this work. For example, DEMS [43] tries to ensure simultaneous subflow completion via a "two-way" splitting approach: the two subflows transfer the data in opposite directions, one from the beginning of an application layer chunk and the other from the end. As such, its design is very different from the design of MuSher, which tries to ensure the optimal packet scheduling ratio between the two subflows. More importantly, DEMS needs to know the application chunk boundaries, and hence, it requires changes to applications, thus violating one of the primary design principles of MPTCP – being transparent to applications.

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 into account 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 [47], 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 §III-D.3. Nonetheless, in contrast

to *MuSher*, [47] requires cross-layer information from the wireless driver and per-device calibration.

MPTCP performance over Internet and sub-6 GHz links. A number of works have evaluated different aspects of MPTCP performance under various scenarios [9]–[13], [44], [47]–[52]. 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., [10]–[13], [44]) show that heterogeneous paths result in significant performance degradation. In contrast, our measurement study in §III-C.2 shows that MPTCP works well in heterogeneous 802.11ac/802.11ad networks.

Combining mmWave and sub-6 GHz links. Very little work has been done towards leveraging MPTCP in networks involving mmWave links. Two recent works [14], [15] 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 authors in these works actually argued that the two radios should never be operated simultaneously and proposed a cross-layer approach [14] and the use of MPTCP *backup* mode [15] to select the best interface at any given time. Similarly, the work in [53] proposes an application-aware, seamless WiGig/WiFi handover solution using 802.11ad's FST. In contrast to all these works, *MuSher* combines both the 802.11ad and 802.11ac interfaces simultaneously instead of using one of them as a backup. The only work, other than *MuSher*, that simultaneously utilizes 802.11ac and 802.11ad radios, is Jigsaw [54], a cross-layer approach for 4K live video streaming, implemented in the Linux bonding driver. In contrast, this work leverages MPTCP for a generic solution, transparent to applications.

The work in [55], [56] explored MPTCP performance in 5G mmWave 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*. The work in [57] found, using the same simulators as in [55], [56], that the cause of the poor performance is the excessive congestion window growth in Slow Start phase, and proposed a BDP Estimation Based Slow Start (BESS) algorithm to address this problem. In contrast, to the best of our knowledge, our work is the first to provide an extensive experimental study of the performance of MPTCP over mmWave WLANs links, use 802.11ad and 802.11ac radios concurrently, and show that MPTCP works well in static scenarios regardless of the congestion control algorithm.

VII. 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 the performance degradation of MPTCP. Our evaluation in a wide range of scenarios showed that *MuSher* improves MPTCP throughput by up to 130% and it can accelerate recovery time from a link failure by an order of magnitude, compared to the default *minRTT* scheduler.

Acknowledgment

The contents of this work are solely the responsibility of the authors and do not represent the opinions or views of Microsoft.

References

[1] T. Nitsche, A. B. Flores, E. W. Knightly, and J. Widmer, "Steering with eyes closed: Mm-wave beam steering without in-band measurement," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 2416–2424.

[2] M. K. Haider and E. W. Knightly, "Mobility resilience and overhead constrained adaptation in directional 60 GHz WLANs: Protocol design and system implementation," in *Proc. 17th ACM Int. Symp. Mobile Ad Hoc Netw. Comput.*, Jul. 2016, pp. 61–70.

[3] S. Sur, X. Zhang, P. Ramanathan, and R. Chandra, "BeamSpy: Enabling robust 60 GHz links under blockage," in *Proc. USENIX NSDI*, 2016, pp. 193–206.

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

[5] *Use Multipath TCP to Create Backup Connections for iOS*. [Online]. Available: https://support.apple.com/en-us/HT201373

[6] *Commercial Usage of Multipath TCP*. Accessed: Feb. 12, 2020. [Online]. Available: http://blog.multipath-tcp.org/blog/html/2015/12/25/commercial_usage_of_multipath_tcp.htm

[7] *Hybrid Access Solution*. Accessed: Aug. 18, 2021. [Online]. Available: https://www:tessares:net/solutions/hybrid-access-solution/

[8] *IEEE 802.11ad, Amendment 3: Enhancements for Very High Throughput in the 60 GHz Band*, Standard IEEE 802.11 Working Group, 2012.

[9] C. Raiciu *et al.*, "How hard can it be? Designing and implementing a deployable multipath TCP," in *Proc. 9th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2012, pp. 399–412.

[10] Y.-C. Chen, Y.-S. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley, "A measurement-based study of MultiPath TCP performance over wireless networks," in *Proc. Conf. Internet Meas. Conf.*, Oct. 2013, pp. 455–468.

[11] S. Ferlin, T. Dreibholz, and O. Alay, "Multi-path transport over heterogeneous wireless networks: Does it really pay off?" in *Proc. IEEE Global Commun. Conf.*, Dec. 2014, pp. 4807–4813.

[12] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan, "WiFi, LTE, or both?: Measuring multi-homed wireless internet performance," in *Proc. Conf. Internet Meas. Conf.*, Nov. 2014, pp. 181–194.

[13] S. K. Saha *et al.*, "Multipath TCP in smartphones: Impact on performance, energy, and CPU utilization," in *Proc. 15th ACM Int. Symp. Mobility Manage. Wireless Access*, Nov. 2017, pp. 23–31.

[14] S. Sur, I. Pefkianakis, X. Zhang, and K.-H. Kim, "WiFi-assisted 60 GHz wireless networks," in *Proc. 23rd Annu. Int. Conf. Mobile Comput. Netw.*, Oct. 2017, pp. 28–41.

[15] K. Nguyen, M. G. Kibria, K. Ishizu, and F. Kojima, "Feasibility study of providing backward compatibility with MPTCP to WiGig/IEEE 802.11ad," in *Proc. IEEE 86th Veh. Technol. Conf. (VTC-Fall)*, Sep. 2017, pp. 1–5.

[16] C. Raiciu, M. Handley, and D. Wischik, *Coupled Congestion Control for Multipath Transport Protocols*, document RFC 6356, 2011.

[17] R. Khalili, N. Gast, M. Popovic, and J.-Y. Le Boudec, "MPTCP is not Pareto-optimal: Performance issues and a possible solution," *IEEE/ACM Trans. Netw.*, vol. 21, no. 5, pp. 1651–1665, Oct. 2013.

[18] Q. Peng, A. Walid, J. Hwang, and S. H. Low, "Multipath TCP: Analysis, design, and implementation," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 596–609, Feb. 2016.

[19] S. K. Saha *et al.*, "Fast and infuriating: Performance and pitfalls of 60 GHz WLANs based on consumer-grade hardware," in *Proc. 15th Annu. IEEE Int. Conf. Sens., Commun., Netw. (SECON)*, Jun. 2018, pp. 1–9.

[20] H. Assasa, S. Kumar Saha, A. Loch, D. Koutsonikolas, and J. Widmer, "Medium access and transport protocol aspects in practical 802.11 ad networks," in *Proc. IEEE 19th Int. Symp. World Wireless, Mobile Multimedia Netw. (WoWMoM)*, Jun. 2018, pp. 1–11.

[21] S. K. Saha, S. Aggarwal, R. Pathak, D. Koutsonikolas, and J. Widmer, "MuSher: An agile multipath-TCP scheduler for dual-band 802.11ad/AC wireless LANs," in *Proc. 25th Annu. Int. Conf. Mobile Comput. Netw.*, Oct. 2019, pp. 1–16.

[22] S. Ferlin, O. Alay, O. Mehani, and R. Boreli, "BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks," in *Proc. IFIP Netw. Conf. (IFIP Netw.) Workshops*, May 2016, pp. 431–439.

[23] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure, "Experimental evaluation of multipath TCP schedulers," in *Proc. ACM SIGCOMM Workshop Capacity Sharing Workshop*, Aug. 2014, pp. 27–32.

[24] *Iptables*. Accessed: Oct. 2, 2021. [Online]. Available: http://www:netfilter:org/projects/iptables/index:html

[25] *cfg80211 Subsystem: Scanning and BSS List Handling*. Accessed: Sep. 15, 2020. [Online]. Available: https://www.kernel.org/doc/html/v4.12/driver-api/80211/cfg80211.html#scanning-and-bss-list-handling

[26] Y.-S. Lim, E. M. Nahum, D. Towsley, and R. J. Gibbens, "ECF: An MPTCP path scheduler to manage heterogeneous paths," in *Proc. 13th Int. Conf. Emerg. Netw. EXperiments Technol.*, Nov. 2017, pp. 147–159.

[27] H. Wu, O. Alay, A. Brunstrom, S. Ferlin, and G. Caso, "Peekaboo: Learning-based multipath scheduling for dynamic heterogeneous environments," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 10, pp. 2295–2310, Oct. 2020.

[28] *ECF MPTCP Scheduler*. Accessed: Jan. 9, 2022. [Online]. Available: https://github.com/multipath-tcp/mptcp/blob/mptcp_trunk/net/mptcp/mptcp_ecf.c

[29] Q. D. Coninck and O. Bonaventure, "Multipath QUIC: Design and evaluation," in *Proc. 13th Int. Conf. Emerg. Netw. Exp. Technol.*, Nov. 2017, pp. 160–166.

[30] H. Assasa, S. K. Saha, A. Loch, D. Koutsonikolas, and J. Widmer, "Medium access and transport protocol aspects in practical 802.11ad networks," in *Proc. IEEE WoWMoM*, Jun. 2018, pp. 1–11.

[31] S. H. Baidya and R. Prakash, "Improving the performance of multipath TCP over heterogeneous paths using slow path adaptation," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2014, pp. 3222–3227.

[32] J. Hwang and J. Yoo, "Packet scheduling for multipath TCP," in *Proc. 7th Int. Conf. Ubiquitous Future Netw.*, Jul. 2015, pp. 177–179.

[33] N. Kuhn, E. Lochin, A. Mifdaoui, G. Sarwar, O. Mehani, and R. Boreli, "DAPS: Intelligent delay-aware packet scheduling for multipath transport," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2014, pp. 1222–1227.

[34] Y. Cao, Q. Liu, G. Luo, and M. Huang, "Receiver-driven multipath data scheduling strategy for in-order arriving in SCTP-based heterogeneous wireless networks," in *Proc. IEEE 26th Annu. Int. Symp. Pers., Indoor, Mobile Radio Commun. (PIMRC)*, Aug. 2015, pp. 1835–1839.

[35] D. Ni, K. Xue, P. Hong, and S. Shen, "Fine-grained forward prediction based dynamic packet scheduling mechanism for multipath TCP in lossy networks," in *Proc. 23rd Int. Conf. Comput. Commun. Netw. (ICCCN)*, Aug. 2014, pp. 1–7.

[36] D. Ni, K. Xue, P. Hong, H. Zhang, and H. Lu, "OCPS: Offset compensation based packet scheduling mechanism for multipath TCP," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2015, pp. 6187–6192.

[37] F. Yang, P. Amer, and N. Ekiz, "A scheduler for multipath TCP," in *Proc. 22nd Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2013, pp. 1–7.

[38] F. Yang, Q. Wang, and P. D. Amer, "Out-of-order transmission for in-order arrival scheduling for multipath TCP," in *Proc. 28th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, May 2014, pp. 749–752.

[39] T. Shreedhar, N. Mohan, S. K. Kaul, and J. Kangasharju, "QAware: A cross-layer approach to MPTCP scheduling," in *Proc. IFIP Netw. Conf. (IFIP Netw.) Workshops*, May 2018, pp. 1–9.

[40] H. Zhang, W. Li, S. Gao, X. Wang, and B. Ye, "ReLeS: A neural adaptive multipath scheduler based on deep reinforcement learning," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2019, pp. 1648–1656.

[41] X. Corbillon, R. Aparicio-Pardo, N. Kuhn, G. Texier, and G. Simon, "Cross-layer scheduler for video streaming over MPTCP," in *Proc. 7th Int. Conf. Multimedia Syst.*, May 2016, pp. 1–12.

[42] B. Han, F. Qian, L. Ji, and V. Gopalakrishnan, "MP-DASH: Adaptive video streaming over preference-aware multipath," in *Proc. 12th Int. Conf. Emerg. Netw. EXperiments Technol.*, Dec. 2016, pp. 129–143.

[43] Y. E. Guo, A. Nikravesh, Z. M. Mao, F. Qian, and S. Sen, "Accelerating multipath transport through balanced subflow completion," in *Proc. 23rd Annu. Int. Conf. Mobile Comput. Netw.*, Oct. 2017, pp. 141–153.

[44] A. Nikravesh, Y. Guo, F. Qian, Z. M. Mao, and S. Sen, "An in-depth understanding of multipath TCP on mobile devices: Measurement and system design," in *Proc. 22nd Annu. Int. Conf. Mobile Comput. Netw.*, Oct. 2016, pp. 189–201.

[45] A. Nikravesh, Y. Guo, X. Zhu, F. Qian, and Z. M. Mao, "MP-H2: A client-only multipath solution for HTTP/2," in *Proc. 25th Annu. Int. Conf. Mobile Comput. Netw.*, Aug. 2019, pp. 1–16.

[46] X. Zhu, J. Sun, X. Zhang, Y. E. Guo, F. Qian, and Z. M. Mao, "MPBond: Efficient network-level collaboration among personal mobile devices," in *Proc. 18th Int. Conf. Mobile Syst., Appl., Services*, Jun. 2020, pp. 364–376.

[47] Y.-S. Lim, Y.-C. Chen, E. M. Nahum, D. Towsley, and K.-W. Lee, "Cross-layer path management in multi-path transport protocol for mobile devices," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2014, pp. 1815–1823.

[48] C. Paasch, R. Khalili, and O. Bonaventure, "On the benefits of applying experimental design to improve multipath TCP," in *Proc. 9th ACM Conf. Emerg. Netw. Exp. Technol.*, Dec. 2013, pp. 393–398.

[49] B. Han, F. Qian, S. Hao, and L. Ji, "An anatomy of mobile web performance over multipath TCP," in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol.*, Dec. 2015, pp. 1–7.

[50] Q. De Coninck, M. Baerts, B. Hesmans, and O. Bonaventure, "Poster: Evaluating Android applications with multipath TCP," in *Proc. 21st Annu. Int. Conf. Mobile Comput. Netw.*, Sep. 2015, pp. 230–232.

[51] Q. D. Coninck, M. Baerts, B. Hesmans, and O. Bonaventure, "Observing real smartphone applications over multipath TCP," *IEEE Commun. Mag.*, vol. 54, no. 3, pp. 88–93, Mar. 2016.

[52] Q. D. Coninck, M. Baerts, B. Hesmans, and O. Bonaventure, "A first analysis of multipath TCP on smartphones," in *Proc. Passive Act. Meas. Conf. (PAM)*, 2016, pp. 57–59.

[53] Y.-Y. Li, C.-Y. Li, W.-H. Chen, C.-J. Yeh, and K. Wang, "Enabling seamless WiGig/WiFi handovers in tri-band wireless systems," in *Proc. IEEE 25th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2017, pp. 1–2.

[54] G. Baig *et al.*, "Jigsaw: Robust live 4K video streaming," in *Proc. 25th Annu. Int. Conf. Mobile Comput. Netw.*, Aug. 2019, pp. 1–16.

[55] M. Polese, R. Jana, and M. Zorzi, "TCP in 5G mmWave networks: Link level retransmissions and MP-TCP," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, May 2017, pp. 343–348.

[56] M. Polese, R. Jana, and M. Zorzi, "TCP and MP-TCP in 5G mmWave networks," *IEEE Internet Comput.*, vol. 21, no. 5, pp. 12–19, Sep. 2017.

[57] Y. Liu, X. Qin, T. Zhu, X. Chen, and G. Wei, "BESS: BDP estimation based slow start algorithm for MPTCP in mmWave-LTE networks," in *Proc. IEEE 88th Veh. Technol. Conf. (VTC-Fall)*, Aug. 2018, pp. 1–5.