# Accelerating Multipath Transport Through Balanced Subflow Completion

Yihua Ethan Guo, Ashkan Nikravesh, Z. Morley Mao, Feng Qian[†], Subhabrata Sen[‡]

University of Michigan [†]Indiana University [‡]AT&T Labs – Research
{yhguo, ashnik, zmao}@umich.edu fengqian@indiana.edu sen@research.att.com

## ABSTRACT

Simultaneously using multiple network paths (*e.g.,* WiFi and cellular) is an attractive feature on mobile devices. A key component in a multipath system such as MPTCP is the scheduler, which determines how to distribute the traffic over multiple paths. In this paper, we propose DEMS, a new multipath scheduler aiming at reducing the data chunk download time. DEMS consists of three key design decisions: (1) being aware of the chunk boundary and strategically decoupling the paths for chunk delivery, (2) ensuring simultaneous subflow completion at the receiver side, and (3) allowing a path to trade a small amount of redundant data for performance. We have implemented DEMS on smartphones and evaluated it over both emulated and real cellular/WiFi networks. DEMS is robust to diverse network conditions and brings significant performance boost compared to the default MPTCP scheduler (*e.g.,* median download time reduction of 33%–48% for fetchingfi les and median loading time reduction of 6%–43% for fetching web pages), and even more benefits compared to other state-of-the-art schedulers.

## 1 INTRODUCTION

Simultaneously using multiple network paths such as cellular and WiFi to accelerate data transfer is an attractive feature on mobile devices. It is supported by many commercial products such as Apple Siri [4], Gigapath by Korean Telecom [1], and Samsung Download Booster [2]. Researchers have also devised multipath strategies for applications such asfi le download [24, 29], web browsing [15, 16], and video streaming [9, 17, 38]. Currently the most widely used multipath solution is MPTCP [12], which enables unmodified applications to leverage multipath by adding a shim layer to the TCP interface. MPTCP establishes a *subflow* over each network path. The MPTCP sender distributes the data onto the subflows; the receiver reassembles the data into the original byte stream and delivers it to the app transparently.

MPTCP (or any transport-layer multipath scheme) has a complex protocol stack. In this work, we focus on optimizing the *schedulers*, which determine how the data is distributed onto the subflows. MPTCP supports different types of schedulers. For example, the *MinRTT* scheduler attempts to deliver the data as soon as possible by choosing a subflow with the smallest RTT unless its congestion window is full; the *ReMP* scheduler [13] boosts the reliability by duplicating packets over all subflows. There also exist schedulers that consider other dimensions such as energy efficiency [36], path priority [17] and receiver buffer occupancy [25].

Despite these efforts, we found that the multipath scheduler design is far from being optimal. In a pilot experiment conducted in §3, we observe that surprisingly, under representative WiFi/LTE network conditions, the MinRTT scheduler inflates the download time for a medium-sizedfi le by up to 33% compared to the optimal scheduling decision derived offline. In real-world networks withfl uctuating bandwidth or latency, MinRTT may perform even worse (up to 7.5x download time increase, 49% median increase compared to optimal scheduling) as shown in §7. Regarding the root cause, our key insight is that for such afi le download, oftentimes the subflows do *not* complete at the same time at the *receiver* side. This inevitably leads to suboptimal performance: if Subflow A completes earlier than Subflow B, one could achieve a shorter overall download time by offloading some of the traffic from Subflow B to A. Therefore, achieving simultaneous subflow completion is a necessary condition for minimizing the data transfer time.

The key contribution of this paper is the design, implementation, and evaluation of DEMS (**DE**coupled **M**ultipath **S**cheduler[1]), a new multipath packet scheduler aiming at reducing the data chunk download time over multiple paths. A *data chunk* is simply a block of application-defined bytes, which is a very common data transfer workload in a wide range of applications, *e.g.,* fetching an image, JavaScript, MP3fi le, or video chunk. The key idea behind DEMS is to *achieve simultaneous subflow completion at the receiver side through strategic packet scheduling over decoupled subflows* in order to minimize the chunk download time. Accomplishing this seemingly straightforward task, however, faces several challenges. First, MPTCP by default treats the user data as a continuous stream without even knowing the chunk boundary, letting alone performing any optimization for it. Second, the scheduler works at the sender side while we need to balance thefl ow completion time at the receiver side so there exists a "visibility gap" between the sender and receiver. Third, the task is further complicated by thefluctuating network conditions in wireless networks. Our judicious design addresses all above challenges as follows.

• As afi rst prerequisite for optimizing the chunk download time, DEMS is aware of the boundary of a chunk. Since how the data within a chunk is delivered does not matter (as long as the chunk can

[1]Note here "decoupled scheduling" is different from the decoupled congestion control in MPTCP.


*MobiCom '17, October 16–20, 2017, Snowbird, UT, USA*

ACM ISBN 978-1-4503-4916-1/17/10…$15.00
https://doi.org/10.1145/3117811.3117829

be reassembled correctly), DEMS can employfl exible and efficient schemes to split the chunk over subflows. For example, if only two paths are involved, one path can send data from the beginning in the forward direction, and the other path sends data from the end in the backward direction. Doing so simplifies our design and facilitates the performance by decoupling the subflows (§4.1).

• We devise a technique to ensure simultaneous subflow completion on the receiver side. To achieve this, at the *sender* side, DEMS strategically introduces a timing offset between the two subflows with different RTTs so that the last packets across all subflows will arrive at the receiver at the same time. The timing offset is dynamically determined based on the network latency and bandwidth dynamics (§4.2).

• Thefl uctuation of bandwidth and latency may still cause some differences in completion times across the subflows. To minimize this negative performance impact, DEMS performs data *reinjection*: if one subflowfi nishes earlier, it can "help" other subflows by transmitting a small portion of data (typically towards the end of the chunk download) that was already assigned to another subflow. Such data may be redundant (*i.e.,* transferred twice over two subflows) but it helps further reduce the overall download time (§4.3). We develop a method that adaptively determines the amount of the redundant data to strike a sweet spot between the performance and the additional data transmission due to reinjection (§4.4).

DEMS can work with any data transfer size and/or traffic pattern. The ideal workload on which DEMS yields the highest benefits is downloading application data chunks, which are very common to mobile traffic workloads. A wide range of mobile applications such as web browsing and video streaming involve downloading such data chunks (*e.g.,* an HTTP object or a video segment).

We integrated the DEMS components into a holistic system (§5) and implemented it on commodity mobile devices (§6). We conducted extensive evaluations over both emulated and real cellular/WiFi networks. The results indicate that DEMS is robust to diverse network conditions (including challenging multipath environments) and oftentimes brings significant performance boost compared to the state-of-the-art. Below highlights some key results.

• In stable network conditions, compared to MinRTT, DEMS reduces download time by up to 74%, 57% and 21% for 256KB, 1MB and 4MB download, respectively, under different combinations of delay/bandwidth of the paths. DEMS exhibits even better performance compared to ReMP, round-robin, and the best single path approach (§7.2, §7.5).

• In changing network conditions, DEMS reduces the median download time of 256KB and 1MBfi les by 12% to 46%, compared to MinRTT. Meanwhile, our adaptive reinjection scheme effectively reduces the redundant bytes by 48% to 86% compared to the naïve reinjection scheme while maintaining similar performance (§7.3).

• We conductedfi eld tests at 5 real-world locations such as a parking lot and a grocery store. DEMS reduces the download time by up to 88%, 83% and 77% for 256KB, 1MB and 4MBfi le, respectively, compared to MinRTT (the median reductions are 33%, 48%, and 42%, respectively). The real-world results are even better than the in-lab emulation results. Compared to wired networks, wireless networks like WiFi and cellular can sometimes exhibit high delay and bandwidth dynamics [22, 23]. Our evaluation demonstrates that compared to existing multipath schedulers, DEMS is able to robustly handle such environments (§7.4).

• DEMS reduces the median web page load time (across 10 popular websites) by 6% to 43% (median: 25%) under real network conditions, compared to MinRTT (§7.6).

Overall, ourfi ndings indicate that *strategically performing decoupled packet scheduling with balanced subflow completion can significantly improve the multipath transport performance*, which translates into better user QoE and improved energy efficiency (due to shortened radio-on time [21]). The remaining sections will focus on the motivation (§3), algorithm design (§4), system integration (§5), implementation (§6), and evaluation (§7) of DEMS. We discuss related work in §2 and limitations in §8.

## 2 RELATED WORK

**Characterization of Mobile Multipath.** Several prior efforts focus on characterizing the multipath performance over mobile networks. Chen *et al.* [6] studied thefi le download performance using MPTCP over 3G/4G and WiFi. Deng *et al.* [11] compared the performance between single-path and multipath in the mobile context. Han *et al.* [15] investigated the interaction between MPTCP and web protocols such as HTTP/1.1 and SPDY. Nika *et al.* [28] characterized the energy efficiency and performance of radio bundling (*i.e.,* multipath) in outdoor environments. Some other recent studies examined MPTCP performance [8] and its impact on applications [29] through crowd-sourced measurements.

**Multipath Schedulers** have been experimentally shown to affect download performance [28, 30, 31]. Several multipath scheduling algorithms have been proposed for different application scenarios. ECF makes scheduling decision using both congestion window size and RTT to avoid undesirable idle transmission periods and achieve higher aggregate throughput [26]. Compared to ECF, DEMS relies on chunk-based data transfer to decouple subflows and employs adaptive reinjection to combat variable network conditions. MPRTP focuses on real-time content delivery over multipath by adapting to the changing path characteristics [34]. DAPS aims at reducing receiver's buffer blocking time over multiple wireless networks [25]. ReMP duplicates the same packet onto all paths to reduce latency and to improve reliability [13]. eMPTCP takes energy consumption into consideration when making scheduling decisions [36]. Another energy-aware MPTCP scheduling algorithm was proposed in [32]. Compared to the above work, DEMS aims at reducing the download time for data chunks through a set of novel techniques.

**Applications of Mobile Multipath.** Previous studies have leveraged the MPTCP API extension to support enhanced socket options [18],fi ne-grained control on transport behavior [19], and multipath over UDP [5]. Some other studies investigated how to better use multipath for different applications, such as video streaming [7, 9, 17, 38], web browsing [16], smooth handover [10], and traffic sharing across users [27]. In contrast, DEMS is a general-purpose multipath scheduler that can benefit a wide range of applications.

## 3 BACKGROUND AND MOTIVATION

After providing necessary background in §3.1, we reveal the performance inefficiency caused by MPTCP's scheduling algorithm in §3.2, which leads to the key principle of DEMS in §3.3: ensure simultaneous subflow completion.

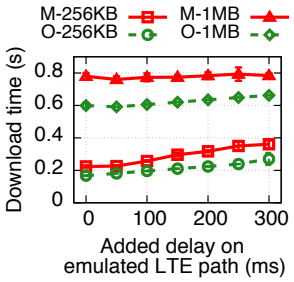



**Figure 1: Compare chunk download time (M: MinRTT, O: Optimal).**

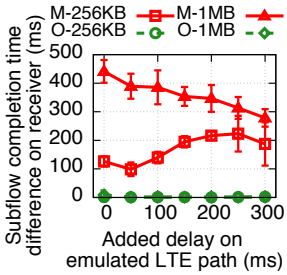



**Figure 2: Compare subflow completion time on receiver.**

## 3.1 Multipath TCP and its Schedulers

Multipath TCP (MPTCP [12]) enables simultaneous usage of multiple network paths (*a.k.a.* subflows). In the remainder of this paper, we primarily focus on two paths, as they correspond to the most common mobile multipath usage scenarios: jointly using WiFi and cellular on a smartphone or WiFi and Bluetooth on a wearable. We discuss how DEMS can be extended to more than two paths in §8.

MPTCP has a complex transport protocol stack consisting of several components: subflow management, packet scheduling, congestion control,fl ow control, *etc.*, among which we focus on optimizing the *schedulers*. A multipath scheduler takes packets from applications (stored in the "meta buffer") and determines on which subflow to transmit each packet. MPTCP currently supports three schedulers: round-robin, ReMP (sending the same data to all subflows for better reliability), and MinRTT. Among them, MinRTT is the default scheduler aiming at reducing the overall data transfer time. As long as the congestion window allows, MinRTT favors the subflow with the smallest RTT so that the packet can be delivered as soon as possible. The MinRTT scheduler is simple and robust, and has registered wide usage in practical systems [1, 4].

## 3.2 Can We Further Improve MinRTT?

Consider the common task of downloading a data chunk, which can be an image, a Javascript, an audio snippet, or a video chunk, over multipath. Our apparent goal is to minimize the download time. We conduct a pilot experiment to demonstrate (1) the impact of the scheduler on the download time, and (2) the potential room for improving MinRTT.

The experiment was conducted on a laptop with both WiFi and Ethernet connectivity, which emulate WiFi and LTE networks respectively using Linux `tc`. The network characteristics were chosen based on recent large-scale measurements of metropolitan LTE [22] and WiFi [35] users[2]. We use our user-level MPTCP testbed (to be described in §6) to provide the multipath support for the laptop.

We use the above setting to download a medium-sizedfi le of {256KB, 1MB} using the MinRTT scheduler and an optimal scheduling computed offline as follows. We download $p$% of thefi le over WiFi and $1-p$% over cellular. Tofi nd the best $p$ that leads to the shortest (*i.e.*, optimal) download time, we perform an "exhaustive search" by conducting many experiments covering the full range of $p \in [0, 100]$. We also vary the latency difference between the two paths by inflating the emulated LTE path, as it is common to have diverse path characteristics in mobile networks [11]. Note that except for the scheduling algorithm, the MinRTT and the optimal schemes share the same configurations (*e.g.*, the initial congestion window and congestion control algorithm) to ensure apples-to-apples comparison.

The results are shown in Figure 1. As shown, the scheduling decision clearly impacts the performance. Surprisingly, compared to the optimal case, MinRTT increases the download time by up to 33%. Note our experiment assumes stable network condition, whereas in real-world networks withfl uctuating bandwidth or latency, the gap between MinRTT and the optimal case can be even larger (up to 7.5x download time increase, 49% median increase compared to optimal scheduling) (§7). Also note that researchers have proposed other MPTCP schedulers such as deadline-aware scheduler [17], energy-efficient scheduler [36], real-time content scheduler [34] and buffer-blocking-aware scheduler [25]. They usually sacrifice chunk download time performance for other properties such as path priority and energy consumption, so we do not compare with them here.

By examining the results at the packet timing level, we identified a key reason why MPTCP yields suboptimal performance to be that the subflows do not complete at the same time at the receiver side. Figure 2 plots the two subflows' completion time difference when using the two schedulers. For MinRTT, the difference ranges from 100ms to 450ms while in the optimal scheme, the last bytes on the two subflows almost always arrive at the receiver simultaneously.

## 3.3 Ensuring Simultaneous Subflow Completion and its Challenges

Having all subflows complete at the same time at the receiver side is a necessary condition for achieving the optimal performance. The reason can be easily shown through *proof by contradiction*: suppose in an optimal scheme, Subflow Afi nishes earlier than Subflow B; in that case Subflow B can further "offload" some bytes to Subflow A, leading to an even shorter download time. But why cannot the MinRTT scheduler achieve this same-time-completion property? The reasons are multi-fold as explained below.

- When making a scheduling decision, MinRTT only considers the latency of each subflow without taking into consideration the bandwidth. Oftentimes, despite one subflow having a higher RTT than other subflows or even having a full congestion window (so it is temporarily unavailable), its higher bandwidth allows it to drain data from its sender buffer quickly. As a result, choosing the higher-RTT subflow can actually lead to lower end-to-end latency.
- Under practical network conditions (wireless in particular), the RTT and bandwidth are often highlyfl uctuating, leading to unbalanced subflow completion time. This factor is largely not taken into account by MinRTT.
- MinRTT is stateless in that the scheduling decision of the current packet does not explicitly depend on the previous packets. We will show that by strategically leveraging the information of previously transmitted packets, the scheduling decisions and thus the overall performance can be improved.

[2]WiFi: uplink 2020kbps, downlink 7040kbps, RTT 50ms;
LTE: uplink 2286kbps, downlink 9185kbps, RTT 70ms.

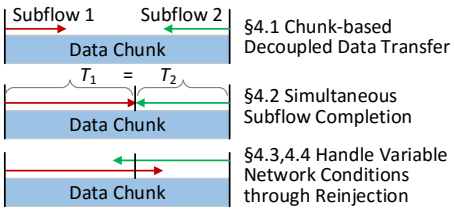



**Figure 3: Key design decisions of DEMS.**

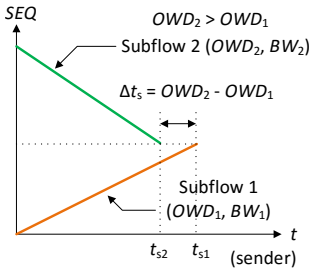



**Figure 4: Achieve simultaneous subflow completion (sender-side view).**

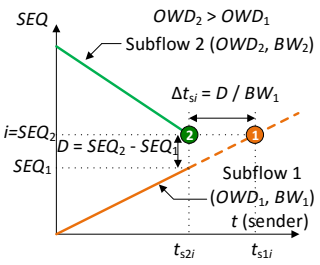



**Figure 5: Choose a subflow with an earlier estimated data arrival time (sender-side view).**

## 4 THE DEMS ALGORITHM

We propose DEMS, a new scheduling algorithm for reducing the data chunk download time over multipath. As shown in Figure 3, the key design decisions of DEMS include the following. (1) DEMS is aware of the chunk boundary, and it strategically decouples the paths for chunk delivery (§4.1). (2) DEMS ensures simultaneous subflow completion at the receiver side (§4.2). (3) DEMS allows a path to trade a small amount of redundant data for performance (§4.3, §4.4). We next elaborate them in this section.

### 4.1 Chunk-based Data Transfer

In DEMS, by default, data is delivered to the application on a *per-chunk* basis. A (data) chunk consists of a block of bytes defined by the application, which can be, for example, an image, a Javascript, an audio snippet, or a video chunk. At the sender side, after the sender app pushes the chunk to the multipath meta buffer, DEMS treats all data in the meta buffer as a chunk by default, thus being fully transparent to applications. At the receiver side, when the chunk is fully received, it is then delivered to the application. We will describe in §5 implementation alternatives for making DEMS aware of the chunk boundaries informed by applications through a simple API.

As long as a chunk can be correctly reassembled, bytes within the chunk can be delivered in any order. The data chunk is thus split into different parts that are distributed onto different paths for delivery. For the common scenario involving two paths, we design a "two-way" splitting approach: the two paths transfer the data in opposite directions, one from the beginning and the other from the end; when they "meet" each other, the chunk is fully downloaded. This approach is intuitive and parameterless. Furthermore, it helps improve the multipath performance by *decoupling* the two subflows. In conventional MPTCP, subflows are tightly coupled; a stall (*e.g.,* due to packet loss) in one subflow may slow down other subflows due to their limited and shared meta receive window whose size is difficult to set [29]. DEMS, on the other hand, decouples the two subflows by allowing each subflow to freely and independently transfer the data until the very end when subflows meet and merge. The receive window (16 MB by default) is configured to be larger than a typical chunk size so it will not become a performance bottleneck during a chunk transmission [29]. In rare cases when the application data is larger than the receive window, the data will be split into multiple chunks for transmission.

### 4.2 Simultaneous Subflow Completion

Now let us consider how to simultaneously complete subflows at the receiver side. Recall that the high-level idea is to introduce a timing offset at the sender to compensate the heterogeneous delay across both subflows.

Let usfi rst assume that the one-way delay (OWD) of both subflows can be accurately predicted. Let $OWD_1$ and $OWD_2$ be the OWD of Subflow 1 and 2, respectively, where $OWD_2 > OWD_1$. Let $t_{s1}$ and $t_{s2}$ be the time when the last byte is transmitted over Subflow 1 and 2, respectively. Let $t_{r1}$ and $t_{r2}$ be the time when Subflow 1 and 2 receives the last byte, respectively, *i.e.,* the subflow completion time at the receiver. If no packet reordering or loss happens, we have:

$$t_{r1} - t_{s1} = OWD_1 + t_{\text{offset}} \quad (1)$$

$$t_{r2} - t_{s2} = OWD_2 + t_{\text{offset}} \quad (2)$$

where $t_{\text{offset}}$ is the clock difference between the sender and receiver (handling network conditionfl uctuation caused by packet losses/reordering will be discussed in §4.3 and §4.4). Thus, the subflow completion time difference is:

$$t_{r2} - t_{r1} = (t_{s2} - t_{s1}) + (OWD_2 - OWD_1) \quad (3)$$

Clearly the receiver-side subflow completion time difference depends on both the sender-side transmission completion time difference denoted as $\Delta t_s = t_{s2} - t_{s1}$ and the forward-path one-way delay difference denoted as $\Delta OWD = OWD_2 - OWD_1$. To ensure simultaneous subflow completion *i.e.,* $t_{r2} = t_{r1}$, we need:

$$t_{s1} - t_{s2} = OWD_2 - OWD_1 = \Delta OWD \quad (4)$$

Namely, at sender side, the subflow with a larger OWD mustfinish transmission $\Delta OWD$ earlier than the other subflow.

Recall that DEMS employs the "two-way" data split scheme where the two subflows start from the two ends of the chunk and move towards each other. This process is illustrated in Figure 4, which plots the *sender-side* view of the data transfer progresses at both subflows (the X axis is time and Y axis is the sequence number). Assuming the constant bandwidth, the data transfer curves are linear. Subflow 2 stops transmission at $t_{s2}$. Subflow 1 then spends $\Delta OWD$ time to transmit all its data,fi nishing at $t_{s1}$. To translate this into the *receiver-side* view is easy: by shifting the two curves towards the right by $OWD_1$ and $OWD_2$ respectively, they will meet at $t_{s1} + OWD_1 = t_{s2} + OWD_2$, indicating they complete simultaneously at the receiver side.

Now we describe how to incorporate the above idea into the DEMS scheduler design, which needs to handle two tasks: assigning each packet (with its byte range) to a subflow and deciding when

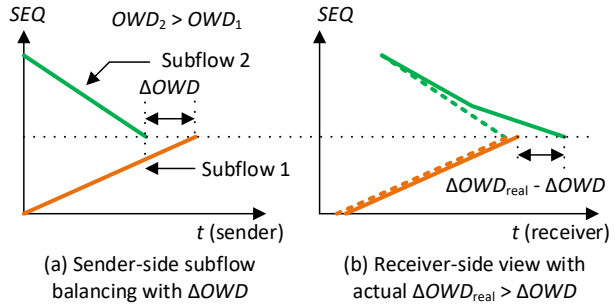



**Figure 6: Impact of inaccurate $\Delta OWD$ estimation.**

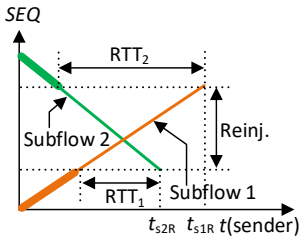



**Figure 7: A simple reinjection scheme.**

to stop a subflow at the sender. The former is trivial as the chunk has already been split in "two-way". So we focus on the latter task. The basic idea is as follows. When the subflow with a larger OWD can transmit a packet over the network, we estimate its arrival time over both subflows, and then choose the subflow with an *earlier* packet arrival time for actual transmission. We next show that this strategy will make the large-OWD subflow stop transmission when Equation (4) holds, thus resulting in simultaneous subflow completion shown in Figure 4.

Consider a general scenario depicted in Figure 5 where Subflow 2 with a larger OWD can now transmit a byte whose sequence number is $i = SEQ_2$. Should this byte be immediately transmitted over Subflow 2 (marked as ❷) or later over Subflow 1 (marked as ❶) so that Subflow 2 can stop now? If we choose Subflow 2, the byte's estimated arrival time at the receiver side is:

$$est(t_{r2i}) = t_{s2i} + OWD_{2i} + t_{\text{offset}} \tag{5}$$

where $t_{s2i}$ is the current sender-side timestamp and $OWD_{2i}$ is the current estimation of $OWD_2$. Now consider choosing Subflow 1 to transmit $SEQ_2$. Since Subflow 1 is currently working on a byte with a smaller sequence number $SEQ_1$, $SEQ_2$ has to be buffered in the meta buffer and gets transmitted at ❶ after all bytes from $SEQ_1$ to $SEQ_2 - 1$ are transmitted (over Subflow 1). Therefore, the estimated arrival time of $SEQ_2$ over Subflow 1 is:

$$est(t_{r1i}) = t_{s2i} + \Delta t_{si} + OWD_{1i} + t_{\text{offset}} \tag{6}$$

The buffering delay, $\Delta t_{si}$, is computed as:

$$\Delta t_{si} = \frac{SEQ_2 - SEQ_1}{BW_{1i}} \tag{7}$$

where $BW_{1i}$ is Subflow 1's current bandwidth estimation.

As said, we stop Subflow 2 and choose Subflow 1 when the estimated byte arrival time of the larger-OWD subflow (Subflow 2) is later than the smaller-OWD subflow (Subflow 1) *i.e.,* $est(t_{r2i}) > est(t_{r1i})$. In this situation, we have:

$$\Delta t_{si} = t_{s1i} - t_{s2i} < OWD_{2i} - OWD_{1i} \tag{8}$$

This is the same criteria as that in Equation (4), which guarantees the same receiver-side subflow completion time. Next, plugging Equation (7) into (8) yields:

$$D = SEQ_2 - SEQ_1 < (OWD_{2i} - OWD_{1i})BW_{1i} \tag{9}$$

$D$ corresponds to the number of bytes remaining to be transmitted in the meta buffer (imagine $SEQ_1$ and $SEQ_2$ as two "pointers" moving towards each other). We use Equation (9) in our system to determine when to stop the transmission for the larger-OWD subflow as the two-way chunk download approaches to its end. Note the small-OWD subflow can always transmit new packets before meeting with the large-OWD subflow, as long as the congestion window allows.

## 4.3 Handling Variable Network Conditions

So far we assume that both subflows' OWD and the smaller-OWD subflow's bandwidth (*i.e.,* those on the right-hand side of Equation (9)) can be accurately predicted. Apparently this assumption does not always hold in practice in particular for wireless networks. Figure 6 illustrates the impact of inaccurate network condition estimation. At the sender side shown in Figure 6(a), DEMS makes scheduling decisions based on its estimated $\Delta OWD$. Now assume Subflow 2's OWD increases over time, causing the sender to underestimate $\Delta OWD$. In consequence, at the receiver side shown in Figure 6(b), the actual data reception time over Subflow 2 (in solid line) deviates from the expected time shown in the dashed line. As a result, Subflow 1 completes early than Subflow 2, leading to suboptimal chunk download time. Similar reasoning can be made for $\Delta OWD$ overestimation and inaccurate bandwidth estimation.

DEMS tolerates unbalanced completion of subflows under variable network conditions. It also takes a key design decision of performing *reinjection*: instead of having a full stop when all bytes of a chunk are transmitted, a subflow may further "overshoot" its portion by sending a small number of bytes that are beyond the meeting point of the two subflows, as illustrated in Figure 7. These bytes are redundant because they have already been transmitted over the other subflow. The purpose of reinjection is to trade redundant data for better performance: under uncertain network conditions, if the reinjected (redundant) data arrives earlier than its original copy, the overall download time is reduced.

A key challenge here is to carefully control how many bytes to reinject: reinjecting too little data incurs suboptimal performance, while reinjecting too much causes unnecessary battery drain or data plan usage (for cellular networks). In the extreme case adopted by MPTCP's redundant scheduler (ReMP) that duplicates *every* byte onto the secondaryfl ow, MPTCP falls back to the "best single path" approach.

In DEMS, reinjection only occurs near subflows' meeting point. Wefi rst present a simple reinjection approach that gives a reasonable "upper bound" for the number of reinjected bytes. In this approach, after the two subflows meet, they perform reinjection in their corresponding directions until all bytes of the chunk are either acknowledged or reinjected. As shown in Figure 7, Subflow 2 keeps reinjecting data until $t_{s2R}$ when it hits the byte that was

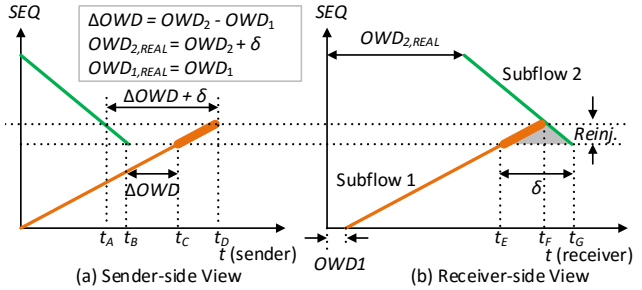



**Figure 8: The adaptive reinjection scheme. All OWD values are exaggerated in the plot for illustration purpose.**

transmitted and acknowledged over Subflow 1 (it takes $RTT_1$ for the ACK to arrive). Similarly, Subflow 1 stops reinjection at $t_{s1R}$ when the remaining data has been acknowledged by Subflow 2. At $t_{s1R}$, any byte within the chunk has either been acknowledged (shown as thick lines in Figure 7, or has a redundant copy infl ight (shown as thin lines). By analyzing the "X" shape in Figure 7 (assuming the last byte on each subflow is not lost or reordered), we can compute the total number of reinjected bytes to be:

$$Reinj = \frac{BW_1 BW_2}{BW_1 + BW_2}(RTT_1 + RTT_2) \tag{10}$$

where $BW_1$ and $BW_2$ are the bandwidth of the subflows. Note they are the slopes of the two lines in Figure 7.

Equation (10) gives a reasonable upper bound of the reinjection overhead, which however is still too high as to be evaluated in §7. More importantly, this reinjection approach is not adaptive in that the reinjection behavior remains the same regardless of the network conditionfl uctuation. Ideally, when thefl uctuation is low (high), we should reinject less (more) data given that the subflow completion time balancing technique introduced in §4.2 is more (less) reliable.

## 4.4 Adaptive Reinjection

We design a scheme that performs adaptive reinjection while maintaining good performance. Let usfi rst consider a scenario where $\Delta OWD$ is underestimated: its real value, denoted as $\Delta OWD_{\text{REAL}}$, is $\Delta OWD$ (the estimated version) plus $\delta$. We use $OWD_1$ and $OWD_2$ to denote the estimated OWD for each subflow, and use $OWD_{1,\text{REAL}}$ and $OWD_{2,\text{REAL}}$ to denote their real values, respectively. Since what really matters is the prediction accuracy of the *difference* between OWDs (*i.e.,* $\Delta OWD$), without loss of generality we assume $OWD_{1,\text{REAL}} = OWD_1$ and $OWD_{2,\text{REAL}} = OWD_2 + \delta$, for the ease of presentation.

Under the above setting, let usfi rst examine the sender side illustrated in Figure 8(a). Based on the completion time balancing technique introduced in §4.2, Subflow 1 and 2 would stop at $t_C$ and $t_B$ respectively where $t_C - t_B = \Delta OWD$. However, $\Delta OWD$ is underestimated. As a result, if we switch to the receiver's view in Figure 8(b), we will see that Subflow 1 completes $\delta$ time units before Subflow 2. This is because (assuming there is no clock drift between sender and receiver) $t_E = t_C + OWD_1$ and $t_G = t_B + OWD_{2,\text{REAL}}$, so $t_G - t_E = (OWD_{2,\text{REAL}} - OWD_1) - (t_C - t_B) = \delta$. Now let us consider how to perform reinjection. In Figure 8(b), to minimize the download time, Subflow 1 only needs to keep reinjecting data until it meets Subflow 2. The reinjected portion is highlighted in bold.

Now we derive the sender-side reinjection policy by shifting the reinjected portion back to Figure 8(a). When Subflow 1 reinjects the last byte at $t_D$, the byte's original transmission time (by Subflow 2) is $t_A$. As shown, $t_D - t_A = (t_B - t_A) + (t_C - t_B) + (t_D - t_C) = (t_G - t_F) + (t_C - t_B) + (t_F - t_E) = \Delta OWD + \delta$, which is the threshold for stopping reinjection. In other words, when $\Delta OWD$ is *underestimated* by $\delta$, the *smaller-OWD* subflow keeps reinjection until the to-be-reinjected byte was transmitted more than $\Delta OWD + \delta$ time units ago. That is (using the notions introduced in §4.2):

$$\Delta t_{\text{si}} = t_{\text{s1}i} - t_{\text{s2}i} > OWD_{2i} - OWD_{1i} + \delta \tag{11}$$

Let the number of the reinjected bytes be $r$. By examining the gray triangle in Figure 8(b), we have $r/BW_1 + r/BW_2 = \delta$, which leads to $r = \delta BW_1 BW_2/(BW_1 + BW_2)$ where $BW_1$ and $BW_2$ are the subflows' bandwidth.

The counterpart scenario of $\Delta OWD$ overestimation can be derived in a similar way (proof omitted): when $\Delta OWD$ is *overestimated* by $\delta$, the *larger-OWD* subflow keeps doing reinjection until the to-be-reinjected byte was transmitted less than $\Delta OWD - \delta$ time units ago, which is:

$$\Delta t_{\text{si}} = t_{\text{s1}i} - t_{\text{s2}i} < OWD_{2i} - OWD_{1i} - \delta \tag{12}$$

The reinjection overhead is also $r = \delta BW_1 BW_2/(BW_1 + BW_2)$. In reality we can only estimate (with a certain level of confidence) that $\Delta OWD_{\text{REAL}}$ falls in the range of $[\Delta OWD - \delta, \Delta OWD + \delta]$. DEMS thus lets both subflows to reinject according to Equation (11) and (12). The overall reinjection overhead is:

$$Reinj = 2\delta \frac{BW_1 BW_2}{BW_1 + BW_2} \tag{13}$$

These redundant bytes also help tolerate the inaccurate bandwidth prediction for the smaller-OWD subflow, as well as help recover from packet losses (within the reinjected byte range) quickly. Compared to (10), Equation (13) is usually much smaller. It is also adaptive as it is a function of $\delta$ that can be configured based on the predictability of the network condition. For example, $\delta$ can be set to $k \cdot StdDev(\Delta OWD)$.

## 4.5 Put Everything Together

We now walk through Algorithm 1, which combines chunk-based transfer (§4.1), subflow completion time balancing (§4.2), and adaptive reinjection (§4.4). The scheduling algorithm works at the sender side (the receiver-side logic is trivial). The input consists of the two network paths and a meta buffer that stores the packetized chunk data to be transmitted. The algorithm is invoked whenever either subflow can transmit a packet (*i.e.,* has some empty congestion window space). It makes a decision of transmitting a new packet, reinjecting a previously transmitted packet, or withholding transmission. Also note it only processes unacknowledged packets.

Line 7–13 handles the subflow with a smaller OWD. Recall that in §4.2, the default behavior is to always transmit a new packet over a small-OWD subflow whenever possible. Line 11–13 deals with the reinjection scenario according to Equation (11). Line 15–18 processes the subflow with a larger OWD. According to Equation (9), we may need to skip the large-OWD subflow to achieve simultaneous subflow completion. Line 16 performs adaptive reinjection over the large-OWD subflow according to Equation (12).

**Algorithm 1:** The DEMS scheduling algorithm

**Input**: subflow $i \in \{1, 2\}$ that can transmit packet, packets in the meta buffer $metaBuf[j]$, $j \in [0, m]$.
**Output**: packet $packet$ to transmit over subflow $i$.

```
1  packet ← GetNextUnAckedPacketOnThisSubflow(i);
2  smallOwdNo ← GetSmallOWDSubflowNo();
3  largeOwdNo ← GetLargeOWDSubflowNo();
4  ΔOWD ← GetΔOWD();
5  δ ← GetΔOWDVar();
6  trans ← false;
7  if i == smallOwdNo then
8      if not packet.tx [i] then
9          trans ← true;
10     else
11         delay ← ts − packet.ts [largeOwdNo];
12         if delay ≤ ΔOWD + δ then
13             trans ← true;
14 else
15     bw ← GetSubflowBW(smallOwdNo);
16     thres ← (ΔOWD − δ) ∗ bw;
17     if GetUntransmittedSize(metaBuf) ≥ thres then
18         trans ← true;
19 if trans then
20     Transmit(packet);
21     packet.ts [i] ← GetCurrTimestamp();
22 else
23     packet ← NULL;
```

## 5 SYSTEM DESIGN

We now elaborate on how to integrate the DEMS algorithm into a real system. Figure 9 plots the system diagram. At the sender side (right), the chunk data coming from the application is stored in the meta buffer, and is then split, scheduled, and transmitted by the packet scheduler. Working with the packet scheduler, the reinjection manager keeps track of packets' transmission states and makes decisions on adaptive reinjection. We also design a module for measuring and predicting network conditions ($\Delta OWD$ and bandwidth). The application can also optionally specify the chunk size through an API (to be elaborated next). The receiver side logic is much simpler. It passively receives/acknowledges the data, reassembles it in the receiver-side meta buffer, and delivers the in-order data to the application. Note that data over the (decoupled) subflows are acknowledged separately using the per-subflow ACK numbers. Meanwhile, similar to MPTCP, at the receiver meta buffer, the global sequence number carried by each packet is used to mark which portion within the chunk has been received. While Figure 9 illustrates one-way data transfer, our system supports full-duplex data transmission.

**Interaction with Applications.** DEMS provides applications with an optional hint API (through a socket option) to specify the chunk boundary, allowing DEMS to work on the chunks sequentially and to minimize the download time for each of them. Alternatively, if no hint is provided, DEMS leverages a heuristic that treats all data in the meta buffer as a single chunk. One issue here is that more data may arrive at the tail of the meta buffer from the app when the transmission is in progress. To handle this, we modify the packet fetch function (Line 1 in Algorithm 1) to let one subflow always fetch unacknowledged packets from the head of the meta buffer and the other subflow fetch from the tail of the

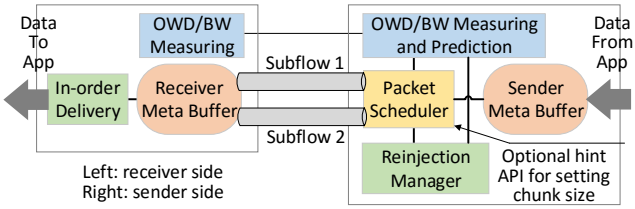



**Figure 9: System diagram of DEMS.**

meta buffer. The meta buffer is realized using a circular queue to allow efficient space reuse. In this way, DEMS is fully transparent to both the client-side and server-side applications. When there are multiple connections transmitting data simultaneously, DEMS can use a separate meta buffer for each connection and take bytes from each meta buffer in a round robin manner so that each connection is given a fair share of the network bandwidth. Other scheduling strategies such asfl ow prioritization can also be realized in DEMS.

**$\Delta OWD$ Measurement and Prediction.** OWD measurement requires cooperation between the sender and the receiver. The sender records the timestamp of each outgoing packet; the receiver records the reception time and sends it and the sequence number back to the sender through an encapsulated control message (§6). OWD is then estimated at the sender using exponential weighted moving average (EWMA) with $\alpha$ empirically chosen to be 0.25. Note that estimated OWD contains the sender-receiver clock drift, which is nevertheless cancelled out when a $\Delta OWD$ sample is calculated. Also, in wireless networks (cellular in particular), network latency is often correlated with the number of bytes-in-flight (BIF) [14, 23, 39]. We thus bin OWD samples for each subflow separately using measured BIF with a bin size of 10KB. The samples in each bin are processed separately using EWMA to facilitate a more accurate OWD estimation for each subflow at different BIF levels. We set $\delta$, the parameter controlling adaptive reinjection (§4.4), to be the standard deviation of $\Delta OWD$. The subflow bandwidth is also estimated using EWMA over measured samples. Note DEMS only needs the bandwidth estimation for the subflow with a smaller OWD (see Equation (9)). DEMS can also incorporate more sophisticated prediction methods (*e.g.,* those taking special consideration of network uncertainties [20]) to improve the prediction accuracy.

**Congestion Control and Packet Losses.** DEMS is compatible to any congestion control (CC) algorithm such as decoupled CC, LIA [37] and OLIA [24]. In our experiments we use decoupled CC that mobile multipath typically uses [15, 29]. Also, DEMS is robust to packet losses. Since the network condition prediction is performed in an online manner, delay or bandwidthfluctuation caused by (real or spurious) packet losses can be quickly picked up and reflected in the scheduling decision changes. We demonstrate in §7.4 that DEMS works well under diverse real-world scenarios including those with poor network conditions.

## 6 IMPLEMENTATION

DEMS can be directly integrated into MPTCP as a new scheduler. However, from the perspective of conducting real-world evaluations, MPTCP has two issues:fi rst, most of today's commercial Internet servers do not yet support MPTCP, making testing real

workload difficult; second, MPTCP uses special TCP options that are often blocked by commercial cellular middleboxes [29].

To facilitate real-world test, we implemented a multipath TCP proxy infrastructure in C/C++. Between the proxy and the client host, multipath is realized as multiple conventional TCP connections each corresponding to a subflow established over a network path. For uplink traffic, at the client side, an application TCP connection's data is transparently split over the subflows using a custom light-weight encapsulation protocol (as opposed to using special TCP options); the data is then merged at the proxy and delivered to the server over conventional single-path TCP to ensure server transparency (assuming the client-proxy paths are the bottleneck). The downlink traffic is handled symmetrically. We have replicated MPTCP's three schedulers: MinRTT, round-robin, and ReMP by precisely following their algorithms. Compared to MPTCP, our proxy-based approach offers the same performance (based on our lab test[3]) while providing server transparency and middlebox friendliness.

We then implemented DEMS on our multipath proxy infrastructure. Most of the scheduling logic is implemented in the user space. Some low-level functionalities such as OWD/bandwidth measuring and prediction are implemented in the kernel through a light kernel module. Overall DEMS is lightweight: our implementation (not including the base proxy system) consists of 970 LoC (450 LoC for the scheduler and 520 LoC for network condition measurement/prediction). DEMS is generally compatible with Linux-based systems.

# 7 EVALUATION

We systematically evaluate DEMS under various settings including different network setups (emulated vs. real networks), different network conditions (stable vs.fl uctuating), different workload (raw chunk download vs. real application workload), different clients (smartphone vs. laptop), *etc.*

## 7.1 Experimental Setup and Methodology

We study three variants of DEMS: DEMS-B, DEMS-S, and DEMS-F. They all employ chunk-based data transfer, decouple the subflows in the "two-way" manner, and attempt to ensure simultaneous subflow completion (§4.1 and §4.2). Their differences are the reinjection policy. DEMS-B ("Basic") does not perform reinjection; DEMS-S ("Simple reinjection") employs the naïve reinjection strategy described in §4.3; DEMS-F ("Full") performs the adaptive reinjection described in §4.4.

Our evaluation focuses onfi le download (upload can be handled by DEMS symmetrically). We set up our multipath proxy (developed in §6) on a commodity server with 4-core 3.6GHz CPU, 16GB memory, and 64-bit Ubuntu 16.04. The meta buffers at both the proxy and the receiver side are configured to be sufficiently large to avoid performance degradation due to limited buffer size. For apples-to-apples comparison, all schedulers use the same decoupled TCP congestion control (TCP CUBIC) and same subflow-level TCP send/receiver buffer sizes (8MB by default). Unless otherwise noted, the application server (hosting afi le or web contents) is near the proxy, and uses only single-path. The RTT between the proxy and the app server is configured to be 4ms, which is the median RTT between a mobile ISP gateway (where our proxy can be deployed) and 30 popular content providers' servers based on a recent measurement study [33]. The client-proxy paths covering the "last-mile" wireless hops are thus the bottleneck.

We evaluated DEMS over both emulated and real multipath environments of WiFi and cellular. For emulation, we use Linux `tc` to throttle the bandwidth and to add extra delay on the client-proxy paths. By default, we use the network condition profiles from large-scale measurements of metropolitan LTE [22] and WiFi [35] users (same numbers as those used in §3.2). To emulate in-network buffering, we keep a 50ms bottleneck buffer for WiFi and a 500ms bottleneck buffer for LTE (set based on [23]) using `tc`.

We use two devices for evaluation: a laptop and a smartphone. The laptop is an HP EliteBook 840 with 1.90GHz dual-core CPU and 8GB memory, running Linux 3.18. When doing emulations over the laptop, we use Ethernet to emulate the LTE network, and use WiFi as it is. The smartphone is a Nexus 6p running Android 7.0 on Linux 3.10. We use its WiFi and cellular interfaces for experiments.

Recall in §5 that an application can interact with DEMS either using or not using a hint API. We adopt the non-API mode so DEMS is fully transparent to our applications. Finally, note that all evaluations were conducted on our multipath proxy infrastructure instead of MPTCP (recall in §6 that we replicated all MPTCP's schedulers on our infrastructure).

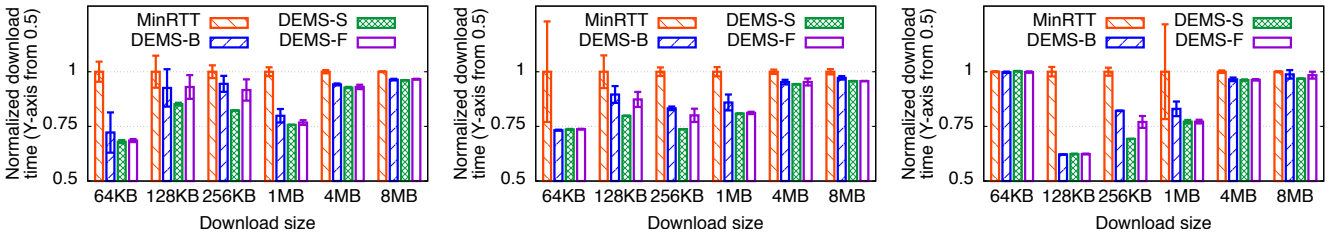
## 7.2 Stable Network Conditions

Wefi rst study the performance of DEMS under stable network conditions using emulation. The workload isfi le download. For each test, we repeat the download for 10 times and report the average value. We consider different delay differences and bandwidth of the two paths. Unless otherwise noted, we use MinRTT, the default MPTCP scheduler, as the comparison baseline.

**Different Delay Differences.** Figure 10 compares the download time across the three DEMS variants and MinRTT for different file sizes. We consider three RTT combinations to cover different subflow delay differences. All three DEMS variants outperform MinRTT. Among the three variants, DEMS-S achieves the highest download time reduction (15% to 38% compared to MinRTT) for medium-sizedfi les (128KB to 1MB) due to its aggressive reinjection behavior. The performance gain brought by DEMS-F decreases slightly in most cases. However, Figure 11, which compares the size of reinjected data incurred by DEMS-S and DEMS-F, clearly indicates that DEMS-F strikes a much better balance between the performance and the reinjection overhead: compared to DEMS-S, DEMS-F reduces the reinjected data size by 23% to 100%. Regarding DEMS-B, it slightly falls behind DEMS-F by up to 8%. The difference is small because the network conditions are stable here.

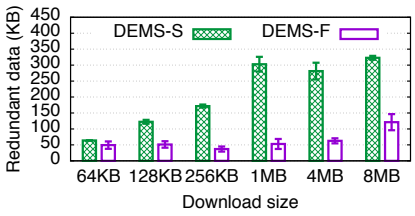
Figure 10 indicates that DEMS brings more download time reduction as the two paths' RTT difference becomes larger. This is because a larger RTT difference usually indicates a larger $\Delta OWD$ that leads to more room for DEMS to balance the subflow completion time. Figure 10 also shows that DEMS's benefits are maximized when thefi le size is small or medium. For largefi les such as 8MB, all four schemes exhibit similar performance. The reason, as mentioned in §8, is that the subflow completion time difference is dwarfed

[3]In the test, the server is co-located with the proxy to ensure MPTCP and our approach traverses the same network paths.




(a) WiFi RTT 50ms, LTE RTT 70ms (b) WiFi RTT 50ms, LTE RTT 270ms (c) WiFi RTT 50ms, LTE RTT 470ms

**Figure 10: Compare performance between DEMS and MinRTT on downloadingfi les with different sizes (laptop, emulation). The WiFi and LTE bandwidth are 7040kbps and 9185kbps respectively.**




(a) WiFi 50ms, LTE 70ms (b) WiFi 50ms, LTE 270ms

**Figure 11: Compare redundant data between DEMS-S and DEMS-F (laptop, emulation).**

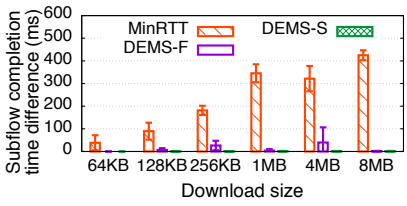
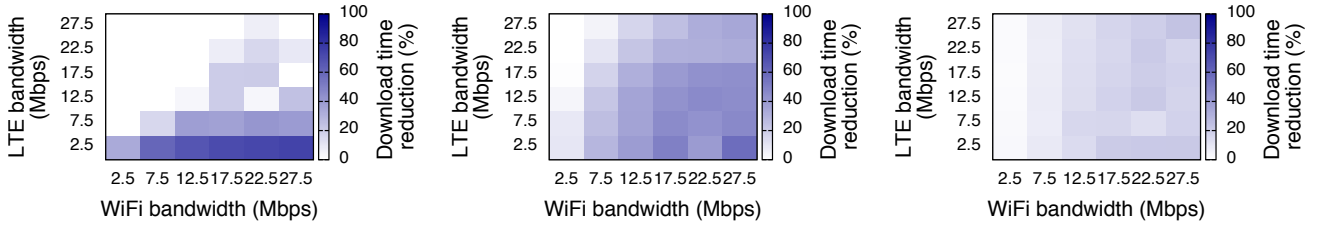



**Figure 12: Compare subflow completion time difference (laptop, emulation, WiFi RTT 50ms, LTE RTT 270ms).**




(a) 256KB download (b) 1MB download (c) 4MB download

**Figure 13: Download time reduction brought by DEMS-F compared to MinRTT under 36 bandwidth combinations (laptop, emulation, WiFi RTT 50ms, LTE RTT 70ms).**

by the long download duration. Nevertheless, we do expect that for future faster networks such as 5G network, DEMS will benefit large transfers. For the 64KBfi le download in Figure 10(c), all four schemes have the same performance because in this case the best strategy is to use the best single path.

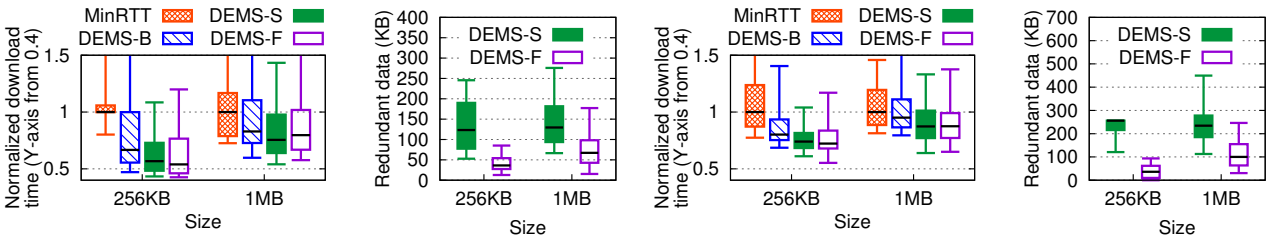
Figure 12 plots the subflow completion time difference (measured at the receiver) between the two paths when three schedulers are used. It confirms that our algorithm in §4.2 effectively achieves simultaneous subflow completion, which leads to shorter chunk download time compared to MinRTT.

**Different Bandwidth Combinations.** To understand the impact of the paths' bandwidth on DEMS, We compare the download time of DEMS-F and MinRTT under 6*6=36 bandwidth combinations of WiFi and emulated LTE networks, for threefi le sizes (256KB, 1MB, and 4MB). The bandwidth of each path varies from 2.5Mbps to 27.5Mbps. The heat maps in Figure 13 visualize the download time reduction brought by DEMS-F compared to MinRTT. We observe two trends. First, as the WiFi bandwidth increases, the overall download time decreases. Therefore the optimizable portion of DEMS (*i.e.,* the two subflows' completion time difference) becomes more prominent, leading to more perceived performance enhancement. Second, increasing the LTE bandwidth also shortens the download time. However, it also reduces the in-network queuing delay and henceforth reduces $\Delta OWD$. Therefore whenfi xing the WiFi bandwidth and increasing the LTE bandwidth, the download time reduction incurred by DEMS is not prominent. Overall, DEMS-F achieves up to 74%, 57%, and 21% of download time reduction for 256KB, 1MB and 4MB download, respectively, compared to MinRTT.

## 7.3 Varying Network Conditions

We next evaluate how DEMS performs under changing network conditions. We considerfl uctuating latency and bandwidth separately.

**Varying Latency.** We conduct experiments at a location on our campus where the RTTs of both WiFi and LTE experience high variance (the *stddev* of RTT is up to 40% and 50% of the mean for WiFi and LTE, respectively) while the bandwidth is sufficiently




(a) BW Profile 1: download time (b) Profile 1: redundant data (c) BW Profile 2: download time (d) Profile 2: redundant data

**Figure 14: Compare different scheduling algorithms under varying network conditions (laptop, trace-driven emulation).**

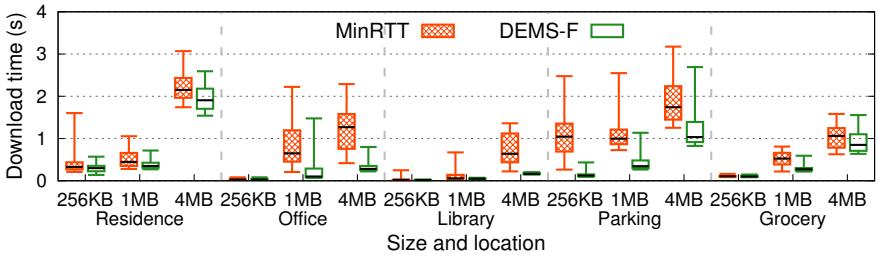



**Figure 15: Compare performance of different scheduling algorithms (smartphone, real WiFi/LTE atfi ve locations).**

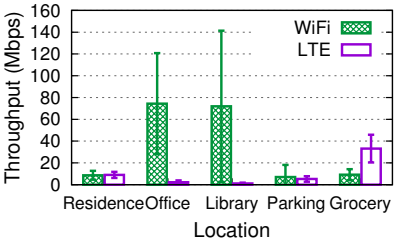



**Figure 16: Downlink throughput of real WiFi and LTE atfi ve locations.**

high. We then throttle the bandwidth to 7040kbps and 9185kbps for WiFi and cellular respectively before launching thefi le download experiments (256KB and 1MB, each repeating 10 times). The results are the following (figure not shown). DEMS-F and DEMS-S achieve similar download time that is 14% to 27% lower compared to MinRTT, respectively. However the reinjected data incurred by DEMS-F is much smaller – only 15% to 53% compared to DEMS-S. Also DEMS-B's download time is 12% worse compared to DEMS-F due to the reinjection performed by DEMS-F, which is particularly useful when it is difficult to accurately predict the network condition.

**Varying Bandwidth.** We take a "record and replay" approach to realistically emulate the varying bandwidth. We collect 5-minute bandwidth traces of both paths from two campus locations. The bandwidth at both locations is highly variable, with their *stddev* being up to 54% and 92% of the mean bandwidth for WiFi and LTE, respectively. Thefi rst location has overall lower bandwidth compared to the second. We then replay (emulate) the two bandwidth traces in our lab while maintaining stable baseline RTT. The results are plotted in Figure 14. Compared to MinRTT, DEMS-F reduces the median download time by 12% to 46% for the two bandwidth profiles, as shown in subplot (a) and (c). DEMS-S only marginally outperforms DEMS-F, at the cost of reinjecting much more data (1.9x to 7.1x) as shown in subplot (b) and (d). Also both DEMS-F and DEMS-S yield better results than DEMS-B, again because reinjection helps absorb the uncertainty of the varying network conditions.

## 7.4 Field Test under Real-World Settings

We now performfi eld test to assess DEMS under real-world settings. We went tofi ve locations: residential apartment, office, campus library, parking lot, and grocery store to conduct experiments of downloadingfi les of different size (256KB, 1MB, and 4MB). For eachfi le size, we used the Nexus 6p smartphone to repeatedly performfi le download using DEMS-F and MinRTT back to back for 5 minutes over multipath (cellular network provided by a large U.S. carrier and commercial WiFi). The results are shown in Figure 15, which indicates that the network conditions at thefi ve locations are indeed very diverse. Overall we obtained encouraging results: compared to MinRTT, DEMS-F reduces the download time by up to 88%, 83%, and 77% for 256KB, 1MB and 4MB download, respectively; the median download time reduction is 33%, 48%, and 42%, respectively. The download time variance is also reduced by DEMS-F.

The above improvements are attributed to the different path characteristics between WiFi and cellular networks that are not handled well by MPTCP. Compared to wired networks, wireless networks like WiFi and cellular can sometimes exhibit high delay and bandwidth dynamics [22, 23] that pose challenges to multipath schedulers. To illustrate this, Figure 16 shows the downlink throughput of each subflow calculated every 200ms when downloading 4MBfi les. As shown, at many locations, WiFi and LTE networks exhibit different and highly variable bandwidth that can affect their OWD and the OWD prediction accuracy. As shown in Figure 17, oftentimes the relative OWD of WiFi and LTE are indeed highly variable, and largerfi le downloads have even higher variabilities. Thefl uctuation of bandwidth and delay makes it hard to accurately predict OWD. For example, when downloading 4MB files at the *Residence* location, the relative OWD of LTE ranges from 100ms to 1.3s, causing its prediction error to reach up to 800ms. DEMS does not fully rely on such predictions and instead employs adaptive reinjection described in §4.4 to more robustly handle the performance variability of many wireless network settings.

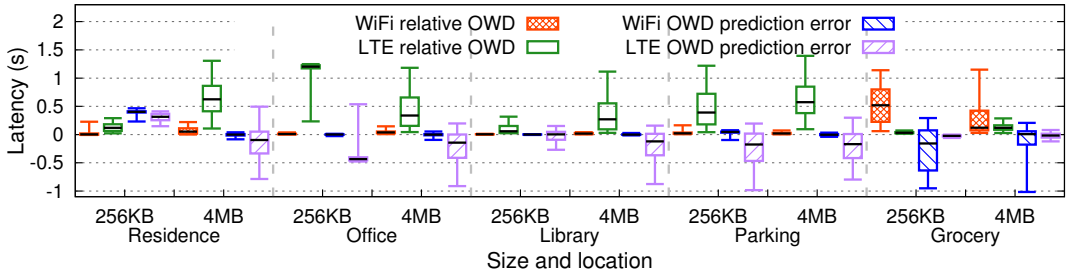
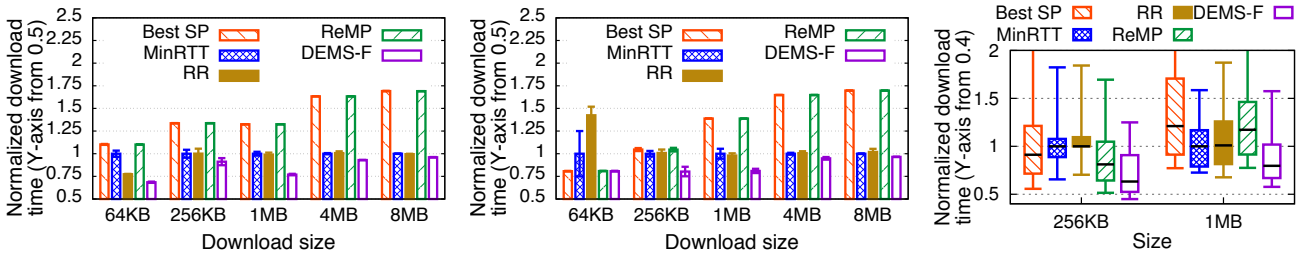



**Figure 17: Relative OWD and prediction error of real WiFi and LTE networks atfi ve locations.**




(a) Stable condition: WiFi 50ms, LTE 70ms (b) Stable condition: WiFi 50ms, LTE 270ms (c) Replay Bandwidth Profile 1

**Figure 18: Compare performance among DEMS-F and four other schedulers (laptop, emulation).**

## 7.5 Compare with Other Schedulers

Besides studying MinRTT, We further compare DEMS-F with three other scheduling approaches: round-robin, ReMP [13], and using the best single path to download afi le. Figure 18(a) and (b) compare the file download time using thefi ve schemes for two emulated stable network conditions. Figure 18(c) conducts a similar comparison under varying network condition (replaying Bandwidth Profile 1 collected in §7.3).

We observe that DEMS-F outperforms all other four schedulers. Besides this, in most cases, MinRTT achieves similar performance compared to round-robin (RR). This is likely explained by two reasons. First, in our implementation, the round-robin selection starts with the low-RTT path; starting from the high-RTT path will inflate the download time for smallfi les. Second, since we are performing bulk data transfer, it is unlikely that *both* paths have empty congestion window space at the same time. So during most of the time both MinRTT and RR have only one choice for path selection. Figure 18 also indicates the apparent limitation of the best single-path approach: compared to using multipath, using only one path significantly inflates the download time due to insufficient bandwidth. Regarding ReMP, it blindly duplicates *every* byte onto all subflows, making it essentially fall back to the "online" version of the best single-path approach. While ReMP and DEMS-F both transmit redundant data, DEMS-F's reinjection strategy is much more adaptive and strategic. For a 4MBfi le transfer, DEMS-F only transmits 2% of the redundant bytes sent by ReMP.

## 7.6 Web Browsing Performance

All experiments so far usefi le (raw data chunk) download as the workload. We now investigate how DEMS helps improve the QoE of web browsing, one of the most popular applications on mobile devices. A typical web page may consist of tens of objects, and downloading them faster would improve the QoE. We conduct web page loading experiments using off-the-shelf Chrome browser (53.0.2785.124) on our Nexus 6p smartphone. We picked 10 popular websites and use their mobile-version landing pages as the target web pages. The 10 websites cover diverse categories including news, education, travel, shopping, and government. We use the page load time (PLT), which is programmatically measured by the Chrome debugging interface, as the QoE metric. To make the experiments reproducible, we use Google Page Replay [3] to take a snapshot of each landing page and host it on our web server that is near the multipath proxy (§7.1). We compare the PLT of the 10 landing pages loaded by DEMS-F and by MinRTT under real-world network condition. The experiments were performed at a location on our campus where the RTTs and bandwidths of both WiFi and LTE experience high variance, similar to the network condition described in §7.3. We load each web page for 10 times and report the median PLT. As shown in Figure 19, compared to MinRTT, DEMS brings the per-page median PLT reduction of 6% to 43% across the 10 pages (median: 25%). The results indicate that DEMS can significantly improve the web QoE under real-world settings.

To understand why DEMS helps reduce the PLT, we take a closer took at the web object transmission pattern. Figure 20 exemplifies two waterfall diagrams for the same webpage when MinRTT and DEMS-F are used as the multipath scheduler. As shown, DEMS-F effectively reduces the reception duration (*i.e.,* download time) for many objects. Since many of them are on the critical path for web page loading, the overall PLT is effectively reduced. Also note that one key difference betweenfi le download and web page loading is that for the latter, computation and network activities are interleaved. Therefore, although the entire web page may be large, the web server has to intermittently feed data to the transport layer.

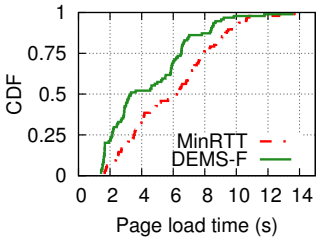
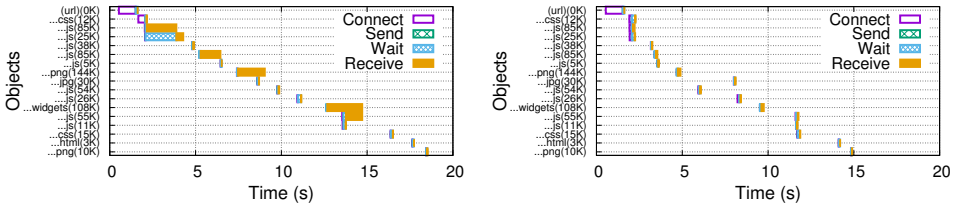



**Figure 19: Compare web page load time when using DEMS-F and MinRTT (smartphone, real WiFi/LTE).**




(a) MinRTT (b) DEMS-F

**Figure 20: Example waterfall diagrams for MinRTT and DEMS-F (some objects are omitted for better illustration).**

This naturally forms many small/medium-sized chunks that can be well optimized by DEMS.

Recall in §5 that an application can interact with DEMS either using or not using a hint API. Since it is challenging to modify the Chrome browser source code, we adopt the non-API mode in our experiments. In reality, since an HTTP(S) persistent connection may carry multiple web objects back to back, we thus envision that using the hint API to inform DEMS of the boundary between objects can provide additional benefits of minimizing the delivery time of each object. We leave this as future work[4].

## 7.7 System Overhead

We compared CPU utilization across the three schedulers: DEMS-F, MinRTT, and ReMP by running them on the smartphone. The workload is to upload a largefi le as fast as possible, with the aggregated bandwidth being around 50Mbps. Compared to the other two simpler schedulers, the additional CPU overhead incurred by DEMS-F is unnoticeable. For download traffic, since the mobile device acts as a receiver, the overhead is even more negligible.

# 8 DISCUSSIONS

**Integration with MPTCP.** While we implement DEMS on our own multipath TCP proxy infrastructure, DEMS can also be directly integrated into MPTCP. Since MPTCP provides a modular scheduler framework [30], the core logic of DEMS can be implemented as a new scheduler module. To facilitate OWD measurement (§5), the receiver side can leverage MPTCP options in TCP headers to carry necessary information used for estimating OWD at the sender side. The network prediction functionality can be implemented as a general API in MPTCP for all schedulers to use.

**Applicability.** DEMS can work with diverse traffic patterns and transfer sizes despite the ideal usage scenario being chunked data transfer, a very common workload as mentioned in §1. As DEMS shows the most promise for small-to-medium-size downloads, it can benefit a wide range of today's mobile applications (*e.g.,* video streaming) that involve downloading chunks from a few hundreds KBs to a few MBs. For very largefi les (*e.g.,* tens of MBs), when the network bandwidth is limited (a few Mbps), the subflow completion time difference is dwarfed by the long download duration, leading to overall small relative improvement brought by DEMS. Nevertheless, we do expect for future faster networks such as 5G with bandwidth of hundreds of Mbps, DEMS will significantly benefit large transfers.

**Limitation.** We discuss some limitations of DEMS. First, our current design focuses on two subflows. This was driven by realistically the most common mobile multipath use cases today. Nevertheless, the core concepts of DEMS (*e.g.,* chunk-based transfer, simultaneousfl ow completion, and adaptive reinjection) are also applicable to more than two paths, though involving more complexities. For example, instead of using the "two-way" splitting, each subflow can transmit packets in an interleaved (but still decoupled) manner. To achieve simultaneous subflow completion, a subflow can decide to stop transmission if the remaining data in the meta buffer can be delivered earlier by some other subflows with smaller OWD (some coordination algorithms need to be developed). In this way, at the sender side, the subflow with the largest and the smallest OWD will be thefi rst and the lastfi nishing subflow, respectively. This can compensate the delay differences among the subflows, leading to simultaneous subflow completion at the receiver side. We leave fleshing out the details on this to future work.

Second, the current DEMS design focuses primarily on improving download time, a critical factor impacting user QoE. In the future, we plan to explore additional dimensions like energy and limiting data usage on a particular subflow. This could possibly be done by adopting concepts from other special-purpose schedulers such as energy-aware [36] and subflow-priority-aware [17] schedulers.

# 9 CONCLUSION

Compared to single-path, multipath transport brings more complexities due to not only more involved paths but also their sophisticated interactions. Through judicious algorithm design, system integration, and extensive evaluation, we demonstrate that by strategically scheduling the packets, we can improve the multipath performance significantly (*e.g.,* median download time reduction of 33%–48% for fetchingfi les and median loading time reduction of 6%–43% for fetching web pages under real-world settings compared to MinRTT). In our future work, we plan to extend DEMS in several aspects (§8).


# ACKNOWLEDGEMENTS

We would like to thank our shepherd, Professor Prasun Sinha, and the anonymous reviewers for their valuable comments and suggestions. This research was supported in part by the National Science Foundation under grants CCF-1438996, CCF-1629347, CNS-1345226, and CNS-1566331.


[4] In HTTP/2, objects might be multiplexed together. To handle this, multiplexed objects can be treated as one chunk.

## REFERENCES


[1] 2015. In Korean, Multipath TCP is pronounced GIGA Path. http://blog.multipath-tcp.org/blog/html/2015/07/24/korea.html. (2015).

[2] 2016. Samsung Download Booster. http://www.samsung.com/uk/support/skp/faq/1061358. (2016).

[3] 2017. Google Web Page Replay Tool. https://github.com/chromium/web-page-replay. (2017).

[4] 2017. iOS: Multipath TCP Support in iOS 7. https://support.apple.com/en-us/HT201373. (2017).

[5] Maciej Bednarek, Guillermo Barrenetxea, Mirja Kühlewind, and Brian Trammell. 2016. Multipath Bonding at Layer 3. In *Proceedings of the 2016 Applied Networking Research Workshop*.

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

[7] Yung-Chih Chen, Don Towsley, and Ramin Khalili. 2014. MSPlayer: Multi-Source and multi-Path LeverAged YoutubER. In *ACM CoNEXT 2014*.

[8] Quentin De Coninck, Matthieu Baerts, Benjamin Hesmans, and Olivier Bonaventure. 2016. A First Analysis of Multipath TCP on Smartphones. In *PAM 2016*. Springer.

[9] Xavier Corbillon, Ramon Aparicio-Pardo, Nicolas Kuhn, Géraldine Texier, and Gwendal Simon. 2016. Cross-layer Scheduler for Video Streaming over MPTCP. In *ACM MMSys 2016*.

[10] Andrei Croitoru, Dragoş Niculescu, and Costin Raiciu. 2015. Towards WiFi Mobility without Fast Handover. In *USENIX NSDI 2015*.

[11] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. 2014. WiFi, LTE, or Both? Measuring Multi-homed Wireless Internet Performance. In *ACM IMC 2014*.

[12] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. 2013. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824. (2013).

[13] Alexander Frömmgen, Tobias Erbshauser, Alejandro P. Buchmann, Torsten Zimmermann, and Klaus Wehrle. 2016. ReMP TCP: Low Latency Multipath TCP. In *IEEE ICC 2016*.

[14] Yihua Guo, Feng Qian, Qi Alfred Chen, Zhuoqing Morley Mao, and Subhabrata Sen. 2016. Understanding On-device Bufferbloat for Cellular Upload. In *ACM IMC 2016*.

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

[16] Bo Han, Feng Qian, and Lusheng Ji. 2016. When Should We Surf the Mobile Web Using Both Wifi and Cellular?. In *ACM All Things Cellular Workshop 2016*.

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

[18] Benjamin Hesmans and Olivier Bonaventure. 2016. An Enhanced Socket API for Multipath TCP. In *Proceedings of the 2016 Applied Networking Research Workshop*.

[19] Benjamin Hesmans, Gregory Detal, Raphaël Bauduin, Olivier Bonaventure, et al. 2015. SMAPP: Towards Smart Multipath TCP-enabled APPlications. In *ACM CoNEXT 2015*.

[20] Brett D Higgins, Kyungmin Lee, Jason Flinn, Thomas J Giuli, Brian Noble, and Christopher Peplin. 2014. The Future is Cloudy: Reflecting Prediction Error in Mobile Applications. In *MobiCASE 2014*.

[21] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *ACM MobiSys 2012*.

[22] Junxian Huang, Feng Qian, Yihua Guo, Yuanyuan Zhou, Qiang Xu, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2013. An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance. In *ACM SIGCOMM 2013*.

[23] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. 2012. Tackling Bufferbloat in 3G/4G Networks. In *ACM IMC 2012*.

[24] Ramin Khalili, Nicolas Gast, Miroslav Popovic, Utkarsh Upadhyay, and Jean-Yves Le Boudec. 2012. MPTCP is Not Pareto-optimal: Performance Issues and a Possible Solution. In *ACM CoNEXT 2012*.

[25] Nicolas Kuhn, Emmanuel Lochin, Ahlem Mifdaoui, Golam Sarwar, Olivier Mehani, and Roksana Boreli. 2014. DAPS: Intelligent Delay-aware Packet Scheduling for Multipath Transport. In *IEEE ICC 2014*.

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

[27] Catalin Nicutar, Dragos Niculescu, and Costin Raiciu. 2014. Using Cooperation for Low Power Low Latency Cellular Connectivity. In *ACM CoNEXT 2014*.

[28] Ana Nika, Yibo Zhu, Ning Ding, Abhilash Jindal, Y. Charlie Hu, Xia Zhou, Ben Y. Zhao, and Haitao Zheng. 2015. Energy and Performance of Smartphone Radio Bundling in Outdoor Environments. In *Proceedings of the 24th International Conference on World Wide Web*.

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

[30] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. 2014. Experimental Evaluation of Multipath TCP Schedulers. In *ACM SIGCOMM Capacity Sharing Workshop (CSWS) 2014*.

[31] Bahar Partov and Douglas J Leith. 2016. Experimental Evaluation of Multi-path Schedulers for LTE/wifi Devices. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*.

[32] Qiuyu Peng, Minghua Chen, Anwar Walid, and Steven Low. 2014. Energy Efficient Multipath TCP for Mobile Devices. In *ACM MobiHoc 2014*.

[33] Feng Qian, Vijay Gopalakrishnan, Emir Halepovic, Subhabrata Sen, and Oliver Spatscheck. 2015. TM3: Flexible Transport-layer Multi-pipe Multiplexing Middlebox without Head-of-line Blocking. In *ACM CoNEXT 2015*.

[34] Varun Singh, Saba Ahsan, and Jörg Ott. 2013. MPRTP: Multipath Considerations for Real-time Media. In *ACM MMSys 2013*.

[35] Joel Sommers and Paul Barford. 2012. Cell vs. WiFi: On the Performance of Metro Area Mobile Connections. In *ACM IMC 2012*.

[36] Yeon sup Lim, Yung-Chih Chen, Erich M. Nahum, Don Towsley, Richard J. Gibbens, and Emmanuel Cecchet. 2015. Design, Implementation and Evaluation of Energy-Aware Multi-Path TCP. In *ACM CoNEXT 2015*.

[37] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. 2011. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *USENIX NSDI 2012*.

[38] Jiyan Wu, Chau Yuen, Bo Cheng, Ming Wang, and Jun-Liang Chen. 2016. Streaming High-quality Mobile Video with Multipath TCP in Heterogeneous Wireless Networks. *IEEE Transactions on Mobile Computing* 15, 9 (2016), 2345–2361.

[39] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. 2015. Adaptive Congestion Control for Unpredictable Cellular Networks. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. 509–522.