MP-H2: A Client-only Multipath Solution for HTTP/2

Ashkan Nikravesh[★] Yihua Guo[†] Xiao Zhu[★] Feng Qian[‡] Z. Morley Mao[★]

[★]University of Michigan – Ann Arbor [†]Uber Technologies, Inc. [‡]University of Minnesota – Twin Cities

ABSTRACT

MP-H2 is a client-only, HTTP-based multipath solution. It enables an HTTP client to fetch content (an HTTP object) over multiple network paths such as WiFi and cellular on smartphones. Compared to MPTCP, MP-H2 offers several key advantages including server transparency, middlebox compatibility, and friendliness to CDN, anycast, and load balancing. MP-H2 strategically splits the file into byte range requests sent over multipath, and dynamically balances the workload across all paths. Furthermore, MP-H2 leverages new features in HTTP/2 including stream multiplexing, flow control, and application-layer PING to boost the performance. MP-H2 also supports multi-homing where each path contacts a different CDN server for enhanced performance. Evaluations show that MP-H2 offers only slightly degraded performance (6% on average) while being much easier to deploy compared to MPTCP. Compared to other state-of-the-art HTTP multipath solutions, MP-H2 reduces the file download time by up to 47%, and increases the DASH video streaming bitrate by up to 44%.

ACM Reference Format:

Ashkan Nikravesh, Yihua Guo, Xiao Zhu, Feng Qian, and Z. Morley Mao. 2019. MP-H2: A Client-only Multipath Solution for HTTP/2. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19), October 21–25, 2019, Los Cabos, Mexico.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3300061.3300131

1 INTRODUCTION

Multipath transport enables mobile applications to utilize multiple network paths (interfaces) simultaneously to transfer data, leading to significant improvement of performance in terms of throughput, latency, or reliability [21, 24, 29, 31, 33, 39, 44]. Multipath TCP (MPTCP [47]) is the state-of-the-art solution of multipath transport. It is attractive in several aspects such as application transparency, mobile-friendliness, and good performance. However, there are several major hurdles that considerably slow down the deployment of MPTCP. First, MPTCP requires changes on both the client and the server side at the OS kernel level. Second, many middleboxes are not compatible with MPTCP in that they strip MPTCP options from TCP packets [44]. Third, MPTCP is not friendly to many other critical protocols and infrastructures such as Content Distribution Networks (CDN), anycast, and flow-level load balancing. For example, today's DNS-based CDN server selection is not aware of MPTCP; as a result, the selected CDN server is only a good choice for the primary subflow and may incur poor performance for other (secondary) subflows. All above issues among possibly other limitations made MPTCP's deployment extremely slow (§2.1).

This paper presents the design and implementation of MP-H2, a client-only multipath solution for HTTP. MP-H2 enables an HTTP client to fetch content (an HTTP object) over multiple network paths such as WiFi and cellular. Our primary target workload is to fetch a single object with a medium-to-large size such as a Dropbox file, a video chunk, an MP3 song, and an image. MP-H2 strategically splits the object over multipath to minimize the transfer time. We note that there is another type of workload: fetching many small objects when loading a web page. For this workload, a recommended approach for improving the network efficiency over both single [30, 50] and multipath [32] is to merge the small objects into bundles before transmitting them. In this context, the concept of MP-H2 can still be applied to transferring a bundle, but the detailed realization is beyond the scope of this paper.

Compared to MPTCP, MP-H2 provides similar performance while offering several key advantages, thus significantly lowering the bar for its deployment. As a client-side HTTP library extension, MP-H2 brings *no* change to the web server and *minimum* user-level changes to the client; MP-H2 does not change the transport layer (*e.g.,* not using any TCP option), making it fully compatible to today's middleboxes; furthermore, MP-H2 is friendly to CDN, anycast, and load balancing, because subflows in MP-H2 are fully decoupled as regular transport channels. In particular, due to its fully decoupled subflows, MP-H2 can support *multi-homing* where

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6169-9/19/10...$15.00
https://doi.org/10.1145/3300061.3300131

its subflows contact different CDN servers, each selected by the Local DNS server (LDNS) on the corresponding path.

Among a wide range of application protocols, we pick HTTP on which MP-H2 is built, due to the dominance of HTTP(S) on today's Internet. The underlying idea of MP-H2 is straightforward: the client sends multiple byte-range requests to fetch different chunks of the file over multiple paths, and reassembles the received chunks before delivering the file to the web client. HTTP byte range requests have been a standard HTTP feature since the 1990s, and are widely supported for fetching large objects (§3.1). The key technical contribution we made is a client-side scheduler that determines *when to fetch which chunk (in terms of their byte ranges) over which path*, so that the overall download time can be minimized. Designing such a scheduler is challenging mainly due to two reasons: (1) the scheduler resides on the client side, slowing down MP-H2's reaction to changing network conditions; (2) sending HTTP requests incurs additional latency that also lengthens the download time.

To address the above challenges, MP-H2 strategically performs the chunk (byte range) split and schedules their HTTP requests. Chunk split is performed in an online manner based on network condition estimation over all paths. Ideally this leads to all path completing their chunks simultaneously – a necessary condition for optimal download time. In cases where one path completes its assigned chunks earlier than other paths due to imperfect network condition estimation, some of the slower paths' chunks will be dynamically offloaded to the fast path to balance the workload among all paths. Also very importantly, we find that HTTP/2, the latest HTTP standard [19] (also QUIC [14]), provides a set of new features including stream multiplexing, flow control, and application-layer PING that can be leveraged by MP-H2 to significantly improve the performance. We leverage these new HTTP/2 features to realize two core components in MP-H2: pipelining HTTP requests and promptly modifying the byte range of an on-going HTTP transaction, both helping improve the network bandwidth utilization for MP-H2.

We implemented MP-H2 on commodity Android/Linux client devices in ∼2,000 LoC. We have successfully applied MP-H2 to fetch contents from real content providers such as Dropbox and commercial video platforms by changing only 5 to 15 lines of *client-side* application code for invoking the MP-H2 library. We conduct extensive evaluations of MP-H2 over real WiFi and cellular networks by comparing it with MPTCP, mHTTP [37], and MSPlayer [22], with the latter two being state-of-the-art HTTP-based multipath schedulers. Our key evaluation results include the following.

- For downloading a medium-sized HTTP object, under stable network conditions, MP-H2 is 3% to 26% faster than mHTTP [37]. Meanwhile, MP-H2 provides comparable performance to MPTCP (only 0.1% to 11% slower than MPTCP for 1MB+ files). Under variable network conditions, MP-H2 brings a median download time reduction of up to 26% compared to mHTTP and its performance is only 6% slower than MPTCP on average. Under high bandwidth, MP-H2 outperforms mHTTP by 42% to 47% while being only 0.2% to 7% slower than MPTCP. Note that MP-H2 is slightly slower than MPTCP because of MPTCP's server-side scheduling and precise byte-level control at TCP layer – between MP-H2 and MPTCP there is an inherent tradeoff between performance and intrusiveness. We believe MP-H2 strikes a good balance by providing an *easy-to-deploy alternative* to MPTCP.
- We apply MP-H2 to DASH video streaming and compare it with MPTCP and MSPlayer, an HTTP-based multipath scheduler designed specifically for video streaming. Under stable (highly variable) network conditions, MP-H2 can provide up to 44% (25%) higher average video bitrate than MSPlayer, and MP-H2's average bitrate is only 0.3% to 0.9% (2% to 8%) lower than that of MPTCP.
- Under multi-homing, MP-H2 can reduce the file download time by 4% to 19% compared to single-server scenarios.

Overall, we demonstrate that it is entirely feasible to develop a client-only, user-space only, and high-performance multipath solution over HTTP/2 by strategically leveraging its new features. Our solution is readily deployable as a regular browser or HTTP library upgrade.

2 MOTIVATION

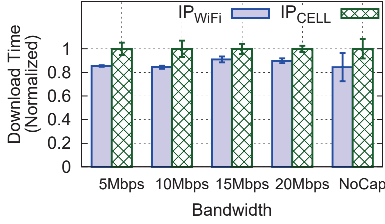
We begin by presenting arguments that motivate MP-H2.

2.1 MPTCP Adoption

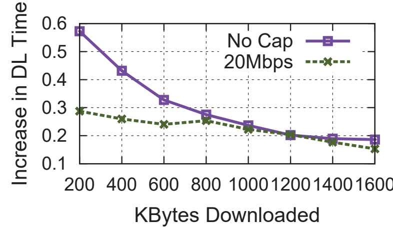
MPTCP can effectively improve the networking performance on mobile devices and data center networks [21, 31, 44, 47]. However, after five years since MPTCP was standardized by IETF, its adoption has been very slow. In 2015, Mehani *et al.* [42] scanned the Alexa top 1M websites and reported that less than 0.1% of the hosts on the Alexa list support MPTCP. Recently (in 2018), we repeated the same measurement on Alexa top 500 websites and observed that this number has not changed much since 2015. Only one out of 500 websites (0.2%) supports MPTCP.

Additionally, today's middleboxes are not friendly to MPTCP. As reported by [44], major U.S. cellular carriers strip MPTCP options from the web traffic. Thus, establishing MPTCP subflows on the cellular interface to the default HTTP/HTTPS ports are not possible. We repeated the same experiments in 2018, and observed that nothing has changed.

Furthermore, MPTCP requires modifications in the kernel stack of both the client and server side. It lacks a flexible interface for apps to enforce their policies [44], as all its logic resides in the kernel space. All these practical limitations may explain why the MPTCP adoption has been slow and motivate us to design MP-H2.



(a) Downloading a 1.6MB video chunk from IP_{WiFi} and IP_{CELL} over WiFi.



(b) Relative download time increase as we download first x bytes of a 1.6MB video chunk from IP_{CELL}, compared to downloading from IP_{WiFi}.

Figure 1: **Impact of server selection on downloading Netflix video chunks.**

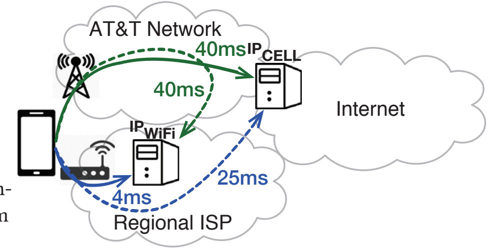


Figure 2: **Netflix server selection in MPTCP.**

2.2 CDN Server Selection

We investigate the performance of Content Delivery Network (CDN) server selection in the context of multipath.

To accelerate the delivery of web contents (*e.g.*, a web page or a video chunk) with global reach to the users, CDN replicates the content across servers at multiple geographically distributed locations. Then a user's request will be directed to a server which is usually close to the user, in order to deliver the best user experience. Most CDNs rely on DNS to select the best server and to balance the load on their servers.

DNS-based server selection can still work in multipath, but it may not be optimal. When fetching a URL using MPTCP, the DNS request to resolve the domain is only sent from the interface of the *primary* path (as opposed to secondary paths). The CDN's authoritative DNS server is thus not aware of the existence of other paths. As a result, the selected CDN server may not be a good choice with respect to the non-primary paths, leading to potentially suboptimal performance.

2.2.1 Case Study: Netflix. Modern CDNs deploy their servers in strategic locations. For instance, Netflix, which accounts for more than 37% of all downstream Internet traffic in North America [15], has its own CDN (Open Connect [11]). It partners with mainstream U.S. ISPs and deploys its server infrastructures inside the ISPs' access networks, or peers directly with ISPs at one of their peering exchange points. When these content servers are located inside an ISP, the multipath performance might become suboptimal: the traffic of other subflows will be "detoured" into the primary interface's access network, making the latency of other subflows increase. We will shortly explain this in detail.

Figure 2 shows a real example (as measured by us) of Netflix deployment for a regional broadband ISP and a cellular carrier (AT&T) in the U.S. A smartphone on our campus is connected to the campus network through WiFi and to AT&T network through cellular. To find where these content servers are located, we use the information included in the reverse domain names of the IP address that video chunks are fetched from [20]. As shown, over WiFi, users' traffic is directed to a content server inside the regional ISP (*i.e.*, IP_{WiFi}) and over cellular, the traffic is directed to a content server in an internet exchange point (*i.e.*, IP_{CELL}). The RTT of our client to IP_{WiFi} is measured to be around 4ms over WiFi and around 40ms over cellular. Similarly, the RTT to IP_{CELL} is around 25ms over WiFi and around 40ms over cellular.

Next, we use the above setup to study the impact of CDN server selection strategy on the network performance over multipath. We consider two approaches: (1) using one server for all paths, and (2) MP-H2's multi-homing approach where the subflows contact different CDN servers.

A Single CDN Server for All Paths. We use the following example to facilitate our discussion. Consider a video streaming session. At the beginning, only cellular (single path) is available so the DNS lookup returns IP_{CELL}, which is then cached by the browser. In the middle of the video playback, WiFi becomes available due to, for example, the user's mobility, allowing the browser to use MPTCP. Note that the cellular is always available so there is no handover. If the browser continues using the cached IP address IP_{CELL}, the multipath performance in terms of the RTT is suboptimal. In the above example, contacting IP_{CELL} leads to 5.25x RTT inflation over WiFi, compared to using IP_{WiFi} (25 vs. 4ms).

To understand how much this extra latency affects the download time of video chunks, we fetch a 1.6MB data chunk from IP_{CELL} and IP_{WiFi}. Figure 1 compares the (single path) download time of fetching the file from these two servers over WiFi under different bandwidth values (capped using Linux `tc`). As shown in Figure 1(a), the extra 21ms latency from the client to IP_{CELL} increases the download time by 10% to 18%. Figure 1(b) shows the time increase for fetching a small fraction of the 1.6MB file. The download time inflation can reach up to 57% for the first 200KB of data.

When multiple paths are available, one may argue for the following approach: performing DNS lookups over *all* paths and pick the best server IP that yields an overall highest "utility" to serve all paths. However this approach may still under-utilize the overall system capacity, because a single CDN server may not provide optimal performance for *all* paths.

Multi-homing. We instead advocate a multi-homing multipath approach: on each path, the client performs a *separate* DNS lookup and fetches the content from the selected CDN server. In other words, the servers selected over different paths might be different. In this way, the client can contact the optimal server over each path, leading to full utilization of each path's capacity (assuming the CDN selection algorithm works properly). In the aforementioned example, this approach leads to using IP_{WiFi} for WiFi and using either IP_{WiFi} or IP_{CELL} over cellular.

Since the original Netflix servers do not support MPTCP, we next conduct emulations to quantify the benefits of the multi-homing scheme. We set up an Apache web server with MPTCP support inside our campus network, which is almost identical to IP_{WiFi} in terms of RTT. To emulate WiFi accessing IP_{CELL}, we use the same server and add an extra 21ms latency to the WiFi traffic using `tc`, to approximate the RTT to IP_{CELL} over WiFi (Figure 2). Over cellular, the latency to IP_{WiFi} and IP_{CELL} is adjusted similarly if needed. Given the above setup, we conduct emulations by downloading files of two sizes (512KB and 1MB) using off-the-shelf MPTCP under two schemes: the solid curves in Figure 3 emulate the multi-homing scenario, and the dotted curves correspond to the suboptimal path selection where the client selects IP_{CELL} over IP_{WiFi} for the WiFi subflow. By comparing them, we see that for the 512KB and 1MB file, the latter scheme increases the average download time by 31% and 9%, respectively. Note that with the recent advances in the access network technology (*e.g.,* 5G), the performance bottleneck is shifting from the last mile to the wide area network, making CDN selection even more important.

2.2.2 Crowd-sourced Measurement Study. To understand how prevalent the above suboptimal server selection is across today's CDNs and ISPs, we conduct an IRB-approved crowd-sourced measurement study using a custom measurement app we released on Google Play. To minimize its intrusiveness, our measurement is limited to measuring the latency.

In this study, we investigate six popular CDN providers: Google, LimeLight, Amazon CloudFront, Incapsula, Akamai, and Fastly. For each CDN provider, we find at least one popular domain that it hosts, by loading top Alexa pages and examining the domain names of their embedded objects. Then, we schedule the following measurement task, which takes the domain names as input, to run every hour on users' Android devices. For each domain name, a device first performs DNS lookups of the domain over WiFi and cellular to obtain the corresponding IP addresses as IP_{WiFi} and IP_{CELL}, respectively. If IP_{WiFi} and IP_{CELL} are different, the client will then measure the RTT difference between the paths toward IP_{WiFi} and IP_{CELL} over WiFi and then over cellular.

We collected the data for more than 5 months since September 2017. In total, 123K measurements are collected from 138 devices, covering 71 carriers across the world. We find that for 48% of the measurements, the domains are resolved to different IPs over WiFi and cellular. Across different CDNs, the percentage of DNS lookups that resolved to different IPs ranges from 21% to 88% (Incapsula: 21%, LimeLight: 35%, Fastly: 47%, CloudFront: 77%, Google: 87%, Akamai: 88%).

The latency measurement results are shown in Figure 4. The first (second) subplot measures the RTT increase over WiFi (cellular) when the client switches from IP_{WiFi} (IP_{CELL}) to IP_{CELL} (IP_{WiFi}) for server selection over the WiFi (cellular) path. As shown, over WiFi (Figure 4(a)), the median increase is between 10% to 37% across the CDN providers. The median increases for cellular (Figure 4(b)) are less than those for WiFi (6% to 17%), as the RTT over cellular is usually longer. Both figures also exhibit long tails.

2.3 Anycast and Load Balancing

MPTCP is not compatible with load balancers and anycast servers. In anycast, a single IP address is assigned to multiple hosts in different locations and this IP address is announced through BGP. When using MPTCP, the handshake traffic of primary and secondary subflows may be routed to different physical servers that advertise the same IP address. Since the server that is reached by the secondary subflow is not aware of the MPTCP connection and is not maintaining the connection state, the secondary subflow will not be established. In the case of load balancing, we have a similar situation, where the subflows with different 5-tuple, *i.e.,* protocol, src/dst address, src/dst port, may be assigned to different servers. This problem is recently addressed by [26], but the solution requires changes in the implementation of anycast and load balancers, as well as modifications in the OS kernel, making it difficult to be adopted.

2.4 Server Transparency and Flexible Transport Protocol Support

Adding multipath capabilities to transport protocols has shown to be an effective way to improve their performance and reliability. For instance, besides MPTCP, Multipath QUIC (MPQUIC) [23], which is a recent extension to the QUIC protocol [14], allows hosts to exchange data over multiple interfaces or networks over a single QUIC connection.

A main limitation of these multipath solutions is that adopting them requires modifying the server and client-side code. For MPTCP, it requires modifying the kernel on both sides; for MPQUIC, it also requires modifying the server-side code. Instead, MP-H2 takes a different approach by realizing the multipath logic in the application layer (client-side only) and on top of HTTP. Doing so has two benefits. First, HTTP is the dominating application protocol of today's Internet and it (including HTTP/2 and QUIC) can be realized upon different transport protocols (*e.g.,* TCP or UDP). Thus, an

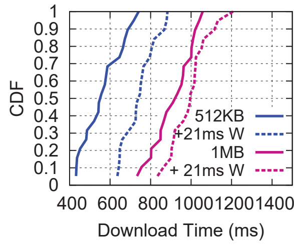
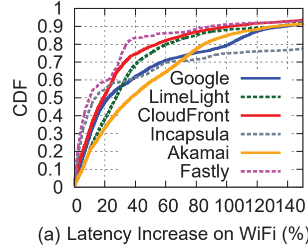


Figure 3: Impact of suboptimal server selection on MPTCP performance (emulation).



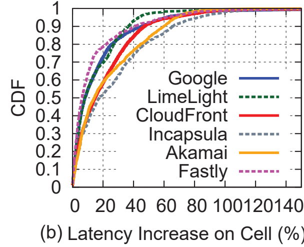


Figure 4: Crowd-sourced measurement of multipath server selection. (a) Latency increase of IP_{CELL} compared to IP_{WiFi} over WiFi. (b) Latency increase of IP_{WiFi} compared to IP_{CELL} over cellular.

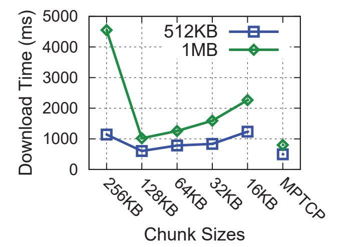


Figure 5: Multipath performance w/ fixed chunk size when fetching 512KB and 1MB files.

HTTP-based multipath solution can automatically work with any transport layer protocol and even their mixture (*e.g.,* one subflow over TCP, the other subflow over QUIC). Second, incorporating a new HTTP library with multipath support requires only small changes to the client-side application. It does not incur any change at lower layers (transport, IP, *etc.*) on the client side or any change to the server, thus significantly lowering the deployment bar.

3 DESIGN OF MP-H2

We now describe our design of MP-H2 in detail.

3.1 System Overview

MP-H2 is a client-side, HTTP-based multipath solution for fetching HTTP objects over multiple heterogeneous paths. We describe our design for two paths. We name them WiFi (the primary path) and cellular (the secondary path) just for good readability; MP-H2 does not impose constraints on the actual link types. Later we show how our design can be generalized to more than two paths in §3.3.

The main goal of MP-H2 is to provide a multipath solution that is easy-to-deploy (requiring minimal changes to the client and no server-side change). Furthermore, to provide flexibility in terms of supporting various transport layer protocols, we design MP-H2 at the HTTP layer. This allows changing the underlying transport protocols transparently. HTTP, which accounts for more than 88% of today's mobile Internet traffic [35], has two major versions: HTTP/1.1 [27] and HTTP/2 [19]. As its name suggests, MP-H2 is designed on top of HTTP/2 due to two reasons. First, HTTP/2 is getting increasingly popular. Second, HTTP/2 introduces new features such as multiplexing, flow control, and app-layer PING that provide perfect building blocks for MP-H2 (§3.4).

High-level idea. To simultaneously utilize WiFi and cellular paths to fetch an HTTP object, we can send separate HTTP byte-range requests over both paths to fetch different portions (*i.e., chunks*) of the file. Then, on the client side, we reassemble the received chunks before delivering the whole object to the web client. The key logic of MP-H2 that we detail in the remainder of this section is the client-side scheduler design, which dictates *when to fetch which chunk (in terms of their byte ranges) over which path*. Contrary to MPTCP, we assume a stateless, single-path server as our scheduling logic resides fully on the client side. In MP-H2, different paths are completely decoupled and independent from the transport-layer perspective. Therefore, different chunks can be fetched from different servers. In other words, MP-H2 naturally supports the "multi-homing" concept described in §2.2.1.

To fetch different portions of a file, MP-H2 relies on HTTP byte range requests, a built-in feature of HTTP (both HTTP/1.1 and HTTP/2) since the 1990s. All mainstream web servers support this feature. We also investigate the Alexa top 50 global sites that provide video contents, and found that all of them support byte range request for fetching video data. Also more than half of these sites use HTTP/2 or SPDY (an early version of HTTP/2).

Requirements and challenges. To minimize the download time, we have to fully utilize the bandwidth provided by both paths. This translates to two requirements:

- **R1**: both paths need to complete their transfers simultaneously (as measured from the receiver side). Otherwise we can derive an even better download schedule by moving some bytes from the slow path to the fast path.
- **R2**: over each path, the bandwidth needs to be fully utilized. Chunks need to be consecutively downloaded without network idle periods between them.

Compared with MPTCP, placing the scheduler on the client side complicates the design, making achieving the above two goals challenging. First, it is difficult to achieve **R1**, because of the potential fluctuating network condition on both downlink and uplink, which affects both the data plane and the control-plane communication between the scheduler and the sender. In particular, the control-plane communication, which slows down MP-H2's reaction to changing network conditions, does not exist in MPTCP. Second, achieving **R2** is trivial in MPTCP but challenging in MP-H2, because sending HTTP requests incurs idle periods on downlink.

There are other unique challenges that MP-H2 faces. First, we will describe later in §3.4 that complex cross-layer interactions between the transport layer and HTTP complicate our design. Second, typically MP-H2 has no prior knowledge of the object size, which is a critical piece of information that the scheduler needs to craft byte range requests.

3.2 Suboptimal Alternative Designs

We begin with a simple design: split the file into *two equal-sized* chunks, each then being fetched over one path using an HTTP byte-range request. This clearly does not satisfy **R1** unless both paths have the same bandwidth and latency.

To better account for the heterogeneity among the paths, another approach would be to split the file into *smaller* and fixed-sized chunks (*e.g.,* 200KB each) and fetch the chunks sequentially on each path [37]. With smaller chunk sizes, more chunks will be downloaded over the fast path, allowing us to get closer to **R1**. However, this also means more HTTP requests, which incur longer idle periods (at least one RTT per request), thus leading to **R2** not being satisfied.

We implemented the second design. Figure 5 shows how download time changes with smaller chunk sizes, when a client downloads a 512KB and a 1MB file over cellular (10Mbps) and WiFi (5Mbps), using two concurrent persistent HTTP sessions, one over each path. For both files, the download time initially decreases as we reduce the chunk size from 256KB to 128KB (improving on **R1**). But with smaller chunk sizes, the download time increases due to the overhead of HTTP requests (degrading on **R2**). Note that the optimal chunk size differs over different network conditions and file sizes, making this approach even less attractive.

3.3 The Basic Design of MP-H2

We now detail the design of MP-H2's client-side scheduler. We begin with an ideal scenario where both the file size and network condition of both paths are perfectly known. In this case, we only need to divide the file into two chunks, each transferred over one path, of different sizes. The chunk sizes can be calculated by solving the equation of $T_i = T_j$, where T_i is the time taken to download the chunk with size D_i on path i. Since the scheduler has to send an HTTP request before receiving each chunk, we have $T_i = RTT_i + \frac{D_i}{BW_i}$, where BW_i and RTT_i are the bandwidth and RTT of path i, respectively. Then by letting $T_i = T_j$ (j is the other path), we calculate the chunk size D_i as:

$$D_i = \frac{D \cdot BW_i + BW_i BW_j (RTT_j - RTT_i)}{BW_i + BW_j} \tag{1}$$

where $D = D_i + D_j$ is the file size. Note that since we split the file into D_i and D_j, we only need to send one HTTP request over each path. One can show that this fetching scheme satisfies both **R1** and **R2** (proof omitted).

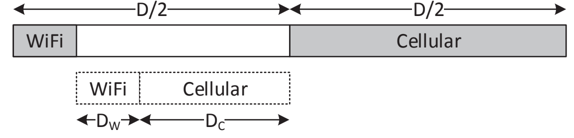
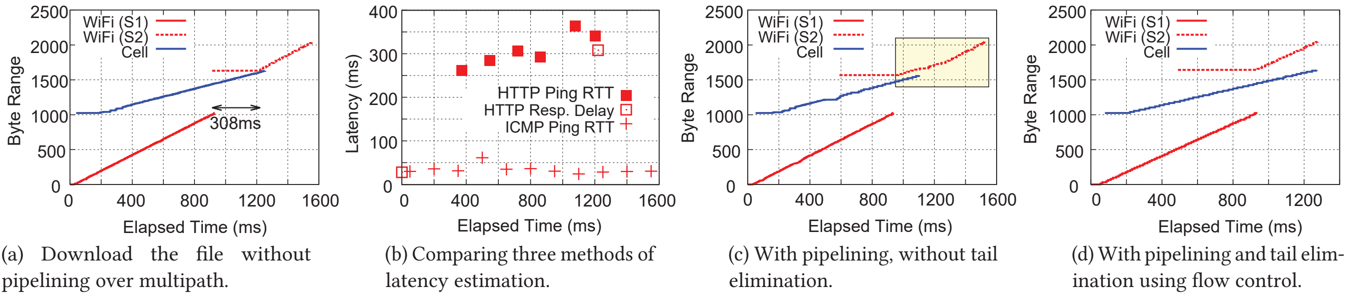


Figure 6: **Two iterations of the byte range re-adjustment. The gray area shows the downloaded parts of the data chunks.**

The above scheme is ideal in that the client needs to know in advance both the file size and the network conditions of both paths. Now let us relax these requirements.

Regarding the network condition information, MP-H2 predicts both paths' bandwidth and RTT by computing the moving average of the bandwidth and latency samples collected passively. One challenge here is the *bootstrapping* phase: at the very beginning of the file transfer, MP-H2 has no knowledge of either path's network condition so it cannot use Equation 1 to split the file. To address this issue, MP-H2 initially performs an even split *i.e.,* $D_i = D_j = D/2$, allowing both paths to start accumulating samples of network condition readings. Due to the heterogeneity of the paths, typically one path will finish earlier than the other. In this case, the *fast* path helps the *slow* path through downloading part of the data that was originally assigned to the slow path. Also, since at this moment we already have samples for network condition estimation, we can apply Equation 1 to update the byte ranges. This process is illustrated in Figure 6. As shown, initially each path is assigned to handle half of the file. Then assume the cellular path completes its portion earlier than WiFi does. Accordingly, MP-H2 uses Equation 1 to compute D_C and D_W, the updated bytes-to-fetch for cellular and WiFi, respectively. Subsequently, on the *fast* (cellular) path, MP-H2 sends a *new* byte-range request to download the D_C bytes assigned to that path; on the *slow* (WiFi) path, MP-H2 *re-adjusts its byte range end* to the number of bytes that have already been downloaded plus D_W, in order to avoid the overlap between the slow and fast paths. If the bandwidth and RTT predictions are accurate, the above process will complete in two iterations as illustrated in Figure 6. Otherwise, due to imperfect network condition estimation, some path will again complete sooner than the other path. This leads to another iteration involving re-computing the byte ranges, re-adjusting the byte range end for the slow path, and issuing a new request over the fast path. The procedure may be repeated for multiple iterations until the difference between the estimated completion time of the two paths is less than a threshold.

Now we describe how to deal with the missing file size information. A simple way for the client to learn the file size is to issue an HTTP HEAD request, which will be replied with the HTTP response header containing the `Content-Length` field. Its drawback is that it incurs one additional RTT. MP-H2 instead employs a more efficient approach as follows. The



(a) Download the file without pipelining over multipath.

(b) Comparing three methods of latency estimation.

(c) With pipelining, without tail elimination.

(d) With pipelining and tail elimination using flow control.

Figure 7: Performance of the MP-H2 scheduler for downloading a 2MB file. The WiFi and cellular bandwidth are 10Mbps and 5Mbps, respectively. The Y-axis shows the byte ranges transferred over different paths.

client initially requests for the *whole* file over the primary (WiFi) interface using a regular HTTP GET to obtain the file size as well as to start receiving the data. When the file size becomes available, MP-H2 issues a request for the second half of the file over the secondary (cellular) path. For what happens next, let us consider two cases. First, if WiFi is slower, then anyway WiFi's byte range end needs to be adjusted (recall that the slow path's byte range end needs to be adjusted) so we save one RTT compared to using HTTP HEAD. Second, if cellular is slower, then MP-H2 adjusts WiFi's byte range end to the midpoint of the file (where the cellular path initially starts) to prevent WiFi's overshoot. This is done in a "lazy" manner when the WiFi's progress hits the midpoint, and only takes place once in the first iteration.

We focus on two paths in the current design, as it is the most common mobile multipath use case today. Our design can be applied to more than two paths. For example, over three paths, the file is split into three chunks for each path to fetch. Once the file size and estimated network conditions are available, the scheduler can find the sizes of chunks to be downloaded over each path by solving the equations of $T_i = T_j$ and $T_j = T_k$, where k is the third path, and deriving a solution similar to Equation 1.

3.4 More Design Aspects of MP-H2

In §3.3, we left two key building blocks unrealized: issuing a new request and adjusting the HTTP byte range end. We describe their design in this section.

3.4.1 Issuing New Requests over the Fast Path. Recall that in each iteration, once the download of a chunk is completed on the fast path, the scheduler needs to send an HTTP request to fetch a new data chunk offloaded from the slow path. Sending a new request incurs a downlink idle period time of one RTT due to the HTTP request, causing inefficiencies for **R2**. For example, Figure 7(a) shows downloading a 2MB file on a smartphone connected to 10Mbps WiFi and 5Mbps LTE. The three curves in the figure correspond to the one chunk transferred over cellular (the slow path) and the two chunks (S1 and S2) transferred over WiFi (the fast path). When WiFi completes S1 at around 910ms, it sends a request for S2, and gets the response at 1220ms. As shown, the WiFi path wastes 308ms on waiting for S2's response. To address this issue, we take a *pipelining* approach, whose basic idea is to send the new request *one (application-layer) RTT before* the previous chunk download finishes (on the fast path). By doing so, the idle period incurred by requesting the next chunk is masked by the on-going data transfer of the previous chunk. If the RTT estimation is correct, the client can immediately start receiving the data on the fast path when the download of the previous chunk finishes, thus facilitating **R2**.

In HTTP/1.1, pipelining HTTP transactions has numerous restrictions and is virtually not used in practice [7, 8] so the client has to establish multiple TCP connections to support parallelism, incurring high overhead. Fortunately, the HTTP/2 protocol suite (including QUIC) provides inherent protocol support for intra-connection parallelism through *multiplexing* streams within the same connection. Therefore, in MP-H2, to send a new request, the fast path reuses its connection by creating a new stream associated with it.

Next, let us consider how to predict the application layer RTT, which, as mentioned above, corresponds to the time from sending a request to receiving the first byte of its response. The challenge here is to accurately sample them for prediction. We consider three possible approaches. First, performing direct sampling of the HTTP response delay suffers from a low sampling rate. For example, in Figure 7(a), there are only two samples over WiFi (26ms and 308ms), making prediction difficult. Second, an alternative approach is to use transport-layer or network-layer RTT samples (*e.g.*, UDP/ICMP Ping) to approximate the app-layer RTT. Doing so may provide a high sampling rate but low accuracy. Figure 7(b) compares the ICMP Ping RTT ("+") versus the HTTP response delay ("□"), with the latter being significantly higher. This is attributed to the additional buffering delay that a client-issued UDP/ICMP Ping does not experience. Such delays are typically incurred at the server application and/or the server-side TCP send buffer [43, 46].

Given neither approach above works well, we resort to the third approach of using HTTP/2 PING frames, part of the HTTP/2 specification [19]. HTTP/2 PING is mainly designed

for determining whether an idle connection is still alive. Unlike TCP Ping or ICMP Ping, HTTP/2 PING operates at the app layer, so its frame experiences the same delay as the HTTP response delay, which includes the potentially high buffering delay on the server side – exactly what we need.

In MP-H2, the client uses HTTP/2 PING frames to compute the application-layer RTT. Specifically, the client sends a PING frame every 100ms over each path, and uses the moving average of the last 5 samples to estimate the app-layer RTT of the corresponding path. Note that the overhead of sending these PING frames is negligible given their small sizes. The app-layer RTT is then used to approximate the HTTP response delay. As shown in Figure 7(b), the delay of the second HTTP response ("□") is similar to the recent HTTP PING samples collected over the WiFi path ("■").

3.4.2 Adjusting the HTTP Byte Range End. Now we consider how to adjust the HTTP byte range end. Recall from §3.3 that this needs to be done under two scenarios to avoid downloading the same data over both paths: (1) in the first iteration, adjust the primary path's byte range end to prevent it from overshooting the midpoint, and (2) in every iteration, adjust the slow path's byte range end when a fast path finishes.

Neither HTTP/1.1 nor HTTP/2 provides a direct way to modify the byte range end of an on-going byte-range transaction. In HTTP/1.1, the only way to realize this appears to be terminating the underlying TCP connection and establishing a new one. In HTTP/2, we find a more efficient method by leveraging the HTTP/2 `RST_FRAME` frame, which cancels an on-going stream without tearing down the underlying transport-layer channel. The client can simply send this frame once it reaches the desired byte range end.

Figure 7(c) repeats the experiment in Figure 7(a), with the integration of pipelining (§3.4.1) and byte range adjustment using `RST_FRAME`. We first make a positive observation that both mechanisms indeed work: over WiFi, S2 is requested before S1 completes, and the byte range end of S1 (the primary path) in the first iteration is properly adjusted so it does not overshoot the cellular chunk. However, we notice that S2 yields a lower goodput than S1 does as indicated by their different slopes. We discover that this is caused by cross-layer interactions between TCP and HTTP/2 streams. Specifically, upon the reception of `RST_FRAME`, the server application stops sending data over the closed stream. However, all its remaining data in the TCP send buffer, which can become fairly large, will still be transmitted to the client [43]. Because the server application does not have any control over the in-kernel TCP buffer, such unwanted bytes, which are called "tail bytes" [43], cannot be removed.

As a side effect of using `RST_STREAM`, the tail bytes are wasteful: they consume the network bandwidth but are discarded by the client. As a result, in Figure 7(c), the tail bytes of S1 cause reduced goodput for S2 as shown in the highlighted area. Next, we describe a novel solution to realize byte range end adjustment without incurring tail bytes. The basic idea is to leverage *HTTP/2 flow control* to limit the amount of data sent to the client. Our approach is fully compatible with the HTTP/2 specification [19].

Different from TCP flow control, HTTP/2 flow control provides *app-level* APIs to regulate the delivery of individual streams. It can be used to prevent the sender from overwhelming the receiver with data it may not want or be able to process [7]. It applies to individual streams with the following mechanism. When an HTTP/2 connection is established, the client and server advertise their initial stream-level flow control windows using the `SETTINGS` frame[1]. A stream's window on the sender side imposes a cap on the amount of data it can send to the receiver side over that stream. Note different streams may have different window sizes. For an HTTP/2 download, the server-side window size is reduced whenever the server sends a `DATA` frame to the client, and is incremented when the server receives a `WINDOW_UPDATE` frame from the client. When the window size becomes ≤ 0, the sender cannot send new `DATA` frames over the corresponding stream until it receives a `WINDOW_UPDATE` frame that causes the window to become positive.

To ensure the server does not go beyond the desired byte range end, MP-H2 limits its window size in such a way that after the server sends the last byte in the desired byte range, its window size becomes zero. As a result, the server emits no byte outside the byte range into TCP send buffer or the network. We realize this by strategically issuing the `SETTING` and the `WINDOW_UPDATE` frames to adjust the server-side window size[2]. Note that this also needs to be done in advance – one HTTP PING RTT (§3.4.1) before the completion time of the stream according to the desired byte range end, to ensure the server's timely reception.

We repeat the experiment in Figure 7(c) while changing the byte range adjustment mechanism from using `RST_FRAME` to using HTTP/2 flow control. We confirm that the tail bytes are almost fully eliminated. This is reflected in Figure 7(d) where the goodput of S1 and S2 are almost identical.

3.5 Put Everything Together

We now describe the overall scheduling algorithm (Algorithm 1), which is triggered when the client receives data on either path. Note that for a given object being downloaded, at any given time, we have at most one connection over each path, and each connection may include at most two HTTP/2

[1]With the `SETTINGS_INITIAL_WINDOW_SIZE` parameter.

[2]A practical limitation is that `WINDOW_UPDATE` can only *increase* a window. So when we need to decrease a particular stream's window, we first send a `SETTING` frame to reduce all streams' windows to 0, and then use multiple `WINDOW_UPDATE` to adjust each individual stream's window. Note all above control frames can be sent in a single batch.

Algorithm 1: The MP-H2 Client-side Scheduling Algorithm.

```
Input: Interface this and other that transmit data over either
       WiFi or cellular. This function is called when we
       receive bytes on this interface.
Output: A new stream to be created on the fast interface if
        this is faster than other, otherwise NULL.
1  thisDLTime ← bytesToDL(this) / getBw(this);
2  otherDLTime ← bytesToDL(other) / getBw(other);
3  if otherDLTime − thisDLTime > δ then
4      if thisDLTime ≤ estimatedRespDelay(this) &
        numStreams(this) < 2 then
5          newByteRangeToDLOnThis ←
            computeBytesRatio(bytesToDL(other) ,
            getBw(other), getBw(this),
            estimatedRespDelay(this));
6          adjustByteRangeEnd(other,
            newByteRangeToDLOnThis.start −1);
7          if getType(this) == WiFi & isInitialStream(this)
            then
8              adjustByteRangeEnd(this, filesize/2);
9          return createNewStream(this,
            newByteRangeToDLOnThis);
10 return NULL;
```

streams (due to pipelining). We first estimate the completion time of the chunks that are being downloaded (Line 1 and 2)[3]. Then, we create a new stream over the current path, *i.e., this*, if all following conditions hold (Line 3 and 4): (1) the current path is faster, *i.e.,* expected to complete its chunk sooner than the *other* path does, (2) the difference of the expected completion time between the two paths is larger than a threshold δ (empirically set to 50ms), and (3) the time taken to complete the current chunk is not longer than the estimated HTTP response delay, so that the client can start receiving the next chunk as soon as the current chunk completes. This is the pipelining optimization described in §3.4.1. If all three conditions hold, we use a slightly modified version of Equation 1 to estimate the size of the next chunk to be downloaded on the fast path by letting $RTT_i + \frac{D_i}{BW_i} = \frac{D_j}{BW_j}$ where i is the fast path (Line 5). We do not need RTT_j here because an on-going transfer is already occurring on the slow path. Meanwhile we adjust the byte range end of the chunk that is being downloaded on the slow path to avoid overlaps between the two paths (Line 6). §3.4.2 details how this is achieved. Recall from §3.3 that there is another case where we need to adjust the byte range end due to a lack of the file size information: when the fast path (*i.e., this*) is the primary path and the current stream is the first stream created on this path, we adjust its byte range end to the midpoint of the file to prevent an overshoot (Line 7 and 8).

[3]The `getBW` function is realized as follows. The client continuously measures each path's throughput by generating a throughput sample when at least 20KB of new data is downloaded. The path's bandwidth is then estimated by computing a weighted moving average of the most recent samples collected over a window of 500KB.

3.6 Additional Options of MP-H2

We discuss two additional options that MP-H2 provides for further improving the performance. First, MP-H2 allows a web client to provide the size of a file before requesting it. Having the size information beforehand offers two benefits. First, it allows the scheduler to send the byte-range requests over all paths at the same time, thus saving one RTT over secondary paths. Second, the scheduler does not need to invoke the extra logic of dealing with missing file size (§3.3), leading to improved performance as well. Note that in practice, file size information is oftentimes available in several contexts such as DASH video streaming [33, 52, 53] and web-based applications [16, 18].

The second option that MP-H2 provides is to cache the network condition estimation (bandwidth and latency) across multiple HTTP transactions. This is similar to TCP caching various parameters across connections (unless `tcp_no_metrics_save` [9] is set in `sysctl`). Having accurate network condition estimation before an HTTP transaction allows MP-H2 to use Equation 1 to find an appropriate chunk size for each path, instead of blindly performing a half-half split at the beginning. This may help reduce the number of iterations *i.e.,* the number of byte-range requests issued. We evaluate the effectiveness of both options in §5.

4 IMPLEMENTATION

We implement MP-H2 on commodity Android devices, targeting a common usage scenario of concurrently using cellular and WiFi. To access and transfer data on both paths simultaneously, we use an API [3] that was introduced in Android 5.0 (87% of Android devices run Android 5.0 or higher [1] as in 8/2018). Using this API, applications can request access to a `Network` object containing a set of `Capabilities` [12]. These capabilities include different types of transports (*e.g.,* cellular, Ethernet, Bluetooth) and/or properties (*e.g.,* unmetered or non-restricted network). If a network with specified capabilities is available, it will be provided to the application through a callback.

We implement MP-H2 on top of OkHttp, a popular HTTP client for Android and Java applications [13]. We modify its source code to expose some internal APIs to the MP-H2 scheduler. They include reading a stream's ID and sending HTTP/2 PING, `WINDOW_UPDATE`, and `SETTINGS` frames.

Our implementation consists of about 2,000 lines of code in Java. Except enabling HTTP/2 and byte range requests, our implementation does not change any configuration on the server. No server-side source code modification is required. MP-H2 provides client-side applications with interfaces that

are similar to those provided by other popular HTTP client libraries. We have verified that MP-H2 works well with two mainstream web servers: Apache and Nginx.

5 EVALUATION

We extensively evaluate MP-H2 under various settings including different network conditions, server configurations, and workloads over commercial WiFi and LTE networks. We also quantitatively compare MP-H2 with MPTCP and existing HTTP-based multipath solutions.

5.1 Integration with Commercial Services

Before examining the performance, we first demonstrate that MP-H2 can indeed work with commercial services without server-side changes. We consider two applications: downloading a file from Dropbox and watching videos from commercial video content providers. Note that in both cases we have no access to the server and therefore cannot compare MP-H2 with MPTCP. The comparisons will be done in the remaining subsections using our controlled server.

For Dropbox download, we develop a simple Android app that calls MP-H2. Our app can download any Dropbox file given its URL over the Internet. For video streaming, we slightly modify ExoPlayer [5], a popular open-source DASH video player used by more than 10k mobile apps [6] by adding the MP-H2 support. We have successfully tested our MP-H2-enabled ExoPlayer with Dailymotion [4], a popular video streaming service. Note for some other video streaming services such as YouTube, their DASH manifest files are protected, so they cannot even work with the vanilla ExoPlayer.

Finally we note that the client-side app change is small. For our Dropbox client and ExoPlayer, we only modified 5 and 15 lines of code, respectively, in order to add MP-H2.

5.2 Experimental Setup and Methodology

We now investigate the performance of MP-H2.

We study three variants of MP-H2: (1) MP-H2+FS, with the file size given by the application, (2) MP-H2+FS+BW/L, with the file size known and the history bandwidth and latency information available, (3) MP-H2, with *no* knowledge on the file size or network condition, making it fully transparent to the application. We compare them with MPTCP, mHTTP [37], and MSPlayer [22], with the latter two being the state-of-the-art HTTP-based multipath schedulers.

Testbed. We set up our multipath testbed using two Nexus 5 phones with access to commercial WiFi and LTE networks, and a commodity server with 4-core 3.6GHz CPU, 16GB memory, and 64-bit Ubuntu 16.04 with Linux kernel 4.9.60. We install the latest version of Android-compatible MPTCP (version 0.89.5) on one Nexus 5 phone running Android 4.4.4[4].

[4]We did not use a newer device or Android version because the latest Android-compatible MPTCP version only works with Nexus 5 and Android 4.4.4 at the time when this study was conducted [10].

The server runs the latest MPTCP (v0.94) available at the time of experiments. For both the phone and server, MPTCP uses the MinRTT multipath scheduler, WiFi as the primary path, and decoupled congestion control to ensure fair comparisons with MP-H2. To evaluate MP-H2 and compare it with existing HTTP-based multipath solutions, we implement MP-H2, mHTTP, and MSPlayer on top of OkHttp library on the second Nexus 5 phone running Android 5.0, the minimum required version for multipath transfers on Android (§4). We installed Apache 2.4.29 with HTTP/2 module on the same server. We have verified that the two phones have almost the same single-path TCP performance over both WiFi and cellular networks under the same network condition.

We evaluate MP-H2 under three settings: (1) stable network conditions, (2) highly fluctuating network conditions by replaying bandwidth profiles collected from various public locations, and (3) a real-world setting without any bandwidth throttling. Note that all three settings use commercial WiFi (Comcast) and LTE (AT&T) as the underlying "base" networks, on which additional bandwidth throttling is imposed for (1) and (2). We next detail each setting. For (1), we use constant bandwidth and latency (*e.g.,* 5Mbps WiFi and 10Mbps LTE, imposed by Linux `tc` running on the server side), whose values match recent large-scale measurements of LTE and WiFi [35, 51]. For (2), to capture the bandwidth dynamics of real networks, we collect bandwidth traces (both public WiFi and commercial cellular) at five locations including coffee shops, residential apartment, and campus library. The captured bandwidth is highly variable. The average WiFi (cellular) bandwidth across these locations ranges from 4.3Mbps (1.4Mbps) to 7.2Mbps (8.3Mbps), with the standard deviation being up to 57% and 60% of the mean bandwidth for WiFi and cellular, respectively. The average RTT between the phone and the server is around 28ms and 65ms over WiFi and LTE, respectively. We then replay these bandwidth traces over commercial WiFi and LTE networks to realistically emulate the fluctuating bandwidth. The replay is done using Linux `tc`, and is performed at a location with very good network conditions (60Mbps WiFi bandwidth provided by Comcast and 40Mbps LTE bandwidth provided by AT&T), to make the replay experiments reproducible while capturing the latency dynamics from commercial wireless networks. For (3), we conduct experiments in a residential building with around 18Mbps WiFi bandwidth (Comcast) and 32Mbps LTE bandwidth (AT&T), without applying additional throttling.

5.3 Single File Download

Stable Bandwidth Conditions. We begin with evaluating the performance of MP-H2 under stable bandwidth conditions. The workload is downloading files of different sizes (*e.g.,* video chunks and large web objects). For each experiment, we repeat the download 30 times and report the 25th,

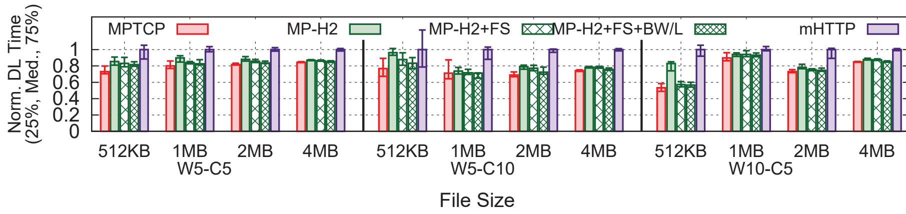


Figure 8: **Performance of different schedulers under three bandwidth settings.**

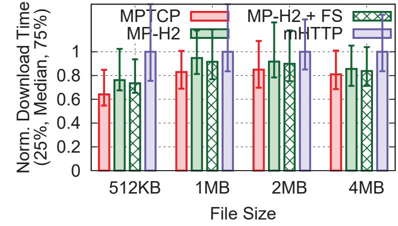


Figure 9: **Schedulers under real BW profiles.**

50th, and 75th percentiles of the download time. We consider different bandwidth combinations of the two paths. For mHTTP, it uses a default chunk size of 1024KB. Since the file size can be smaller than that, we also try three smaller chunk sizes (128KB, 256KB, and 512KB) for mHTTP and report the one with the best performance. For all experiments, we use persistent TCP or MPTCP connections.

Figure 8 compares the download time across three MP-H2 variants, mHTTP, and MPTCP for different file sizes. We consider three bandwidth combinations of WiFi and LTE: (5Mbps, 5Mbps), (5Mbps, 10Mbps), and (10Mbps, 5Mbps). All three MP-H2 variants outperform mHTTP, achieving 3% to 26% faster download time across all different file size and bandwidth combinations. For 4MB file, for example, MP-H2 yields a faster download time between 353ms and 807ms. Such an improvement is attributed to MP-H2 strategically performing byte range split and pipelining the HTTP requests, instead of employing fixed chunk sizes and sequential requests as done by mHTTP. Meanwhile, all MP-H2 variants provide comparable performance to MPTCP, only 0.1% to 11% longer than MPTCP's download time for 1MB+ files. If we consider the best MP-H2 variant (MP-H2+FS+BW/L), its download time is only 0.1% to 3% higher for 1MB+ files, compared to MPTCP. For 512KB file, the difference between MP-H2 and MPTCP is slightly larger (2% to 25%). This comparison shows that there is an intrinsic tradeoff between performance and intrusiveness: MPTCP's server-side scheduling at the transport layer can perform precise byte-level control at the sub-RTT time granularity, while MP-H2's client-side scheduling at the HTTP layer has to be much more coarse-grained, leading to worse performance despite MP-H2's various optimizations. Overall, the results show that MP-H2 outperforms mHTTP, and strikes a good balance by providing an easy-to-deploy alternative to MPTCP with only slight sacrifice of performance.

Among the three MP-H2 variants, MP-H2+FS+BW/L achieves the best performance for all file sizes as explained in §3.6. MP-H2+FS slightly falls behind MP-H2+FS+BW/L by up to 6%, as the network conditions are stable here.

Varying Bandwidth Conditions. We next evaluate how MP-H2 performs under fluctuating network conditions by replaying bandwidth traces (both WiFi and LTE) we collected at the five public locations as described in §5.2.

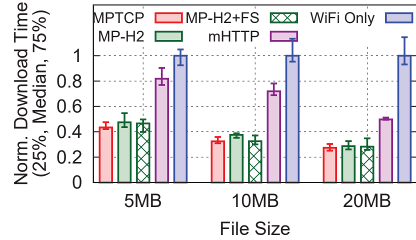


Figure 10: **Compare 5 schedulers under unthrottled bandwidth.**

Figure 9 shows the download time of different file sizes. Compared to mHTTP, MP-H2 and MP-H2+FS reduce the median download time by 5% (67ms) to 23% (216ms) and 9% (107ms) to 26% (240ms), respectively. Similar to the stable bandwidth conditions, MP-H2 and MP-H2+FS yield small download time increase compared to MPTCP when the bandwidth is fluctuating, within 14%∼18% (3%∼5%) for the 512KB (4MB) file, respectively. We also observe that the download time difference between MP-H2 and MPTCP is larger when file size is smaller. This is explained by two reasons. First, delaying the request over the cellular (secondary) path incurs a higher performance impact as the file size gets smaller. Second, because of their client-side and application-layer nature, all MP-H2 variants adjust the scheduling decisions less frequently than MPTCP does, and thus react slower to the fluctuating network conditions.

Real-world Networks without Throttling. We further evaluate MP-H2 under unthrottled bandwidth. Recall from §5.2 that this experiment was conducted in a residential building with the WiFi and LTE bandwidth being around 18Mbps and 32Mbps, respectively. We pick the workload of downloading large files of 5, 10, and 20MB using the five schemes shown in Figure 10. For each combination of the file size and download scheme, we repeat the download 30 times. Figure 10 shows that MP-H2 outperforms mHTTP by 42% to 47% in terms of the median download time. A 42% improvement translates to 0.8 and 2.9 seconds of shorter download time for a 5MB and 20MB file, respectively. The median download time of MP-H2 (MP-H2+FS) is only 4% to 15% (0.2% to 7%) higher than MPTCP. The results indicate MP-H2's good performance when the bandwidth is high.

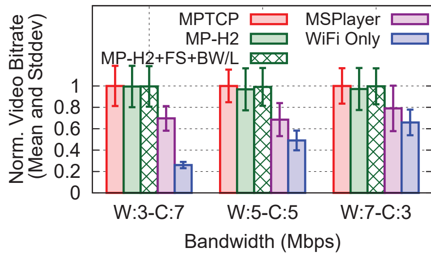


Figure 11: Compare the video quality of different schedulers under three capped bandwidth profiles.

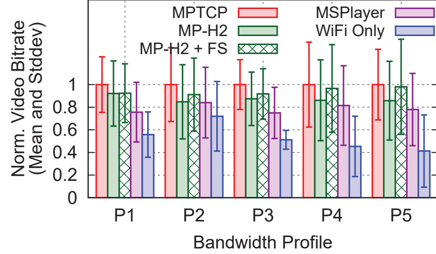


Figure 12: Compare the video quality of different schedulers under real bandwidth profiles.

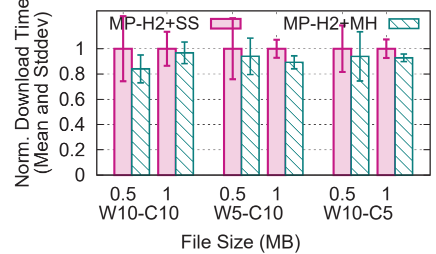


Figure 13: Impact of multi-homing on downloading a 0.5MB/1MB file under different bandwidth settings.

5.4 Video Streaming

We now study how MP-H2 helps improve the QoE of video streaming by comparing MP-H2 to MPTCP and MSPlayer [22], which is an HTTP-based multipath scheduler designed specifically for video streaming. It adjusts the chunk size of the fast path based on the throughput ratio between WiFi and cellular. The chunk size for the slow path is doubled or halved based on the comparison between the current bandwidth and its estimated value. We modify ExoPlayer [5] by adding the MP-H2 support (§5.1) as well as the MSPlayer scheduling algorithm. We also instrument ExoPlayer to retrieve playback statistics such as video bitrate and rebuffering (stall) events. For each experiment, we use ExoPlayer to play a 10-min ABR video hosted on our server. The video is segmented into 6-second chunks each having 9 video bitrates, ranging from 253kbps to 10Mbps, as described in the manifest file of the video. To evaluate the performance of MP-H2+FS and MP-H2+FS+BW/L, we include the size of each video chunk into the DASH manifest file.

Stable bandwidth conditions. Figure 11 compares the average video bitrate of the five schemes under three bandwidth combinations of WiFi and LTE: (3Mbps, 7Mbps), (5, 5), and (7, 3). In all three bandwidth combinations, the overall available bandwidth (WiFi+LTE) is sufficient to stream the highest video bitrate. As shown in Figure 11, MP-H2+FS+BW/L always performs better than MSPlayer by providing up to 44% higher video bitrate. This is attributed to the fact that MSPlayer does not pipeline its requests and thus suffers from bandwidth under-utilization due to the idle time between consecutive data chunks. Compared to MPTCP, MP-H2+FS+BW/L achieves almost the same video quality, with the video bitrate being only 0.3% to 0.9% lower. For this set of experiments, the rebuffering ratios (*i.e.,* stall duration over the entire video length) are consistently low: 0.1%, 0.3%, and 0.08% for MP-H2, MPTCP, and MSPlayer, respectively.

Changing bandwidth conditions. Figure 12 plots the results under five real bandwidth profiles. Overall the results are also encouraging: compared to MSPlayer, MP-H2+FS achieves up to 25% higher video bitrate, and its bitrate is only 2% to 8% lower compared to MPTCP. We also observe that the plain MP-H2 scheme slightly falls behind MP-H2+FS by 6% in terms of the video bitrate. This is due to a lack of file size information and delayed request on the secondary path. Similar to the stable bandwidth conditions, the rebuffering ratios of all schedulers here are also very small (below 0.8%).

5.5 Multi-homing

One eminent feature of MP-H2 that MPTCP does not support is multi-homing. To demonstrate the benefit of multi-homing in the context of multipath, we consider two scenarios. In the first scenario (MP-H2+SS, "SS" stands for "single server"), MP-H2 downloads the content from a single server (server A), which is identified by resolving the domain name over the primary path (WiFi). In the second scenario (MP-H2+MH, "MH" stands for "multi-homing"), we consider the scheme where the domains are resolved over each path separately and as a result, MP-H2 may fetch the content from multiple servers, server A for WiFi and server B for cellular, that provide the best performance for each path.

In our experiment, we set up the two servers (A and B) where the baseline RTT of WiFi and LTE between the client and the corresponding server is 28ms and 65ms, respectively (based on our collected traces, see §5.2). We add 20ms extra delay to the LTE downlink from server A to the client, to account for the delay penalty of LTE accessing server A. To see why this setting is realistic, note that the 20ms delay is selected based on our crowd-sourced measurement in §2.2.2, and can be translated to ~30% increase in the latency over the cellular path that is a common case in the real world as shown in Figure 4(b). Figure 13 shows the results of downloading 512KB and 1MB files over different bandwidth settings under MP-H2+SS and MP-H2+MH. Compared to using a single server, MP-H2 with multi-homing reduces the average file download time by 4% to 19%. This shows that MP-H2 can boost the performance by leveraging CDN multi-homing.

5.6 Pipelining HTTP Requests

MP-H2 utilizes pipelining, which sends a new request one app-layer RTT before the on-going chunk transfer finishes,

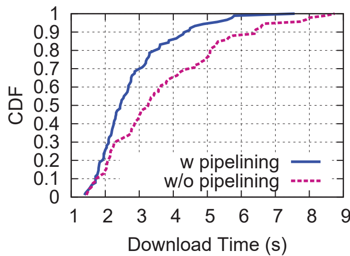


Figure 14: Compare performance of MP-H2 with and without HTTP pipelining.

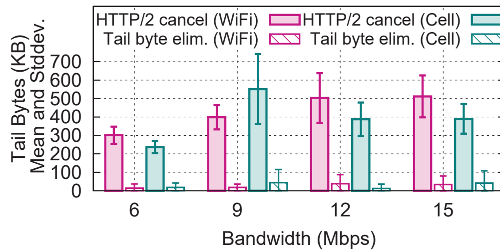


Figure 15: Compare the tail byte elimination mechanism based on HTTP/2 flow control with that using the RST_STREAM frame.

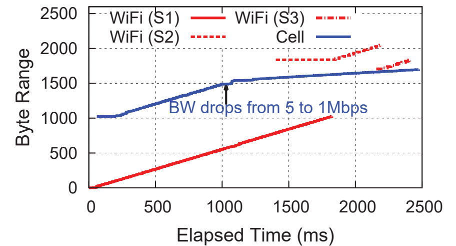


Figure 16: A case study of how MP-H2 reacts when the bandwidth of one path drops to 1Mbps when downloading a 2MB file.

in order to avoid the idle time between the download of consecutive chunks (§3.4.1). To show the benefit of pipelining, we compare the performance of MP-H2 with and without pipelining. To do so, we download a 2MB file using each approach under five bandwidth profiles we collected (§5.2). For each test, we repeat the file download 30 times. As shown in Figure 14, the pipelining approach improves the median download time by 25% compared to the non-pipelining approach, where the scheduler sends the request of the next chunk *after* the previous chunk download is completed.

5.7 Tail Byte Elimination

We compare the tail byte elimination mechanism based on HTTP/2 flow control with that using the RST_STREAM frame that cancels an HTTP/2 stream (§3.4.2). Our experiment is as follows. We start downloading a 2MB file over a single path (WiFi or LTE with stable network condition), and stop the download when the first half of the file is transferred by updating the byte range end. Ideally only 1MB data should be observed over the downlink but tail bytes will further inflate that. Figure 15 shows the comparison of the number of tail bytes incurred by the two schemes above. When using the HTTP/2 stream cancellation mechanism, the tail bytes over WiFi and cellular range from 237KB to 551KB. Using HTTP/2 flow control reduces the tail bytes to 12KB to 44KB. The tail bytes are not completely eliminated by HTTP/2 flow control due to non-perfect estimation of the application RTT and bandwidth.

5.8 Adapting to Abrupt Network Changes

To demonstrate how MP-H2's scheduling algorithm responds to abrupt network changes, we conduct the following experiment. Initially, both paths have 5Mbps bandwidth. Around $t = 1050ms$, we decrease the cellular bandwidth to 1Mbps. As shown in Figure 16, around $t = 1386ms$, the scheduler creates another stream (S2) over WiFi, based on its estimation of both paths' bandwidth (using a weighted moving average). Initially the scheduler over-estimates the cellular bandwidth (to be 2.65Mbps). This results in requesting a smaller chunk over the fast path (WiFi). Later, as more bandwidth samples are collected, the scheduler obtains a more accurate bandwidth estimation and creates another stream (S3) to achieve simultaneous completion on both paths. Due to MP-H2's continuous bandwidth monitoring and various optimizations (§3), despite the sudden bandwidth drop, MP-H2 is largely capable of realizing both requirements (**R1:** simultaneous path completion and **R2:** full bandwidth utilization). We repeat the experiment using MPTCP under the same setting, and observe that the download time difference between MPTCP and MP-H2 is only 4% (averaged over 10 runs).

6 RELATED WORK

Besides the built-in MPTCP schedulers such as minRTT and round-robin [45], as well as mHTTP [37] and MSPlayer [22] that we quantitatively compare MP-H2 with, there are other multipath schedulers in the literature. DEMS [29] is an MPTCP scheduler that achieves simultaneous subflow completion by performing "two-way" data transfers in opposite directions within a file. ECF [41] is designed for heterogeneous paths and considers both congestion window and RTT to decide which subflow to use. eMPTCP [40] takes energy models of WiFi and cellular into consideration when making scheduling decisions. MPRTP [49] is designed as an extension to RTP and targets real-time content delivery over multipath. All schemes above require server modifications. Instead, MP-H2 is a client-side scheduler, making it readily deployable through simple HTTP library upgrades.

Similar to mHTTP [37], there exist other HTTP-based multipath schedulers aiming at improving the video streaming performance. Kaspar *et al.* [36] proposed to pipeline byte range requests over HTTP/1.1 to accelerate fetching large video files. It also calculates an optimal chunk size to minimize the idle time between consecutive requests. However, HTTP/1.1 pipelining has various limitations and is not widely used in practice [7, 8]. Koo *et al.* [38] and Go *et al.* [28] proposed multipath schedulers for downloading segments of a DASH video. Both schemes formulate multipath scheduling as an optimization problem to maximize the video quality under the energy consumption and cellular data usage constraints. Compared to MP-H2, both of them only make

coarse-grained scheduling decisions in that neither scheduler splits a video segment into smaller chunks sizes. Also neither scheduler explicitly considers the requirement of fully utilizing the available bandwidth *i.e.,* **R2** defined in §3.1. In addition, none of the above work focuses on HTTP/2.

Recently, there have been some application-specific extensions to the MPTCP protocol. For instance, MP-DASH [33] considers the cost of each path in DASH video streaming. Its cost-aware scheduling logic can be possibly incorporated into MP-H2. In addition, researchers very recently propose multipath QUIC [23]. Operating at the application layer, MP-H2 can instead bring multipath support to any transport protocols or even their mixture (§2.4).

A topic relevant to multipath is interface selection. For example, Socket Intents [48] enable apps to provide their interface preferences through the socket API. Other work along this line includes [25, 34, 44]. MP-H2 can also provide a similar interface at the HTTP layer to allow apps to switch between single and multipath based on various contexts.

7 DISCUSSIONS

IP-based Content Customization. MP-H2 relies on issuing multiple HTTP byte-range requests over different interfaces to accelerate the data transfer. However, although not common, the server may customize the content based on the client's IP address; distributed CDN servers may also return different contents based on the same URL for purposes such as geographic-based content customization. In these cases, assembling byte-range responses in general may lead to corrupt data. Note this happens to responses obtained over not only multipath but also single path, because the IP address of a client may change over time due to mobility or DHCP. Our recommendation is that the server should make byte-range responses non-cacheable, or simply disable the byte-range feature for objects that are customized based on the client's IP address or other network states.

When there is no customization of byte-range responses, MP-H2 can be safely applied. MP-H2 is compatible with various security-enhancing mechanisms such as TLS Client Authentication [17] and Channel-bound Cookies [2].

Fairness. We discuss potential fairness issues brought by MP-H2 at three levels. First, at the HTTP/2 stream level, MP-H2 does introduce a higher level of parallelism due to pipelining (§3.4.1). The additionally created streams may thus utilize extra network resources. However, we expect its impact on stream-level fairness is very limited due to several reasons. (1) Pipelining occurs only for a short duration as illustrated in Figure 7(c); (2) tail elimination further mitigates the impact as shown in Figure 7(d); and (3) the impact is further bounded by limiting the number of per-object streams over each path to 2 (Line 4 in Algorithm 1). Second, at the TCP connection level, since MP-H2 does not establish additional TCP (sub)flows compared to applying the vanilla MPTCP, there should not be additional inter-connection fairness issues. Third, at the mobile device level, compared to single path over WiFi, MP-H2 offloads traffic to cellular networks whose resources may be scarce when an excessive number of devices are sharing the same base station or RAN. This is a generic problem of any mobile multipath scheme, and cross-device fairness can typically be ensured by the network (*e.g.,* through proportionally fair scheduling at e-NodeB).

HTTP/1.1 Fallback. Although HTTP/2 is getting increasingly popular, many of today's web servers still only support HTTP/1.1. In order to be compatible with HTTP/1.1 servers, MP-H2 can still use the high-level scheduling approach described in §3.3, but all HTTP/2-specific features have to be disabled. Specifically, (1) the client cannot use pipelining when issuing a new request (§3.4.1), and (2) the client can only adjust the HTTP byte range end by terminating the existing connection and issuing a new request (§3.4.2). The above simplifications inevitably lead to performance degradation compared to full-fledged MP-H2 for HTTP/2.

Multipath Upload. Another limitation of MP-H2 is its lack of support for data upload over multipath. In theory, one can develop a client-side scheduling algorithm that splits an HTTP PUT transaction over multiple paths using byte range requests. There are two practical limitations though. First, not many servers accept byte-range HTTP PUT requests; second, a data upload typically does not support multi-homing.

MP-H2 for QUIC. Porting MP-H2 to QUIC, whose most features are very similar to those of HTTP/2, is relatively easy. In particular, QUIC also provides stream multiplexing, PING frames, and connection-level flow control [14] that are the key building blocks of MP-H2.

8 CONCLUDING REMARKS

We address an important problem of *lowering the deployment bar of mobile multipath and making it friendly to today's infrastructures* such as NAT, CDN, and anycast. To this end, we develop MP-H2, a client-side, user-space multipath solution for HTTP, the *de-facto* application-layer protocol today. The key design philosophy of MP-H2 consists of (1) fully decoupling all paths at the transport layer, (2) judiciously balancing all paths' workload from the client side, and (3) strategically leveraging new HTTP/2 features to maximize the bandwidth utilization. Extensive evaluations show that MP-H2 brings similar performance to MPTCP and considerably outperforms other HTTP-based multipath schedulers.

ACKNOWLEDGEMENTS

We thank our shepherd, Peter Steenkiste, and the anonymous reviewers for their valuable comments. This research was supported in part by the National Science Foundation under grants CCF-1628991, CNS-1629763, and CNS-1566331.

REFERENCES

[1] Android platform versions. https://developer.android.com/about/dashboards/index.html.

[2] Channel-bound cookies. http://www.browserauth.net/channel-bound-cookies.

[3] ConnectivityManager | Android Developers. https://developer.android.com/reference/android/net/ConnectivityManager#requestnetwork.

[4] Dailymotion - Explore and watch videos online. https://www.dailymotion.com/.

[5] ExoPlayer. https://google.github.io/ExoPlayer.

[6] Exoplayer 2 – why, what and when? https://medium.com/google-exoplayer/exoplayer-2-x-why-what-and-when-74fd9cb139.

[7] High Performance Browser Networking, O'Reilly. https://hpbn.co/.

[8] HTTP/2 Frequently Asked Questions. https://http2.github.io/faq/.

[9] Kernel Documentation: networking/ip-sysctl.txt. https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt.

[10] Multipath tcp - linux kernel implementation. https://multipath-tcp.org/pmwiki.php/Users/Android.

[11] Netflix open connect. https://openconnect.netflix.com/en/.

[12] NetworkCapabilities | Android Developers. https://developer.android.com/reference/android/net/NetworkCapabilities.

[13] Okhttp. http://square.github.io/okhttp/.

[14] Quic, a multiplexed stream transport over udp. https://www.chromium.org/quic.

[15] Streaming services now account for over 70% of peak traffic in North America, Netflix dominates with 37%. https://tinyurl.com/yafk38w3.

[16] The Web App Manifest. https://developers.google.com/web/fundamentals/web-app-manifest/.

[17] Tls client authentication. http://www.browserauth.net/tls-client-authentication.

[18] Web App Manifest. https://www.w3.org/TR/appmanifest/.

[19] M. Belshe, R. Peon, and E. M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, Internet Engineering Task Force, 2015.

[20] T. Bottger, F. Cuadrado, G. Tyson, I. Castro, and S. Uhlig. Open connect everywhere: A glimpse at the internet ecosystem through the lens of the netflix cdn. In *Proc. of ACM SIGCOMM Computer Communications Review (CCR)*, 2018.

[21] 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. of IMC*, 2013.

[22] Y.-C. Chen, D. Towsley, and R. Khalili. MSPlayer: Multi-Source and multi-Path LeverAged YoutubER. In *Proc. of CoNEXT*, 2014.

[23] Q. De Coninck and O. Bonaventure. Multipath QUIC: Design and Evaluation. In *Proc. of CoNEXT*, 2017.

[24] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. WiFi, LTE, or Both?: Measuring Multi-Homed Wireless Internet Performance. In *Proc. of IMC*, 2014.

[25] S. Deng, A. Sivaraman, and H. Balakrishnan. All your network are belong to us: A transport framework for mobile network selection. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, page 19. ACM, 2014.

[26] F. Duchene and O. Bonaventure. Making Multipath TCP friendlier to Load Balancers and Anycast. In *Proc. of IEEE ICNP*, 2017.

[27] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, 1999.

[28] Y. Go, O. C. Kwon, and H. Song. An Energy-Efficient HTTP Adaptive Video Streaming With Networking Cost Constraint Over Heterogeneous Wireless Networks. *IEEE Transactions on Multimedia*, 17(9):1646–1657, 2015.

[29] Y. E. Guo, A. Nikravesh, Z. M. Mao, F. Qian, and S. Sen. Accelerating Multipath Transport Through Balanced Subflow Completion. In *Proc. of ACM MobiCom*, 2017.

[30] B. Han, S. Hao, and F. Qian. Metapush: Cellular-friendly server push for http/2. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, pages 57–62. ACM, 2015.

[31] B. Han, F. Qian, S. Hao, and L. Ji. An Anatomy of Mobile Web Performance over Multipath TCP. In *Proc. of CoNEXT*, 2015.

[32] B. Han, F. Qian, and L. Ji. When should we surf the mobile web using both wifi and cellular? In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, pages 7–12. ACM, 2016.

[33] B. Han, F. Qian, L. Ji, and V. Gopalakrishnan. MP-DASH: Adaptive Video Streaming Over Preference-Aware Multipath. In *Proc. of CoNEXT*, 2016.

[34] B. D. Higgins, A. Reda, T. Alperovich, J. Flinn, T. J. Giuli, B. Noble, and D. Watson. Intentional networking: opportunistic exploitation of mobile network diversity. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, pages 73–84. ACM, 2010.

[35] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance. In *Proc. of SIGCOMM*, 2013.

[36] D. Kaspar, K. Evensen, P. Engelstad, and A. F. Hansen. Using HTTP Pipelining to Improve Progressive Download over Multiple Heterogeneous Interfaces. In *IEEE International Conference on Communications*, 2010.

[37] J. Kim, R. Khalili, A. Feldmann, Y. Chen, and D. Towsley. Multi-source multi-path HTTP (mhttp): A proposal. *CoRR*, abs/1310.2748, 2013.

[38] J. Koo, J. Yi, J. Kim, M. A. Hoque, and S. Choi. Seamless Dynamic Adaptive Streaming in LTE/Wi-Fi Integrated Network under Smartphone Resource Constraints. *IEEE Transactions on Mobile Computing*, 2018.

[39] H. Lee, J. Flinn, and B. Tonshal. Raven: Improving interactive latency for the connected car. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 557–572. ACM, 2018.

[40] Y.-s. Lim, Y.-C. Chen, E. M. Nahum, D. Towsley, R. J. Gibbens, and E. Cecchet. Design, Implementation, and Evaluation of Energy-aware Multi-path TCP. In *Proc. of CoNEXT*, 2015.

[41] Y.-s. Lim, E. M. Nahum, D. Towsley, and R. J. Gibbens. ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths. In *Proc. of CoNEXT*, 2017.

[42] O. Mehani, R. Holz, S. Ferlin, and R. Boreli. An early look at multipath TCP deployment in the wild. In *HotPlanet 2015, 6th International Workshop on Hot Topics in Planet-Scale Measurement, in Conjunction with ACM MobiCom 2015*, 2015.

[43] X. Mi, F. Qian, and X. Wang. SMig: Stream Migration Extension For HTTP/2. In *CoNEXT*, 2016.

[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. of ACM MobiCom*, 2016.

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

[46] F. Qian, V. Gopalakrishnan, E. Halepovic, S. Sen, and O. Spatscheck. Tm3: Flexible transport-layer multi-pipe multiplexing middlebox without head-of-line blocking. In *CoNEXT*, 2015.

[47] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proc. of ACM SIGCOMM*, 2011.

[48] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann. Socket intents: Leveraging application awareness for multi-access connectivity. In *CoNEXT*. ACM, 2013.

[49] V. Singh, S. Ahsan, and J. Ott. MPRTP: Multipath Considerations for Real-time Media. In *Proc. of MMSys*, 2013.

[50] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 325–336. ACM, 2014.

[51] J. Sommers and P. Barford. Cell vs. WiFi: On the Performance of Metro Area Mobile Connections. In *Proc. of IMC*, 2012.

[52] S. Xu, S. Sen, Z. M. Mao, and Y. Jia. Dissecting VOD Services for Cellular: Performance, Root Causes and Best Practices. In *Proc. of IMC*, 2017.

[53] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proc. of ACM SIGCOMM*, 2015.