A Multipath QUIC Scheduler for Mobile HTTP/2

Jing Wang, Yunfeng Gao, Chenren Xu✉*

Peking University

ABSTRACT

In recent years, QUIC protocol has shown great advantages for HTTPS over TCP in terms of improving handshake delay and head-of-line blocking. Multipath QUIC (MPQUIC) further opens up the opportunity to leverage path diversity for realizing various optimization goals, especially for mobile access. In this paper, we present a context-aware MPQUIC packet scheduler dedicated to mobile HTTP/2. Specifically, the scheduler takes into account the stream priority (from HTTP/2 dependency tree) for stream-aware downlink packet scheduling by exclusively transferring each stream at a time while maintaining the relative stream completing order. Additionally, ACK packets are scheduled by choosing the path with the lowest one-way delay to reduce overall RTT and expedite loss recovery. Real-world experiments show that our scheduler reduces page load time by up to 8.5% and stream average completion time by up to 12.9% over the status-quo.

CCS CONCEPTS

• **Networks → Transport protocols**; **Application layer protocols**; **Packet scheduling**.

KEYWORDS

QUIC, MPQUIC, multipath, mobile, HTTP/2, scheduler

ACM Reference Format:
Jing Wang, Yunfeng Gao, Chenren Xu. 2019. A Multipath QUIC Scheduler for Mobile HTTP/2. In *3rd Asia-Pacific Workshop on Networking 2019 (APNet '19), August 17–18, 2019, Beijing, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3343180.3343185

*✉: chenren@pku.edu.cn

1 INTRODUCTION

Mobile devices are generating more HTTP traffic than desktops today [1]. Meanwhile, HTTP/2 has been proposed to reduce Page Load Time (PLT) by using multiplexing, concurrency, stream dependencies, header compressions and server push. Up to now, 148,612 sites truly support HTTP/2 [2]. However, its improvement is blurred due to its complexity of loading a page caused by interleaving network transfer and local computation (*e.g.,* parsing HTML/JavaScript/CS), especially in mobile cases [3]. Optimizing HTTP alike traffic is highly necessary, as Google reported in [4] that additional delays in PLT significantly reduced the number of searches, and an extra delay of 500 milliseconds can decrease up to 20% traffic for some popular content provider [5].

QUIC protocol is proposed to improve transport performance for HTTP/2 traffic and to enable rapid deployment by elevating all the flow/congestion control logic to the user space. Today, QUIC accounts for up to 9.1% of the current Internet traffic [6]. By leveraging the native multiple interfaces (*e.g.,* WiFi and cellular) on mobile devices, the design of multipath QUIC [7, 8] can easily facilitate bandwidth aggregation from multiple independent paths. However, the PLT of HTTP/2 web involves complicated inter-object downloading dependencies and additional computation delay (*e.g.,* from JS), which is not considered in the native (MP)QUIC design, but can consequently lead to head-of-line blocking. As another aspect, in (MP)TCP operation, ACK and the corresponding data packet are often binded in the same path, which incurs out-of-order delay and postpones loss recovery when a link (*e.g.,* WiFi) experiences high variations of RTT.

In this paper, we present a downlink-uplink co-designed context-aware MPQUIC packet scheduler that exploits several unique characteristics of QUIC and HTTP/2 to address the issues aforementioned. For downlink scheduling, the server effectively leverages the priority information from the dependency tree provided by HTTP/2 to reduce the blocking period in the default weighted round-robin scheduling. Specifically, We first determine the stream deliver sequence considering both stream priority and size to guarantee the fairness by calculating the completion sequence of all streams in the weighted round robin algorithm. Then we optimize completion time for each specific stream by allocating path quotas to it. This not only makes all streams complete as the original sequence, but also transfers each stream exclusively and makes it complete on different paths simultaneously, which minimizes its completion time. For uplink scheduling, we send all ACK packets through the path with the lowest uplink latency based on dynamic probing. By breaking the tie between data and ack on the same path, our solution can reduce RTT and speed up loss recovery.

Contributions.

• We design a downlink-uplink co-designed scheduler for mobile HTTP/2. It leverages the unique protocol characteristics such as the dependency tree and flexible path assignment to collaboratively reduce the page load time.

• We implement a prototype of our co-designed scheduler and validate it by running simulated and real-world experiments. Our real-world experimental results (§4) show we can reduce page load time by up to 8.5% and reduce stream average completion time by up to 12.9%.

2 BACKGROUND

HTTP/1.x has several performance issues with establishing multiple TCP connections. The issues include unnecessary handshakes, superfluous header retransmissions, and slow start. HTTP/2 aims to address those issues by only using one TCP connection per origin with separate frames. The Header Frames contain the HTTP header and other control messages. The Data Frames contain request/response bodies. HTTP/2 uses HPACK to compress header metadata and delivers all the Frames in parallel. HTTP/2 also introduces a dependency-based prioritization mechanism to schedule streams. The client can specify weights for specific streams and dependencies between streams. Thus, the server will send a stream only after the completion of its dependent streams. The weights also give the server hints about the resource (*i.e.,* bandwidth) allocation proportion among streams. HTTP/2 also enables other features like server push to speed up the page load procedure.

Google's Quick UDP Internet Connection (QUIC) is an application-layer transport protocol. It provides reliable, high-performance, and encrypted in-order packet delivery. It outperforms TCP from two aspects. Firstly, QUIC implements a loss detection and recovery mechanism, more flexible and efficient than TCP. One QUIC frame can acknowledge up to 256 packet ranges while SACK in TCP option can only deal with 2-3 blocks. Secondly, it eliminates head-of-line blocking. When a loss or out-of-order occurs on one stream, it blocks all other streams in HTTP/2 + TCP, because TCP has to maintain connection level in-order delivery. Whereas, QUIC allows other streams to continue transmitting without being blocked. Although QUIC is a transport protocol, it has many features designed to work with HTTP/2. QUIC implements stream multiplexing and its data streams carry the HTTP/2 data frames. Besides, QUIC uses a header stream in the entire connection to handle HTTP/2 header frames. When a header frame arrives at the server, either the server or the client opens new data streams to transmit data frames.

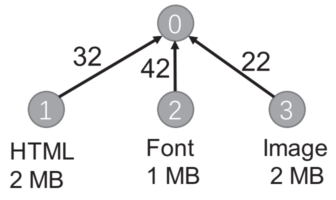


Figure 1: Dependency Tree Example.

MPQUIC [7] further adds multipath capability to QUIC. It adds several control frames and a path scheduler for path management. The path scheduler arranges packets on paths and is set to MIN-RTT by default. MPQUIC exhibits potential to outperform MPTCP with QUIC's inherent features. If a packet loss occurs, MPTCP can retransmit the packet on another path, but the original subflow will still retransmit the packet due to connection level in-order delivery, delaying the following packets and wasting network resources. In contrast, MPQUIC is able to reschedule lost packets on a different path flexibly, accelerating loss recovery.

3 DESIGN

We design a scheduler for both downlink and uplink. The downlink scheduler arranges downlink streams on different paths. It considers path characteristics (i.e., bandwidth, RTT) and stream priorities from HTTP/2 dependency tree. The uplink scheduler arranges ACK Frames by choosing the path with the lowest uplink delay. Thus, it can reduce overall RTT and speed up loss recovery.

3.1 Stream-Aware Downlink Scheduler

To load a web page, the browser needs to download a series of objects, including Javascript, CSS, images, and etc. The objects priority has a large impact on page loading. QUIC doesn't provide any priority schemes, but should make use of the priorities offered by overhead layers [10]. (MP)QUIC has access to the priority and length of each stream. Having this knowledge therefore allows us to determine stream delivery sequences and reduce completion time.

3.1.1 Stream Prioritization

HTTP/2 defines the dependency tree, a prioritization mechanism, to describe object priorities. In a dependency tree, nodes represent streams and links represent the dependencies. A children stream should wait until its parents finish the delivery. Each node has a weight number from 1 to 256, identifying their relative priorities. An inactive parent stream allocates its resources to its children stream.

The de facto MPQUIC scheduler neither supports the dependency tree nor takes action on the priority. Weighted

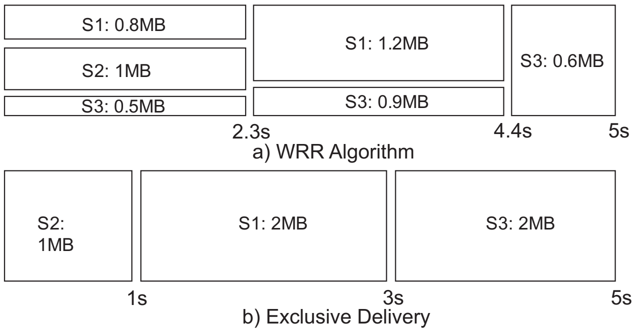


Figure 2: Stream Schedule Policy.

Round Robin (WRR) algorithm gives each stream a fair share. Each parent node in the dependency tree applies a scheduler based on the WRR algorithm. If the node has data to send and can send data, it consumes its resources. Otherwise, it distributes the resources to its children according to their priorities. In this way, several streams are simultaneously transferred on the same path. This algorithm considers the stream priority and the fairness not to starve any streams. However, the concurrent transfer can enlarge the average stream completion time.

During a web browsing procedure, a stream has minor usage before the stream's completion. It can benefit more if we keep fairness and use exclusive transmission. For example, in Fig. 1, stream 0 has 3 children streams which are HTML, font and image, with the weight of 32, 42, and 22, respectively, according to the Firefox priority scheme. Up to now, different browsers use different policies to build their dependency trees. It depends on the server to decide how to use the priority hints. In our work, we choose the priority scheme from a widely used browser Firefox. In this scheme, different object types are grouped into different priority classes. Assuming the bandwidth is 1 MBps, streams 2, 1, 3 would complete transmitting at 2.3, 4.4 and 5s respectively with WRR algorithm. However, if we transmit the streams 2, 1, 3 exclusively, they can complete at 1, 3, and 5s respectively, significantly reducing the average stream completion time. Note that if a client wants stream 1 to finish first, it can change the stream weight to a higher number (*e.g.,* 100).

Based on this intuition, our scheduler first calculates the stream completion sequence in WRR based algorithm using the information of stream length and dependency tree, and then exclusively deliver each stream.

3.1.2 Stream Bytes Allocation

We have already determined the transfer order of all the streams. Our goal is to make each path complete data transfer at the same time for a stream. We allocate an appropriate proportion of bytes on each path for each stream. When estimating stream completion time for each path, we consider bandwidth, RTT, and queuing time of previous streams.

Suppose we allocate S_{ij} bytes on path j to transfer stream i (here streams are renumbered to match their transfer order). The completion time of the stream on this path is

$$T_{ij} = RTT_j + S_{ij}/BW_j$$

Meanwhile, the total queuing time of the previous streams on this path is:

$$Q_{ij} = \sum_{k<i} T_{kj}$$

We can formalize our goal as

$$\begin{cases} T_{i1} + Q_{i1} = T_{i2} + Q_{i2} = \cdots = T_{iN} + Q_{iN} \\ \sum S_{ij} = SIZE_i \end{cases} \tag{1}$$

where N is the number of paths, $SIZE_i$ is the remaining size of stream i, RTT_j and BW_j are RTT and bandwidth of path j.

The solution to (1) is:

$$S_{ij} = BW_j \cdot \left(\frac{SIZE_i - \sum_k BW_k(RTT_j + Q_{ik})}{\sum_k BW_k} - RTT_j - Q_{ij}\right)$$

Q_{ij} can be calculated when previous streams are allocated. We use the smoothed RTT and the congestion window size provided by QUIC to estimate path RTT and bandwidth. Note that the theoretical solution may be non-integer or negative. Currently, we round down the numbers and convert negative numbers to zero, then pick a non-zero number randomly and adjust it to keep the sum of allocated bytes correct.

3.1.3 Put Them Together

We now describe how we integrate the algorithms in §3.1.1 and §3.1.2 into one scheduler. In the beginning, scheduling is triggered when the server has new data to send or receives an ACK. We first traverse through the paths in ascending order of RTT and try to send as many packets as possible on each path. When sending a packet, we traverse through the streams in the order determined by §3.1.1. For each stream, we apply a sending limit according to the allocation mechanism described in §3.1.2. When no bytes can be sent, we switch to the next stream; When the congestion window is full or no streams are available on this path, we turn to the next path. We show our algorithm in Alg. 1. When sending is finished, we update the remaining size of each stream, rearrange stream order, and reallocate bytes. New data arriving or path changing also triggers the rearrangement.

3.2 Latency-aware Uplink Scheduler

MPTCP can improve connection reliability but may not perform bandwidth aggregation well when paths are extremely heterogeneous. The throughput of MPTCP can be worse than single path TCP with disperse path capacity. MPQUIC mitigates the problem by eliminating head-of-line blocking and flexibly retransmitting lost packets on faster paths. MPQUIC adds a Path ID field into the ACK frame and thus enables acknowledging QUIC packets on any path in the same session.

Algorithm 1 Stream Earliest Completion Scheduler

```
Sort available paths with RTT
$p \leftarrow$ firstPath
while $p$ is not null do
  $s \leftarrow$ firstStream
  while ($s$ is not null) and (cwnd not full) do
    $quota \leftarrow$ bytes allocated on $p$ for $s$
    $swnd \leftarrow$ sending window of $s$
    $size \leftarrow$ remaining bytes of $s$
    $size \leftarrow min(quota, swnd, size)$
    if $size > 0$ then
      Send packet
    else
      $s \leftarrow$ nextStream
    end if
  end while
  $p \leftarrow$ nextPath
end while
Update stream remaining size
Rearrange stream order and allocation
```

But by far, an ACK frame is still sent on its own path. In our work, we estimate uplink delay for all paths and send ACK frames on the path with the lowest uplink delay. It reduces the RTT for slower paths without disturbing the congestion control algorithm. Our scheduler takes two steps:

- Estimate uplink delay. Duplicated ACK packets are sent simultaneously on all uplink paths periodically or when a new path is created. The server monitors duplicated ACKs and gets the uplink delay for all paths. Note that it's not reasonable to duplicate all packets containing ACK frames blindly on all paths, because doing so will generate considerable unnecessary traffic when path numbers grow. Another option is to modify the ACK frame format and to add a timestamp into it. But in this way, it will not be compatible with the legacy protocol. Clients using our scheme can work well with legacy servers.
- Change ACK Return Path. MPQUIC server receives and monitors duplicate ACKs, from which it keeps the uplink delay information for all paths up-to-date; Then it tells the client which path ACK frames should be sent on. We add a *CHANGE_RETURN_PATH_FRAME* control frame to designate an ACK path for each downlink path explicitly. Note that the ACK path ID here is transparent to the congestion window and each round trip path is identified by the downlink path ID.

Reducing RTT brings benefits of better loss recovery because the protocol needs less time to detect a packet loss event. It is also more friendly to latency sensitive applications.

4 EVALUATION

We evaluate our system with both controlled experiments in simulated environments and a real-world situation where a mobile device uses WLAN and LTE interfaces together. Note that our experiments are only for prototype validation. We will do a thorough evaluation in the future. We implement our scheduler based on the prototype of mp-quic [11], with 1187 lines of golang code. mp-quic is implemented based on the golang version of QUIC quic-go [12]. Neither mpquic nor quic-go supports HTTP/2 dependency tree or any other prioritization mechanism. They simply use a round-robin scheduler for streams. We add necessary interfaces for stream prioritization and use Weighted Round Robin as our baseline.

4.1 Experimental Setup

For the controlled experiments, we use Mininet simulation platform [13] running on GIGABYTE MKLP7AP-00 with i7-6500U. We use two paths on behalf of LTE and WIFI paths, but our scheduler is scalable to multiple paths. We empirically choose two scenarios 1) two paths with the same bandwidth 5 *Mbps* and One Way Delay (OWD) 10 *ms* and 2) two paths with different bandwidth 3 *Mbps*, 7 *Mbps* and different OWD 10 *ms*, 50 *ms* respectively. The aggregated bandwidth remains at 10 *Mbps*. We also do the experiment in these two scenarios under a non-negligible path loss of 2%. Fig. 7 shows our real-world experiment's testbed.

Server. We deploy MPQUIC on the Alibaba Cloud with 4 vCPU, 16GB RAM, 100 *Mbps* peak bandwidth.

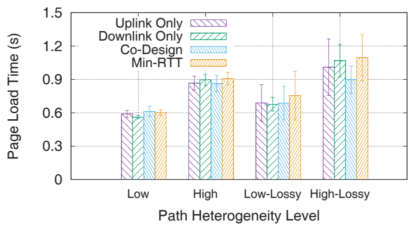
Client. We tether one Android phone (Xiaomi 6) to a laptop (Dell Latitude E7470) via USB 3.0. The laptop is connected to WiFi AP in CERNET [14]. The phone is equipped with a SIM card of one popular mobile carrier in China.

4.2 Experimental Results

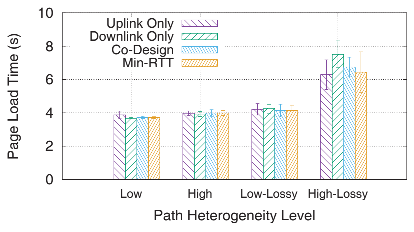
We evaluate our stand-alone uplink/downlink scheduler and the co-designed scheduler. Then we compare them with the default MIN-RTT scheduler. Schedulers without our downlink optimization are combined with the Weighted Round Robin stream scheduler. We summarize the path parameters

HETEROGENEITY	PATH 1	PATH 2	NOTE
Low	5 Mbps Bandwidth, 10 ms OWD, 0 PLR	5 Mbps Bandwidth, 10 ms OWD, 0 PLR	All parameters are same
Low and Lossy	5 Mbps Bandwidth, 10 ms OWD, 2% PLR	5 Mbps Bandwidth, 10 ms OWD, 2% PLR	Packet Loss Rate is non-negligible
High	3 Mbps Bandwidth, 10ms OWD, 0 PLR	7 Mbps Bandwidth, 50 ms OWD, 0 PLR	Bandwidth and OWD varies
High and Lossy	3 Mbps Bandwidth, 10ms OWD, 2% PLR	7 Mbps Bandwidth, 50 ms OWD, 2% PLR	Packet Loss Rate is non-negligible

Table 1: Parameter setting for experiments.



(a) Google



(b) Amazon

Figure 3: Simulation Experiments.

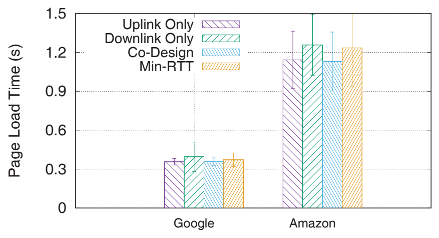
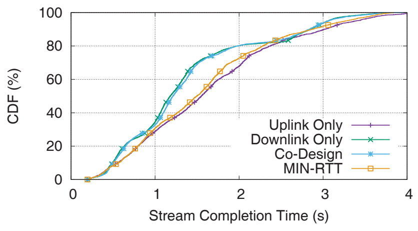
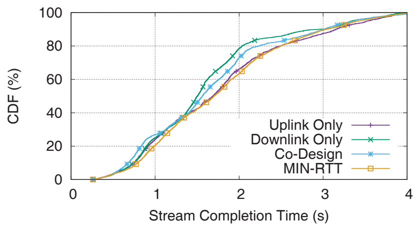


Figure 4: Real-World Experiments.



(a) Low Heterogeneity Paths.



(b) High Heterogeneity Paths.

Figure 5: Simulation Experiments.

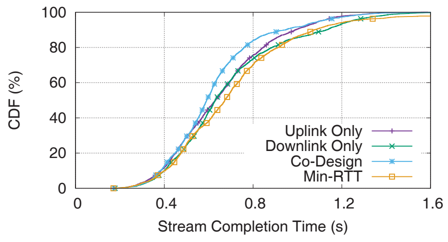


Figure 6: Real-World Experiments.

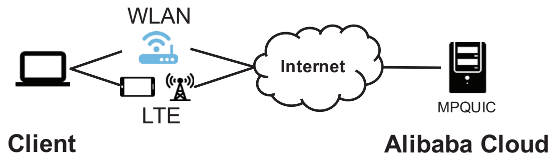


Figure 7: Evaluation testbed.

of interest, including bandwidth, one-way delay, and packet loss rate (PLR) in Tab. 1.

We first capture copies of web pages supporting HTTP/2 from two famous real-world websites in Tab. 2 (a). We use webpagetest [15] to download all the objects and their dependency tree, and then deploy a simple QUIC server in golang to host the pages. Then we write a client in golang to fetch HTTP/2 websites. We evaluate the impacts of different schedulers by analyzing the completion time of all the web objects (*i.e.,* Page Load Time) and the completion time of each stream under different network scenarios. Note that the completion time of a stream depicts the duration from the page request to the stream completion. Reducing stream completion time can potentially speed up the page loading procedure in a browser. For example, a browser can parse completed CSS and JavaScript files earlier. Sequential computing is time-consuming on a mobile device and sometimes can block the delivery of the following objects. Thus, reducing the stream completion time can speed up the loading of the whole page and benefit user's QoE.

Page Load Time. Fig. 3 shows the results of Page Load Time (PLT) for Google and Amazon under different scenarios simulated by Mininet. For the Google web page, uplink scheduler benefits significantly in the loss scenarios where two paths are heterogeneous. It can reduce up to 7.8% PLT for Google. It is because decreased RTT leads to lower RTO; thus, a packet loss can be detected in a shorter time compared to default MPQUIC. Note that retransmission of a lost packet has already performed on the fastest path in MPQUIC, we keep this feature in our method to recover loss fast. Our downlink scheduler reduces the PLT by 11-78 ms because our scheduler lets the stream wait for the faster path if it can complete earlier by waiting instead of being transmitted on the slower path, similar to ECF. The Co-designed scheduler combines the uplink and downlink scheduler, and it reduces up to 18.2% of PLT. Amazon has a large number of objects, and in this case, scheduler spends almost the same PLT with WRR + MIN-RTT. But we believe it can benefit more in real browsers since the stream sequence will have a big impact on PLT because of the blocked local computation. We repeat the experiments in the real world. Results in Fig. 4 shows that our co-designed scheduler reduces page load time by 3.9% for Google and 8.5% for Amazon. The reduction of PLT mainly attributes to the ACK offloading from WIFI to LTE.

Stream Completion Time. We analyze the stream completion time and draw the CDF of all the objects from Google and Amazon for two different scenarios, where two paths are under different heterogeneities. As shown in Fig. 5, our scheduler significantly outperforms the MIN-RTT scheduler. Because we exclusively transfer each stream so it can complete faster. Note that we remain fair because the stream completion sequence in our scheduler is the same as the

WRR algorithm. We achieve this by pre-calculating the sequence and transferring streams exactly as we calculated. When paths are heterogeneous, our scheduler is likely to estimate bandwidth inaccurately and has lower benefits than in low heterogeneity scenario. We repeat the experiments in the real world. Results in Fig. 6 show that the co-designed scheduler can reduce stream average completion time by up to 12.9% for all objects. ACK offloading reduces RTT on WIFI path, resulting in estimating bandwidth more accurately, thus it further reduces the stream completion time.

5 DISCUSSION

Dependency Tree. The structure of the dependency tree can have significant impacts on the web page processing procedure since it can affect the stream completion sequence. The question of which policy is more beneficial remains to be explored/answered. Some works [16] [17] focus on the order of web objects to optimize QoE but they are not designed for HTTP/2. Up to now, browsers use different policies to build dependency trees, and servers decide how to use the priority hints. However, Jiang et al. [18] showed that fewer HTTP/2 servers than expected had implemented priority mechanisms. Our scheduler implements a priority mechanism by calculating stream delivery sequences using the dependency tree, and can work better when the client has a better policy on stream priority.

Non HTTP alike Traffic. Our scheduler is optimized for web applications and takes advantage of stream length and priority information from the client. When no such information is available for non-HTTP alike traffic, our scheduler compatibly falls back to default round robin algorithm.

Mobility. When mobility exists, the bandwidth and RTT will dynamically change with time. Estimating path bandwidth and RTT inaccurately would be harmful to page loading. The scheduler needs to adjust the quota on each path dynamically to solve the problem, and we leave this as future work.

6 RELATED WORK

A number of multipath packet schedulers have been proposed to reduce out-of-order packets and improve the overall performance. STMS [19] schedules packets strategically to make them arrive at the client in order. Earliest Completion First (ECF) scheduler [20] utilizes information including estimated RTT, bandwidth and available data to decide whether to wait for a faster path. It solves the problem of faster paths being potentially blocked by slower ones due to path heterogeneity. For a single stream, our optimization goal is similar to ECF, but we take stream length into consideration and apply a byte limit on paths instead of blindly waiting. Our scheduler is also aware of multiple streams along with their priorities. We also propose uplink scheduling.

MPQUIC's [7] proposing, being aware of streams and flexible enough to schedule frame and packets, further provides the opportunity to optimize the scheduler. To our knowledge, few studies in recent years have done work on optimizing schedulers. MPQUIC uses the min-RTT algorithm to choose a path for each packet. Rabitsch et al. [21] proposed a stream-aware scheduler for heterogeneous paths. It allocates a fair share of the aggregated bandwidth to each stream according to HTTP/2's dependency tree and determines which path the stream should be sent on. But its sending path for a stream will not be changed once assigned, and several streams are delivered on the same path simultaneously, delaying the local computation and causing relatively long page load time. Our work differs from theirs because we exclusively transfer each stream in the order we expected. We also make each stream complete sending at the same time on different paths to minimize stream completion time.

Website	# Objs	Total Size (MB)
Google	14	0.39
Amazon	256	3.76

(a) Website

Path	WIFI	LTE
Bandwidth (Mbps)	57.4 ± 7.0	65.9 ± 6.9
RTT (ms)	214 ± 54	77.8 ± 30

(b) Path

Table 2: Website and path parameter.

7 CONCLUSION

It becomes more and more popular to browse websites on mobile devices. User-space implementation makes deploying MPQUIC, a protocol designed to improve web browsing experience, on mobile phones extensively possible. Optimizing the scheduler over MPQUIC can significantly improve HTTP/2 performance. Our co-designed scheduler estimates the path bandwidth and RTT, and it also considers stream priority. The scheduler allocates an appropriate number of bytes to be transferred on each path for each stream, making all the paths complete transmission for a stream at the same time. Each stream will be transmitted exclusively in a predefined order. We keep it fair by pre-determining the stream sequence according to both stream priority and length, and transferring streams to make their completion sequence exactly the same as we expected. Experimental results show that both our uplink and downlink schedulers can significantly reduce stream completion time and page load time.

ACKNOWLEDGMENTS

This work is supported in part by National Natural Science Foundation of China (Grant No. 61802007).

REFERENCES

[1] Mobile web browsing overtakes desktop for the first time. https://www.theguardian.com/technology/2016/nov/02/mobile-web-browsing-desktop-smartphones-tablets, Nov 2016.

[2] Http/2 dashboard. http://isthewebhttp2yet.com/.

[3] Yi Liu, Yun Ma, Xuanzhe Liu, and Gang Huang. Can http/2 really help web performance on smartphones? In *IEEE SCC*, 2016.

[4] Eric Schurman and Jake Brutlag. Performance related changes and their user impact. In *velocity web performance and operations conference*, 2009.

[5] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. How far can client-only solutions go for mobile browser speed? In *ACM WWW*, 2012.

[6] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. A first look at quic in the wild. In *Springer PAM*, 2018.

[7] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *ACM CoNEXT*, 2017.

[8] Tobias Viernickel, Alexander Froemmgen, Amr Rizk, Boris Koldehofe, and Ralf Steinmetz. Multipath quic: A deployable multipath transport protocol. In *IEEE ICC*, 2018.

[9] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext transfer protocol version 2 (http/2). Technical report, 2015.

[10] draft-ietf-quic-transport-14. https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-14.

[11] Quentin De Coninck et al. Multipath quic. https://github.com/qdeconinck/mp-quic, 2018.

[12] Lucas Clemente et al. A quic implementation in pure go. https://github.com/lucas-clemente/quic-go, 2018.

[13] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *ACM CoNEXT*, 2012.

[14] Xing Li, Congxiao Bao, Maoke Chen, Hong Zhang, and Jianping Wu. The china education and research network (cernet) ivi translation design and deployment for the ipv4/ipv6 coexistence and transition. Technical report, 2011.

[15] Patrick Meenan. Webpagetest. https://www.webpagetest.org/.

[16] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. Speeding up web page loads with shandian. In *USENIX NSDI*, 2016.

[17] Weiwang Li, Zhiwei Zhao, Geyong Min, Hancong Duan, Qiang Ni, and Zifei Zhao. Reordering webpage objects for optimizing quality-of-experience. *IEEE Access*, 2017.

[18] Muhui Jiang, Xiapu Luo, TungNgai Miu, Shengtuo Hu, and Weixiong Rao. Are http/2 servers ready yet? In *IEEE ICDCS*, 2017.

[19] Hang Shi, Yong Cui, Xin Wang, Yuming Hu, Minglong Dai, Fanzhao Wang, and Kai Zheng. {STMS}: Improving {MPTCP} throughput under heterogeneous networks. In *USENIX ATC*, 2018.

[20] Yeon-sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. Ecf: An mptcp path scheduler to manage heterogeneous paths. In *ACM CoNEXT*, 2017.

[21] Alexander Rabitsch, Per Hurtig, and Anna Brunstrom. A stream-aware multipath quic scheduler for heterogeneous paths. In *ACM EPIQ*, 2018.