SMig: Stream Migration Extension For HTTP/2

Xianghang Mi
Indiana University
Bloomington, IN
xmi@iu.edu

Feng Qian
Indiana University
Bloomington, IN
fengqian@indiana.edu

Xiaofeng Wang
Indiana University
Bloomington, IN
xw7@indiana.edu

ABSTRACT

HTTP/2 is quickly replacing HTTP/1.1, the protocol that supports the WWW for the past 17 years. However, HTTP/2's connection management and multiplexing schemes often incur unexpected cross-layer interactions. In this paper, we propose SMig, an HTTP/2 extension that allows a client or server to migrate an on-going HTTP/2 stream from one connection to another. We demonstrate through real implementation that SMig can bring substantial performance improvement under certain common usage scenarios (*e.g.,* up to 99% of download time reduction for small delay-sensitive objects when a concurrent large download is present).

CCS Concepts

•**Networks** → **Application layer protocols;**

Keywords

HTTP/2; Stream Migration; Head-of-line Blocking

1. INTRODUCTION

HTTP, the key protocol supporting the World Wide Web, has been evolving. Currently, the most widely deployed HTTP version is HTTP/1.1 [16] standardized 17 years ago. As web pages became rich and complex, HTTP/1.1 started to exhibit performance issues. To address them, several new web protocols have been proposed recently. In particular, HTTP/2 [10], the next version of HTTP, has been standardized in 2015 and is replacing HTTP/1.1 quickly. From July 2015 to June 2016, the fraction of websites using HTTP/2 has increased from 0.3% to 8.1% [8]. Within the Alexa top 100 websites, 31% of them support HTTP/2 [25].

HTTP/2 introduces several new features such as a binary protocol format, header compression, and server push. Among them, a particularly notable feature is *multiplexing*, which consolidates multiple concurrent requests into a single TCP connection. By contrast, HTTP/1.1 only supports serving requests sequentially over a TCP connection[1]. Therefore, HTTP/1.1 needs multiple concurrent connections to support concurrency. Prior studies have shown multiplexing often effectively improves the page load time [27, 26, 23]. However, studies also revealed that multiplexing may incur undesired interaction with other layers, leading to suboptimal performance. Examples include being vulnerable to losses [26], poorly interacting with the cellular radio state machine [15], and under-utilizing the network bandwidth [23]. Note many of these issues also exist in other multiplexing-based web protocols such as QUIC [13].

In this paper, we propose SMig (Stream MIGration extension), an extension that improves the performance and enables new use cases for HTTP/2. In the current HTTP/2 paradigm, (1) objects are usually multiplexed over a single connection; (2) an object transfer including both its request and response must be bound to the same connection. SMig instead allows the delivery of an object to be *migrated* from one connection to another at any time. For example, the request and response can be delivered over different connections; or all bytes except for the first 100KB of an object can be migrated to a different connection. We make the following contributions throughout this paper.

- • In §2, we motivate SMig using concrete use cases. (1) When large and small objects are multiplexed together, HTTP/2 suffers from severe sender-side head-of-line (HoL) blocking, toward which no effective solution exists. (2) We found that (explicitly or implicitly) canceling a large file download over HTTP/2 incurs a significantly larger amount of wasted traffic compared to doing so over HTTP/1.1. This can be effectively addressed by SMig. (3) We also describe other new use cases enabled by SMig such as adaptive multipath.
- • In §4, we present the design of SMig, which is lightweight, backward compatible, and incrementally deployable. In SMig, a migration can be initiated by either the server or a client, and the migration incurs no additional delay under the common usage scenario. A possible concern of SMig is that it creates additional connections. However, their numbers are still significantly less than those in HTTP/1.1. More

CoNEXT '16, December 12-15, 2016, Irvine, CA, USA

DOI: http://dx.doi.org/10.1145/2999572.2999583

[1] An exception is HTTP Pipelining, which has various limitations and was not widely deployed.

importantly, since usually only large objects are migrated, the connection management overhead is amortized by the large transfer carried by a migrated connection.

• We have implemented a prototype of SMig and integrated it with our custom HTTP/2 client and server applications. In §5, we evaluate SMig over wired networks and commercial cellular networks. The results indicate that SMig can bring substantial performance improvement under certain common usage scenarios (*e.g.,* up to 99% of download time reduction for small delay-sensitive objects when a concurrent large download is present, and up to 90% of reduction of wasted traffic when canceling a large object transfer).

2. MOTIVATING EXAMPLES

We first give an overview of the multiplexing mechanism in HTTP/2. HTTP/2 encapsulates HTTP transactions into *streams* each carrying one transaction. A stream is a bidirectional flow of bytes consisting of a series of *frames* carrying the actual data. Each frame also contains a stream ID to allow the receiver to de-multiplex the streams. Next, we describe a few scenarios where applying SMig is beneficial.

2.1 Multiplexing Large and Small Objects

Our prior study [23] identifies an issue of the multiplexing mechanism in SPDY [6], the predecessor of HTTP/2. When a large object download (often delay-tolerant) is multiplexed with small object transfers (usually delay-sensitive), the latter's performance will degrade dramatically. This is attributed to the head-of-line (HoL) blocking at the TCP send buffer that significantly affects HTTP/2 performance. Due to multiplexing, large and small transfers *share* the same send buffer that is inherently FIFO. Therefore, small transfers will be blocked by the large download as long as the TCP send buffer size is not trivially small. Note this is different from another well-known HoL blocking happening at the *receiver* side caused by packet losses or severe out-of-order [26].

Here we further confirm that server-side HoL blocking exists in HTTP/2. We conducted experiments over a wide range of HTTP/2 server implementations (latest versions as of June 2016) including Nghttp2 1.11, LiteSpeed 5.0, Nginx 1.10.1, and H2O 2.0.0. We found all of them, many of which are production-level implementations, are vulnerable to sender-side HoL blocking. It is also worth mentioning that HTTP/2 can be deployed either at a web server or at a proxy. In the former scheme, objects belonging to the same domain are multiplexed together; while in the latter, all traffic from the same client browser is multiplexed, making the sender-side HoL blocking an even more severe issue: a large download will affect the entire browser's performance.

Next, we conduct measurements to reveal that for real websites, it is quite common that large and small objects are hosted under the same domain (so HTTP/2 will use the same connection to deliver them concurrently, causing potential sender-side HoL blocking). The measurement is performed as follows. We study the Alexa top-1500 websites. For each website, we crawl all objects that belong to the first three levels of its *website object tree*, which has its landing page's HTML file as the root node (Level 1). Within the tree, Object Y is a child of Object X if and only if X and Y have the same domain name and X contains the URL of Y (therefore X must be an HTML page and all objects in the tree have the same domain name as the landing HTML page). We first observe that the vast majority of objects are small, with the 25-th, 50-th, and 75-th percentiles measured to be 18KB, 40KB, and 115KB, respectively, across all websites. Prior studies [24] show that for mobile versions of web pages, their object sizes are even smaller. On the other hand, Table 1 lists the number of websites (*i.e.,* their object trees with up to three levels of objects) that contain at least one large object whose size is at least $X \in \{1, 5, 10, 25, 50\}$ MB. As shown, many websites contain large objects that are under the same domain name as the landing page. If such large objects are concurrently fetched with other small objects, sender-side HoL blocking will occur. As will be demonstrated in §5.1, even a mid-sized file of 1MB can cause severe HoL blocking.

Contain Objects with Size...	# Websites
$\geq$ 1MB	209
$\geq$ 5MB	43
$\geq$ 10MB	21
$\geq$ 25MB	10
$\geq$ 50MB	4

Table 1: Websites among the Alexa top-1500 sites that contain large objects in the first three levels of their website object trees.

Scenario	Tail Bytes
1. HTTP/1.1, cancel DL	150 KB
2. HTTP/2, cancel DL	2.5 MB
3. HTTP/2, close browser	150 KB

Table 2: Tail bytes in three scenarios.

2.2 Canceling HTTP/2 Download

We found that compared to HTTP/1.1, after a single HTTP/2 file download is canceled, the client may still receive a large number of bytes. To demonstrate this, we conduct three experiments under the same network condition (emulated 10Mbps link, ~80ms RTT): (1) cancel an on-going large HTTP/1.1 file download in Chrome browser while keeping the browser open, (2) cancel an on-going HTTP/2 download of the same size while keeping the browser open, and (3) cancel an HTTP/2 download by directly closing the browser. The cancellation is done manually as how a normal user would do. We then measure the number of bytes delivered to the client (we call them "tail bytes") after taking the cancellation action. As shown in Table 2, 2.5 MB of tail bytes appear in Scenario (2) while much fewer tail bytes appear in Scenario (1) and (3). The issue of Scenario (2) is also severe in cellular networks (up to a 20x difference compared to Scenario 3) as will be shown in §5.2. This causes bandwidth waste and monetary cost since cellular customers are billed by bytes.

When canceling a file download in HTTP/2, the client sends a **`RST_STREAM`** control frame to shut down its corresponding stream. However, the stream's underlying connection *cannot* be closed, because a TCP connection in HTTP/2 is persistent and long-lived [10] *i.e.,* it needs to be shared by all streams of the same domain (or all traffic of the entire browser, if a proxy is used). Therefore, although the server application stops delivering data of that closed stream to TCP, all remaining data in the TCP send buffer will still be transferred to the client. In contrast, when canceling a file download in HTTP/1.1 or by closing the browser, the TCP connection (with all data in its send buffer) is immediately torn down, leaving much fewer tail bytes to arrive.

It is important to note that download cancellation occurs frequently in the real world. It is not limited to user explicitly hitting a "cancel" button. Instead, it usually happens *implicitly*, such as user skipping a song in a media player, repositioning a video playback, navigating to another page before the current page finishes loading a large object, pausing background synchronization, *etc.* All above scenarios will incur more tail bytes in HTTP/2 than in HTTP/1.1.

2.3 Download Accelerator Using Multipath

Mobile devices are usually equipped with multiple interfaces. Samsung recently introduced a new feature called Download Booster to its Android devices, for accelerating large HTTP download ($>$ 30MB) using both Wi-Fi and cellular [5]. Download Booster is realized using concurrent HTTP byte range requests: requests for different ranges of a large file are simultaneously sent over two connections, one over Wi-Fi and one over cellular. This approach has two limitations. First, since the byte ranges are pre-calculated, Download Booster usually cannot achieve the optimal download time (*e.g.,* when cellular finishes sooner than Wi-Fi). Second, download booster does not work with files whose sizes are not known beforehand. In §4.1, we discuss how SMig can easily work with existing off-the-shelf multipath solutions to addresses the above limitations.

Summary. Due to multiplexing, connection management in HTTP/2 differs from that in HTTP/1.1 significantly. The HTTP/2 RFC [10] recommends that clients "should[2] not open more than one HTTP/2 connection to a given host and port pair". However, our findings indicate that blindly using the single connection may lead to performance degradation (§2.1), unnecessary network traffic (§2.2), and obstacles to realizing new use cases (§2.3). We describe how SMig can be used in the above scenarios in §4.1.

3. RELATED WORK

Our proposal complements existing work of modeling [27, 26], measuring [13, 25], optimizing [17, 15], and applying [12, 9, 14] emerging protocols such as HTTP/2, SPDY, and QUIC. Next, we further motivate SMig by explaining the limitations of existing solutions to the sender-side HoL blocking problem described in §2.1.

Stream Prioritization. HTTP/2 supports assigning to streams different priorities, which "can be used to select streams for transmitting frames when there is limited capacity for sending" [10]. Stream prioritization however does not help mitigate the sender-side HoL blocking when the shared buffer is at the lower (*e.g.,* transport) layer.

Shrinking TCP Buffer can mitigate the sender-side HoL blocking and reduce the tail bytes. But doing so may cause performance degradation as the TCP send buffer size limits the TCP congestion window that is often highly fluctuating. This makes estimating the right TCP buffer size difficult.

Priority Queue Support for TCP. Nowlan *et al.* proposed uTCP [21], which adds unordered delivery and multiqueue support to TCP. Leveraging uTCP, the server can avoid HoL blocking by directing large and small transfers to different queues. However, uTCP requires changing the OS kernel. More importantly, uTCP is incompatible with TLS cipher suites using chained encryption where a TLS record cannot be decrypted until all prior records are processed.

Packet Late Binding. In our prior work [23], we built TM^3, a multiplexing proxy without sender-side HoL blocking. The basic idea is packet late binding: TM^3 moves the multiplexer deep into the OS kernel so that an outgoing packet is not filled with real data until it exits shared buffers. This essentially "skips" shared buffers and thus eliminates the HoL blocking. However, as a general transport-layer proxy, TM^3 is designed to transparently multiplex concurrent HTTP/1.1 (or any short-lived) TCP connections, and it cannot optimize an HTTP/2 flow that has already been multiplexed at the application layer. Also, the late binding mechanism enforces restrictions on frame size and format. TM^3 also requires OS kernel modification.

Other Protocols. Sender-side HoL blocking may also occur at other shared buffers such as Qdisc and driver buffer. But their sizes are usually much smaller. For example, TCP Small Queues (TSQ [7]) can be applied to limit the per-connection Qdisc buffer size with little performance degradation. Also, HoL blocking occurs in other multiplexing-based web protocols such as SPDY [6] and QUIC [4]. QUIC employs UDP as the transport layer. Because UDP buffers are also FIFO, sender-side HoL blocking still exists [23].

4. SMig DESIGN

The Stream Migration Extension (SMig) is an extension for HTTP/2 allowing *an on-going stream to be migrated from one connection to another*. Either the server or a client can initiate a migration. As an application protocol extension, SMig only adds lightweight logic to HTTP/2 and requires no change to the underlying OS. Note that the HTTP/2 specification indeed permits extending the protocol by adding new frame types or new settings [10].

4.1 Usage Scenarios of SMig

A common usage scenario of SMig is to migrate the response of an HTTP transaction. Consider the issues described in §2.1 and §2.2. Suppose the client sends one or more HTTP/2 requests to the server over a multiplexed

[2] The word "should" in RFC means "there may exist valid reasons in particular circumstances to ignore a particular item" [11].

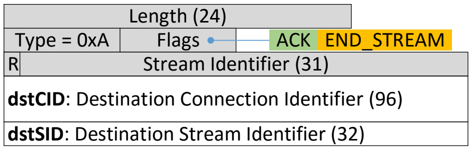
Length (24)
Type = 0xA
Flags
ACK
END_STREAM
R
Stream Identifier (31)
dstCID: Destination Connection Identifier (96)
dstSID: Destination Stream Identifier (32)

Figure 1: Format of the **MIGRATE** frame. The gray fields belong to the fixed 9-byte header of any HTTP/2 frame.

connection. If any of the requested object(s) are large and there are a non-trivial number of small objects being (or to be) transferred over the same connection, then the server will initiate migrations by moving each large object to a separate connection so that small and large objects are not mixed together. In other words, in this scenario, for a migrated stream, (1) its (usually small) request and large response are delivered over different connections, and (2) its large response uses a dedicated connection. Note the entire migration process is transparent to the upper layer. As a result, (1) as large transfers are migrated to different connections with separate buffers, they will not create HoL blocking for small object transfers; (2) since each large transfer now uses a dedicated connection without being multiplexed, to cancel its download, its underlying connection can be directly torn down, leading to much fewer tail bytes.

A counter-argument toward the above solution is, for large objects, if the client can initiate their requests over separate connections in the first place, then why do we still need SMig? The key reason is that *it is difficult for a client to know the size of an object beforehand*, so the client has to follow the default paradigm by sending all requests over a multiplexed connection. On the other hand, the server usually knows the sizes of their hosted objects (the vast majority of objects have the "Content-Length" field in their response headers). This makes it trivial for the server to make the migration decision based on the object size. Nevertheless, there do exist objects whose precise sizes are not known by server beforehand. We discuss how they can be handled in §4.4.

SMig can also be leveraged to enable new use cases for HTTP/2. For example, to overcome the limitations of Download Booster (§2.3), SMig can be used with MPTCP [3], the *de-facto* multipath solution with off-the-shelf Linux implementation. MPTCP transparently splits the byte stream of a TCP connection into multiple coupled paths (*e.g.,* one over Wi-Fi and one over cellular). However, it is well known that MPTCP provides little benefit for small file download [18]. The server can thus adopt the following strategy to use MPTCP in an adaptive manner. By default, for saving energy, only single path TCP over Wi-Fi is used for file download. If the server finds the file to be large, it can employ SMig to migrate the transfer to an MPTCP connection that reduces the overall download time.

4.2 Stream Migration

We begin with introducing a new type of control-plane frame defined by SMig. A **MIGRATE** frame is used to expresses the intent of a stream migration, or to acknowledge a migration initiated by a peer. As shown in Figure 1, there

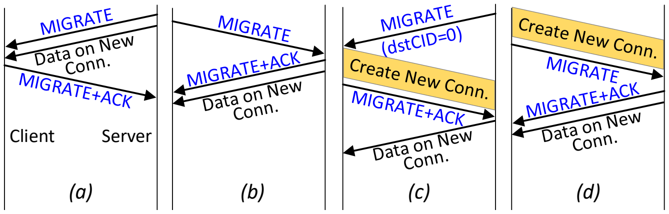

Figure 2: Four stream migration scenarios (only downlink data of the stream after the migration is shown).

C1, S1	C1, S1	C2, S2	C1, S1	C1, S1	C2, S2
Frame 1	Frame 2	Frame 4	Frame 3	MIG-ACK	Frame 5

Figure 3: Frames received by the server during migration.

are two key fields in a **MIGRATE** frame: **dstCID** is the connection identifier (**CID**) of the destination connection that the stream is migrated to; **dstSID** is the stream identifier (**SID**) that the stream will be assigned to after it migrates to **dstCID**. **SID** and its numbering convention are already defined in the HTTP/2 specification. In SMig, **CID** identifies a connection. It is a 96-bit number generated when a connection is established, as to be detailed in §4.3. Note that **SID** is defined within the namespace of **CID** so a stream between two hosts is uniquely identified by (**CID**,**SID**).

The gray fields in Figure 1 belong to the fixed 9-byte common header of any HTTP/2 frame. The "Stream Identifier" field is the **SID** of the stream (in its original connection) to be migrated. The "Flag" field defines two types of flags. The **END_STREAM** flag can be set to half-close a stream to be migrated (an HTTP/2 stream is bidirectional; either side can close the one-way data channel to its peer). The usage of the **ACK** flag is explained next.

We now detail the stream migration procedure. Suppose a stream with **SID**=S_1 on connection **CID**=C_1 needs to be migrated. We first assume (1) the migration is initiated by the server (the common case), and (2) there exists an idle connection **dstCID**=C_2 that the stream can be migrated to. Note that neither assumption is mandatory for SMig. We will describe client-initiated migration and the scenario where no idle connection exists soon. The message exchange is shown in Figure 2(a). The server first generates an unused stream ID **dstSID**=S_2 within C_2. It then sends the corresponding **MIGRATE** frame with **ACK**=0 to the client over S_1 on C_1. The main purpose of the **MIGRATE** frame is to inform the peer of the **dstSID** and **dstCID** so that the migrated stream can be seamlessly handled. **MIGRATE** also ensures the cross-connection ordering of frames by marking the last downlink (server to client) frame transferred over the old connection (example shown soon). All subsequent downlink frames must be transferred over the new connection C_2 using the new **SID** S_2. Also as shown, the data over the new connection can be piggybacked with the **MIGRATE** frame.

Upon the reception of the **MIGRATE** frame, the peer (in this case, the client) acknowledges it by sending an identical **MIGRATE** with **ACK**=1. Despite TCP ensures reliable delivery of the original **MIGRATE** frame, this **ACK** is still needed because (similar to the downlink case) as the last uplink frame transferred over (C_1,S_1), this MIGRATE with ACK ensures the cross-connection ordering of frames. Figure 3 shows a possible sequence of uplink frames received by the server. As shown, Frame 3 and 4 are out of order. The server needs to buffer all frames received on the new connection (Frame 4 in this example) until the reception of MIGRATE with ACK, without which the server has no way to know when the old stream ends. Note frames do not have sequence numbers; their ordering *within* a connection is guaranteed by TCP.

Despite not a common use case, a migration can also be initiated by a client by following a similar procedure: the client sends a MIGRATE and the server acknowledges it with a MIGRATE with ACK, as shown in Figure 2(b). Note regardless of who initiates the migration, the connection that carries the migrated object is always initiated by the client.

Next we consider scenarios where there does not exist an idle connection. If the client initiates the migration, it first creates the connection before sending MIGRATE, as shown in Figure 2(d). If the server initiates the migration, it sends a MIGRATE with dstCID=0. The client will then create a new connection on behalf of the server before sending a MIGRATE with ACK and a valid dstCID, as shown in Figure 2(c).

Migration Overhead. We analyze the migration overhead for the scenarios in Figure 2. We consider the delay incurred by migration on the downlink data (the uplink cases are symmetric). In plot (a), when the server initiates a migration to an existing connection, the migration takes no additional delay as the data over the new connection can be piggybacked with the MIGRATE frame. The same case happens in (b) except that the migration will be delayed by one RTT. If the connection needs to be created, the overhead is higher: in both (c) and (d), it takes the connection establish time (including the SSL/TLS handshake delay) plus one RTT before data appears on the new connection. During this period, data transmission of the stream being migrated is paused. However, note that since usually only large objects are migrated, such delay is dwarfed by the long object transfer time. To eliminate the connection establishment overhead, a client application can always be ready for migration by maintaining one idle connection for certain (host, port) pairs (detailed in §4.3). Note SMig never affects the performance of objects that are *not* migrated.

4.3 Other Design Considerations

CID Generation. When creating a new HTTP/2 connection, an SMig-capable client generates a 96-bit CID, and embeds it to a SETTINGS frame sent to the server. A CID has two parts. The first 64-bit string is called AppID, which is used for distinguishing multiple applications (apps) on the same host; the last 32 bits identify a particular connection within an app. When generating a CID, a client app thus needs to ensure that (1) the CID is unique among all currently established connections to the same (host, port) pair, and (2) all CID belonging to the same app session have the same AppID. When a stream is migrated from one connection to another, their CID must have the same AppID. This prevents a stream from being migrated to a connection belonging to a different app running on the same client. The SETTINGS frame also ensures backward compatibility: an SMig-capable server must ACK it so both sides know SMig is enabled; otherwise it will be ignored per HTTP/2 specification. The SETTINGS frames can be piggybacked with the very first request and response so they do not incur additional delay.

Internal State Migration. When a stream migrates to a new connection, its internal states are migrated together with the stream. They include header compression state [22], flow control state, stream priority *etc.* Note that there is no need to migrate the states at TLS and TCP layers. Optimization can be made though to allow a migrated connection to cache a subset of the old connection's lower-layer states (*e.g.,* certain congestion control parameters) for better performance.

Idle Connection Management. As described in §4.2, a client can optionally "cache" idle connections to reduce the migration overhead. One issue here is to decide the set of domains whose idle connections will be cached. Here the tradeoff is between performance and connection management overhead. Consider two extreme cases. Caching an idle connection for every domain essentially doubles the number of connections, while not performing any caching may slightly delay a migration as illustrated in Figure 2(c)(d). A possible strategy here is to leverage historical information to predict on which domains migrations are more likely to happen. Then the browser will only create additional idle connections for those domains.

Interplay with Server Push. Server push is a new feature introduced in HTTP/2. Using server push, a server can preemptively push (*i.e.,* send) resources to a client without requiring the client to request for the resource. Server push enables early resource discovery and thus can potentially reduce the page load time [17]. To push an object, the server first sends a PUSH_PROMISE frame on an existing client-initiated stream. The PUSH_PROMISE frame contains a *Promised Stream ID* as well as the header information of the to-be-pushed object. After that, the server initiates the new stream over which the object's data is transferred. SMig works well with server push. The procedure for migrating a pushed object is largely the same as that for migrating a regular object as illustrated in Figure 2. Note if the migration is initiated by the server, the server must send the MIGRATE frame after the PUSH_PROMISE frame, and set the stream identifier in the MIGRATE frame to be the ID of the promised stream, which is the stream to be migrated.

Simultaneous MIGRATE. Consider a corner case where both sides send MIGRATE frames simultaneously for the same stream. They can be reconciled if their dstCID are the same (or server-side dstCID is zero) despite their dstSID being different (details omitted). To avoid the case where two simultaneous MIGRATE frames contain different dstCID, SMig requires that when multiple idle connections are available, the one with the smallest CID should be picked as dstCID.

Security. To our knowledge, SMig brings no new security vulnerability to HTTP/2. A possible concern is information leak: an adversary can infer a possible migration by observing data being sent over a new connection. It can further infer that, for example, the client is downloading large files. Nevertheless, we believe this is not a big concern because the leaked information is insignificant.

4.4 The Migration Policy

SMig provides the protocol support for stream migration. Applications (*e.g.,* web server) also need migration *policies*, which are expected to be simple, concise, and easy to configure. We next exemplify policies for mitigating sender-side HoL blocking and reducing tail bytes.

The policy executes on the server side. When an HTTP request arrives, the server checks the object size s, and the number of on-going and pending HTTP transactions on the same connection n. A simple policy is to invoke migration if both s and n are larger than pre-defined thresholds. A more adaptive policy is to further consider the network condition: the server measures the network bandwidth b (*e.g.,* using existing methods [19]) and performs migration when both s/b and n are larger than pre-defined thresholds where s/b is the estimated HoL blocking time (an upper bound).

The above policy assumes that the object's size s is known by the server. To handle the less common case where the file size is not known, the server can use robust heuristics to roughly estimate the size. Note even in this case, the server has more knowledge than the client so server-initiated migration is still helpful. An alternative approach is to allow up to t bytes (a pre-defined threshold) of an object with unknown size to be multiplexed into the existing connection. If the object size turns to be larger than t, the remaining part of the object will be migrated. This approach does not need file size estimation but it may incur slight HoL blocking caused by the first t bytes.

5. IMPLEMENTATION AND EVALUATION

We have implemented our custom HTTP/2 client and server, which are user-level applications for Linux/MacOS ($\sim$7.5K C++ LoC). They conform to the HTTP/2 specification except that a small number of advanced features such as server push were left as future work. We then implemented the SMig extension ($\sim$1K LoC) and integrated it with our client and server.

Our evaluation testbed consists of the following. The client is a commodity Macbook with 2.7GHz Intel Core i5 CPU and 8GB memory; the server is a Ubuntu 14.04 machine with 3GHz Intel Core2 Duo E8400 CPU and 4GB Memory. We use default TCP settings unless otherwise mentioned. We conduct experiments over two types of networks: an emulated 10Mbps link with 50ms RTT and a commercial LTE network provided by a large cellular ISP. The cellular connectivity is provided to the laptop by a tethered LTE smartphone. We next use this testbed to evaluate SMig, focusing on addressing the issues described in §2.1 and §2.2 (the use case in §2.3 is more self-explained).

One limitation here is our experiments were not conducted on commercial browsers and servers (integrating them with SMig is our on-going work). Nevertheless, we believe SMig will work with them effectively given that the SMig logic is simple and does not depend on a particular server/browser implementation.

5.1 Mitigating Sender-side HoL Blocking

We first evaluate how well SMig mitigates sender-side HoL blocking under the following setting. The client keeps fetching a 10KB object every 1 sec. During this process, the client also fetches a large file (50 MB) using the four migration schemes: (1) *NoMig*: the large file is still multiplexed with small objects without migration; (2) *MigSW*: the migration is initiated by the server and the whole response is migrated immediately after the request is received by the server; (3) *MigSP*: the migration is initiated by the server and only part of the response (after the first 100KB) is migrated. This corresponds to the scenario where the large file's size is not known so migration is performed in a "lazy" manner; (4) *MigCP*: the migration is initiated by the client and part of the response (after the first 100KB) is migrated.

Figure 4 measures the small object download time over emulated wired network with the default server-side TCP send buffer configuration (min: 4KB, default: 16 KB, max: 1MB). We repeat the experiment for 10 times, each downloading 25 small objects after the large file's request is sent. We report the average download time across all runs (the variation is small). As shown, SMig dramatically reduces the download time for small objects, which are usually delay-sensitive, by up to 90%. Figure 6 repeats the above experiments over cellular network, and qualitatively similar results are observed.

Next, instead of using the default TCP send buffer, we increase it (min: 1MB, default: 4MB, max: 8MB) and do the experiments again, with results shown in Figure 5 and 7 for wired and cellular networks, respectively. Note that it is a common practice of network administrators to increase TCP buffers [2, 1] for improving the network performance. For HTTP/2, however, a key downside is that this exacerbates sender-side HoL blocking. In Figure 5 and 7, SMig reduces the small file download time by up to 93% and 92%, respectively. We observe *MigCP* leads to worse performance than other migration schemes do. This is because when a migration is initiated by the client, the server's send buffer already becomes heavily occupied, making small objects vulnerable to HoL blocking before the buffer drains.

Impact of Object Sizes. The above experiments use 10KB small objects. Figure 8 plots how SMig accelerates small file download with different sizes (wired network, large TCP buffer) when a concurrent large file download (50MB) is present. The Y axis corresponds to the ratio of the download time for *MigSW* to that for *NoMig*. As shown, decreasing the small object size makes SMig more effective *e.g.,* for 1KB small files, SMig reduces their download time by 95%. This is because when the size of small objects decreases, their download time without blocking is reduced while their HoL blocking time largely remains the same. Over cellular network, we observe even more download time reduction – up to 99% brought by SMig (figure not shown).

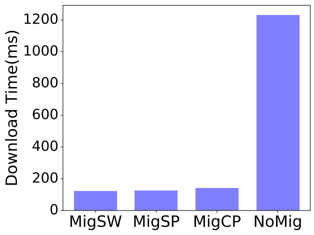

Figure 4: SMig's impact on small file download (default TCP buffer, wired)

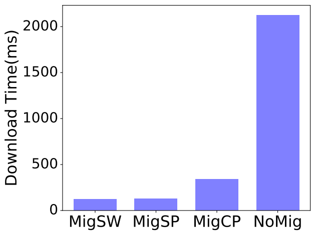

Figure 5: SMig's impact on small file download (large TCP buffer, wired)

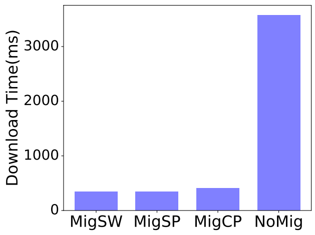

Figure 6: SMig's impact on small file download (default TCP buffer, cellular)

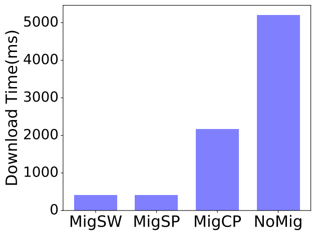

Figure 7: SMig's impact on small file download (large TCP buffer, cellular)

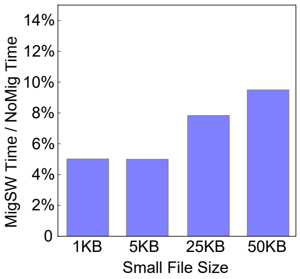

Figure 8: SMig's impact on small file download. Changing small file sizes (large TCP buffer, wired)

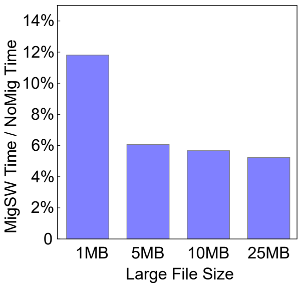

Figure 9: SMig's impact on small file download. Changing large file sizes (large TCP buffer, wired)

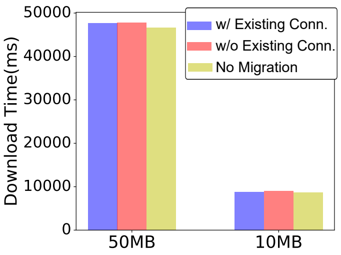

Figure 10: SMig's impact on migrated large file download time (default TCP buffer, wired network)

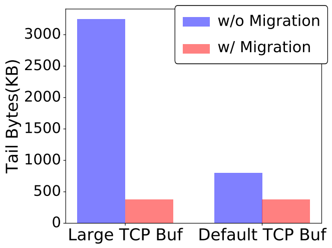

Figure 11: Impact of SMig on incurred tail bytes (cellular network, default/large TCP buffer)

On the other hand, when the large file's size changes, the HoL blocking duration changes accordingly. Figure 9 plots the impact of SMig on 10KB file download time when downloading different large files over wired networks. As shown, even when simultaneously fetching a mid-sized file of 1MB, SMig can effectively reduce the 10KB file download time by 88%. We observe similar results over LTE networks. For example, on LTE, even downloading a 10MB file over HTTP/2 (*e.g.,* an HD video chunk or a podcast audio) can block small objects for more than 5 seconds. This can be effectively mitigated by SMig.

Impact of Large File Download Time. SMig incurs no impact on objects that are not migrated. For migrated (large) objects, the performance impact of SMig on them is small. Figure 10 plots the large object (50MB and 10MB) download time under three scenarios: (1) no migration, (2) migrating the object to an existing idle connection (Figure 2(a)), and (3) creating a new connection and then performing migration (Figure 2(c)). The download time increase in Scenario (2) and (3) is less than 2.5% compared to Scenario (1). The slight increase is due to two reasons. First, in Figure 2(c), the large file download needs to be paused while the new connection is being established. Second, the new connection needs to experience an additional slow start.

5.2 Reducing Tail Bytes

Recall that in the current HTTP/2 paradigm, an HTTP/2 connection is by default long-lived. As a result, even when a large stream is closed, its data in the TCP buffer cannot be removed, leading to many tail bytes. After migrating a large object to a dedicated connection, canceling its download can be done by directly tearing down its connection, resulting in much fewer tail bytes. Figure 11 measures the tail bytes without and with migration for LTE network. When the TCP buffer is large, migration helps cut the tail bytes by 90%. Under the scenario of default TCP buffer, migration still reduces the tail bytes by 53%. The findings are consistent with the measurement over wired networks (Table 2). Note SMig cannot eliminate all tail bytes because they also include all "in-flight" bytes being transmitted in the network when the TCP connection is closed/reset. We also note that even with migration, cellular incurs more tail bytes than the emulated wired network does. Besides the apparent reason of their different bandwidth-delay products, another reason is that there exist buffers *inside* cellular networks causing the tail bytes to inflate [20]. But as shown, the main contributor of tail bytes is still the on-device TCP send buffer whose impact can be eliminated by SMig.

6. CONCLUDING REMARKS

We have designed and implemented SMig, a novel HTTP/2 extension that substantially improves the HTTP/2 performance in certain common usage scenarios. It also enables several new use cases for HTTP/2. SMig is backward compatible, incrementally deployable, and incurs negligible runtime overhead. We are currently working on adding SMig support to the Chrome browser and Nginx HTTP/2 server.

Acknowledgements

We would like to thank our shepherd, Ramesh Sitaraman, and the anonymous reviewers for their valuable comments and suggestions. This research was supported in part by the National Science Foundation under grant CNS-1566331.

7. REFERENCES

[1] How To: Network / TCP / UDP Tuning. https://wwwx.cs.unc.edu/~sparkst/howto/network_tuning.php.

[2] Linux Tune Network Stack (Buffers Size) To Increase Networking Performance. http://www.cyberciti.biz/faq/linux-tcp-tuning/.

[3] MultiPath TCP - Linux Kernel implementation. http://www.multipath-tcp.org/.

[4] QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic.

[5] Samsung Download Booster. http://www.samsung.com/uk/support/skp/faq/1061358.

[6] SPDY Protocol Version 3.1. http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-1.

[7] TCP Small Queues (TSQ). http://lwn.net/Articles/507065/.

[8] Usage of HTTP/2 for Websites. https://w3techs.com/technologies/details/ce-http2/all/all.

[9] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *NSDI*, 2015.

[10] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, 2015.

[11] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, 1997.

[12] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *NSDI*, 2015.

[13] G. Carlucci, L. D. Cicco, and S. Mascolo. HTTP over UDP: an Experimental Investigation of QUIC. In *ACM SAC*, 2015.

[14] W. Cherif, Y. Fablet, E. Nassor, J. Taquet, and Y. Fujimori. DASH fast start using HTTP/2. In *NOSSDAV*, 2015.

[15] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a SPDY'ier Mobile Web. In *CoNEXT*, 2013.

[16] R. Fielding, J. Gettys, J. Mogul, H. F. L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1 . RFC 2616, 1999.

[17] B. Han, S. Hao, and F. Qian. MetaPush: Cellular-Friendly Server Push For HTTP/2. In *All Things Cellular Workshop*, 2015.

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

[19] Q. He, C. Dovrolis, and M. Ammar. On the Predictability of Large Transfer TCP Throughput. In *SIGCOMM*, 2005.

[20] H. Jiang, Y. Wang, K. Lee, , and I. Rhee. Tackling Bufferbloat in 3G/4G Networks. In *IMC*, 2012.

[21] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Amin, and B. Ford. Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-Compatible with TCP and TLS. In *NSDI*, 2012.

[22] R. Peon and H. Ruellan. HPACK: Header Compression for HTTP/2. RFC 7541, 2015.

[23] 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.

[24] F. Qian, S. Sen, and O. Spatscheck. Characterizing Resource Usage for Mobile Web Browsing. In *MobiSys*, 2014.

[25] M. Varvello, K. Schomp, D. Naylor, J. Blackburn, A. Finamore, and K. Papagiannaki. Is The Web HTTP/2 Yet? In *PAM*, 2016.

[26] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? In *NSDI*, 2014.

[27] K. Zarifis, M. Holland, M. Jain, E. Katz-Bassett, and R. Govindan. Modeling HTTP/2 Speed from HTTP/1 Traces. In *PAM*, 2016.