CSCI-P 538 Fall 2016 Project 3
TM^3: A Multiplexing Proxy for TCP
(Engineering Track B)

A Deadline

December 2 2016 23:59:59 EST. This is a hard deadline and no extension will be given. Any clarification queries should be sent to `p538fall16-l@list.indiana.edu`.

B Teamwork

Up to two students can work together on this project. In other words, a team may consist of one or two students. In the latter case, both students must contribute to the project, and a detailed description of each student's role must be submitted (see §E.4).

C Introduction

Proxies, or middleboxes, are widely used in today's Internet, for boosting application performance, enhancing security, and providing other value-added features. Proxies can be deployed at different layers such as application (*e.g.,* a web proxy) or transport (*e.g.,* a TCP proxy).

In this project, you will build a transport-layer multiplexing proxy, called TM^3 (**T**ransparent **M**ulti-pipe **M**ultiplexing **M**iddlebox), that enhances TCP performance. As you already know, multiplexing is a commonly used technique in computer networks: multiple data streams are combined into one single logical or physical channel, in order to share its expensive resource. For example, TCP connections between two end hosts are multiplexed into a single end-to-end IP path (so we need port numbers); new web protocols such as HTTP/2 multiplex many HTTP transactions into the same TCP connection to increase the bandwidth utilization; in DSL, voice and data are also multiplexed onto the same physical link.

The motivation and full story of TM^3 are described in a research paper attached at the end of this document. You are strongly encouraged to read it before starting working on this project. Note that this course project is a highly simplified version of the original TM^3 work (more precisely, the features you are to implement only cover Section 3.2 and 3.3 in the paper). So from the technical implementation's perspective, you must follow this project description instead of the original research paper, whose purpose is just to provide you with the "big picture".

D The TM^3 Architecture

The multiplexing technique you are to implement is to *explicitly multiplex multiple TCP connections onto a fixed number of TCP connections*. The overall architecture is illustrated in Figure 1. TM^3 consists of two key components: a *local proxy* (LP) and a *remote proxy* (RP). LP resides on the client host, and RP is deployed on a middlebox. In cellular networks, for example, RP can be integrated with web proxies. In DSL, RP can be placed at broadband ISP's gateway. LP and RP transparently split an end-to-end TCP connection into three segments: *(i)* a *local connection*[1] between client applications and LP, *(ii)* one or more *pipes* between LP and RP, and *(iii)* a *remote connection* between RP and the remote server. The pipes serve as the transport layer between the client and the middlebox, thus covering the "last mile" that is usually the bottleneck link.

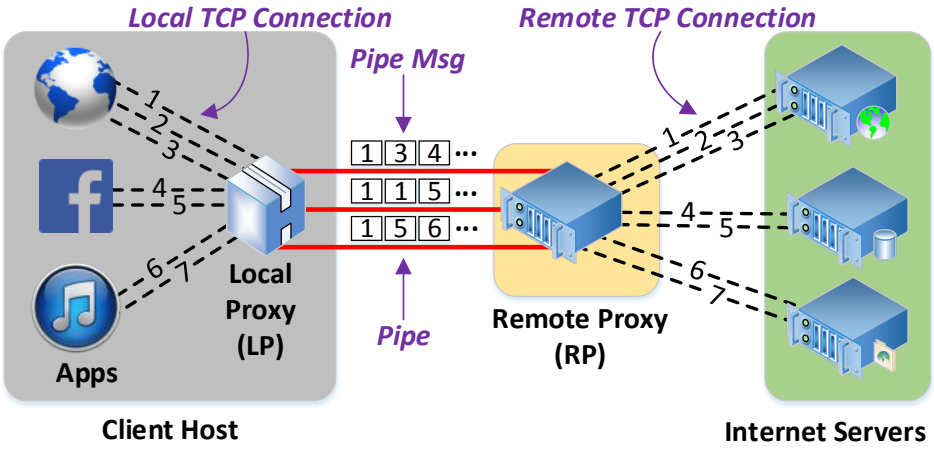

Figure 1: The TM^3 architecture.

Pipes are bidirectional and long-lived, and they are shared by all connections across all applications by multiplexing. Uplink traffic from client to server is multiplexed onto the pipes at LP, and de-multiplexed at RP (reversely for downlink traffic). Similarly, downlink traffic is multiplexed at RP and de-multiplexed at LP. Multiplexing and de-multiplexing are done transparently all applications, which, in other words, are completely aware of the underlying TM^3 architecture.

A *pipe message* is the atomic unit transferred on a pipe. As shown in Figure 2, there are three types of pipe messages to support basic data transfer over pipes: Data, SYN, and FIN. A data pipe message ("data message") carries user data; the other two messages are used for TCP connection management (described shortly below).

A data message has an 8-byte header including payload length, connection ID, and sequence number, followed by the actual payload. Connection ID (connID) is used to uniquely identify each connection. A valid connID is a positive 16-bit integer. The length of a message includes the 8-byte header, so it is at least 9, and at most $4096 + 8 = 4104$, since a pipe message can contain at most 4KB of data. An empty message that does not contain any data is not allowed. All header fields in TM^3 are encoded in little-endian.

[1]In the rest of this document, for brevity, a "connection" means an *application-issued* TCP connection unless otherwise noted.

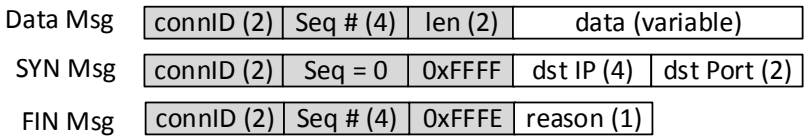

Figure 2: Pipe message format with headers (8 bytes) shaded.

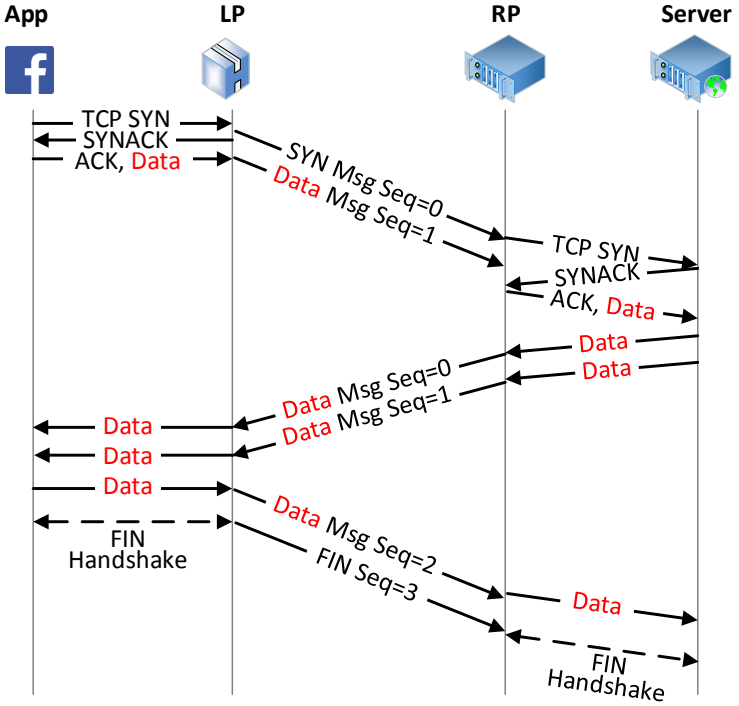

Figure 3: Example of data communication in TM^3.

Since pipe messages created from one TCP connection can be concurrently delivered by multiple pipes in an arbitrary order, each pipe message of the same TCP connection needs a sequence number that increases per message so received frames are never delivered out of order. The sequence numbers for both directions of a connection are independently maintained, and they always start from 0. The first uplink message of a connection is always a SYN message, which has a sequence number of 0. Note this differs from TCP sequence numbers that increase per byte, since byte-ordering within a pipe message is guaranteed by TCP that a pipe uses.

When a remote connection has any downlink data coming, the RP encapsulates the TCP payload into one or more data messages. For each message, the RP selects a pipe and transfers the message over that pipe. When an uplink data message arrives on a pipe, the RP extracts the TCP payload and delivers it to the corresponding remote connection, if the message's sequence number matches the expected sequence number of the remote connection. Otherwise the message is buffered and delivered when the sequence number gap is filled. Multiplexing and de-multiplexing at LP are performed in the same manner.

Next, we describe TCP connection management in TM^3.

Connection establishment. When LP receives a TCP SYN packet from an application, it immediately replies with SYN-ACK to establish the local connection. The LP subsequently sends a SYN pipe message (Figure 2) containing an unused connection ID, the remote host IP address, and the remote host port number to RP. Upon the reception of the SYN pipe message, the RP establishes the connection to the remote host on behalf of the client host. As shown in Figure 3, the local connection setup takes virtually no time. The SYN and data pipe messages are sent back to back. This helps reduce the connection setup delay over the pipes.

Connection closure. When LP receives a TCP FIN or RST packet from a local connection, it immediately terminates the local connection by completing the FIN handshake (no handshake for the RST case). Meanwhile, the LP sends a FIN pipe message, whose "reason" field (Figure 2) indicates whether the closure is caused by FIN (0x88) or RST (0x99), to the RP. Upon the reception of the FIN pipe message, the RP terminates the remote connection correspondingly. Connection closure initiated by server is handled similarly. Half-open connection is not supported by TM^3.

Note you do not need to worry about other TCP features such as flow control and congestion control, as they are already transparently handled by TCP.

E Problem Description

Your job is to implement the remote proxy (RP) in C/C++/Java. You will be given the binary code of the LP, as well as a model implementation of RP (also in binary), to help you debug/test your code.

E.1 Task 1: Set Up the Working Environment

You need to set up a dedicated Ubuntu virtual machine (VM) to run the LP, and a dedicated cloud server to host the RP, by following the instructions below. You must use 64-bit Ubuntu 14.04 for both LP and RP (installation image provided below), because my model implementations do not run on other Linux distributions.

LP Setup. Download the Ubuntu image from here: `http://www.cs.indiana.edu/~fengqian/ubuntu-14.04.3-desktop-amd64.iso`. Use a virtual machine software (*e.g.,* VMware or VirtualBox) to create the VM.

Log onto the VM and type the following commands (you need Internet connectivity):

```
sudo apt-get update
sudo apt-get install g++
```

If you are using VirtualBox and find the screen size is too small, do the following:

```
sudo apt-get install virtualbox-guest-dkms
sudo apt-get remove libcheese-gtk23
```

```
sudo apt-get install xserver-xorg-core
sudo apt-get install -f virtualbox-guest-x11
```

Reboot the VM, and then do

```
sudo apt-get install ubuntu-desktop
```

You do not need to install other updates prompted by Ubuntu. Also, follow the link to enable a shared folder between your guest Ubuntu and your host OS: `http://helpdeskgeek.com/virtualization/virtualbox-share-folder-host-guest/`.

(Note the above instructions are specifically for VirtualBox.)

RP Setup. Create an Amazon EC2 instance of Ubuntu Server 14.04 LTS. Make sure you use a "free tier" instance that gives you one-year free trial. Open TCP port 6000 that will be used by the pipes.

Log onto the server using the following command on any Linux terminal:

```
ssh -i key.pem ubuntu@52.23.220.80
```

where `key.pem` is the key pair name you downloaded when creating the EC2 instance. The default user name is `ubuntu`. `52.23.220.80` is the server's public IP address. Then type the following commands:

```
sudo apt-get update
sudo apt-get install g++
```

E.2 Task 2: Run the Model Implementation

We first copy the model implementation to LP and RP.

1. Copy `remote_proxy` to RP. This is the RP program.

2. Copy `local_proxy` to LP. This is the LP program.

3. Copy `tm3.c`, `Makefile`, and `tm3_mod.pl` to LP. Type `make` under the same directory. It will generate a file called `tm3.ko`. This is a Linux kernel module that transparently forwards traffic to LP.

Now let's conduct two experiments. Assume the RP's public IP address is 52.23.220.80. The first experiment is illustrated in Figure 4 where we download data from our home-made server using TM^3.

1. Copy `server.cpp` to the RP host. Build it by `g++ ./server.cpp -o ./server`. This is the server program. Start it by `./server 6001 10` (port 6001 needs to be opened using the EC2 management interface).

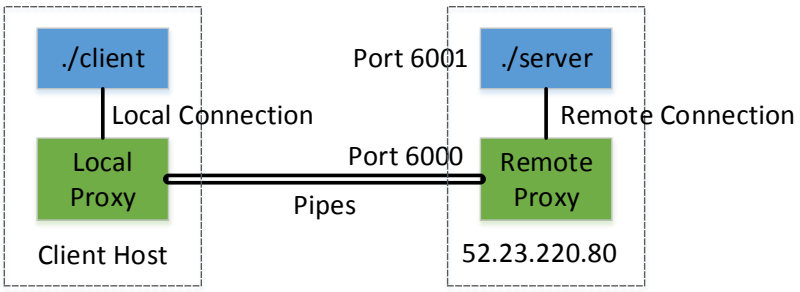

Figure 4: Experiment 1: download data from our home-made server using TM^3.

2. Copy `client.cpp` to the LP host. Build it by `g++ ./client.cpp -o ./client`. This is the client program we will be using. Start it by `./client 52.23.220.80 6001 10 4096 4096`. The client will receive 10 responses, each having a size of 4096 bytes, from the server.

3. Start RP by `./remote_proxy 4`. This starts RP with 4 pipes. Also start the server program by `./server 6001 10` if is not yet up. Note the server program and RP are co-located on the same host.

4. Start LP by `./local_proxy 52.23.220.80 4`. This starts LP with 4 pipes, and connect them with the RP. If you run the client program now, the LP and RP programs will not output anything. This is because the traffic is not yet forwarded to pipes.

5. On the LP host, do `sudo perl tm3_mod.pl 52.23.220.80 eth0 6001`. Ignore the error message "rmmod: ERROR: Module tm3 is not currently loaded". This will turn on traffic redirection *i.e.,* redirecting all `eth0` traffic from/to server port 6001 to the pipes. In general, the `tm3_mod.pl` script is used as follows. The first argument is the public IP address of RP, the second argument is the name of the primary interface, and the remaining arguments are the server ports whose redirection will be turned on.

6. Now, run the client again. You will see output produced by LP and RP. This indicates traffic was being carried by the pipes over port 6000. You can verify that using `tcpdump`.

7. Stop LP or RP by `Ctrl+C`.

8. To turn off traffic redirection, do `sudo rmmod tm3`. If you do not turn off traffic redirection, your traffic will not be sent out if TM^3 is not running.

The second experiment is illustrated in Figure 5 where we use TM^3 to fetch data from real web servers.

1. Start RP with four pipes: `./remote_proxy 4`

2. Start LP: `./local_proxy 52.23.220.80 4`

3. On the LP host, enable traffic redirection for port 80 (HTTP) and 443 (HTTPS): `sudo perl tm3_mod.pl 52.23.220.80 eth0 80 443`

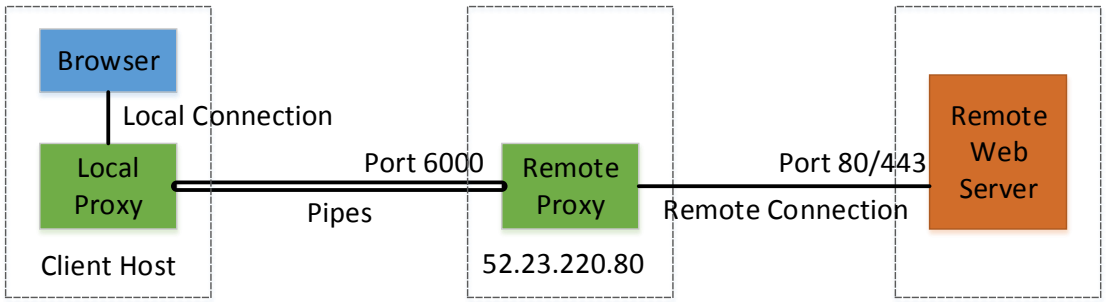

Figure 5: Experiment 2: browse the web using TM^3.

4. Now open a browser and type a URL. You will see output produced by LP and RP. This indicates web traffic was being carried by the pipes.

5. Stop LP or RP by `Ctrl+C`.

6. Turn off traffic redirection by `sudo rmmod tm3`.

When you run the above experiments, you are also encouraged to use `tcpdump` and `Wireshark` to study the TM^3 protocol format. This will help you better understand how TM^3 works.

E.3 Task 3: Build the TM^3 Remote Proxy (RP)

Your RP takes only one argument: the number of pipes. For example, the following command starts your proxy with 4 pipes (assuming the compiled executable is `a.out`).

```
./a.out 4
```

When your RP starts, listen on port 6000 on n incoming TCP connections from LP where n is the number of pipes (*i.e.,* four in the above example). After these n pipes are established, your RP can start processing messages from the LP, or data from remote servers. No pipe message will arrive before all n pipes are established. Pipes are long-lived. They never close unless LP or RP exits.

Your implementation will be assessed by four levels described below, with each level supporting more features (and apparently, being more difficult) than the previous one. Be prepared that it may not be easy to debug Level 3 and in particular Level 4.

When you are working on this task, be sure to proceed with these four levels in order, and move to the next one only when the current level is fully working. Otherwise bugs will be mixed together, making it very difficult to fix them. Also, you must use non-blocking (recommended) or multi-threaded I/O because single-threaded blocking I/O does not work even for Level 1 (think of why).

Level 1: Sequential Application Connections over One Pipe

As the minimum requirement, this level lays the groundwork for your work. Your RP should support sequentially uploading and downloading data that is delivered by one (application) connection over

one pipe. Below are expected behaviors of your RP.

- When receiving a SYN message from LP, establish the connection to the remote server.
- When receiving a data message from LP, extract the data and deliver it to the server.
- When receiving data from the server, encapsulate it into one or more data messages and send them over the pipe to LP.
- When receiving a FIN message from LP, close the remote connection. You can ignore the "reason" field in the FIN message.
- When the connection is closed by the server, send a FIN message to LP. Set the "reason" field correspondingly based on how the connection is closed (0x88 for FIN and 0x99 for RST).

Level 2: Sequential Application Connections over Multiple Pipes

At this level, your RP needs to support multiple pipes. A single application TCP connection is multiplexed onto multiple pipes as illustrated in Figure 6.

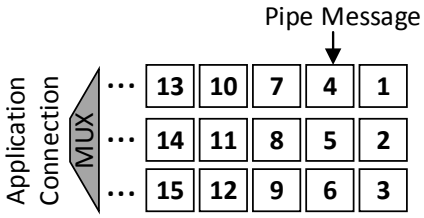

Figure 6: Multiplex a connection onto three pipes in using round robin.

- Select one of the pipes when sending a pipe message to LP. You can use any pipe selection strategy (*e.g.,* round robin or random). If a pipe is blocked, try another. You only pause sending when all pipes are blocked. Note a pipe message cannot cross pipes *i.e.,* you cannot send part of it over one pipe and the other part over another pipe.
- In Level 1, since there is only one pipe, all pipe messages will arrive in order (so the sequence numbers are not useful). However, when being delivered over multiple pipes, pipe messages can arrive out-of-order. When this happens, you need to buffer the messages (similar to how TCP buffers out-of-order segments), and deliver them when the sequence number gap is filled.

Level 3: Concurrent Application Connections over Multiple Pipes

In Level 1 and 2, at any time, there can be at most one active application connection. Now you are required to support multiple (up to 256) concurrent application connections. The workflow for each connection is the same as before, but the challenge is to handle the concurrency by carefully designing your algorithm and data structures. Some of the data structures such as out-of-order message list and sequence numbers need to be maintained on a per-connection basis.

Level 4: Real Workload over Multiple Pipes

There is no new feature to be added in Level 4. Therefore, if your Level 3 implementation is perfect, then you will automatically pass this final level. In this level, we will stress test your RP over real workload. For example, the client host runs a browser that visits real websites as shown in Figure 5, watch a YouTube video, or download some very large files. Your code needs to handle diverse situations such as a remote connection fails to establish, and a local or remote connection shuts down in the middle of data transmission. Note that throughout this project, you may assume that the connectivity between LP and RP never fails. You may also assume the LP never sends ill-formed pipe messages to RP.

E.4 Task 4: Write a Summary

Write a short document (less than one page) summarizing your progress in Task 3, including three points: (1) the level you have reached; (2) the functionalities that your RP can support; (3) a list of known bugs. Please be honest for as false claims will be easily identified in our testing.

If your team involves two students, you further need a paragraph that: (1) lists both students' names; (2) gives a detailed description of each student's contribution. Please be specific on who did what. A single sentence like "both students contributed equally" is not sufficient. The final grade will take this description into consideration.

F Submit Your Work

Your submission must include the following:

- Task 1: nothing to submit.
- Task 2: a pcap trace of TM^3 traffic generated by my model implementation (name it `tm3.pcap`).
- Task 3: all your source code (`*.cpp;*.c;*.h;*.java`).
- Task 4: the short document in §E.4 (name it `readme.txt`).

Compress everything above into a single file called `project3B.zip`, `project3B.bz2`, or `project3B.tar.gz` (depending on the compression program you use). Submit it to Canvas. Do not include any executable in your submission. Double check your submission to make sure it includes all files.

If a team involves two students, only one student (either one) needs to submit the team's work. If we receive submissions from both students, we will only grade the latest version (based on its timestamp) submitted before the deadline.

G Run and Test Your Program

To compile your program in Linux, type:

```
gcc (a list of your .c files) -O1
```

or

```
g++ (a list of your .cpp files) -O1
```

or

```
javac (a list of your .java files)
```

It will generate an executable file named `a.out`. The switch `-O1` tells the compiler to do some performance optimization for your code. For C/C++, you need to include `-lpthread` if you have included `<pthread.h>` in your program. Besides `pthread`, you are not allowed to use any other 3rd-party library. Similar to Project 2, your program can only use built-in C/C++/Java libraries, including the C++ Standard Template Library (STL).

If you use C/C++, we strongly recommend you work with an update-to-date GNU C/C++ compiler (gcc/g++ version 4.3 or above). Do not use other C/C++ compilers that might be incompatible to gcc/g++. For Java, please use JDK version 7.0 or above. Recall that you must use Ubuntu 14.04 for this project (for both LP and RP).

We will test your code under the same environment (VM client and EC2 server) for all levels you have accomplished.

H Grading

- You get 5 points when finishing Task 2. To prove that, use `tcpdump` to capture a pcap trace containing some TM^3 traffic and include it in your submission. Studying the trace will greatly help you understand the protocol and how LP and RP are supposed to work.

- Task 3 is worth 90 points with the following grading policy.

  You get 40 points for reaching Level 1.

  If you reach Level 1, you get additional 20 points for reaching Level 2.

  If you reach Level 2, you get additional 20 points for reaching Level 3.

  If you reach Level 3, you get additional 10 points for reaching Level 4.

Partial credits will be given to your working but buggy implementation.

- You get 5 points by submitting the document required in Task 4.

I Honor Code

Students must follow the IU honor code (`http://www.iu.edu/~code/code/responsibilities/academic/index.shtml`). In no case may your code be copied from another student or a third-party source on Internet. We will use an anti-plagiarism software to detect code "shared" among students. Any violations of the honor code will be dealt with strictly, including but not limited to receiving no credit for the entire project.

J Appendix: the Original TM^3 Paper

See the next page.

TM^3: Flexible Transport-layer Multi-pipe Multiplexing Middlebox Without Head-of-line Blocking

Feng Qian Vijay Gopalakrishnan* Emir Halepovic* Subhabrata Sen* Oliver Spatscheck*
Indiana University
Bloomington, IN
fengqian@indiana.edu

*AT&T Labs – Research
Bedminster, NJ
{gvijay, emir, sen, spatsch}@research.att.com

ABSTRACT

A primary design decision in HTTP/2, the successor of HTTP/1.1, is object multiplexing. While multiplexing improves web performance in many scenarios, it still has several drawbacks due to complex cross-layer interactions. In this paper, we propose a novel multiplexing architecture called TM^3 that overcomes many of these limitations. TM^3 strategically leverages multiple concurrent multiplexing pipes in a transparent manner, and eliminates various types of head-of-line blocking that can severely impact user experience. TM^3 works beyond HTTP over TCP and applies to a wide range of application and transport protocols. Extensive evaluations on LTE and wired networks show that TM^3 substantially improves performance *e.g.,* reduces web page load time by an average of 24% compared to SPDY, which is the basis for HTTP/2. For lossy links and concurrent transfers, the improvements are more pronounced: compared to SPDY, TM^3 achieves up to 42% of average PLT reduction under losses and up to 90% if concurrent transfers exist.

CCS Concepts

•Networks → Network protocol design; Middle boxes / network appliances;

1. INTRODUCTION

In May 2015, version 2 of the Hypertext Transfer Protocol (HTTP/2) was published as RFC 7540 [17], introducing significant changes over HTTP/1.1. The evolution to HTTP/2 was driven by various performance issues of HTTP/1.1 as web pages became rich and complex. In particular, it is known that HTTP/1.1 inefficiently interacts with the underlying transport layer. Improving such an interplay is one of the key focuses of HTTP/2.

The design of HTTP/2 draws heavily from SPDY [12]. The key feature of SPDY (HTTP/2)[1] and other recent proposals [10, 3, 7] is to multiplex multiple object transfers over a single transport-layer connection on which HTTP transactions are efficiently pipelined. Since only one connection is used per-domain, overheads such as TCP/SSL handshake incurred by short-lived connections in HTTP/1.1 are significantly reduced. These protocol changes help improve the main metric of page load time (PLT), especially for complex web pages with many small objects.

However, due to tight coupling between multiplexing and the use of a single TCP connection, HTTP/2 underperforms in lossy environments, remains vulnerable to different types of head-of-line (HoL) blocking, and may suffer from bandwidth under-utilization (§2). Addressing these issues is imperative at this time when HTTP/2 is expected to be used by applications ranging from web browsers to mobile apps.

In this paper, we focus on the following questions: *(i)* can we do better by removing the limitation of a single TCP connection, *(ii)* can different types of HoL blocking affecting HTTP/2 be avoided, *(iii)* can transport-related inefficiencies be independently addressed while considering HTTP needs, and *(iv)* is there a more flexible solution that works beyond just HTTP/SPDY over TCP and applies to a broader variety of application and transport protocols?

Our findings, based on extensive evaluations in real settings, are that answers to all questions are positive. Taking the high-level approach of decoupling the optimization of HTTP from the underlying transport layer, we propose a new architecture called TM^3 (Flexible Transport-layer Multi-pipe Multiplexing Middlebox), for improving performance of all applications. TM^3 adopts positive ideas from HTTP/2 and SPDY, but overcomes many limitations in existing mul-

[1]Since SPDY and HTTP/2 are similar, statements about SPDY in this paper also apply to HTTP/2 unless otherwise noted. We conduct most experiments using SPDY whose implementation is presumed to be more stable than HTTP/2 at this moment.

CoNEXT '15, December 01-04, 2015, Heidelberg, Germany
 ISBN 978-1-4503-3412-9/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2716281.2836088

tiplexing proposals by improving the transport layer.

First, instead of using a single connection, TM^3 leverages multiple concurrent channels (called "pipes") for multiplexing. This provides two benefits. *(i)* It makes TM^3 robust to non-congestion losses that usually affect only one or a subset of pipes, and *(ii)* it improves bandwidth utilization by enabling *inverse-multiplexing*, where an application connection leverages all pipes to transfer data. To mitigate the aggressiveness of using multiple pipes, we use host-based congestion control that manages all pipes' congestion windows in an aggregated manner.

Second, TM^3 eliminates various types of head-of-line (HoL) blocking. In particular, we found that in SPDY and HTTP/2, due to large shared FIFO queues at the sender, when concurrent large and small (usually delay-sensitive) objects are multiplexed together, the load time for small objects can be tens of times worse compared to HTTP/1.1. We propose an effective solution to eliminate this type of HoL blocking by performing multiplexing *after* most shared queues. Our solution works with diverse unmodified transport-layer protocols and requires minimal change to the OS.

Third, as a transport-layer multiplexing solution, by leveraging endhost operating system support, TM^3 can transparently improve performance for *all* applications (even for those using non-HTTP protocols) without requiring any change to existing applications or servers. TM^3 is generally applicable to any type of network, with highest benefits for moderate to high latency links (*e.g.,* cellular or DSL).

Fourth, TM^3 brings improved flexibility. Pipes can be realized by diverse transport protocols, and heterogeneous pipes can coexist. The proxies can then dynamically decide which pipe(s) to use based on factors such as network conditions and QoE requirements. TM^3 thus provides a practical platform for realizing various optimizations and policies.

We implemented TM^3 and extensively evaluated it over commercial LTE and emulated wired networks. The key results are summarized as follows.

- TM^3 substantially reduces individual file download time for 2MB, 300KB, and 50KB files by 32%, 53%, and 58%, respectively, over LTE. When used with HTTP/1.1 over LTE, TM^3 improves Page Load Time (PLT) by an average of 27%. When used with SPDY, the average PLT reduction is 24%, across 30 popular websites.
- Over a lossy link (10Mbps with 50ms RTT), when TM^3 is used with SPDY, the median improvements of PLT are 22%, 33% and 42% for 0.5%, 1% and 1.5% of random loss rates, respectively. TM^3 outperforms QUIC at handling losses, based on comparing with another recent study of QUIC [18].
- TM^3 brings significant benefits when concurrent transfers exist. Due to the sender-side HoL blocking being eliminated, the PLT can be reduced by 75% to 90% over LTE.
- We conduct case studies to show the transparency and the flexibility of TM^3. TM^3 incurs negligible runtime overhead and very small protocol overhead of no more than 1.5%.

2. EXISTING MULTIPLEXING SCHEMES

We motivate TM^3 by revealing limitations of existing multiplexing protocols. Among many candidates including SST [20], SCTP [30], SPDY [12], QUIC [10], and HTTP Speed+Mobility [3], we first focus on SPDY that is a concrete implementation forming the basis of HTTP/2. We then discuss QUIC, a more recent multiplexing protocol, in §2.2.

A distinct feature of SPDY is it supports multiple outstanding requests on one TCP connection. SPDY encapsulates HTTP transactions into *streams*, such that a stream carries usually one transaction, and multiple streams are *multiplexed* over one TCP connection. In this way, bootstrapping overheads of short-lived TCP connections in HTTP/1.1 are significantly reduced, leading to more "dense" traffic on the multiplexed connection. SPDY also supports request prioritization and header compression.

SPDY has been shown to significantly reduce PLT. However, SPDY still suffers from a few limitations. *(i)* Its performance degrades under packet losses due to the use of a *single* TCP connection [32], which aggressively cuts its congestion window upon a loss (either real or spurious [19]). *(ii)* Even without loss, a single connection can still suffer from bandwidth under-utilization. For example, slow start needs to be performed after an idle period (typically one RTO). Compared to multiple connections performing slow start simultaneously (as the case in HTTP/1.1), the congestion window growth of a single connection is slower.

2.1 Head-of-line Blocking in Multiplexing

Compared to HTTP/1.1, multiplexing can also cause various types of head-of-line (HoL) blockings to be described below. We refer to them as Type-L, Type-S, and Type-O in the rest of the paper. They can severely affect multiplexing performance and TM^3 can eliminate all three types of blockings. We leave describing the solutions to §4.

Type-L (Loss) Blocking. Occurring at the receiver side, Type-L blocking is usually caused by packet loss over TCP-based multiplexing channel. TCP ensures in-order delivery at byte level. But such a guarantee is too strict for multiplexing. HoL blocking happens when packet loss in frame A[2] prevents a later frame B from being delivered to upper layers at receiver, where A and B belong to different streams.

Type-S (Sender) Blocking. When being multiplexed together, a large data transfer may substantially hinder the delivery of a small transfer, because both transfers share several FIFO queues at application, transport, and/or link layer at the sender. Assuming their sizes are q_1, q_2, and q_3, respectively, when the bottleneck link is saturated, the queues will build up, causing HoL blocking delay of at least $(q_1 + q_2 + q_3)/b$ for newly arrived small transfers where b is the bottleneck link bandwidth (in contrast, q_1 and q_2 are usually negligible in HTTP/1.1). By instrumenting the source code of SPDY module for Apache (`mod_spdy` 0.9.4.1 [11]),

[2]A *frame* is the atomic transfer unit in SPDY and HTTP/2. A stream contains one or more frames.

Table 1: 4KB web object load time (in second) with and w/o concurrent bulk download, averaged across 20 measurements.

Concurrent Download?	5Mbps BW		10Mbps BW		20Mbps BW	
	HTTPS	SPDY	HTTPS	SPDY	HTTPS	SPDY
NO	0.05	0.05	0.05	0.05	0.05	0.05
YES	0.22	17.02	0.14	8.40	0.09	4.00

we verified q_1 can take up to several MBs. In cellular networks, q_2 can also grow to several MBs to accommodate large in-network buffers [22]. q_3 is by default 1000 packets unless TCP Small Queues (TSQ) [14] is enabled.

Type-S blocking has been measured in other contexts including the Tor overlay network [21] and synthetic application [25]. However, to our knowledge, there has been no its study in real settings of SPDY or HTTP/2. We thus conduct an experiment as follows. We perform two concurrent transfers between a laptop and a web server (Ubuntu 14.04/Apache 2.4) with SSL and SPDY [11] configured: one transfer is downloading a large file, and the other is fetching a 4KB object. The client uses Google Chrome browser (stable release 39.0.2171). The end-to-end RTT is 50ms, and we vary server's bandwidth using `tc`. Table 1 measures the object load time (from sending HTTP request to receiving the whole 4KB data). When there is no concurrent bulk download, HTTPS and SPDY show similar performance. Surprisingly, when the bulk download is in progress, the transfer time of the small object over SPDY increases dramatically (44 to 77 times of that for HTTPS).

It is worth mentioning that there are two ways to deploy SPDY: using a server plugin (described above) and using a SPDY proxy. In the former scenario, multiplexing is performed at a per-domain basis. When a SPDY proxy is used, all traffic between the browser and the proxy will be multiplexed into a single TCP connection. We repeat the experiments presented in Table 1 for SPDY proxy (compiled from the Chromium Project [15]) and observe very similar response time inflation. In this case, the bad situation gets even worse: as long as the user downloads some large data from *any* website, response time for *all* websites can become unacceptably long.

Next, we show that even a medium-sized file download in the background is sufficient to trigger latency inflation for SPDY. We repeat the experiment with various background file download sizes. Table 2 indicates a 500KB concurrent file download can already inflate the 4KB object load time by 8 times. We also tested several real applications. Figure 1 compares the 4KB object load time between HTTP (using Squid HTTP proxy [13]) and SPDY (using Chromium SPDY proxy) when simultaneously watching a YouTube HD video. The 75th percentiles of object load time for HTTP and SPDY are 0.2 and 2.5 sec, respectively, yielding a 12.5x difference.

Type-O (Out-of-order) Blocking occurs when application data is multiplexed to *multiple* connections. It does not affect HTTP/2, SPDY, or QUIC that use one connection for multiplexing. However, it affects a naive design of TM^3, as described in detail in §4.3.

File Size	Load Time
0	0.05 sec
500KB	0.45 sec
1MB	0.94 sec
2MB	1.75 sec
4MB	3.33 sec
8MB	6.70 sec

Table 2: 4KB object load time with concurrent download of various sizes, averaged across 20 runs (10Mbps link, 50ms RTT).

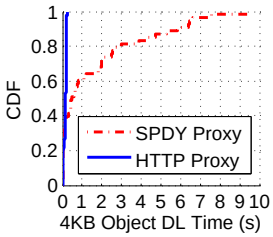

Figure 1: 4KB object load time with YouTube HD streaming (10Mbps link, 50ms RTT).

2.2 From SPDY to QUIC

QUIC [10] is a new protocol proposed in 2012 for further improving web performance. Although its key idea is still multiplexing, it differs from SPDY in many positive aspects. *(i)* At transport layer, it uses reliable UDP that allows unordered delivery. *(ii)* It supports zero-RTT connection setup when the client revisits a server. *(iii)* It uses forward error correction (FEC) to cope with losses. *(iv)* It supports pacing-based congestion control to better balance between throughput and delay in particular for long-lived flows. Recent study indicates QUIC outperforms SPDY in many scenarios [18].

QUIC is not without limitations. Although using UDP addresses Type-L blocking, QUIC still suffers from Type-S blocking. Even though QUIC recovers from losses more aggressively, its performance may still degrade due to the same reason as SPDY's: using a *single* multiplexing channel. As a result, at high loss rate, QUIC's PLT is often worse than (up to 40%) HTTP/1.1 [18]. Also, QUIC's FEC incurs non-trivial overhead. Turning on FEC in QUIC version 21 consumes up to 1/3 of available bandwidth even when there is no loss. Since QUIC is still experimental, in this paper we focus on quantitative comparison between TM^3 and SPDY, with qualitative remarks on QUIC.

3. TM^3 DESIGN

3.1 Overview

TM^3 aims to address limitations described in §2. Table 3 compares between TM^3 and SPDY/HTTP2 in terms of key design decisions (top) and the resultant features (bottom).

- TM^3 is a transparent transport-layer proxy. It performs multiplexing for all TCP connections on an endhost. In SPDY's terminology, a "stream" in TM^3 is an application-issued TCP connection, and a "frame" is a chunk of the connection's user data. Multiplexing at transport layer allows unmodified apps (*e.g.,* mobile apps) to benefit from TM^3.
- Instead of using one TCP connection, TM^3 multiplexes data into *multiple* concurrent transport channels called *pipes*. Doing so improves bandwidth utilization and robustness to losses. To bound the aggressiveness of concurrent pipes,

Table 3: Comparing between TM^3 and HTTP/2 (SPDY).

	TM^3	HTTP/2 (SPDY)
Works at which layer	Transport	App.
Transport protocol	Flexible	TCP
Number of multiplexing channels	Multiple	One
When multiplexing happens	Late	Early
Transparent to applications	Yes	No
Robust to random loss	Yes	No
Free of HoL blocking	Yes	No
Efficient bw util. for bursty traffic	Yes	No
Dynamically change multiplexing channel	Yes	No
App-layer features (*e.g.,* compression)	No	Yes

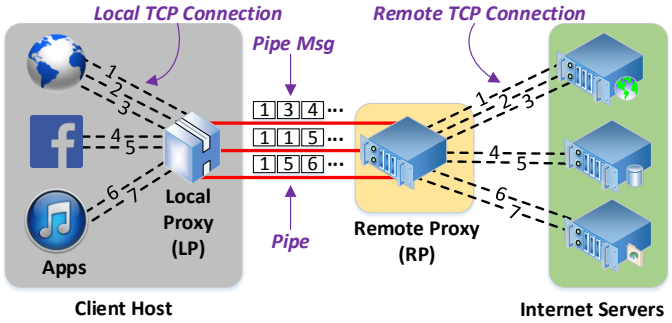

Figure 2: The TM^3 architecture.

TM^3 employs aggregated congestion control by treating all pipes as one "virtual" connection.

- TM^3 addresses all three types of HoL blocking described in §2.1. In particular, we design a new OS mechanism to completely eliminate Type-S and Type-O blocking by postponing multiplexing after shared queues. It requires very small changes to the OS kernel and can work with unmodified transport protocols such as TCP and SCTP. This part will be described in detail in §4.

- TM^3 provides transport-layer flexibility. Pipes can be realized by any transport protocol, and mappings from user data to pipes can be dynamically adjusted. Such flexibility enables new use cases such as dynamic transport protocol selection, and makes testing new pipe realizations very easy.

3.2 Pipes and the Multiplexing Scheme

As depicted in Figure 2, TM^3 consists of two key components: a *local proxy* (LP) and a *remote proxy* (RP). LP resides on the client host, and RP is deployed on a middlebox. In cellular networks, for example, RP can be integrated with web proxies [33]. In DSL, RP can be placed at broadband ISP's gateway. LP and RP transparently split[3] an end-to-end TCP connection into three segments: *(i)* a *local connection*[4] between client applications and LP, *(ii)* one or more *pipes* between LP and RP, and *(iii)* a *remote connection* between RP and the remote server. The pipes serve as the transport layer between the client and the middlebox, thus covering the "last mile" that is usually the bottleneck link. Also note that TM^3 essentially relays traffic at transport layer so it *can* support SSL/TLS although TM^3 cannot decrypt the traffic.

Pipes are bidirectional and long-lived, and they are by default shared by all connections across all applications by multiplexing. Uplink traffic from client to server is multiplexed onto the pipes at LP, and de-multiplexed at RP (reversely for downlink traffic). Due to LP, multiplexing and de-multiplexing are done transparently for all applications.

A *pipe message* is the atomic unit transferred on a pipe. As shown in Figure 3, we currently designed four types of pipe messages to support basic data transfer over pipes: Data, SYN, SYNACK, and FIN. A data pipe message ("data message") carries user data; the other three messages are used for TCP connection management (§3.3). More message types can be added for supporting additional control-plane features such as flow control and setting exchange.

Figure 3: Pipe message format with headers (8 bytes) shaded.

A data message has an 8-byte header including payload length, connection ID, and sequence number, followed by the actual payload. Connection ID (connID) is used to uniquely identify each connection, whose mapping to connID is synchronized between LP and RP. Since pipe messages created from one TCP connection can be concurrently delivered by multiple pipes in arbitrary order, each pipe message of the same TCP connection needs a sequence number that increases *per message*. Note this is not needed by SPDY that uses one connection for multiplexing.

When a local connection has any uplink data coming, the LP encapsulates the TCP payload into one or more data messages. For each message, the LP selects a pipe and transfers the message over that pipe. The least occupied pipe (*i.e.,* a pipe with the least buffer occupancy) is selected. This helps balance the usage of all pipes and avoid pipes experiencing loss. When a downlink data message arrives on a pipe, the LP extracts the TCP payload and delivers it to the corresponding local connection, if the message's sequence number matches the expected sequence number of the local connection. Otherwise the message is buffered and delivered when the sequence number gap is filled. Multiplexing and de-multiplexing at RP are performed in the same manner.

Using multiple pipes provides three key benefits.

- Multiple pipes allow for both *multiplexing* and *inverse multiplexing*. When many short-lived TCP connections are present (*e.g.,* HTTP/1.1 over TM^3), they will be multiplexed

[3]Because of this, sometimes an application may think the data is sent to the server but it is still buffered at LP or RP and may never reach the server. This is common with all TCP-splitting approaches, and it does not affect the correctness of HTTP, which explicitly acknowledges that a request has been processed.

[4]In the rest of the paper, for brevity, a "connection" means an *application-issued* TCP connection unless otherwise noted.

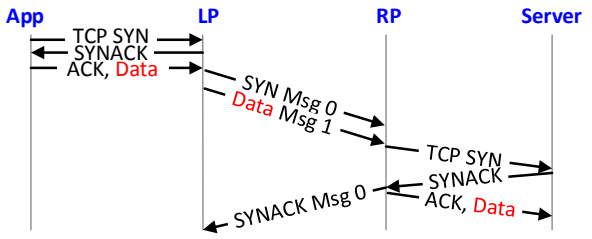

Figure 4: TM³ Connection establishment. The data message (Seq #1) is piggybacked with the SYN Message (Seq #0).

onto fewer pipes to avoid connection setup and slow start overhead. When few TCP connection(s) exist (*e.g.,* HTTP/2 over TM³), they will be inverse-multiplexed onto multiple pipes that make better use of available bandwidth than a single connection does. Note TM³ itself does not distinguish between the two schemes, since an individual connection's data is *always* distributed on to multiple pipes. Whether multiplexing or inverse-multiplexing is performed only depends on the number of connections.

- Multiple pipes are more robust to losses than a single pipe, because congestion control of each pipe is largely independent to others' (although TM³ uses aggregate congestion control to tame their overall aggressiveness, §3.4). Therefore, the impact of throughput reduction caused by non-congestion loss is mitigated by localizing loss to one pipe (or very few pipes) while other pipes remain undisturbed. This differs from, for example, using a single SCTP pipe with multiple streams that are managed by the same congestion control instance.

- Multiple pipes provide great flexibility at transport layer. Pipes can be realized by any transport-layer protocol with message-oriented ordering and reliability guarantees. Furthermore, TM³ allows switching among pipes realized by different protocols on the fly, depending on the network conditions, traffic pattern, or application requirements. For example, SCTP can prevent Type-L blocking in high-loss environment with the drawback of being a heavy-weight protocol [24]. Thus, TM³ can temporarily employ SCTP in high-loss situations, and use TCP pipes by default without interrupting upper-layer applications. We study this in §6.5.

3.3 TCP Connection Management

Connection establishment. When LP receives a TCP SYN packet from an application, it immediately replies with SYN-ACK to establish the local connection. The LP subsequently sends a SYN pipe message (Figure 3) containing an unused connection ID, the remote host IP address, and the remote host port number to RP. The RP then establishes the connection to the remote host on behalf of the client, and sends back a SYNACK message to LP to indicate the successful establishment of the remote connection.

As shown in Figure 4, the local connection setup takes virtually no time. Inspired by TCP Fast Open [26] and QUIC, we allow SYN and data pipe messages being sent back to back, for achieving 0-RTT connection setup over pipes. Two issues need to be resolved. First, the data piggybacked with the SYN message will be wasted if the remote connection establishment fails. Second, an ill-behaved LP may send a SYN message with a non-existing server address, followed by a large amount of data, potentially exhausting the buffer space at RP. To address both issues, we limit the amount of data allowed to send to RP before the reception of the SYNACK message, at both per-connection and per-host basis. The two limits are known by both LP and RP so violation leads to an error.

Connection closure. When LP receives a TCP FIN or RST packet from a local connection, it immediately terminates the local connection. Meanwhile, the LP sends a FIN pipe message, whose "reason" field (Figure 3) indicates whether the closure is caused by FIN or RST, to the RP. Upon the reception of the FIN pipe message, the RP terminates the remote connection correspondingly. Connection closure initiated by a server is handled similarly.

3.4 Managing Aggressiveness of Pipes

Compared to a single pipe, employing multiple pipes can be more aggressive. Because each pipe runs congestion control independently without explicit coordination with other pipes, the aggregated congestion window across pipes is more likely to overshoot the bandwidth-delay product. To address this, we propose to employ host-based congestion control (CC) [16] by treating pipes as "virtual" connections, and control the *aggregated* congestion window across all pipes. The host-based CC co-exists and loosely couples with per-pipe CC, which provides more fine-grained control of, for example, the congestion window growth. In literature, numerous CC algorithms have been proposed, and many of them can be easily converted to host-based CC. We conduct a case study in §6.6.

4. ELIMINATING HOL BLOCKING

Recall in §2.1 that existing multiplexing schemes such as SPDY and QUIC suffer from at least one type of HoL blocking. We describe how TM³ eliminates them.

4.1 Type-L (Loss) Blocking

Type-L blocking can be eliminated by using transport protocols (*e.g.,* SCTP or QUIC) that support out-of-order delivery as pipe realization. For example, SCTP serves a role similar to that of TCP while allowing an upper layer to define multiple message-oriented streams that can be delivered independently, thus eliminating Type-L blocking.

4.2 Type-S (Sender) Blocking

Recall that Type-S blocking is caused by shared FIFO queues at sender side. We first discuss why existing approaches (to our knowledge) are not ideal for our purposes.

Reducing shared queue size can mitigate Type-S blocking [21]. However, doing so (in particular, reducing TCP send buffer size, which caps the congestion window size) may cause side effects such as degraded performance.

Replacing a FIFO queue with a priority queue also helps, as Type-S blocking will not happen among data with different priorities. We discuss doing so at different layers.

At application layer, SPDY supports at most 8 priorities. Our inspection of the Chrome source code indicates that the priority is statically determined by the content type (*e.g.,* a binary object always has higher priority than an image). Therefore a large object can still block small objects of the same priority.

At transport layer, the TCP/UDP/SCTP output queue (send buffer) is by nature FIFO. Making them support priority queues is difficult but doable. For example, uTCP [25] modifies TCP internals to add multi-queue support. However, to satisfy the flexibility requirement, TM^3 should be able to work with diverse and unmodified transport protocols as pipe realization. So uTCP's approach is not ideal for us.

At link layer, the Linux queuing discipline (qdisc) uses a FIFO queue (`pfifo_fast`) by default. Although multiple qdisc can be easily configured, making them work with multiplexing is challenging due to the difficulty of specifying the filter rules (*i.e.,* which traffic goes to which queue). TCP 4-tuples cannot be used as a filter since multiplexed traffic is no longer separated by TCP connections. Even the per-packet ToS bits are inadequate since several pipe messages (or frames in SPDY) with different priorities may be multiplexed into one packet in both TM^3 and SPDY.

It is important to note that to completely eliminate Type-S blocking, *priority queues need to be set up at all above layers*. This greatly increases the solution complexity, and is another reason why we did not take this approach.

4.2.1 Starvation-free Late Multiplexing (LMux)

As discussed above, multiplexing creates unique challenges for eliminating Type-S blocking. We design an OS mechanism called Starvation-free Late Multiplexing (LMux). Unlike existing approaches that perform multiplexing before most shared queues in user space as shown in Figure 5a, LMux postpones multiplexing until the packet is dequeued from most shared queues. LMux achieves three goals: *(i)* eliminate Type-S blocking without starvation (explained shortly), *(ii)* work with diverse unmodified transport protocols that realize pipes, and *(iii)* incur minimal change to the OS, as well as small computation and bandwidth overhead.

LMux is a self-contained feature deployed only at sender, and is transparent to the receiver. Figure 5b illustrates its workflow in 5 steps. *(i)* When data comes from a TCP connection, TM^3 writes blank data to a pipe, generating one or more packets (with blank data) called *containers*. Meanwhile, *(ii)* the real data is copied to the corresponding in-kernel connection buffer. *(iii)* The OS routes the containers as regular packets, which might be queued in various queues. When a container is dequeued from qdisc, *(iv)* the scheduler selects one or more TCP connections based on the scheduling algorithm and their buffered data sizes, and *(v)* the pipe message header(s) and their actual data are written

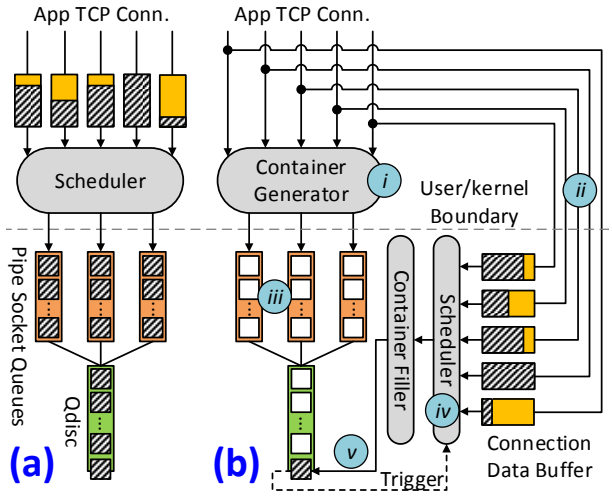

Figure 5: Multiplexing (a) without LMux, and (b) with LMux.

to the container. Note that a pipe message never crosses containers *i.e.,* a container must contain one or more *full* pipe message(s). Otherwise the receiver cannot assemble data in multiple containers into a full message because there is no in-order delivery guarantee across pipes.

How does LMux eliminate Type-S blocking? Consider a scenario where a bulk data transfer A is in progress, saturating all shared FIFO queues. Now some small data with high priority from another connection B arrives and the data is copied to its connection buffer (Step *ii*). Then immediately (Step *iv* and *v*), B is scheduled and its data is pushed to a container leaving the qdisc. It is important to note that the container(s) carrying B's data were generated earlier by A, and have already traversed the shared queues *before* B's data arrives, so B does not need to wait for queuing (and later, in return, A will occupy some container(s) generated by B). In contrast, if the same scheduling decision is made before the shared queues as shown in Figure 5a, B's data can be promptly pushed to the shared queues in which, however, it will still be blocked by A's bulk data.

Generality. LMux only alters the data plane. It does not affect any transport-layer control plane behavior (*e.g.,* congestion control) as the protocol stack is completely unaware of LMux. This makes LMux *generally applicable to different unmodified transport protocols*. The container filler only needs to be aware of the pipes' protocol (*e.g.,* TCP or SCTP) and fills their containers accordingly (§5).

Determining the number of containers is a key challenge for LMux design. Generating too many containers wastes bandwidth, while having too few causes *starvation i.e.,* some data cannot be sent out due to a lack of outbound packets. For u bytes of user data, a naive solution is to emit $k_{\min} = \lceil u/(m-h) \rceil$ containers where h is the pipe message header length (8B) and m is the maximum container size *i.e.,* the maximum packet payload size. The u bytes will be split into $k_{\min}$ containers each containing a pipe message (the last container may be smaller than m). The total container space required is thus $u + hk_{\min}$. But this may still cause

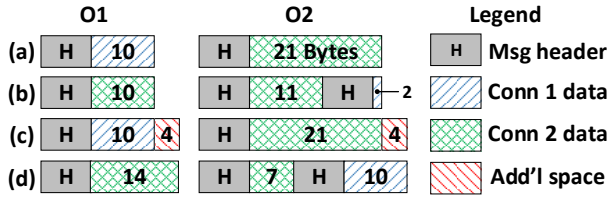

Figure 6: Example: different ways of filling containers.

starvation. As mentioned before, when multiple connections exist, a connection may use containers generated by other connections (this is the key reason why Type-S blocking can be eliminated). In this case multiple messages may share a container, and more container space for headers is required. This is the key reason why precisely determining the number of containers is difficult.

For example, assume 10 and 21 bytes of user data come from Connections 1 and 2, respectively, which then generate two containers of O_1=18B and O_2=29B (h=8B), as shown in Figure 6. No starvation happens if each connection uses the container generated by itself as shown in Figure 6a (so each container holds one message). However, depending on the scheduler, Connection 2 may first consume O_1 and then part of O_2. The remaining space in O_2 can only accommodate 2 bytes for Connection 1 due to an additional header, leading to starvation shown in Figure 6b.

To address this problem, one can generate more containers when starvation happens. But doing so delays packet transmission because a waiting period is needed to differentiate real starvation from containers' delayed arrival (*e.g.,* due to busy CPU). Instead, we allocate additional space for *every* container to guarantee starvation never happens. In the previous example, starvation can be prevented by allocating 4 more bytes for each container (Figure 6d) although they may be wasted (Figure 6c). The exact amount of additional space is given by the following theorem (proof omitted): *when there are n concurrent connections, given any scheduling scheme, starvation will not happen iff each container is allocated with at least $\lceil h(n-1)/n \rceil$ additional bytes.* ■

We therefore allocate h additional bytes for each container to ensure no starvation for any n. Given h is only 8 bytes, the incurred protocol overhead is very small. Note the new scheme may require slightly more containers ($\lceil u/(m-2h) \rceil$) than $k_{\min} = \lceil u/(m-h) \rceil$ to prevent starvation.

4.3 Type-O (Out-of-order) Blocking

This type of blocking happens when a large flow is inverse-multiplexed to *multiple* pipes due to pipe messages being out-of-order, which is explained first.

Consider how inverse multiplexing is performed without LMux. When all pipes are saturated, a block of empty space in any pipe results in a train of consecutive pipe messages being pumped into that pipe. Pipes are then treated as normal transport-layer connections by the OS, which schedules the pipes in a way where it tries to send as much data as possible from the same pipe until some limit is reached, *e.g.,* imposed

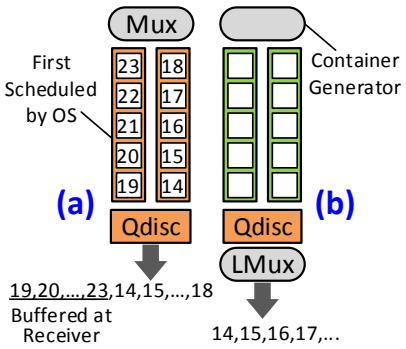

Figure 7: (a) Pipe message out-of-order, assuming the OS schedules the left pipe first then the right pipe; (b) the out-of-order is eliminated by LMux.

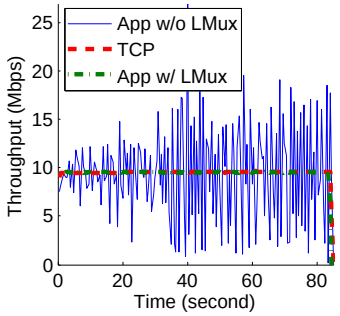

Figure 8: Application and TCP throughput for downloading a 100MB file (4 pipes, 10Mbps, 50ms RTT, default send buffer).

by congestion window or the pipe is empty. Usage of multi-threading inside the kernel further adds uncertainties to such scheduling. As a result, because the OS is unaware of pipe message sequence numbers, messages sent to the network can be severely out-of-order, as illustrated in Figure 7a.

Type-O blocking is caused by such out-of-order and it forces the receiver to buffer a large number of pipe messages to ensure in-order delivery of messages of the same connection. In Figure 7a, message 19 to 23 will be buffered before 18 is received. Type-O blocking has two negative impacts. First, it causes significant receiver-side buffering delay. Based on our measurement, when using 4 pipes, the buffering delay can reach up to 9.6 seconds in LTE, requiring more than 20MB buffer space. Second, the buffering delay has high variation, leading to significant fluctuation of application-layer throughput although the transport-layer throughput remains stable, as shown in Figure 8.

Elimination. Interestingly, LMux also eliminates Type-O blocking. As shown in Figure 7b, pipe messages are sequentially filled into the stream of containers. Because this occurs after OS schedules the transmission of pipes, LMux can guarantee that for each connection, its sequence numbers increase monotonically as pipe messages leaves the endhost.

It is worth mentioning that multipath TCP (MPTCP) also has receiver-side out-of-order issue [28], which however has a different cause of network paths' diverse characteristics. In contrast, Type-O blocking in TM^3 is completely caused by endhost, and can thus be completely eliminated by LMux.

Table 4: Implemented components for the TM^3 prototype.

Component	Type	Lang.	LoC
LP and RP	User-level application	C++	4,500
Traffic forwarder for LP	Kernel module	C	500
LMux	Kernel module	C	1,000
LMux	Kernel source modification	C	30

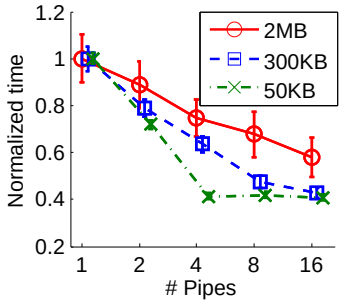

Figure 9: File DL time over LTE with varying # of pipes.

5. IMPLEMENTATION

We implemented the TM^3 prototype on standard Linux platforms, with its components summarized in Table 4.

Both LP and RP are implemented as user-level programs. They share most of the source code. On the client side, to realize the transparency requirement, we built a lightweight kernel module that automatically forwards packets generated by user applications to LP via the loopback interface. It also forwards downlink packets from LP to user applications.

Pipe Realization. We implemented two types of pipes: TCP and SCTP. SCTP eliminates Type-L blocking that affects TCP. However, SCTP is heavy-weight, and only exhibits advantages in high-loss environments [24]. It is often blocked by middleboxes (*e.g.,* in the LTE carrier we tested).

LMux Implementation consists of three components. *(i)* The main LMux logic, including multiplexing, connection scheduling, and container filling are implemented in a kernel module that can be injected to LP, RP, or both. *(ii)* The container generation is implemented in the LP/RP application. *(iii)* We slightly modified the Linux kernel source code by adding a socket option and several `ioctl` commands allowing control-plane communication between user and kernel space. User/kernel shared memory is used for connection buffers. We also add a flag to the `sk_buff` data structure to distinguish among non-TM^3 packets, empty container, and filled container so that, for example, retransmitted containers will not be filled twice. We further implemented a non-LMux version of TM^3 for comparison purposes. Both LMux and non-LMux versions employ the same round-robin scheduling for selecting an application connection although more sophisticated scheduling algorithms can be plugged in.

We implemented container filling logic for TCP and SCTP based on their packet formats. Their main differences are: *(i)* in TCP, there is a one-to-one correspondence between container and packet (as described in §4.2.1), while a container in SCTP is a *data chunk* which is the basic transmission unit of SCTP, and multiple chunks can be bundled in a one packet; *(ii)* for SCTP, LMux also needs to fill in stream ID and stream sequence number (different from the transmission sequence number that is maintained by the OS). The maximum container size is inferred from handshake for TCP and a lightweight probing for SCTP.

Limitation. In our LMux implementation, multiplexing happens after pipe socket buffers but before the qdisc, which is shrunk to 4KB per pipe using TCP Small Queues [14]. This simplifies our implementation (based on Netfilter [6]). We found that unlike reducing TCP buffer, decreasing qdisc buffer occupancy does not impact throughput. Given this is only an engineering issue, we are working on a new version of LMux that addresses this limitation.

6. EVALUATION

We installed LP on a commodity laptop with Ubuntu Linux Desktop 13.10. We consider two types of networks. *(i)* Most experiments were conducted on a commercial LTE network in the U.S. The client laptop obtains LTE access by tethering to a Samsung Galaxy S5 smartphone via USB. The RP is located in a cloud near the cellular network gateway. The experiments took place in two locations (New Jersey and Indiana), under good signal strength (-80 to -75 dBm). We found no qualitative difference between the two locations so we report the Indiana results unless otherwise noted. *(ii)* We also test TM^3 on a wired network. The LP and RP run on two laptops connected by a desktop Gigabit Ethernet switch. We use `tc` to emulate a 10Mbps broadband access link with 50ms RTT between LP and RP. The link parameters are selected by following a recent SPDY study [32]. To ensure fair comparison with other approaches, we use TCP CUBIC (the default TCP variant in Linux and in most of today's web servers) with default parameters, and default qdisc (`pfifo_fast`) unless otherwise noted.

6.1 Single File Download

We first examine TM^3's performance for individual file download, which captures a wide range of application scenarios such as downloading an email attachment or an image. The setup is as follows. A custom server program transfers a fixed-size file to the client over a single TCP connection. The file sizes are 50KB, 300KB, and 2MB.

Figure 9 shows the file transfer time over LTE with different number of pipes, normalized to using a single pipe. The latency between the RP and the file server is set to be 4ms, which is the median latency from RP (located near LTE gateway) to servers of 30 popular websites (§6.2). For each file size, we repeated the experiment 100 times at different times of day. As shown in Figure 9, using multiple pipes reduces download time. For example, using 8 pipes reduces the download time for 2MB, 300KB, and 50KB files by 32%, 53%, and 58%, respectively. Using a traditional TCP-splitting proxy yields results similar to those of using TM^3 with one pipe. TM^3 adds a benefit of eliminating one RTT

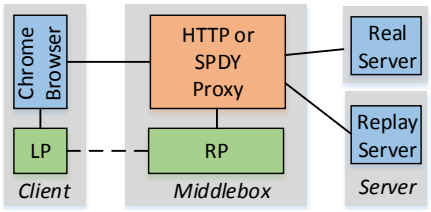

Figure 10: Web performance evaluation setup.

between LP and RP due to SYN/data piggyback (§3.3).

When multiple pipes are used, the improved performance is attributed to inverse multiplexing *i.e.,* the initial congestion window is essentially multiplied by k when k pipes are used. For the 50KB file, its transfer time remains stable for 4 or more pipes because the entire file can fit into the overall initial congestion window (ICW=10*1460B*4≈58KB) so the transfer takes one single RTT. While this appears aggressive, we argue that all today's browsers are already doing this much more aggressively (*e.g.,* Chrome uses up to 32 concurrent connections across all domains). By adopting the merits of both HTTP/1.1 and HTTP/2, TM^3's approach of using a fixed number of pipes *per-host* is in fact less aggressive, more flexible (works for one or many TCP connections), and more controllable (the per-host ICW is bounded by the fixed number of pipes). For long-lived flows, the aggressiveness issue can be mitigated by host-based congestion control.

6.2 Web Browsing

We consider four configurations illustrated in Figure 10.

Config I: HTTP only. The client connects to an HTTP proxy (Squid 3.3.11) running on the middlebox. The HTTP proxy ensures the network path is the same as those of the other three configurations that all need to use the middlebox.

Config II: HTTP+TM^3. It is similar to Configuration I except TM^3 is enabled with 8 pipes, selected based on trends shown in Figure 9. The HTTP proxy and RP are co-located so the forwarding overhead between them is negligible.

Config III: SPDY only. The client connects to a SPDY proxy [15], which is used for two reasons: *(i)* SPDY is not yet used by most web servers today and a proxy is the easiest way to deploy SPDY, and *(ii)* the SPDY proxy brings the ideal usage scenario of the SPDY protocol by getting rid of "hostname sharding" (having one SPDY connection per domain) that makes SPDY behave similarly to HTTP/1.1 [2].

Config IV: SPDY+TM^3. This is the TM^3 version (8 pipes) of Configuration III.

The client uses unmodified Chrome browser. We pick 30 diversely designed websites[5] for automated testing implemented using the Chrome debugging interface. Each experiment consists of testing the 30 landing pages. Each test includes four cold-cache loadings (*i.e.,* caches are cleared) in the aforementioned four configurations, respectively, in a random order. The whole experiment was repeated until we obtain results for at least 50 successful tests for each website.

[5]Website categories: news (5), enterprise portal (4), retailer (3), photo sharing (2), technology (2), government (2), finance (1), online shopping (1), travel (1), movie (1), online radio (1), classified ads (1), sports (1), business review (1), social (1), encyclopedia (1), question answer (1), university (1). 25 of the 30 sites belong to the Alexa Top 100 U.S. sites.

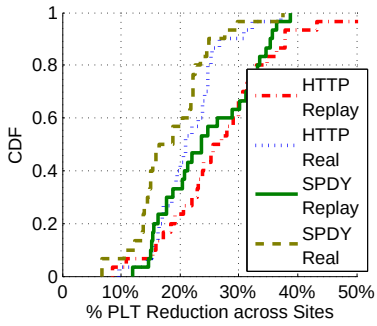

Figure 11: PLT improvement brought by TM^3 (8 pipes), compared to plain HTTP/SPDY, across 30 sites, over commercial LTE, using both real servers and the replay server.

As shown in Figure 10, the HTTP proxy can be connected to either a real web server or a replay server. For the latter, we use a modified Google Web Page Replay tool [1] to record landing pages of the 30 websites and host them on our local replay server. Using replay overcomes issues such as frequent content change of some websites (*e.g.,* `cnn.com`). We measure the RTT from RP (near the LTE gateway) to the real servers, and set the same RTT for the link between the middlebox and the replay server when replaying each website. The 25th, 50th, and 75th percentiles of measured RTTs are 3ms (due to CDN), 4ms, and 39ms, respectively.

6.2.1 Experimental Results

Figure 11 quantifies the effectiveness of TM^3 in four scenarios: using HTTP (SPDY) to load pages from the replay server (real servers). For example, the "HTTP replay" curve plots the distribution of $\frac{p_1-p_2}{p_1}$ across the 30 landing pages on the replay server, where p_1 and p_2 are the average page load time (PLT) for Config I and II, respectively. In all scenarios, the LP and RP communicate over LTE.

HTTP over TM^3 (Config I vs. II). Loading a webpage is a complex process interleaved with network transfer and local computation whose fraction is considerable (35% median estimated by [32]). Therefore the PLT reduction brought by TM^3 is less than that of file download. Nevertheless, for HTTP, TM^3 significantly reduces PLT for all websites. Across the 30 sites, the PLT reduction ranges from 8% to 51% with both mean and median of 27%. The benefits of TM^3 originate from multiplexing that removes overheads of connection setup (on pipes) and slow start overhead for short-lived TCP connections in HTTP/1.1. We observe a positive correlation between PLT reduction and the number of TCP connections, with Pearson correlation coefficient of around 0.5. This indicates that multiplexing, which leads to higher bandwidth efficiency, is more effective for pages transferred by a larger number of TCP connections.

SPDY over TM^3 (Config III vs. IV). Unlike HTTP

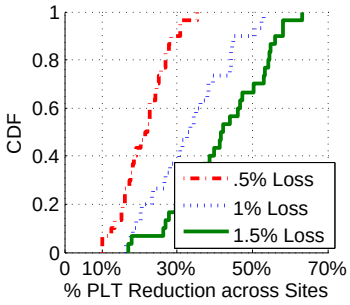

Figure 12: PLT improvement brought by SPDY+TM^3 (8 pipes), compared to plain SPDY, across 30 sites, over 10Mbps link with 50ms RTT and losses, using the replay server.

where the benefits of TM^3 come from multiplexing, for SPDY, TM^3 helps by inverse-multiplexing a TCP connection to multiple pipes. The PLT reduction ranges from 12% to 39% with mean and median being around 24%. Several factors affect the effectiveness of TM^3 for SPDY: page complexity, object sizes, and concurrency of connections.

Replay vs. Real Page Load. When fetching pages from real sites, the PLT reduction achieved by TM^3 is smaller but still considerable. For HTTP, TM^3 shortens PLT by 10% to 37% (mean: 22%, median: 21%). For SPDY, the reduction is between 7% and 37% (mean: 19%, median: 18%). The decreased benefits are attributed to factors including: *(i)* paths between RP and real servers possibly becoming longer and slower (in particular due to 3rd-party domains such as ads), and *(ii)* server delays with dynamic content generation.

6.3 Robustness to Random Losses

Following the previous subsection, we study how TM^3 helps improve web browsing performance under losses.

SPDY over TM^3. Figure 12 quantifies how much SPDY over TM^3 (Config IV) outperforms plain SPDY (Config III) under packet losses. Unlike Figure 11 that uses LTE, experiments of Figure 12 are conducted over an emulated wired link (10Mbps, 50ms RTT) with three loss rates (0.5%, 1%, and 1.5%, chosen based on prior SPDY study [32]), using page replay. As shown, over a lossy link, the advantage of using multiple pipes is even greater, because sporadic non-congestion losses usually only affect one or a subset of pipes, while the performance of the remaining pipes is unaffected. In contrast, for SPDY, any loss will force its only multiplexing channel to experience severe performance degradation. The median PLT reduction brought by TM^3 across the 30 sites for the three loss rates (0.5%, 1%, and 1.5%) are 22%, 33%, and 42%, respectively. More improvements are observed as the loss rate increases.

HTTP over TM^3. We study Config I and II under 2% packet loss, 10Mbps bandwidth, and 50ms RTT. We choose this configuration to compare with a recent study [18], which shows under the same network condition, QUIC's PLT is about 20% (40%) worse than HTTP for medium-size (large-size) page. To ensure apple-to-apple comparison, we generate the same synthetic pages[6], and use the same web server to host the pages, as well as the same client browser, as those used in [18]. The results are shown in Table 5. When 8 pipes are used, HTTP over TM^3 exhibits slightly better performance than plain HTTP/1.1 (due to multiplexing), and significantly outperforms QUIC and HTTP over a single pipe. Note that QUIC yields much smaller PLT than TM^3 with one TCP pipe, because QUIC's congestion control uses a modified loss recovery parameter compared to CUBIC: when loss happens, QUIC's multiplication decrease factor for cwnd reduction is 0.15, while CUBIC uses 0.3 (a similar fix to TCP CUBIC is suggested by [32]). Despite of this, since QUIC uses a single multiplexing pipe, it is still vulnerable to losses as shown in Table 5.

Table 5: Normalized PLT (2% loss, 10Mbps, 50ms RTT) with HTTP/1.1 as the baseline. QUIC results were reported by [18].

Page Size	Plain HTTP	HTTP over TM^3 (1 pipe)	HTTP over TM^3 (8 pipes)	QUIC
Medium	1	2.1	0.95	~1.2
Large	1	2.9	0.99	~1.4

Using multiple pipes exhibits advantages when random (non-congestion) losses happen. It will not help in the case of congestion losses that impact all pipes (this is mitigated by host-based congestion control, §6.6). Nevertheless, non-congestion losses may occur frequently (in particular in Wi-Fi [31]). Even for LTE, loss rate for certain QoS classes can still be high (*e.g.,* 1%) [9], making our approach beneficial.

6.4 Addressing Type-S/O Blocking by LMux

Type-S Blocking. We first evaluate TM^3's effectiveness on mitigating Type-S blocking by fetching a 4KB object while performing background download. Table 6 compares the 4KB object download time with and without LMux, over emulated 10Mbps link with 50ms RTT. In Table 6, we vary the sizes of pipe socket buffer and application buffer. Note that LMux and non-LMux versions of TM^3 use the same round-robin scheduling for connections. When LMux is not used, due to the shared buffers, the delivery of the small object is significantly delayed, and the severity depends on the shared buffer sizes. With LMux, the Type-S blocking is completely eliminated, leading to object load time of around 60ms, which is the same as the object load time when background download is not present. Also, in Table 6, LMux does not impact the background download throughput.

We then repeat the same experiment on LTE and present the results in Table 7, which shows trends similar to those in Table 6 with one difference: in LTE, applying LMux does not always reduce the 4KB object load time to the minimum, which is about 80ms. This is because of deep buffers *inside* LTE networks. When bulk transfers are in progress, they cause in-network "bufferbloat" that cannot be addressed by

[6]The "medium" page used in [18] consists of 40 jpeg images of 2.6KB each, and 7 jpeg images of 86.5KB each; the "large" page consists of 200 2.6KB jpeg images and 17 86.5KB jpeg images.

Table 6: 4KB object load time with background transfer, over 10Mbps link with 50ms RTT, averaged over 30 measurements.

# of Pipes	Pipe Send Buffer	App Buffer	4KB object load time (sec)	
			Without LMux	With LMux
1	default	0	0.24	0.06
1	default	2MB	2.30	0.06
1	2MB	0	1.23	0.06
1	2MB	2MB	2.67	0.06
8	default	0	0.31	0.07
8	default	2MB	1.98	0.07
8	250KB	0	1.22	0.07
8	250KB	2MB	2.75	0.07

Table 7: 4KB object load time with on-going background transfer over LTE, averaged over 30 runs. App buffer is set to 0 to assume no Type-S blocking at app layer.

# of Pipes	Pipe Send Buffer	4KB object load time (sec)	
		Without LMux	With LMux
1	default	0.89±0.20	0.17±0.04
1	4MB	1.06±0.26	0.19±0.05
8	default	1.12±0.62	0.34±0.18
8	500KB	1.46±0.65	0.37±0.19
1*	4MB	3.41±0.45	0.08±0.01
8*	500KB	3.77±0.35	0.10±0.01

*Rate-limit background transfer to 10Mbps.

LMux [22]. To confirm this, we rate-limit the background transfer to 10Mbps to reduce in-network buffer occupancy. As indicated by the last two rows in Table 7, doing so reduces load time to almost the minimum when LMux is used. Similar to Table 6, for all cases in Table 7, LMux has no impact on the background download throughput.

Type-O Blocking. Next, we examine how TM^3 handles Type-O blocking by downloading a 200MB file under various combinations of networks and pipe configurations. Table 8 measures the 75-percentile receiver-side pipe message buffering delay. Without LMux, when multiple pipes are used, the buffering delay increases as pipes' send buffers, which accommodate potentially unordered pipe messages, become larger. With LMux enabled, almost all receiver-side buffering delays are eliminated. We also verified that with LMux, the application-layer throughput matches well the transport-layer throughput, as shown in Figure 8.

Lastly, we quantify how TM^3 helps improve user experience for web browsing when concurrent traffic exists. Figure 13 plots SPDY PLT across 12 websites (randomly chosen from the 30 websites; other sites have similar results) over LTE when background data transfer is present. TM^3 is configured with 8 pipes and the replay server is used to serve the content. LMux helps significantly reduce the PLT, from 75% to 90%, without hurting the bulk download performance (not shown). Repeating the experiments on emulated wired network (10Mbps, 50ms RTT) yields similar PLT reduction from 75% to 90%, which is attributed to the elimination of Type-S blocking, although the background transfer can be impacted by Type-O blocking.

Table 8: 75-th percentile of receiver-side buffering delay of pipe messages. Measured by downloading a 200MB file. Wired link is 10Mbps with 50ms RTT.

NW	# of Pipes	Pipe Send Buffer	Buffering delay (sec)	
			Without LMux	With LMux
Wired	4	default	0.81	0.00
Wired	8	default	1.36	0.00
Wired	4	2MB	3.15	0.00
LTE	4	default	0.97	0.00
LTE	4	2MB	3.33	0.00
LTE	4	4MB	5.13	0.00
LTE	8	default	1.76	0.00
LTE	8	2MB	5.98	0.00

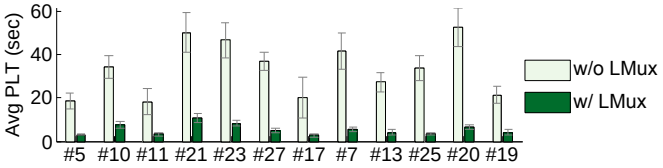

Figure 13: PLT improvement brought by LMux across 12 sites over LTE, averaged over 50 runs, using page replay. TM^3 (8 pipes) is used with SPDY. Background transfer is present.

6.5 Address Type-L Blocking by SCTP Pipes

All prior evaluations use TCP pipes to ensure apple-to-apple comparisons. Here we replace TCP pipes with SCTP pipes to show how SCTP overcomes Type-L blocking.

SCTP exhibits benefits when there are packet losses and multiple concurrent streams exist. We therefore design three experiments as follows. *(i)* Use one connection to download a file of varying sizes with and without loss. *(ii)* Load 12 web pages listed in Figure 13 using replay under 1% loss. *(iii)* Under 1% loss, simultaneously load web page and perform bulk transfer, whose throughput is set to 3Mbps. The link between LP and RP has 10Mbps bandwidth with 50ms RTT. In SCTP (lksctp 1.0.16 [4]), we set the number of streams to 512, and map connections to streams by hashing the connID.

We describe the results. For Experiment *(i)*, TCP and SCTP achieve statistically the same file download time because there is no concurrent transfer. For Experiment *(ii)*, the difference between TCP and SCTP is small, with the average HTTP PLT differing by less than 5% across the 12 sites, when 2 pipes are used. This is likely because the incurred Type-L blocking is usually not severe enough to lengthen the critical path in the object dependency tree (however, SCTP may shorten the time-to-first-byte [25] which we do not evaluate here). SCTP provides no improvement on SPDY that uses one connection. For Experiment *(iii)*, the bulk transfer causes much more severe Type-L blocking when multiplexed with web traffic, making SCTP noticeably outperform TCP. Figure 14 shows the reduction of PLT brought by SCTP pipes compared to TCP pipes across the 12 sites in four scenarios: using HTTP (SPDY) over 2 (8)

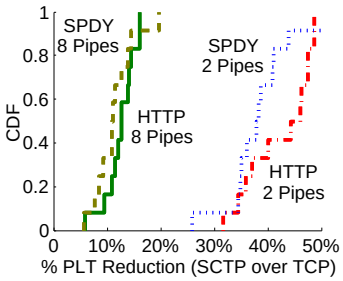

Figure 14: PLT improvement brought by SCTP across 12 sites over a 10Mbps link with 50ms RTT and 1% loss, using replay server. Concurrent data transfer at 3Mbps exists.

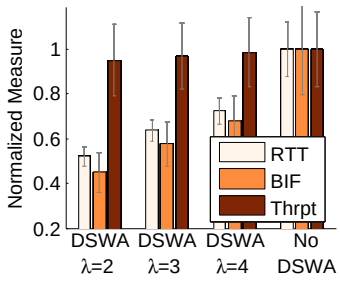

Figure 15: DSWA (Dynamic Send Window Adjustment) over LTE. Averaged over 100 measurements.

pipes. For 2 pipes, using SCTP reduces HTTP (SPDY) PLT by 32% to 49% (26% to 51%). However, if 8 pipes are used, the advantages of SCTP pipes are dampened, with PLT reductions only ranging from 6% to 16% for HTTP and 6% to 20% for SPDY. The explanation is that Type-L blocking does not occur across pipes (*i.e.,* a loss in Pipe 1 never blocks Pipe 2). Therefore, increasing the number of pipes increases chances for delivery without HoL blocking, thus diminishing the difference between SCTP and TCP.

As described in §3.2, changing pipe realization can also be done dynamically. To demonstrate this, we implemented a TM^3 plugin that dynamically switches between SCTP and TCP. We set up n TCP pipes and n SCTP pipes. TCP is used by default and traffic is temporarily directed to SCTP pipes only when the loss rate is greater than a threshold (*e.g.,* 0.5%), so we do not pay the price of SCTP's high overhead when the loss rate is low. The plugin takes less than 100 LoC. We measured no delay during pipe switching, and no disturbance to long-lived connections.

6.6 Host-based Congestion Control

Recall that TM^3 can use host-based congestion control to control the aggressiveness of multiple pipes (§3.4). To demonstrate this, we adapt Dynamic Receive Window Adjustment (DRWA) to multiple pipes. DRWA [22] is a delay-based CC that addresses excessive buffer occupancy (*a.k.a.* "bufferbloat") in cellular and potentially other networks. It limits per-TCP-connection congestion window by putting a cap on the TCP receive window, using end-to-end RTT as indicator. In DRWA, the receive window is continuously adjusted as follows: $R \leftarrow \lambda * RTT_{min}/RTT_{est} * C$. R is the receive window size. C is the estimated congestion window. RTT_{est} and RTT_{min} are currently estimated RTT and the minimum RTT (when in-network queues are almost empty), respectively. Parameter λ controls DRWA's aggressiveness.

We apply the concept of DRWA as follows. *(i)* We adjust the send instead of the receive window, as controlling the sender makes our approach more responsive. This is feasible because the RP (LP) has visibility of all out-going traffic for the entire client host. *(ii)* The window adjustment is performed in an aggregated manner, and each pipe receives an equal share of the overall adjusted window (although more sophisticated approaches exist). Therefore we have $S \leftarrow \lambda * RTT_{min}/RTT_{est} * C/n$ where S is *per-pipe* send buffer size, C is the *total* cwnd, and n is the number of pipes. We call our modified approach DSWA (Dynamic Send Window Adjustment).

We conduct an experiment over LTE by inverse-multiplexing a file transfer to 8 pipes with DSWA enabled (using parameters in the original DRWA paper [22]). We measure average RTT, bytes-in-flight (BIF), and throughput (Thrpt) in Figure 15. Compared to the scenario without DSWA, the RTT and BIF are reduced by 48% and 55% respectively, with only 5% throughput loss when λ is set to 2. Note DSWA can to some extent mitigate Type-S blocking by reducing the overall pipe send buffer, but is far from completely eliminating it compared to LMux. As DSWA is highly adaptive, even for λ=2, often more than 1MB of data remains in pipe buffers, in some cases up to 5MB. We also verified that DSWA has little impact on the web performance studied in §6.2 and §6.3 due the burstiness of web traffic.

6.7 Protocol Overhead

With TM^3 enabled, we run various workloads such as web browsing, file transfer, and video streaming on LTE, Wi-Fi, and wired networks. The measured protocol efficiency, defined as the total user data divided by the total size of all transferred pipe messages, ranges between 98.5% and 98.9%. It is less than 100% is due to pipe message headers and additional bytes used to prevent starvation for LMux.

7. RELATED WORK

Other Multiplexing Protocols besides SPDY and QUIC have been proposed. For example, Structured Stream Transport (SST) [20] is a transport layer protocol that more comprehensively encompasses ideas of multiplexing, hierarchical streams, and prioritization. PARCEL [29] uses multiplexing at the web object level to reduce PLT. Other similar proposals include HTTP Speed+Mobility [3] and Network-Friendly HTTP Upgrade [7], both supporting multiplexing. The PRISM system [23] employs an inverse multiplexer that leverages multiple WWAN (wireless wide-area network) links to improve TCP performance. The inverse-multiplexing approach in TM^3 has a conceptually similar idea but has a different goal of improving performance for short TCP flows over a single bottleneck link.

Late Binding. The high-level idea behind LMux is late binding for the data plane. The concept of late binding has been used in other scenarios. The Connection Manager (CM) system [16] performs late binding at application layer using callbacks. Doing so can avoid additional transport-layer buffering, but requires significant changes to the protocol stacks and the socket APIs, thus defeating our flexibility requirement. In [34], the authors employ interface late binding during handoff, to prevent unnecessary queuing for the old interface. The SENIC system [27] late-binds packet transmissions to the NIC to enable scalable rate limiting. In comparison, in LMux, the packets themselves are not late-bound. It instead performs *data-plane* late binding in the context of multiplexing. In addition, we address the starvation challenge by deriving the precise bound of the number of containers.

Multipath TCP (MPTCP) [28] allows a TCP connection to use multiple paths. TM^3's flexile pipes provide more opportunities for leveraging multipath: pipes can use either MPTCP, or single-path TCP associated with different interfaces. We leave multipath support of TM^3 as future work. Researchers also proposed MPTCP extensions that allow MPTCP to work with one interface [5]. While this is conceptually similar to inverse multiplexing (each MPTCP subflow can be regarded as a "pipe"), TM^3 further allows for both multiplexing and inverse multiplexing, and eliminates various types of HoL blocking in (inverse) multiplexing.

TCP-splitting Proxies or Performance-Enhancing Proxies (PEP) are in wide use in today's Internet [8, 33]. TM^3 takes the TCP splitting concept one important step further, by adding a splitting point on the client host. Doing so enables transparent multiplexing over the last mile link.

8. CONCLUSION AND FUTURE WORK

TM^3 improves application performance by strategically performing transparent multiplexing. It introduces several novel concepts such as flexible concurrent pipes, starvation-free late multiplexing, and inverse multiplexing, providing new insights into improving transport layer to better serve application-layer protocols including HTTP/2. In our future work, we plan to quantitatively compare TM^3 with other multiplexing-based web protocols such as QUIC. We also plan to leverage the pipe flexibility provided by TM^3 to improve user experience in diverse application scenarios.

Acknowledgments

We thank anonymous reviewers for their valuable comments. This research was sponsored in part by Indiana University Faculty Research Support Program – Seed Funding.

9. REFERENCES

[1] Google Web Page Replay Tool. https://github.com/chromium/web-page-replay/.
[2] Google's SPDY Best Practices. http://dev.chromium.org/spdy/spdy-best-practices.
[3] HTTP Speed+Mobility. http://tools.ietf.org/html/draft-montenegro-httpbis-speed-mobility-01.
[4] Linux Kernel SCTP Tools. http://lksctp.sourceforge.net/.
[5] Multipath TCP Support for Single-homed End-systems. https://tools.ietf.org/html/draft-wr-mptcp-single-homed-05.
[6] Netfilter/iptables Project. http://www.netfilter.org/.
[7] Network-Friendly HTTP Upgrade. https://tools.ietf.org/html/draft-tarreau-httpbis-network-friendly-00.
[8] Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. http://tools.ietf.org/html/rfc3135.
[9] Quality of Service (QoS) in LTE. http://4g-lte-world.blogspot.com/2013/01/quality-of-service-qos-in-lte.html.
[10] QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic.
[11] SPDY plugin for Apache 2.4.10. https://github.com/eousphoros/mod-spdy.
[12] SPDY Protocol Version 3.1. http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-1.
[13] Squid HTTP Proxy. http://www.squid-cache.org/.
[14] TCP Small Queues (TSQ). http://lwn.net/Articles/507065/.
[15] The Chromium Projects. http://www.chromium.org/Home/.
[16] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM*, 1999.
[17] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, 2015.
[18] G. Carlucci, L. D. Cicco, and S. Mascolo. HTTP over UDP: an Experimental Investigation of QUIC. In *ACM SAC*, 2015.
[19] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a SPDY'ier Mobile Web. In *CoNEXT*, 2013.
[20] B. Ford. Structured Streams: a New Transport Abstraction. In *SIGCOMM*, 2007.
[21] R. Jansen, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. Never Been KIST: Tor's Congestion Management Blossoms with Kernel-Informed Socket Transport. In *USENIX Security Symposium*, 2014.
[22] H. Jiang, Y. Wang, K. Lee, , and I. Rhee. Tackling Bufferbloat in 3G/4G Networks. In *IMC*, 2012.
[23] K.-H. Kim and K. G. Shin. Improving TCP Performance over Wireless Networks with Collaborative Multi-homed Mobile Hosts. In *Mobisys*, 2005.
[24] P. Natarajan, J. R. Iyengar, P. D. Amer, and R. Stewart. SCTP: An innovative transport layer protocol for the web. In *WWW*, 2006.
[25] 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.
[26] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *CoNEXT*, 2011.
[27] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for End-Host Rate Limiting. In *NSDI*, 2014.
[28] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *NSDI*, 2012.
[29] 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 *CoNEXT*, 2014.
[30] R. Stewart. Stream Control Transmission Protocol. RFC 4960, 2007.
[31] C.-L. Tsao and R. Sivakumar. On Effectively Exploiting Multiple Wireless Interfaces in Mobile Hosts. In *CoNEXT*, 2009.
[32] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, , and D. Wetherall. How speedy is SPDY? In *NSDI*, 2014.
[33] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan. Investigating Transparent Web Proxies in Cellular Networks. In *PAM*, 2015.
[34] K.-K. Yap, T.-Y. Huang, Y. Yiakoumis, N. McKeown, and S. Katti. Late-Binding: How to Lose Fewer Packets during Handoff. In *CellNet*, 2013.