Implementing and Evaluating Stock QUIC Client/Server Performance on DPDK TCP/IP Stack

X.D. Zhai
xingdaz@andrew.cmu.edu

Spencer Baugh
sbaugh@andrew.cmu.edu

ABSTRACT

1. INTRODUCTION

2. BACKGROUND

Our web experience is largely dominated by two factors: how fast the website loads, and how native and instantaneous the interaction feels. The former is determined by the latency of the plumbing, i.e. the network infrastracture, while the latter can be ascribed to the design, architecture and the engineering of the particular web service. To systemically and universally improve the experience, making the plumbing better is crucial.

From the bottom up, companies like Google, Facebook and Netflix have either utilized Content Distribution Networks(CDNs), or deployed their own, to bypass the middle mile, shorten the last mile and bring the content closer to the user. From the top down, Google's Chrome web browser has employed numerous software optimizations such as speculatively querying DNS, establishing TCP connections, and even firing off resource requests based on learned user traffic patterns to minimize the latency on the browser end.[?]

What is left in the middle are the stacks of protocols in need of overhaul for the modern web, and the people from Google are at it again. First, they developed the application layer protocol SPDY to address the pitfalls, particularly relating to latency, of HTTP/1.1 for delivering web pages. The motivation is that changes to transport layer are difficult to deploy because they all live in kernel space, and you get more bang for the buck if all the obvious and egregious handicaps in HTTP/1.1 can be addressed. Table 1 shows some of them and SPDY's remedy.

Table 1: HTTP/1.1 v.s. SPDY[?]

HTTP/1	SPDY
Single request per TCP connection	Multiplexed streams
Exclusively client-initiated requests	Server push
Uncompressed headers	Compression
Redundant headers	Elimination

On the client side, the Chrome browser has support for SPDY, and on the server side, Google donated the `mod_spdy` to the Apache HTTPD codebase to support the protocol. Benchmark testing revealed a 64% reduction in page loading over traditional HTTP/1.1. The SPDY protocol eventually formed the foundation of HTTP/2.

Google's quest for a faster web did not stop at the application protocol layer and the effort eventually spilled into the transport layer. If the main assumption in developing SPDY was "well, there is nothing we can do about suboptimal networking stack, let's do our best to optimize the application protocol," then the implicit assumption this round is "there is nothing we can do about the speed of light, let's reduce the number of trips needed for a connection."[?] Having realized that developing a kernel implementation of the new protocol would be slow for experimentation and deployment, the QUIC developers decided to have this new protocol live in userland and employ UDP layer to move its packets.

Since its stable release in April 2014, Google has ramped up the QUIC traffic to Google's services and analyzed its performance on a larger scale. Its empirical data suggest that it is 5% faster on page load on average, mostly due to 0-RTT connection establishment, and 30% less rebuffering for Youtube videos attributable to better handling of packet loss and stream level flow control.[?]

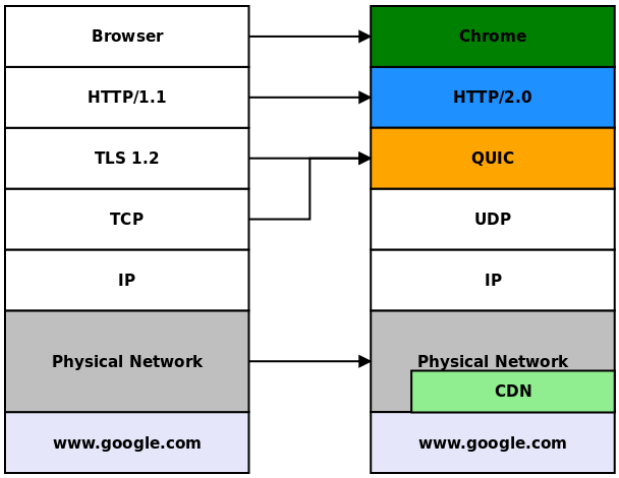

Figure 1: Efforts to make the web go faster

Figure 1 shows a rather promising landscape. It seems that what is standing between us and the promised land of seamless web experience, at least from a networking perspective, are the speed of light (which we can't change), and how fast raw bytes on the wire are processed by the kernel and handed to the application. That is where Intel's DPDK comes in. Data Plane Development Kit, first and foremost, is not a networking stack; it does not provide the ease and comform of Layer-3 forwarding, IPsec, firewalling, etc. It is a collection of low level libraries, NIC drivers and a runtime environment living alongside the kernel for building custom

applications in need of lightening fast packet processing.[?] Notably, pfSense laid out a roadmap where the core of it will be rewritten with DPDK to run at the line rate of 10Gbps interfaces. [?]

Hence, it is our intention to investigate if QUIC's stock client/server performance can be further elevated using DPDK. In this project, we make 4 main contributions:

1. Compare and contrast the difference between QUIC and TCP.
2. Explore and explain the QUIC protocol implementation and that of stock client server in the chromium source code.
3. Explore and explain the architecture of DPDK and DPDK-ANS.
4. Replace all Kernel UDP socket calls in the stock implementation with those of DPDK-ANS. Measure and evaluate the performance on both Linux networking stack and DPDK-ANS, against a TCP equivalent.

3. QUIC V.S. TCP

As a transport layer protocol developed from the ground up, QUIC is packed with new features aimed at incorporating application layer workarounds to TCP's shortcomings(e.g. HTTP/2's stream multiplexing), fixing inherent TCP problems(e.g. retransmission ambiguity), and optimizing for low latency and security.

3.1 Multiplexing and Flow Control

QUIC transplanted two important features from HTTP/2 and implemented them natively in the transport layer: stream multiplexing and dedicated flow controls.

HTTP/2 multiplexes many streams on top of TCP's single byte stream abstraction, but it still suffers from head of line blocking. Within a single TCP connection, there is no distinction between packets of different streams and a lost packet from one stream is still backed up behind those from a stalled one. In QUIC, multiple streams share the same connection and some streams can always make progress in the event of lost packets in others. As a result, QUIC also implements flow control on both the stream level and the connection level. A QUIC receiver keeps track of and advertises the window size for each stream to the sender, as well as the aggregate buffer size, bytes received and offset data. [?]

3.2 Connection establishment

Traditionally, the TCP 3 way handshake is the unavoidable cost of doing business on the web. A further 1.5 RTT is required to establish TLS. When all is said and done, 3 RTTs are wasted before application data actually flows. This is a very low hanging fruit for QUIC to pick off.

QUIC boasts of a 0 RTT for connection establishment largely based on the assumption that the client has talked to the server before and cached the server's configuration. Otherwise, it combines crypto with handshake in a dedicated stream(Stream ID 1) to initiate a connection.

The client sends an initial inchoate hello (CHLO) to the server, specifying among other things, the common certificates it already possesses and cached certificates from previous interaction with the server(if there are any). This allows the server to not send back the whole certificate chain later. The server responds with a rejection (REJ) containing most importantly, server configuration (SCFG), in which it provides a list of available key exchange algorithms and their corresponding public values. Then the client picks an algorithm and its own corresponding public value and resends the CHLO. At this point, the client doesn't even wait for the server's reply and immediately starts sending encrypted application data. [?]

3.3 Congestiong Control

QUIC does not reinvent the wheel when it comes to congestion control. At the time of this writing, it uses a modified implementation of TCP Cubic which is optimized for networks with high bandwidth and high latency. Since Linux 2.6.13, TCP Cubic is the default in the standard Linux distribution. QUIC expands on the TCP Cubic and primarily offers three advantages[?] over it:

1. All packets carry a new sequence number, avoiding the TCP retransmission ambiguity problem for RTT estimation.
2. All ACKs carry delay information between the receipt of the packet and the ACK being sent. Together with the monotonically increasing packet number, precise RTT can be calculated by the other endpoint.
3. ACKs support up to 256 NACK ranges, providing richer signaling than TCP in the event of packet reordering by the network.

3.4 Full Authentication and Encryption

A well known security problem of TCP is that its headers are in plaintext and not authenticated, susecptible to injection and manipulation, by either attackers or middleboxes. Because QUIC packets ride on UDP datagrams, its packets are always authenticated and the payload is encrypted after the handshake. [?]

3.5 Connection Migration

TCP connection is identified by the tuple `source address:port, destination address:port` specifying the end hosts, but not the logical connection between the client application and the web services it is accessing, making TCP unfriendly to mobile clients, e.g. a cell phone steps off WiFi network onto LTE or a laptop moving from one AP to another AP. QUIC addresses this problem by having the client assign a randomly generated 64-bit ID to the logical connection and it outlives all the migrations, enabling uninhibited migration across networks. [?]

4. DPDK AND DPDK-ANS

DPDK is born out of a marriage between the need for software level packet processing at line rate on commodity hardware and continued enhancements introduced by newer generations of Intel microarchitecture. The reality is that NICs at 1/10/40/100 Gb are not uncommon on servers these days but plain vanilla linux kernel is ill equipped to unleash all of their potential. On the other hand, Intel increasingly offers hardware optimizations that promises high performance packet processing but are left mostly untapped by applications. As a result, Intel embedded those features into a set of library with simple API interface and a low overhead run time environment for application developers. Table 2 summarizes the challenges and how DPDK overcome them. [?]

Table 2: Overcoming challenges of line rate packet processing in commodity hardware

Challenges	DPDK's optimizations
OS can't keep up with NIC I/O interrupts	Poll Mode Drivers for NICs
Context switch has high overhead	Bind a SW thread to a HW execution context, and the thread runs to completion.
Memory & PCIe access is slow	Perform batch read to amortize latency. Use SW or HW controlled prefetching. Align data structure to cache line size to minimize bandwidth usage. Ensure contiguous memory access in cache line size increments to minimize cache misses from evictions. Use Direct Data IO for PCIe access and read straight to cache.
Locks/semaphores have high overhead	Use lock free data structures.
Page faults are costly	Use 2MB or 1G Huge Pages in Linux to reduce TLB misses.

DPDK's architecture can largely be broken down into 3 main components: data plane libraries and poll mode drivers(summarized in Table 3), run time environment and the Environment Abstraction Layer(EAL). The run time environment is extremely low overhead and assigns core affinities to software threads which is allowed to run to completion. EAL is a service layer that provides its core libraries and applications a generic interface to system resources such as NICs, memory and PCI bus access, and logging. It straddles user space and kernel space. [**?**]

Table 3: DPDK core libraries

Core libraries	Description
Memory Manager	Allocate memory pools mapped to huge pages. Provides helper methods to pad data objects, stored in rings, for cache alignment and evenly distribute them on DRAM channels
Buffer Manager	Pre-allocate fixed size buffers stored in pools.
Queue Manager	Implements lock free queues for fast access
Flow classification	Utilizes Intel streaming SIMD intrinsics for hash based flow classification
Poll Mode Drivers	1GbE and 10GbE NIC drivers that work without asynchronous, interrupt-based signaling mechanisms

It is clear that DPDK only offers the highly optimized building blocks of a networking stack but just not an implemented one for us to use. It so happens that there is one on the market – DPDK Accelerated Network Stack(`dpdk-ans`). It is a port of FreeBSD's TCP/IP stack to be run in userspace as a DPDK application and it offers a subset (as it is still in active development) of the familiar set of socket API's we've come to love.

ANS fully utilizes DPDK's features. For example, it requires zero copy between NIC and the application. On NICs that support Receiver Side Scaling(RSS), the NIC distributes packets to different logical cores so that the same TCP flow go through the same core, minimizing the need for copying between cores. In addition, each core has its own UDP or TCP stack, making the implementation lock free. In addition, the socket layer tries to evenly distribute sockets to different logical cores so as to maximize the benefits of parallel processing. If there are two applications each using a socket, or a single application using two sockets, those sockets would be put on different cores. Figure 2 presents the complete picture of the logical building blocks of the DPDK TCP/IP stack. [**?**]

While DPDK-ANS promises great performance gain, it also creates its unique problems. First, its socket API is imcomplete; particularly `recvmsg` and `sendmsg` aren't implemented. Secondly, its sockets, and their associated file descriptors, do not live in kernel's universe and therefore, we cannot use system asynchronous I/O facilities such as `epoll` and rely on its analogue in the DPDK-ANS universe. These restrictions created many challenges to us when we were trying to preserse as many functionalities as we could while operating within the bound of the DPDK-ANS.

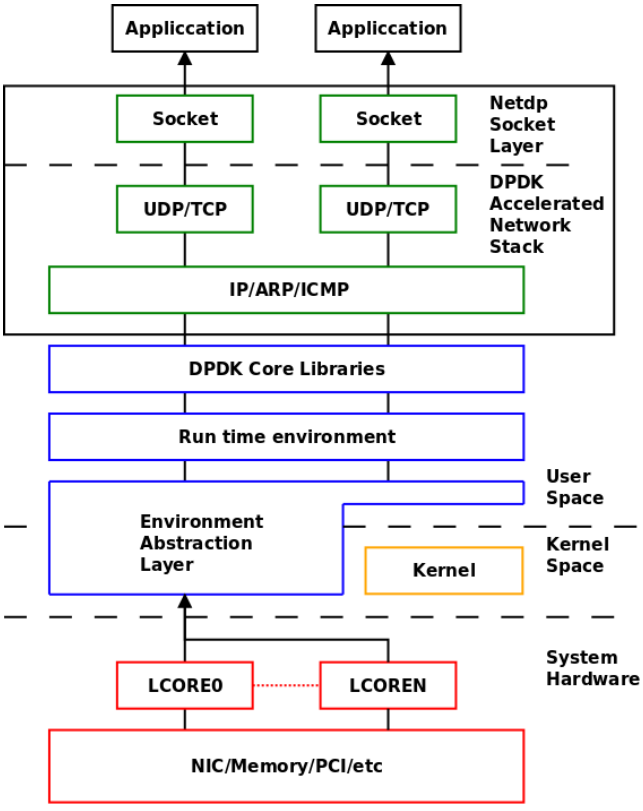

Figure 2: Overview of DPDK and DPDK-ANS architecture

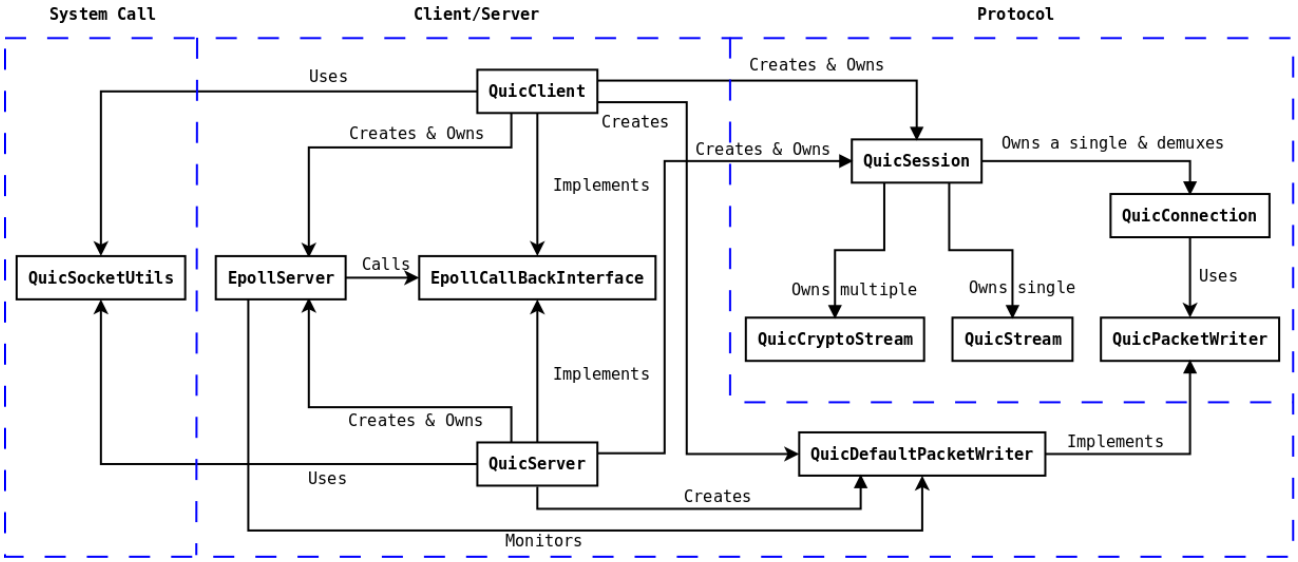

Figure 3: QUIC classes overview

5. IMPLEMENTATION

The QUIC protocol is folded into the Chromium project. Shipped with it is a pair of stock client and server implementation. Our end game was to isolate the protocol from the repo and swapped out all kernel socket calls by the client/server with `dpdk-ans` ones.

First, to tease out the quic protocol implementation and its dependencies into a library is no small task. Luckily, there is a third party standalone library called `libquic`[1] that is built with few modifications and patches from the Chromium project.

Second, we have to make sure that the stock client/server still works with `libquic`. Unsurprisingly and unfortunately, we had very little success due to countless dependency issues after pulling in the stock implementations and trying to build the binary with `libquic`. Our initial progress was greatly thwarted.

Just as we thought this project was stalled, we found out about another project called `quic_toy` that actually used the `libquic` library for performance testing between a watered down version of the stock client/server and a simple TCP counterpart. That was our way in.

5.1 Source Code Overview

Much of our time was spent on understanding the organization of the QUIC source code and looking for socket calls where the packet actually went onto the wire. Here is a overview of important classes in the implementation through the process of creating and using a `QuicClient`. `QuicServer` goes through a similiar process.

On construction, the `QuicClient` is handed a `EpollServer` and the server address. At initialization, the client creates a UDP socket, configures the socket options with utility functions from `QuicSocketUtils`, and finally registers the file descriptor and the callback `EpollCallBackInterface` functions it implements with the epoll server. Here is where most of the system calls are happening.

After concluding with the initialization, the client attempts to connect to the server at the specified address. It first creates a `QuicDefaultPacketWriter` that uses, but may not own, the underlying socket file descriptor, and implements the `QuicPacketWriter` interface. Next, the client creates a `QuicConnection` and hands it a `QuicSession` that demultiplexes the connection among multiple `QuicStream`s. Lastly, the client use a dedicated `QuicCryptoStream` to perform the handshake.

If successful, the client is ready to send and receive data from the server by creating and utilizing either a single or multiple streams. Incoming packets travel from the System Call side through `epoll` callback functions to client or server and onto the underlying protocol suite where states are kept for different streams. The reverse applies for outgoing packets.

5.2 Socket Call Replacements

The protocol classes do not create or own any file descriptors and therefore, we are not too interested in them. On the other hand, socket and epoll system calls largely concentrated in `QuicSocketUtils` and `EpollServer`.

6. BENCHMARK SETUP

Our benchmark setup consisted of two QEMU/KVM virtual machines, quic-server and quic-client, which respectively ran the server and client processes for our benchmarking and profiling. The virtual machines were located on different hosts, so the traffic between them crossed physical network links, rather than an emulated in-kernel bridge; this was desired to increase the realism of our profiling. Each VM had 4 virtual x86_64 CPUs each and 4GB of RAM, and had two Intel Corporation 82540EM NICs, emulated by QEMU, one for traffic and one for control. The VMs ran Ubuntu 14.04, running Linux 3.13, Ubuntu package linux-generic-3.13.0-85.91.

[1]Two months after we started using `libquic`, Google released an unofficial and unsupported `protoquic` standalone library contributed by the current QUIC developers as their side project. This repo is a lot heftier than `libquic`, and sparse with documentation. We decided not to migrate over.

Our testing harness ran a single server process on the server VM, and some number of client processes on the client VM. Each client process connects to the server process, reads off some amount of data sent by the server, then disconnects when the server indicates it is done sending data (with a FIN packet). This is done in a loop in each client process, repeatedly reconnecting, reading data, and disconnecting. The server process, meanwhile, just listens for connections, and immediately sends data followed by a FIN packet to each connecting client.

We used a single server process and client process implementation with both stock libquic using POSIX sockets (referred to hereafter as Linux QUIC) and our modified libquic using DPDK (referred to hereafter as DPDK QUIC). We simply compiled the server and client with both Linux QUIC and DPDK QUIC.

The server process was run under monitoring by the perf profiler. Therefore, we don't believe running under the perf profiler distorts our results. Running under perf allowed us to see how the performance profile of the server changed in different scenarios. Profiling with perf allowed us also to profile into the kernel, revealing how much time was spent in certain functions in the network stack, even down to the level of individual instructions.

We also monitored activity on the server network interface, using `ifstat` on the VM host, to determine the steady data transmission rate in each test scenario.

We tested both Linux and DPDK QUIC clients with both Linux and DPDK QUIC servers, with various different amounts of data transmitted. The parameter controlling how much data was transmitted was the number of times we transmitted the 1300-octet fixed buffer that made up our data; 1300 octets roughly corresponds to the amount of data that could fit in a single packet, so this parameter roughly correponds to the number of packets sent per connection.

7. EVALUATION

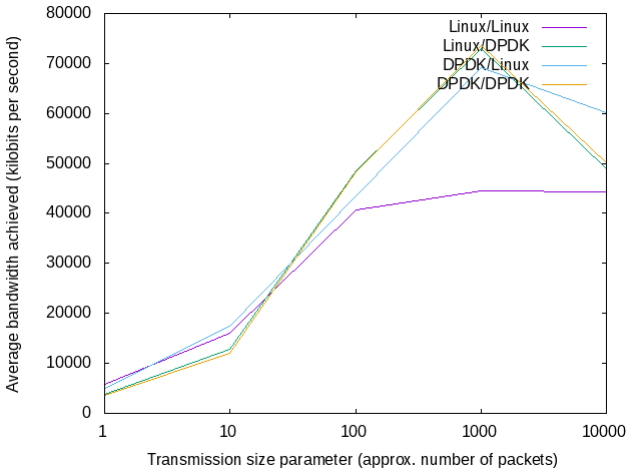

Figure 4: Results of running benchmarks with different client/server setups

A graph of our measurements can be seen in Figure 4.

As can be seen from the graph, when the server side of the benchmarking setup was using DPDK, we saw a significant performance increase in scenarios with moderately large data sizes.

However, at a higher value of our transmission size parameter, the highest we tested, when one or both sides of the benchmarking setup are using DPDK, our performance drops back down below that of Linux. It's possible that this is due to some scaling issue or bug in the DPDK network stack, but we have not yet been able to determine the cause.

Notably, scenarios with a small data transmission size and a DPDK server slightly underperform scenarios with a small data transmission size and a Linux server. These are scenarios where the server spends most of its time doing handshaking and cryptography, and only relatively small numbers of packets are flowing. So we believe that this inferior performance is simply due to DPDK not being optimized for small packet flows; it makes scalable design decisions that have high constant cost.

To gain some insights, we can look at the profile of the server. We've attached to this writeup the top 5 function hotspots for the server in each scenario, as output by the perf profiler. Note that since DPDK runs its network stack in a separate process outside of the kernel, time spent in the network stack is not shown in the profile for the DPDK versions. We do see network stack functions such as `e1000_xmit_frame` in the profile for the Linux versions, courtesy of perf's ability to profile into the kernel.

As the amount of data sent in a single connection increases, the relative overhead of the constant connection setup time decreases. This is visible in our profiling. For benchmarking scenarios with small amounts of data, the hotspots are in cryptographic functions, such as `freduce_coefficients`, `fproduct`, and other math functions starting with "f" (these functions are operations on field elements). In scenarios with larger amounts of data, network operations and datastructure manipulation become more significant, such as `e1000_xmit_frame` and management of the data-waiting-to-be-sent list, respectively.

Nevertheless, even at the largest data size, cryptographic functions remain in the top 5 hotspot functions, though the stream cipher used to encrypt the data replaces the public/private key operations. And even at the smallest data size, network stack function are still hotspots.

Since it is a reasonable assumption that the cryptographic code (being relatively small and standardized) is already well-tuned for performance, the obvious place to optimize in search of performance improvements at all scales is the network stack. Our results show that this can be a fruitful approach.

As well, QUIC internal datastructure manipulations pose a great scalability problem; as the amount of data being sent increases, these datastructure manipulations rapidly become the biggest hotspots. For the development of a scalable QUIC server, optimizations of these datastructures - in particular the ReliableQuicStream pending data list - is critical.

8. FUTURE WORK

Future work will include investigating some of the behaviors uncovered in our benchmarking. In particular, the improved performance but decreased scalability of the DPDK-based QUIC stack is not yet well understood; why does DPDK performance greatly decrease over a certain per-connection-data-size? Profiling and benchmarking DPDK itself will be helpful in determining the cause of this behavior.

Optimizations on the internal QUIC datastructures, especially the significant ones found in our profiling, may help counteract any scalability problems of the DPDK-based QUIC stack.

DPDK has a strong framework for parallel processing of packets; however, the existing QUIC stack is not well suited for parallel procssing. Making the QUIC stack support multithreaded or multiprocess operation would be very helpful.

More prosaically, porting our changes to the QUIC stack from libquic to the Google-associated protoquic will be important to give our work a greater impact; it is also valuable as a way to pickup performance improvements and bugfixes that have happened upstream since our version of libquic was last updated.

9. CONCLUSIONS

10. ACKNOWLEDGMENTS