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

QUIC (Quick UDP Internet Connection) is a new transport layer protocol designed and built by Google in an effort do deliver shorter latency in web loading and video watching experience for their Youtube services. Most notably, it combines crypto and transport handshakes, reducing the time for connection establishment to commonly 0-RTT. QUIC runs in user space as opposed to kernel space, allowing for fast experimentation with protocol. The protocol implementation is folded into the greater Chromium project which comes with a stock client/server to demonstrate the protocol. They are built on top of linux's network stack. In this project, we swapped out linux's network stack with one that is built on top of DPDK (called dpdk-ans) and compared the stock client/server's performance on both linux TCP stack and dpdk-ans. In addition, a similar TCP client/server is also build for comparison. We found blah blah results.

1. INTRODUCTION

QUIC is a new multiplexed and secure transport atop UDP build with HTTP/2 as the primary application protocol. I was desgined from the ground up by Google to deliver shorter latency in web loading and video watching experience for their Youtube services. It avoided all the pitfalls of TCP that make it unsuitable as a transport layer protocol in the modern web era. Put it another, QUIC = TCP + TLS + SPDY with purportedly better performance. Most notably, QUIC provides multiplexing and flow control semantics equivalent to HTTP/2 so HTTP/1 can run on QUIC yielding similar performance gain as HTTP/2, and connection can be established with fewer RTT as TCP.

2. BACKGROUND

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

From the bottom up, companies like Google, Facebook or Netflix have either utilized Content Distribution Networks(CDNs), or deployed their own, to bring the content closer to the user. From the top down, Google's Chrome web browser has employed numerous software optimizations such as (**include a couple of chrome optimizations here**) to greatly enhance the user end experience. Left in the middle are the stacks of protocols in need of overhaul for the modern web.

Put the network stack here. Comparison about what has been done to make replace and innovate in the stack.

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 for delivering web pages. The motivation was that changes to transport layer were difficult to deploy because they all live in kernel space, and you get more bang for your buck if all the obvious shortcomings in HTTP can be addressed. **What are the shortcomings? Put a table here for side by side comparison** SPDY achieved them by encoding more information per transfer via header compression and elimination, and multiplexing multiple HTTP request within the same TCP session to amortize the connection establishment cost. On the client side, the Chrome brower 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. The SPDY protocol eventually formed the foundation of HTTP 2.0 and chrome's support for it is ceased in 2016.

Google's quest for a faster web did not stop at the application layer and set it next target on developing a better transport layer protocol. If the the main assumption in developing SPDY/HTTP2.0 was "well, there is nothing we can do about networking stack latency, so let's do our best to optimize the application protocol," then the implicit assumption in developing the new transport protocol 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 adoption, the QUIC developers decided to have this new protocol live in userland atop UDP sockets for transporting its packets. QUIC was first rolled out in June of 2013 with a stable release on April 2014. Its implementation on the client side was folded into the greater Chromium project and a prototype server was shipped as well for people to experiment with the protocol.

Since its stable release, 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 mostly due to better handling of packet loss and stream level flow control.

It seems that what is standing between us and seamless web experience, at least from a networking perspective, are the speed of light (which we can't change), and how fast raw bits on the wire are processed by the kernel and handed to the application (be it the browser or the server). That is where 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 (**move this to references https://blog.pfsense.org/?p=1588**) where the core of it will be rewritten with DPDK to run at the line rate of 10Gbps interfaces.

Hence, it was our intention to see if QUIC's stock client/server performance can be further lifted using this fast packet processing library. In this project, we make 3 main contributions:

1. Explore and explain the chromium source code, particularly the stock client and server implementation.
2. Explore and explain the architecture of DPDK and DPDK-ANS.
3. Replaced all Kernel UDP socket calls with those of DPDK-ANS.
4. Measured and evaluated the stock client and server's performance on both Linux networking stack and DPDK-ANS, against a TCP equivalent.

3. QUIC VS TCP

This is a section of comparison between QUIC and TCP. How they are different and how is QUIC better at least in theory.

3.1 Connection establishment

Put wireshark packet capture for connection establishment here for comparison.

3.2 Packet Lost Handling

3.3 Congestiong Control

4. DPDK AND DPDK-ANS

This is a section where we put pictures of DPDK and DPDK-ANS architecture and explain what they do.

5. IMPLEMENTATION

This section we explain.

6. MEASUREMENTS

7. EVALUATION

8. CONCLUSIONS

9. ACKNOWLEDGMENTS