Transport-Level Performance of HTTP/3 and QUIC

Shannon Ding
sding2@andrew.cmu.edu

1 Introduction

In this sample-structured document, neither the cross-linking of float elements and bibliography nor metadata/copyright information is available. The sample document is provided in "Draft" mode and to view it in the final layout format, applying the required template is essential with some standard steps.

1.1 The Problem/Opportunity

This project aims to take a practical look at the design choices of HTTP/3 + QUIC, which is a new set of transport protocols for the internet. The Internet Engineering Task Force has classified HTTP/3 + QUIC as a Proposed Standard in June 2022 [1]. All major browsers have implemented a stable version of the protocol in 2019 and 2020 but the protocol is disabled by default. Other networking related libraries are also working on implementing the new standard.

The focus of HTTP3/QUIC designers is determining how their specifications theoretically solve the inefficiencies of older protocols like HTTP2/TCP. The focus of implementers is mostly about checking correctness of their implementation, such as if the data in a message is well formed and correctly ordered. The focus of users like website owners and Internet users is on high level performance metrics like overall time to load and total data sent.

The field could benefit from an intermediate look at lower-level performance metrics and how they relate to the HTTP3/QUIC design specifications.

1.2 Your Approach

I created several scenarios of client server interactions over HTTP/3 and HTTP/2. The scenarios involved sequences of fetching files of different size and either reusing or restarting the connection between the client and server. Then, I collected and analyzed the trace of data sent between the two. I compared and contrasted these low level metrics and tried to connect them to each protocol's design specifications.

1.3 Related Work

Performance Comparison of HTTP/1.1, HTTP/2, and QUIC [2]: This paper runs multiple implementations of HTTP3/QUIC servers arranged in a network and measures high level metrics like time to load. The author's used several different network conditions (percentage of lost packets, bandwidth limitations) as well as served different kinds of websites (number and size of resources part of each website).

A Quick Look at QUIC [3]: The major design decisions of QUIC and provides a good summary of the specifications.

QUIC Insight Tool [4]: In response to the difficultly of debugging QUIC issues, the author builds a tool to describe the low level behavior of a QUIC connection. The tool is a web app that takes in Wireshark trace of a QUIC connection and turns into a visualization. The author demonstrates being able to identify common issues like incorrect parameters, duplicate/missing packets, incorrectly ordered packets, interrupted connections, etc. through the app.

Looking Into QUIC [5]: An article introducing the contents of each QUIC packet. Uses the Wireshark packet analyzer tool to collect the packets.

1.4 Contributions

1. Graphs and metrics that demonstrate the short connection initialization process of HTTP/3 and the significantly lower latency between individual data frames and the entire connection in comparison to HTTP/2.
2. A collection of trace files of various protocols (http1.1, http2, http3) fetching different resources with different types of connections

2 Details

2.1 Evolution

My project was initially an attempt to prove the correctness of HTTP implementations. I read HTTP: The Definitive Guide [6] for background on the structure of the Internet and also specific details on the HTTP specification. After finishing the book, I realized the HTTP specification was deeply extensive: each chapter featured lists and tables of keywords and their required behaviors. The book also described common incorrect behaviors but they were almost all related to middleboxes in the Internet, which are computers aside from the client and server, like proxies and caches. In addition, their unresolved issues are several steps deep.

I moved on to read about HTTP/3 in particular through HTTP/3 Explained [7] and an article from Cloudflare [8] and found more concerns. Analyzing the correctness of implementations involves looking at the contents of data sent. All HTTP3 traffic is encrypted, so analyzing its contents are very difficult as an observer.

Finally, I looked at code for established implementations of QUIC including quiche by Google, msquic by Microsoft, s2n-quic by Amazon, and independent implementations like picoquic and quic-go. Each one had extensive test suite or checked for correctness by testing against a benchmark implementation of QUIC.

Some of the independent implementations explicitly said performance wasn't being prioritized at the moment in the rush to have a correct and working implementation. As a result, I decided to pivot from correctness to performance.

2.2 Goals

The traits of QUIC I wanted to look into were

1. **Performance of HTTP/2 vs. HTTP/3**: HTTP/3 is mainly described through its improvements upon HTTP/2. It's effective to describe HTTP/3's significance by comparing it to HTTP/2.
2. **Performance of reusing connections**: A major improvement in HTTP3 is the more concrete concept of a "stream". Both HTTP2 and HTTP3 have one transport level connection between the client and server. In order to serve multiple resources from a server (such as multiple images on the same webpage) over the single connection, each protocol uses creative ways to multiplex the single connection into separate streams for each independent resource. HTTP3 truly makes the streams independent of each other, unlike HTTP2 where a slowdown in one stream will affect all others (head of line blocking problem). As a result, HTTP3 should be better at moving multiple resources between a client and server.
3. **Performance by size of resource to fetch**: Another major improvement in HTTP3 is lowering the overhead of setting up and maintaining the connection so more of the network traffic is the actual application data from the server. Small resources will reflect the overhead of setting up the connection more. Large resources will reflect the overhead of maintaining the connection more.

2.3 Setup & Procedure

To achieve these goals, several tools were needed:

HTTP/2 and HTTP/3 client: The client should be able to send requests to the server and configure the protocol used and whether the connection should be reused.

HTTP/2 and HTTP/3 server: The server should be able to satisfy the client's configuration. In addition, the server should have several different sizes of resources it can serve.

Network tracing tool: The tracing tool should be able to log all traffic between the client and server including the headers and payload.

Afterward, experiments could be executed and logged on the set up system through a series of repeated steps.

1. Run the server. Set the network tracing tool to log to a file and run the tool.
2. Give the client a test script to execute.
3. Afterward, collect the trace file for analysis. Repeat with more test scripts.

2.4 Experimental Design

To cover the three performance factors I was interested in, I selected some client/server configurations to represent each one:

Protocol: HTTP/3 + QUIC or HTTP/2 + TCP
Connection: Reuse or restart connection
Size: Tiny resource (<10KB), small resource (10KB-1MB), large resource (1MB+)

3 Experimental Setup

3.1 Setup

Code for the experimental setup is in the repository: https://github.com/shannoding/07400

HTTP/2 and HTTP/3 client: curl is a client that uses ngtcp2 to implement HTTP/3 + QUIC. Only the development version of curl has HTTP/3 + QUIC which will need to be built from source. curl depends on quictls' fork of OpenSSL and ngtcp2.

Instructions: https://curl.se/docs/http3.html

I built curl with http3 in Ubuntu 20.04 WSL due to issues with building OpenSSL on Windows.

HTTP/2 and HTTP/3 server: Caddy is an HTTP server with HTTP/3 + QUIC support through the quic-go library.

Instructions: https://caddyserver.com/docs/install

I installed Caddy through Webi on Windows due to Caddy having difficulty accessing services through WSL. In addition, there are issues with WSL2 accessing localhost on Windows so I had to downgrade my WSL2 to WSL in order for the curl client to be able to message the server. The contents and configuration of my Caddy server are at caddy/.

The Caddy server is run on localhost:7443 and has three endpoints:

- */hello* serves a single short string
- */small* serves a 1MB file
- */large* serves a 10MB file

Network tracing tool: Wireshark is a network protocol analyzer that conveniently supports QUIC.

Instructions:
https://www.wireshark.org/docs/wsug_html_chunked/ChBuildInstallWinInstall.html

I installed Wireshark with npcap through the Windows installer. Trace files and "packet dissections" are under /wireshark-captures. I configured it to only look at traffic from the experiment:

- Use the "Adaptor for loopback traffic capture" network interface to only look at localhost traffic
- Filter traffic by tcp.port == 7443 || udp.port == 7443 because the test server is run on localhost:7443 and the two protocols that are being analyzed (HTTP/3 and HTTP/2) are run on top of the TCP and UDP protocols

Once set up, I followed a procedure for all my tests.

1. Run the Caddy server by calling `caddy start`. Start a capture of Wireshark with the correct filters and log to a .pcapng file named after the test.
2. Execute the curl test script (described in the Experimental Design below).
3. Once the test script finishes executing, stop and save the Wireshark trace. Stop the Caddy server by calling `caddy stop`. Repeat with more test scripts.

3.2 Experimental Design

The test scripts are the two bash scripts under tests/.

./closed-test [ENDPOINT] [NUM_REPEATS] [PROTOCOL]

./sim-test [ENDPOINT] [NUM_REPEATS] [PROTOCOL]

./closed-test closes the connection to localhost:7443 between repeated requests. ./sim-test keeps the connection to localhost:7443 open and reuses it between repeated requests. All requests are implemented using curl. Both expect the same three arguments.

[ENDPOINT] is which endpoint on localhost:7443 to call. The valid values are "hello" (tiny resource), "small" (1MB resource), or "large" (10MB resource).

[NUM_REPEATS] is the number of times to request the resource.

[PROTOCOL] is which network protocol to use. Valid values are "http3", "http2", and "http1.1".

The names of trace files that result from a test script follow a [ENDPOINT]-["closed", "sim"]-[PROTOCOL] pattern. For example, the trace from running `./closed-test small 3 http2` which creates three separate http2 connects to localhost:7443/small is saved in a file called small-sim-http2.pcapng.

3.3 Transforming the Trace Files

After collecting all trace files, I further narrowed down the fields I was interested in by having Wireshark generate a "packet dissection", which is an abbreviated log file where each frame is represented by a few fields and frames are ordered like rows in a table.

The fields I displayed in particular are:

- Frame number: Wireshark generated number for the frame. A frame is the very lowest level unit of data transport. A frame can contain multiple TCP/QUIC packets.
- Time: The time a frame was received relative to the first frame Wireshark sees in a session.
- Source port: The locahost port that the frame was sent from.
- Destination port: The localost port that the frame was sent to.
- Frame length: The total length of the frame, including headers.
- Packet type: The HTTP/3 type of packet. Common values include: "Initial", "Handshake"
- Header form: The HTTP/3 type of header. Either "Long" or "Short".
- Connection number: A number HTTP/3 servers use to distinguish different connections.
- Stream number: A number HTTP/3 servers use to distinguish different streams within a connection.
- Info: General string of summary information

The packet dissections are the .csv files under wireshark-captures/

4 Experimental Evaluation

See the additional Figures pdf for the results.

4.1 Surface Analysis

HTTP/3 is significantly faster than HTTP/2. According to Figure 1, HTTP/3 completed the tests in 20.6% of the time it took HTTP/2 on average.

Looking at Figures 2 through 9, there are common stages a connection goes through

- For both HTTP/2 and HTTP/3, there's a connection stage and then an application data sending stage.
- During the connection stage, the client sends its largest packets (the request) and the server replies with similar sized packets.
- Then, the application data sending stage has the server sending very large packets and the client regularly replies with a small ACK.
- The start of the application data sending stage usually is slow but reaches a maximum rate eventually and stays

at that maximum rate until a major event affecting the connection.

- The ACK is also very uniform and consistent although the rate at which the ACK is sent also slows down (allows multiple application data frames before needing to ACK).

As defined in the specs, HTTP/3 has fewer connection start up frames. Likewise, the time it takes for application data to start flowing in an HTTP/3 connection is around 25% of HTTP/2 connections in the four tests.

4.2 Deeper Analysis

4.2.1 HTTP3 frames arrive much faster and closer together than HTTP2

Part of the improvement of building upon UDP is its non blocking nature. Packets inside frames don't have to wait for any others. The HTTP/3 tests used significantly more frames but within a significantly small time range indicating little waiting.

4.2.1 HTTP3 keeps continuous requests more continuous

HTTP/3 has a clearer concept of streams. As expected, during the sim tests, the connection ID remains the same while stream ids change. HTTP/2 has a similar concept, but its underlying use of TCP means it doesn't truly behave like an independent stream of data.

At the start of each HTTP/2 request, the rate of data transfer from the server is low at first but slowly reaches the max rate (TCP slow start). During the sim tests where the HTTP2 client was supposed to stay on the same connection, the client took a while to send the next request and told the server to send less data through "window size changed" messages. As a result, Figure 9, has noticeable dips during the client requests, but less than Figure 7.

The HTTP3 client manages to keep the connection much more continuous. Perhaps it already sent the full request at the very beginning of the connection so it didn't need to re-request. This is strange because curl's documentation says the client sends the request one after another.

For both HTTP/2 and HTTP/3, there is improvement with keeping the connection active.

5 Surprises and Lessons Learned

The extent of HTTP/3 + QUIC speed up was surprising. There are high level metrics showing the speedup but it's much more impactful to see all the waiting and gaps in the HTTP/2 activity graph and the fast paced, continuous transfer in the HTTP/3 graph.

The data indicating HTTP/3 uses significantly more data is surprising to the point of uncertainty. I kept the settings between the HTTP/3 and HTTP/2 the same, except for the protocol used so I'm surprised at the difference.

A lesson learned from a research project in general is how much a single source like a blog article or paper can change a project. I spent a lot of time trying to figure out a experimental setup and after implementing it, found a source that would have been easier and better.

5 Conclusions and Future Work

HTTP/3 enables impressive speed up in client server interactions. Through analysis of the data transfer behavior, the improvement seen in the test cases can largely be attributed to how HTTP3 frames arrive much faster and closer together than HTTP2 thanks to UDP's nonblocking nature and how HTTP3 keeps continuous requests more continuous related to from its progress on streams.

5.1 Next Steps

There were several suspicious conclusions from the data, so it would be helpful to address them first as it may uncover fundamental flaws in the experimental design. In particular, HTTP/3 using more data and HTTP/2 pausing in between simultaneous connections.

The client and server run in this experiment were not real world clients and servers. In particular, neither holds a CA SSL certificate and thus were configured to intentionally ignore the certificate verification step of the protocol. Furthermore, real world websites tend to serve tens of different file types. Our tests only used three.

Lastly, Wireshark interpreted data is not perfect. The QUIC Insight tool [4] addressed many of the issues of Wireshark. More data cleaning or a different trace tool would help the quality of data.

REFERENCES

[1] https://datatracker.ietf.org/doc/html/rfc9114
[2] https://www3.cs.stonybrook.edu/~arunab/course/2017-1.pdf
[3] https://web.cs.ucla.edu/~lixia/papers/UnderstandQUIC.pdf
[4] https://quic.edm.uhasselt.be/files/quicinsight_JonasReynders_August2018.pdf
[5] https://blogs.keysight.com/blogs/tech/nwvs.entry.html/2021/07/17/looking_into_quicpa-pUtF.html
[6] http://www.staroceans.org/e-book/O'Reilly%20-%20HTTP%20-%20The%20Definitive%20Guide.pdf.
[7] https://http3-explained.haxx.se/en
[8] https://blog.cloudflare.com/http3-the-past-present-and-future/