Performance Evaluation of HTTP/3 as an Interservice Communication Mechanism

Bachelor of Science Thesis in Software Engineering and Management

MISLAV MILICEVIC

SOFIA SJÖBLAD

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023

Performance Evaluation of HTTP/3 as an Interservice Communication Mechanism

© MISLAV MILICEVIC, June, 2023.
© SOFIA SJÖBLAD, June, 2023.

Supervisor: MIROSLAW STARON
Examiner: PHILIPP LEITNER

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
University of Gothenburg
Chalmers University of Technology
Gothenburg, Sweden 2023

Performance Evaluation of HTTP/3 as an Interservice Communication Mechanism

1st Mislav Milicevic
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
gusmilicmi@student.gu.se

2nd Sofia Sjöblad
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
gussjoblso@student.gu.se

***Abstract*—The HTTP protocol is widely used for communication and data transfer on the internet. HTTP requests are sent from clients to servers, and the servers respond with the requested data. HTTP is not limited to web pages, but can be used for any service that provides resources when requested. The most recent version of HTTP, HTTP/3, uses a new transport protocol called QUIC, which provides increased performance by improving on the connection handshake and multiplexing processes. This paper aims to perform a performance analysis of HTTP/3 in an interservice communication system, focusing on latency and throughput. The research will compare HTTP/3 to HTTP/2, and also explore the practical implications of the performance analysis. To gain real-world insight, the study will partner with WirelessCar, an automotive software company that maintains high throughput services built on HTTP.**

I. Introduction

A large amount of services on the internet use the HTTP protocol as a means of communication and data transfer. For a common user of the internet, the HTTP protocol is not something that can be visually perceived, however it serves as the foundation of content accessibility. The most common scenario where an internet user would encounter HTTP is when visiting a web page. When we enter a URL into a web browser, an HTTP request is created and sent to a server. After receiving the request, the server creates an HTTP response and sends it back to the web browser. In this case, the response is the web page we see after visiting the URL.

The use of HTTP goes far beyond the retrieval of a web page. In a software engineering context, web page retrieval is an example of a client-server model, for which the HTTP protocol is, more often than not, used. To put the client-server model in simple terms - a client sends requests to a server, the server sends responses to received requests back to the client. In our web page scenario, the web browser would be the client and the server containing the web page would be the server. However, the client isn't always a web browser and the server doesn't always contain web pages.

To generalize the client-server model, we can define the server as any service that can provide resources when they're requested and the client as any service that can request resources and receive them. In many cases, a service can simultaneously be a client and a server. This is a very common occurrence when dealing with systems that implement interservice communication over HTTP.

The HTTP protocol has had several major revisions over the years, the most notable being HTTP/1.1 and HTTP/2. HTTP/1.1 suffers from several issues on the application layer, the most prominent being head-of-line blocking [1] as a side effect of request pipelining [1]. Client implementations mitigate this issue by opening multiple TCP connections to a server. However, opening and maintaining multiple TCP connections can be resource intensive. HTTP/2 solves this issue by multiplexing requests over a single TCP connection [1]. With multiplexing in place, resource intensive requests no longer block subsequent requests. However, multiplexing introduced its own set of issues. If a packet is lost during a request/response transaction, all active transactions across all streams on the same connection are stalled. This occurs due to the fact that TCP's loss recovery mechanism doesn't take into account the parallelism introduced with multiplexing [2]. This is yet another case of head-of-line blocking, only this time on the connection level.

In June 2022, HTTP/3 has become a proposed standard and it offers features designed to overcome the shortcomings of HTTP/2. Instead of TCP, HTTP/3 uses a new transport protocol in the form of QUIC. QUIC is UDP-based and it implements stream multiplexing [3], as opposed to connection multiplexing introduced with HTTP/2. This means that if a packet is lost on one stream, only that stream is stalled, instead of the entire connection. This redesigned transport layer is supposed to provide an increase in performance compared to its TCP counterpart. HTTP/3 uses the same HTTP semantics as HTTP/2 and HTTP/1.1, leaving the means of interaction between a client and a server unchanged, even though the transport layer is completely different.

For this research paper, we will be performing a performance analysis of HTTP/3 in a system where communication between two or more services is conducted over HTTP. Since the HTTP/3 protocol is quite new, there is limited research exploring its performance. The majority of research [6]–[9] at this point in time focuses on the protocol's performance in the browser. Thus, leaving a research gap for studying potential performance benefits of HTTP/3 in an interservice communication system. In an interservice communication system, the focus primarily lies on the number of request/response transactions conducted over a period of time, rather than the amount of data transferred (in the case of the browser). With this in mind, the focus of the performance analysis will be on latency and throughput. As a reference point, we will be using an equivalent system built on top of HTTP/2.

Apart from performance, we are also interested in whether or not these potential performance benefits have enough value for HTTP/3 to be considered a viable option at this point in time, as well as the real-life implications of the performance analysis. While the performance analysis may provide great results in theory, it is difficult to draw a conclusion on the practical application of HTTP/3 based on a performance analysis alone. To get an accurate, real world understanding, we've partnered up with WirelessCar, an automotive software company. WirelessCar develops and maintains a large number of high throughput services during its daily operations and a lot of them are built on top of HTTP. We hope to get valuable insight about the problems we are tackling from the very people that work with these types of services in their everyday work.

II. Background

In this research paper, we conduct a computational experiment to determine whether or not HTTP/3 presents significant performance advantages over its predecessor. In order to understand what the origin of these potential advantages might be, it is important to understand the fundamental design of both HTTP/3 and HTTP/2, and where they differ. According to Thomson et al. [1], there is a link between application performance and the underlying transport of each HTTP version. With this in mind, this section will strongly focus on the underlying transport layers for both HTTP/3 and HTTP/2, how they are used within the protocol implementation and how they differ from one another. Additionally, the HTTP semantics, as well as the HTTP/3 and HTTP/2 application layers will be covered to gain an understanding of how these different layers are integrated.

A. Protocol stack overview

Before going into the specifics of how each version of HTTP protocol works, we should get familiar with the protocol stack that is common across all versions of HTTP. The protocol stack shared by all current HTTP versions consists of 5 parts:

1) HTTP Semantics
2) Application Layer
3) Security Layer
4) Transport Layer
5) Network Layer

Figure 1 is a visual representation of this stack for HTTP/2 and HTTP/3.

B. HTTP Semantics

Over the years, the implementation of the HTTP protocol has changed in numerous ways, but one part that has remained consistent are the HTTP semantics. This immutable nature

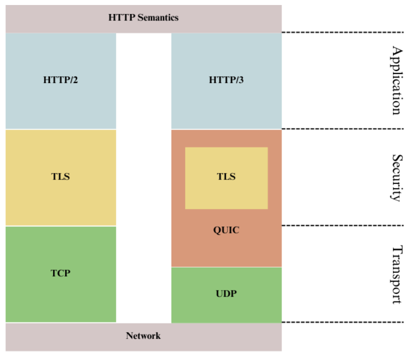


Fig. 1: HTTP stack

of HTTP semantics provides the benefit of hiding underlying details of the protocol implementation while providing clients with a uniform way of accessing resources on a server [12].

Previously, we have discussed the client-server model that HTTP is based on. The concept of clients and servers are a fundamental part of HTTP, but that is only a small fraction of concepts covered by HTTP semantics. To get a better understanding of the core functionality that HTTP provides, we need to examine various core HTTP concepts in greater detail.

Clients and servers exchange data through messages. How these messages are transferred over a connection largely depends on the version of HTTP that is being used. Previously we have mentioned that HTTP semantics abstract the underlying details. One of these abstractions comes in the form of a version independent message format that conveys the same meaning and characteristics of a message over all major HTTP versions, regardless of the underlying transport [12]. A message defined by this format contains:

- control data
- content
- headers
- trailers

When a message is sent over a connection, framing data is sent along with it. Framing data describes the beginning and the end of a message. In versions prior to HTTP/1.1, the framing of a message was linked to the lifecycle of a connection. A connection closure indicated a message end. With HTTP/2 and HTTP/3, connection bound framing is not possible due to the fact that multiple messages can be sent over the same connection. Because of this, the message length is used as a framing mechanism [12].

This message format is consistent across all versions of HTTP, however the syntax for constructing these messages can vary from version to version. For instance, in HTTP/1.1 messages are expressed in raw text, while in HTTP/2 and HTTP/3 binary communication units called frames are used to express messages.

C. Application Layer

When talking about HTTP semantics, we mentioned how every version of HTTP has a syntax defined for constructing messages. In HTTP/1.1 this syntax was expressed through raw text, in HTTP/2 and HTTP/3 a concept of frames is used. Our main focus in this research is the performance of HTTP/3 and how it compares to HTTP/2. With this in mind, in this section we will primarily focus on how messages are created in those two protocols and how those messages are mapped to the underlying transport.

HTTP/2 and HTTP/3 share a very similar frame model. Both protocols support a variety of different frame types that support different aspects of the protocol. When a request/response transaction occurs, the most common frame types we will observe are *HEADER* and *DATA* frames. If we were to map these two frame types to the message model described earlier, *HEADER* frames are meant for transmission of control data, headers and trailers, and *DATA* frames transmit the content of the request/response.

When a request/response is sent over a connection, they can be seen as a sequence of frames. For instance, a server might receive one or more *HEADER* frames and a *DATA* frame after that. This sequence of frames that is exchanged between a client and a server is called a stream. Frames and streams serve as a basis for multiplexing. Multiplexing is a feature that is present in both HTTP/2 and HTTP/3, but its implementation is quite different in both protocols.

Prior to HTTP/2, an HTTP connection could only handle a single request/response at a given moment. This means that if multiple requests were needed to be executed in parallel, multiple connections needed to be opened. In HTTP/1.1 this was partially solved with the introduction of request pipelining, but this introduced an issue of head-of-line blocking in case there was a sufficiently large request at the front of the pipeline blocking execution of requests further down the line. Multiplexing solves this problem by allowing multiple requests to be sent over a single connection in a non-blocking manner. Earlier we mentioned that multiplexing is implemented differently in HTTP/2 and HTTP/3. More specifically, in HTTP/2 multiplexing occurs on the application layer, while in HTTP/3 it occurs on the transport.

HTTP/2 uses TCP as its underlying transport protocol, in which a connection can be seen as a single stream of data. To optimally map multiple streams onto this connections, frames from different streams are interleaved/multiplexed. Figure 2 shows how multiple requests/responses are multiplexed over a single TCP connection.

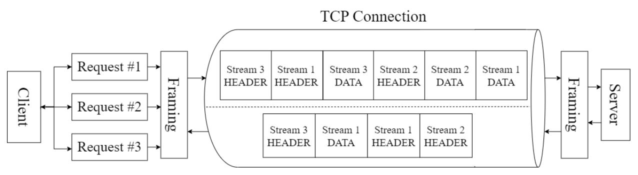


Fig. 2: HTTP/2 Multiplexing

Because HTTP/2 multiplexing is managed on the application layer, HTTP/2 frames contain more data than HTTP/3 frames, for the purpose of managing the stream information of a request.

Another key area where HTTP/2 and HTTP/3 differentiate is the field compression algorithm that they use. Before an HTTP message ends up on the underlying transport, the headers and trailers of that message are compressed to reduce bandwidth usage over the network. HTTP/2 uses a compressor called HPACK [13]. One of the assumptions that HPACK makes during the compression process is that frames across all streams will retain their order. QUIC, the transport used by HTTP/3, does not guarantee this order retention [3]. Using HPACK on an out of order stream would cause head-of-line blocking. To circumvent this, HTTP/3 uses a variation of the HPACK compressor called QPACK [14]. QPACK retains a lot of the core design from HPACK, while introducing support for our of order delivery.

D. Transport Layer

The transport layer is responsible for keeping an open connection between the client and server and transferring data between them. HTTP/3's transport layer uses QUIC while all previous versions of HTTP uses TCP. While TCP and QUIC are both responsible for data transfer over a connection, they offer different properties.

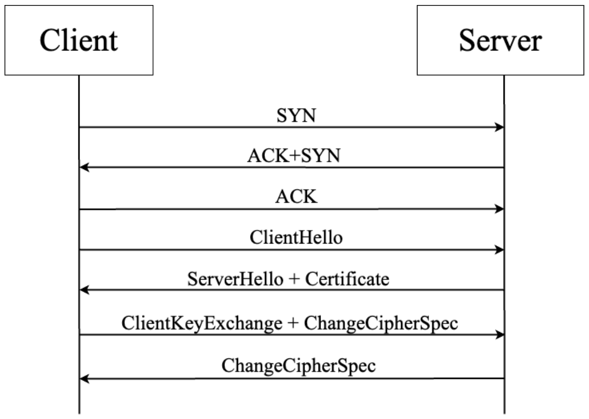
Before a connection between a client and server can be established they undergo an initial handshake process that verifies each party. One of the key differences between TCP and QUIC, is that QUIC has structured that handshake process in a way that allows data to be streamed as soon as possible. QUIC's structure allows clients to cache information about the origin that can be used later to achieve a 0-RTT handshake.

To fully understand how the handshake process introduced by QUIC improves upon the one of TCP, we need to analyze TCP's handshake in more detail. Every octet of data transmitted over a TCP connection has a sequence number [16]. A sequence number being attached to every octet of data allows for every octet to be acknowledged when received. A client and server exchanging data both have an initial sequence number (ISN). For a TCP connection to be established, the ISN of both client and server need to be synchronized. This synchronization is done in a three-way handshake process by exchanging SYN and ACK control bits along with the ISN [16]. A three-way handshake would occur as following (Figure 3):

1) Client sends SYN control bit with ISN to server
2) Server sends ACK control bit with client's ISN and SYN control bit with its own ISN
3) Client sends ACK control bit with server's ISN

With HTTP/2, this process is further extended with a TLS handshake (Figure 3):

1) Client sends a ClientHello [17] message to server
2) Server sends a ServerHello [17] message and a certificate to client
3) Client sends ClientKeyExchange [17] and ChangeCipherSpec [18] messages to server



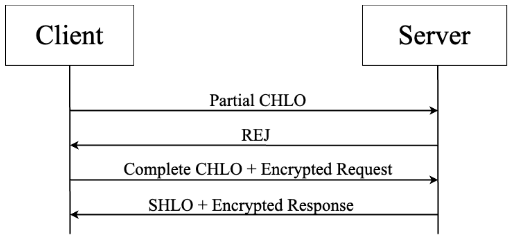
Client
Server
SYN
ACK+SYN
ACK
ClientHello
ServerHello + Certificate
ClientKeyExchange + ChangeCipherSpec
ChangeCipherSpec

Fig. 3: TCP + TLS Handshake

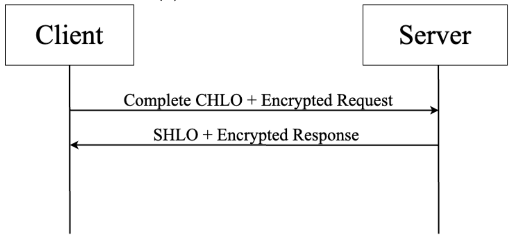
4) Server sends ChangeCipherSpec message to client

To improve on the number of roundtrips needed to establish a connection, QUIC utilizes a 1-RTT to 0-RTT handshake process. The handshake process is conducted by exchanging *ClientHello* (CHLO) , *ServerHello* (SHLO) and *Reject* (REJ) messages between the client and the server (Figure 4). If a client and a server are interacting for the first time, the client does not have the information necessary to initiate a successful handshake. In this case, a 1-RTT handshake will be conducted (Figure 4a). To retrieve this information, a client forces a REJ message from the server by sending a partial CHLO message. The REJ message retrieved from the server contains a server config including the server's long-term Diffie-Hellman [18] public key, a certificate chain authenticating the server and an authenticated-encryption block with the client's public IP [15]. The server's long-term Diffie-Hellman public key and the client's short-term Diffie-Hellman private key are used to calculate the initial keys for the connection. After the initial keys are obtained, the client can initiate the handshake by sending a complete CHLO message containing its short-term Diffie-Hellman public key along with the data encrypted with the initial keys. If the handshake is successful, the server will respond with a SHLO message containing its short-term Diffie-Hellman public key. Once both the client and the server obtain each other's short-term Diffie-Hellman public values, both can calculate the final keys for the connection. To perform a 0-RTT handshake, the client uses the calculated connection keys in all subsequent requests (Figure 4b). When a 0-RTT handshake is successfully performed, latency is inherently reduced because an extra roundtrip is entirely avoided.

Earlier we discussed the process of multiplexing requests and how it solves the problem of head-of-line blocking for HTTP/2. However, what we did not discuss at that point is a drawback of performing multiplexing on the application layer. With HTTP/2, the request/response frames are multiplexed and then sent onto the TCP connection as a stream of packets. Once these packets reach the receiver, they are de-multiplexed into individual requests/responses as shown by figure 2. This is how an optimistic flow of a request/response transaction would



(a) 1-RTT Handshake



(b) 0-RTT Handshake

Fig. 4: QUIC Handshake

run its course. In a more pessimistic scenario, during a request/response transaction a packet would be lost. If this were to happen, TCP's loss recovery mechanism tries to recover the lost packet by retransmitting it over the connection. When we were talking about HTTP/2's application layer, we mentioned HPACK and how frame order retention is crucial to make it work. This means that once the frames are on the connection, they need to be received in the order they were sent. In the case of a lost packet, the only way to retain the order in which frames are received is to block all frames from being processed until the lost packet is retransmitted and received. Consequently, because TCP's loss recovery mechanism is not aware of the existence of multiple streams on the application layer and it only observes a singular stream of data on the connection, when a packet is lost on one stream, all streams are stalled until that packet is retransmitted and received. This is yet another case of head-of-line blocking, only this time on the actual transport.

The multiplexing implementation in HTTP/3 retains a lot of the core concepts introduced in HTTP/2, while tackling issues created by the HTTP/2 implementation. As we mentioned earlier, TCP's loss recovery mechanism is not aware of the existence of multiple streams, causing a stall of all streams in case of a packet loss. HTTP/3 solves this issue by moving the concept of streams to the connection level. QUIC provides its own implementation of streams, meaning in case of a packet loss, the connection is able to determine the stream that is experiencing the loss. To retain stream information across the connection, stream data containing HTTP frames is transported by QUIC STREAM frames [3]. After STREAM frames are created, they are interleaved into one or more QUIC packets. Figure 5 shows how multiple requests/responses are multiplexed over a single QUIC connection.

This implementation of multiplexing significantly reduces the amount of blocking that occurs due to lost packets.

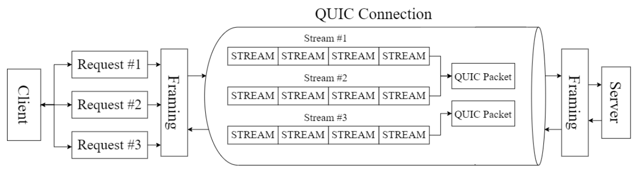


Fig. 5: HTTP/3 Multiplexing

However, it is important to mention that it is still possible for multiple streams to be stalled. Because a QUIC packet can contain STREAM frames from multiple streams, every stream associated with that packet will be stalled in the case of packet loss.

III. Related Work

A. Interservice communication

Software systems are oftentimes composed of multiple services that need to communicate and exchange data to successfully perform operations. This data exchange process is called interservice communication (ISC). Every system that relies on interservice communication has a mechanism over which data is exchanged [4]. Over the years plenty of ISC mechanisms have been brought to the public eye, each with its own set of benefits and drawbacks. However, we are specifically interested in HTTP via REST [5] as an interservice communication mechanism and its performance in that role.

B. Latency, bandwidth and throughput

When talking about services that communicate over a network, it's important to be familiar with the concepts of latency, bandwidth and throughput in order to understand the potential problems that might arise while data is exchanged between services. Latency refers to the time it takes for a packet of data to travel from its source to its destination. Bandwidth refers to the maximum amount of data that can be transferred over a network in a specific amount of time. Throughput refers to the amount of data that was actually transferred over a network in a specified amount of time. [[19]] An optimal network strives to achieve low latency, high bandwidth and high throughput as it maximizes its performance potential.

C. HTTP/3 in the browser

Even though HTTP/3 has become a proposed standard only recently in 2022, there have been a number of early adopters such as Google and Cloudflare [7] that have started integrating the new revision of the protocol as early as 2020. This early adoption has presented an opportunity for various studies [6]–[9] to be conducted, which had a couple of key elements in common. Firstly, the studies specifically focused on the performance of HTTP/3 compared to its predecessors.

In a study conducted by Cloudflare [7], three key metrics were used to evaluate and compare performance between HTTP/3 and HTTP/2: time to first byte (TTFB), 15KB page load time and 1MB page load time. The TTFB benchmark favored HTTP/3, with a 12,4% average performance increase (176 ms vs. 201 ms). The page load time benchmarks had slightly different results. In the 15KB page benchmark, HTTP/3 had a slightly lower average load time with 443 ms compared to HTTP/2's 458 ms (3.3% performance difference). However, in the 1MB page benchmark, HTTP/2 came out ahead with an average load time of 2.30s compared to 2.33s with HTTP/3 (1.3% performance difference). These differences occur due to the different congestion control algorithms Cloudflare used at the time of testing [7]. BBR v1 [10], the HTTP/2 congestion control algorithm, seems to favor larger transfers. On the other hand, CUBIC [11], the HTTP/3 congestion control algorithm, seems to perform better on smaller transfers.

Cloudflare's study [7] was conducted in 2020, at a time when HTTP/3 was just introduced to the world. In 2022, a study on a much larger scale was conducted by Perna et al. [6]. Their dataset consisted of approximately 14,000 websites with approximately 2,600,000 performed visits over the period of a month. Their experiment focused on two key metrics: onLoad (the time when all elements of a page have been downloaded and parsed), SpeedIndex (time at which the visible parts of the page are displayed) and H3-Delta (derived from onLoad and SpeedIndex) [6]. The experiment was conducted under variable network conditions with latency, bandwidth and packet loss as variables. The highest performance gains that were observed during the experiment were in an environment with medium to high latency (50-200ms). In an environment with no additional latency, content served over HTTP/3 performed better 50% of the time. However, in a higher latency environment, that number jumps up to 71% for 50ms added latency and 77% for 100 and 200ms added latency [6]. Environments with low bandwidth saw limited improvements with regards to the onLoad time. At 1Mb/s bandwidth, approximately 57% of websites load faster with HTTP/3 than with HTTP/2 [6]. Tests with higher packet loss showed no clear trend and at times proved to be quite unpredictable when HTTP/3 was enabled on the server.

Aside from performance, the previously conducted studies [6]–[9] share another key element - they have a focus on HTTP/3 performance in the browser. While the performance analysis in the browser can help us form a hypothesis on how HTTP/3 will perform as an ISC in a distributed system, there are enough varying factors between the two that might prove our hypothesis wrong. When accessing web pages through a browser, the amount of transferred data can exceed several MB due to the amount of content that is sent. In a distributed system, the amount of data transferred between two services over a single request/response transaction is a lot smaller, usually in the range of several hundred bytes. Additionally, the content received by the browser is typically delivered through a content delivery network (CDN). The use of a CDN eliminates the increased latency and lowered bandwidth that occur due to the physical distance between the client and the server. In a distributed system, for a number of reasons, the two services exchanging data might be in two completely different geographical regions. As the last differentiating factor, throughput needs to be highlighted. When we talk about content delivery on a website, we are generally more concerned about the amount of data that is transferred over a specific period of time. On the other hand, with interservice communication we are more concerned about the amount of operations that are performed over a specific period of time. In the context of request/response transactions, this is known as request rate. With this in mind, the term request rate will be used interchangeably with throughput.

D. Transport evaluation

A study by Kyratzis et al. [21] stepped out of the browser and tested the performance of QUIC and TCP over LTE networks. In their study, they used the ns-3 network simulator to simulate various transmission conditions under which the two transport mechanism could be tested. Their study concludes that under good or average conditions, QUIC achieves lower file download times and a better throughput, while under poor conditions, the two protocols perform very similar.

Apart from providing the results of their performance evaluation, they also discuss the effects of 0-RTT handshakes and QUIC streams. When using 0-RTT handshakes in their evaluation scenarios, they noted a quicker connection establishment time compared to TCP handshakes. For TCP handshakes, they recorded a 72 millisecond connection establishment time, while for 0-RTT handshakes, they recorded a 22 millisecond connection establishment time. Additionally, the study reports that the introduction of QUIC streams did lead to higher throughput, however the throughput gain decreases as the number of streams gets higher.

This study demonstrates how QUIC's 0-RTT handshake can decrease the overall latency time for a request/response transaction because of the quicker connection establishment time. However, it is important to highlight that since only the connection time is recorded, the study does not provide a full understanding of how QUIC performs compared to TCP in regards to latency.

IV. Research Methodology

A. Goal

Our objective with this study is to identify and analyze the performance differences between HTTP/3 and HTTP/2. As previously discussed, the structure of HTTP/3 is advantageous for improving performance in environments with high latency and limited bandwidth, leading to better throughput. In light of this, we have identified two following key research questions:

RQ1: *To what extent does the performance differ between HTTP/3 and HTTP/2 in a system with interservice communication over HTTP in regards to latency?*

This research question aims to identify the key differences between HTTP/3 and HTTP/2 with respect to latency. While the majority of previous studies have only assessed the performance of HTTP/3 in high-latency environments using browsers, this question will examine its performance as an Inter-Service Communication (ISC) mechanism under varying latency conditions. Given the structural advantages of HTTP/3, it can be hypothesized that it will outperform HTTP/2 in environments with higher latency.

RQ2: *To what extent does the performance differ between HTTP/3 and HTTP/2 in a system with interservice communication over HTTP in regards to throughput?*

This research question aims to identify the key differences between HTTP/3 and HTTP/2 with respect to throughput. As previously mentioned, websites that support HTTP/3 have been tested for their throughput in prior research. When we talk about throughput when accessing websites, we think of it as "how much data has been transferred", whereas with HTTP/3 as an ISC mechanism, the focus lies on the number of requests completed in a specific amount of time. QUIC implements stream multiplexing, which allows for higher throughput connections if an error were to occur on the connection. With this in mind, we can assume that HTTP/3 will outperform HTTP/2 in regards to throughput.

B. Research methodology to be used

For the purpose of this study, computational experiments will be used to identify the performance differences between HTTP/3 and HTTP/2. In order to answer RQ1 and RQ2, we designed two separate experiments. The first experiment benchmarks the latency and the second one benchmarks throughput, each experiment tests for both HTTP/2 and HTTP/3.

C. Latency Experiment

1) Hypothesis:

H1 : There is a performance difference between HTTP/3 and HTTP/2 with higher latency.

H0: There is no performance difference between HTTP/3 and HTTP/2 with higher latency.

2) Variables: The variables being manipulated in this benchmarking experiment are protocol version, server location and payload size. Protocol version has two variations: HTTP/3 and HTTP/2. Payload size is divided into three variations: small (100 bytes), medium (600 bytes), and large (1KB). The server location has two variations: same region as client and a different region as client. These variables are independent variables as they are being manipulated in order to observe their effect on the dependent variable, which is latency.

Our experiment controls for several variables to ensure accurate results. These control variables include the number of open connections, connection reuse, total number of requests, and benchmarking environment (machine hardware and software configuration). To isolate the performance differences that are affected by multiple connections being opened, we limit our system to use only a single HTTP connection.

Additionally, we use a new connection for each request instead of reusing an existing connection. This allows us to account for the time required to establish a connection. The number of requests is 15,000 requests per benchmark (resulting in 60,000 data points in total). Our extraneous variables include the network and compute instance.

3) Implementation: The benchmarks that will be conducted are the following:

- HTTP/2 with client and server in the same region
- HTTP/3 with client and server in the same region
- HTTP/2 with client and server in different regions
- HTTP/3 with client and server in different regions

Each benchmark will be run three times, with each run sending a batch of 5,000 continuous requests for each payload size (small, medium, or large). This means that the 15,000 requests will be sent in three separate batches, rather than all at once. For each request/response transaction, we will record and persist the time the connection was requested (in milliseconds), the time the connection was established (in milliseconds), the time the request was sent (in milliseconds), the time a response was received (in milliseconds), and the calculated roundtrip of the transaction.

The roundtrip of a transaction, R_i, is defined by the following expression:

$$R_i = restime_i - connreq_i$$

where i is the index of the transaction, $restime_i$ is the response time of the i^{th} transaction and $connreq_i$ is the connection request time of the i^{th} transaction.

D. Throughput Experiment

1) Hypothesis:

H1 : There is a performance difference between HTTP/3 and HTTP/2 with respect to throughput.

H0: There is no performance difference between HTTP/3 and HTTP/2 with respect to throughput.

2) Variables: For this experiment, we manipulate two variables: protocol version and payload size. Protocol version has two variations: HTTP/3 and HTTP/2. Identical to the latency experiment, the payload size is varied across three levels: small, medium, and large. The dependent variable in this experiment is the number of processed requests per second, also known as the request rate.

Our control variables for this experiment are the same as in the latency experiment: number of open connections, connection reuse, number of requests, and benchmarking environment. Similarly, we use a single HTTP connection in order to isolate performance. However, since the focus of this experiment is on throughput, we reuse the connection instead of opening a new one for each request. This allows us to observe how the system processes a continuous flow of requests without the overhead of opening and closing connections for each individual request. The number of requests is 60,000 requests for each benchmark (resulting in 120,000 data points in total). Our extraneous variables include the network and compute instance.

3) Implementation: The benchmarks that will be conducted are the following:

- HTTP/2 with client and server in the same region
- HTTP/3 with client and server in the same region

Each benchmark will be run three times, with each run sending a batch of 20,000 continuous requests for each payload size (small, medium, or large). For each request/response transaction, we will record and persist the time the request was sent (in milliseconds) and the time a response was received (in milliseconds).

The current throughput, T_i, is defined by the expression:

$$T_i = \frac{i}{restime_i - reqtime_1}$$

where i is the index of the current transaction, $restime_i$ is the response time of the current transaction and $reqtime_1$ is the request time of the first transaction. This allows us to determine and graph the progression of throughput for the duration of the benchmark:

$$T = [T_1, T_2, T_3, ..., T_n]$$

where n is the total number of requests made during the experiment.

E. Benchmarking Environment

When designing our benchmarking environment, an important design choice we wanted to satisfy was usage of technologies that an everyday developer is likely to use. Additionally, we wanted our experiments to reflect the technology stack that our partner company uses in their day-to-day work. According to the 2022 developer survey conducted by Stack Overflow [20], JavaScript/TypeScript, Python and Java make up the majority of the most popular programming languages in use. From the same survey, we know that AWS is the most used cloud platform. WirelessCar uses AWS as their cloud provider, with the majority of services written in either Java or Python. At the time of writing, Python had a more mature ecosystem in regards to HTTP/3. Our final technology stack for the benchmarking environment has ended up being AWS with a Python client and server.

The first two components we have in our benchmarking system are two EC2 instances acting as a client and a server. The client is a *t2.small* EC2 instance and it contains a Python script that is responsible for generating requests for different types of experiments that need to be conducted. The server is a *t3.small* EC2 instance and it contains a Hypercorn server with HTTP/2 and HTTP/3 support on separate ports (eg. 8000 for HTTP/2 and 4433 for HTTP/3). The two EC2 instances communicate through a network load balancer that routes TCP and UDP data generated by the client instance to the correct port on the server instance. A benchmark is started by tunneling to the client instance and executing the Python script with parameters describing the experiment. These parameters include the protocol version, number of requests, request payload size, etc.

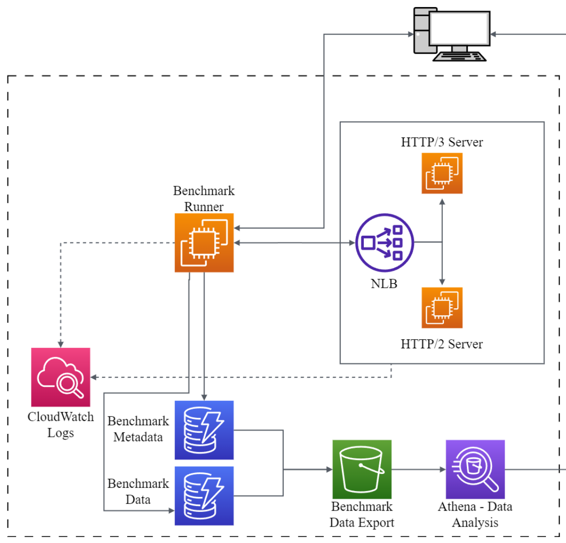


Fig. 6: Benchmarking system

F. Data Collection

We have two DynamoDB tables used for storing information about the conducted experiment (*benchmark-metadata*) and the data generated by the experiment (*benchmark-data*). When an experiment is started, the client stores information about that particular experiment in the *benchmark-metadata* table. The stored information contains properties such as the time the experiment was started, the number of requests performed as a part of the experiment, and a unique identifier for that experiment. After this information is stored, the client instance starts generating request data and sends it to the server through the NLB. When a request/response transaction is completed, data collected from that transaction is stored in the *benchmark-data* table. The data we collect includes the execution time of the request/response transaction, the payload exchanged through the transaction, the time the request was executed, etc.

DynamoDB does not support aggregations, and scanning the entire table to perform manual aggregations on the data can become quite costly. Because of this, we export our collected data into an S3 bucket. With this approach, we only need to scan our DynamoDB tables once instead of every time we want to perform an analysis. After the data is exported to the S3 bucket, it is stored in a JSON-like format called Ion. To make it easier to process data in this format, we use Athena, an AWS analytics service, to parse the exported data and create a SQL data source. Using Athena, we are able to perform various SQL queries on our collected data, perform data aggregation and extract relevant statistics about the data.

Aside from analyzing our data using SQL queries, we use Athena to create Jupyter Notebooks to help us further explore our data. When a SQL query is executed in Athena, the results of that query are stored in an S3 object as a CSV file. Because of this, we are easily able to analyze the results of our queries directly in the Notebooks. Athena integrates directly with Apache Spark, which allows us to perform more advanced analysis of our data. We also use Python packages such as SciPy to perform statistical analysis, as well as Seaborn and Matplotlib packages to visualise our data.

G. Data analysis

After the data collection process for each described experiment, four major groups can be formed by using the protocol version and the client-server distance as the discriminator. For each of these groups, further subgroups can be formed by using the request size as the discriminator. An example of a major group is HTTP/3 with client and server in the same region. A subgroup of this major group is HTTP/3 with client and server in the same region, and only large requests. When performing an analysis, we compare major groups against major groups and subgroups against subgroups. To create a valid group pair for comparison and analysis, the client-server distance and request size must match for both groups, with the protocol version being the variable. This approach allows us to isolate the protocol version as the only factor that might contribute to any differences we might see between the groups.

For every group pair that is identified, the key metric of the experiment that the groups are associated to is used as a basis for further analysis. For experiment 1, the key metric is latency, and for experiment 2, the key metric is throughput. When a group pair is analyzed, their minimum, maximum and mean key metric values are calculated.

1) Pilot run: Before committing to a full experiment cycle, we conducted a pilot run to gain confidence in the functionality of our benchmarking system. The benchmarking system shown in figure 6 is the system that was used to gather results for our research. However, this system originally had a significantly different design during our pilot run. The pilot run allowed us to identify outstanding problems within our system and remedy them before conducting the full set of experiments.

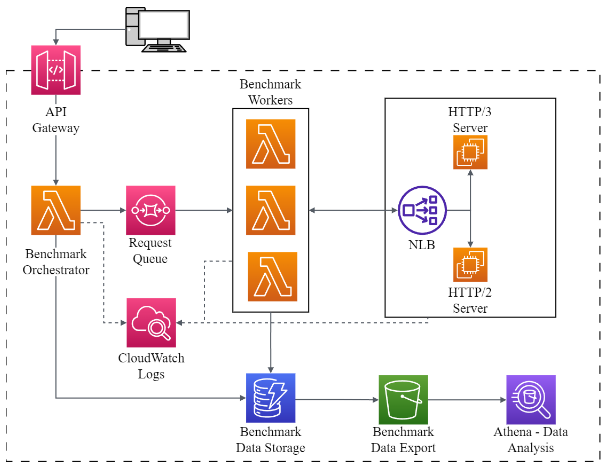


Fig. 7: Legacy benchmarking system

Figure 7 shows the benchmarking system we used during our pilot run. Many parts of the system are identical to the current system, such as the EC2 server instance, the DynamoDB tables and the Athena instance for data analysis. A major part of the system that is different are the components responsible for orchestrating and executing different experiments.

Previously, the responsibilities of orchestrating and executing experiments were split between two AWS Lambda functions that we refer to as *Benchmark Orchestrator* and *Benchmark Worker*. The *Benchmark Orchestrator* would receive experiment parameters through an API Gateway, from which experiment information would be stored in a DynamoDB table, and request data would be generated and sent to a queue. The *Benchmark Worker* would receive the request data from the queue and proceed to execute the requests needed for the experiment. With this design, we experienced several problems that ranged from a minor inconvenience to a problem that made it impossible to continue with the experiments.

The first problem we discovered early on is the API gateway timeout when a large number of requests was being executed. The orchestrator lambda is responsible for returning a response to the API gateway within 30 seconds of it being called. If we tried conducting an experiment with a sufficiently large number of requests, the lambda would take longer than 30 seconds to generate those requests and send them to the queue. This was a minor inconvenience as it only prevented us from seeing when all of the requests were generated and did not actually stop the execution of the experiment. This alone was not enough to justify a redesign of the system at that point in time.

The second problem we discovered much later into the pilot run is the inability to execute HTTP/3 requests from an AWS Lambda. Before trying to create HTTP/3 experiments, we started off with HTTP/2 as it is a more established and more familiar technology. This system design worked well for HTTP/2 experiments, but was completely unusable for HTTP/3. The Python package we are using to create HTTP/3 request, *aioquic*, requires the host to support IPv6, which AWS Lambda at this point in time does not.

Because of this problem, the orchestrator and worker lambdas were replaced by a single EC2 instance which does support IPv6. Because the orchestration and experiment execution were contained within the same instance, the request queue was removed as a consequence. To further simplify the system, we also removed the API gateway and resorted to tunneling into the instance via SSH.

After we were able to conduct the first set of HTTP/3 experiments, we noticed that the HTTP/3 server crashed from time to time. We attributed these crashes to the limited amount of processing resources on the EC2 instance used to host the server. Originally, the server was hosted on a *t2.small* instance. We resolved this issue by upgrading to a *t3.small* instance, which provided us with enough processing resources to not cause any more crashes.

H. Limitations

Our computational experiments are susceptible to threats to external validity due to the nature of our isolated and controlled benchmarking environment. By excluding real-life factors such as network congestion and server load, our results may not accurately reflect the performance and behavior of a system in a practical, real-world setting. Furthermore, the representativeness of our payload sizes may be limited as there exists a virtually infinite number of possible payload variations that are transmitted between services. Given the constraints of time and cost, our experiments are restricted to a limited number of requests. This limitation poses a risk to the external validity of our findings, as we are unable to thoroughly test the protocols to the same extent they are utilized in real-life situations.

The architecture of our benchmarking environment will inevitably influence performance, and due to its specific nature in our test case, generalizing the results may pose challenges. Furthermore, the regions selected for the latency experiment cannot be easily generalized, considering the number of other client/server region combinations.

Due to the variability of network conditions, there is a risk to the reliability of the results when conducting the experiments. However, by ensuring a large enough number of requests, the risk of obtaining significantly skewed data due to extraneous factors is mitigated.

Potential threats to conclusion validity include running an insufficient number of experiment iterations, which may fail to uncover cases contradicting the hypotheses. Additionally, human error during data analysis is another concern, as it relies on human handling.

V. Results

This chapter presents the results obtained from the two experiments conducted in this study. Firstly, the results pertaining to the latency performance of both HTTP/3 and HTTP/2 are presented. This will be followed by the presentation of the throughput results for each respective protocol

A. Latency

Figure 8 is a histogram showing the distribution of all recorded latencies for both HTTP/2 and HTTP/3 with client and server in the same region (eu-west-1). The blue bins show latencies for HTTP/2 requests and the orange bins show latencies for HTTP/3 requests. Each set of bins shows the latency range for a protocol version. Each individual bin shows the number of requests that resulted in a specific latency.

The recorded HTTP/2 latencies range from 29 to 90 milliseconds, with the majority of latencies ranging from 60 to 65 milliseconds. The recorded HTTP/3 latencies range from 22 to 65 milliseconds, with the majority of latencies ranging from 24 to 36 milliseconds.

Figure 9 is a histogram showing the distribution of all recorded latencies for both HTTP/2 and HTTP/3 with client (eu-west-1) and server (us-east-1) in different regions. The blue bins show latencies for HTTP/2 requests and the orange

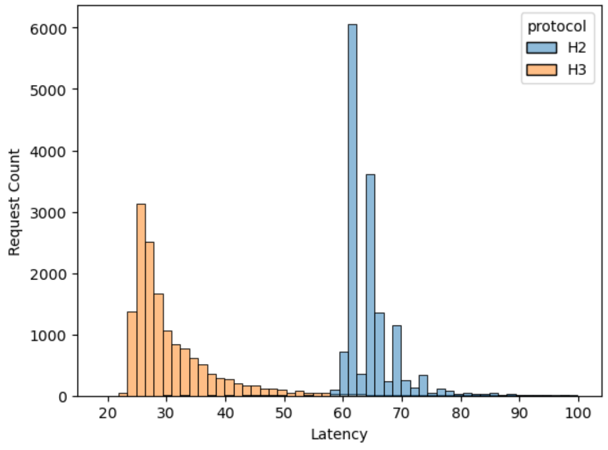


Fig. 8: Latency - Server: eu-west-1

bins show latencies for HTTP/3 requests. Each set of bins shows the latency range for a protocol version. Each individual bin shows the number of requests that resulted in a specific latency.

The recorded HTTP/2 latencies range from 285 to 392 milliseconds, with the majority of latencies ranging from 320 to 335 milliseconds. The recorded HTTP/3 latencies range from 156 to 282 milliseconds, with the majority of latencies ranging from 156 to 165 milliseconds.

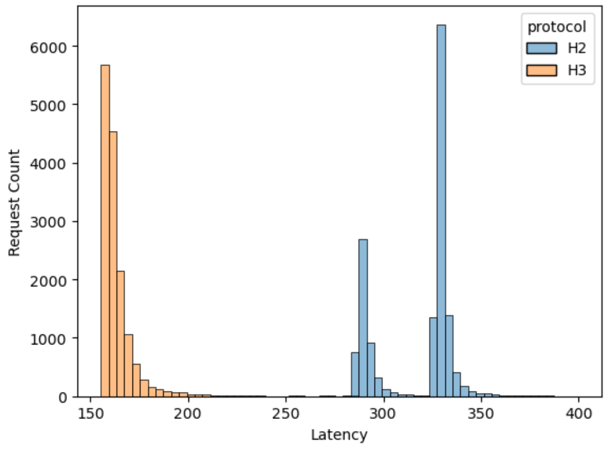


Fig. 9: Latency - Server: us-east-1

B. Throughput

Figure 10 is a line graph showing the progression of throughput for HTTP/2 over 20,000 requests. Each line in the graph shows this progression for the payload size that was used during the benchmark.

At the beginning of the benchmark, we observe very minor oscillations in throughput for each payload size. After approximately 3000 performed requests, the throughput stabilizes at 20 requests per second for each payload size.

Figure 11 is a line graph showing the progression of throughput for HTTP/3 over 20,000 requests. Each line in the graph shows this progression for the payload size that was used during the benchmark. We observe that each payload size exhibited slightly different throughput behaviors.

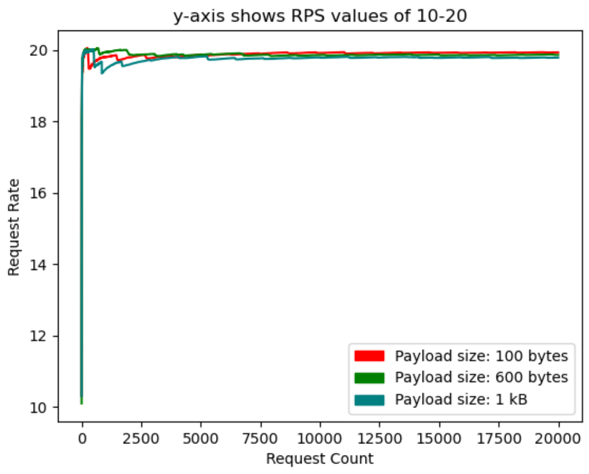


Fig. 10: Throughput - HTTP/2

For the 100-byte payload size, the request rate showed fluctuations throughout the benchmark. Initially, the request rate was observed to be 140 requests per second, which declined rapidly to 100 requests per second for the next 2500 requests. Subsequently, an incline in the request rate was noticed until 7500 requests, reaching 110 requests per second. However, after this point, a drop to 100 requests per second occurred, followed by a steady decline toward 85 requests per second. The minimum and maximum request rates recorded during the observation period were 85 requests per second and 141 requests per second, respectively. The average request rate during the observation period was 102 requests per second.

For the 600-byte payload size, the request rate ranged from 120 to 130 requests per second for the first 7500 requests. For the remainder of the benchmark, a steady decline was observed, with a minimum request rate of 86 requests per second. The minimum and maximum request rates recorded during the observation period were 86 requests per second and 128 requests per second, respectively. The average request rate during the observation period was 107 requests per second.

For the 1 kB payload size, the request rate showed a gradual increase from 120 to 140 requests per second during the first 7500 requests. Until 17000 requests, the request rate gradually decreased to approximately 120 requests per second. For the remainder of the benchmark, a sharp decline to 80 requests per second occurred. The minimum and maximum request rates recorded during the observation period were 73 requests per second and 138 requests per second, respectively. The average request rate during the observation period was 118 requests per second.

VI. Discussion

The results of our experiment suggest that there is a performance benefit when using HTTP/3, in regards to both latency and throughput. While it is interesting to see the

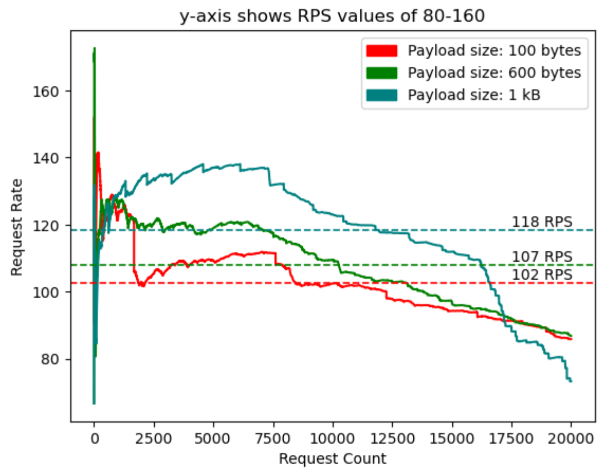


Fig. 11: Throughput - HTTP/3

performance margin between the two protocols, the fact that there is a performance difference is not surprising. Previously conducted studies [6]–[9] found an increase in performance when HTTP/3 and HTTP/2 were matched up in identical scenarios. Said studies specifically focused on performance in the browser, leaving a research gap that our study explored.

In our latency experiments, we noted an average 32 millisecond difference in transaction times when the client and server are in the same region. When placing the client and the server further apart, we noted an average 167 millisecond difference in transaction times. The increase in transaction time difference as the client and server are further apart is inline with the 0-RTT handshake improvements introduced in HTTP/3, as well as the results of a study conducted by Perna et al. [6]. In their study, they observed an increase in performance difference as latency got higher. While in their experiment latency is introduced artificially, the same effect is achieved by moving the client and server physically further apart in our experiments.

Our throughput experiments show a very clear performance difference between the two protocols. With HTTP/2, we were not able to achieve a request rate higher than 20 requests per second. On the other hand, when using HTTP/3 we are able to achieve a request rate as high as 140 requests per second. When we analyze the request rate progression for HTTP/3, we notice a lot of fluctuations throughout the benchmark, unlike HTTP/2 which keeps a stable request rate throughout the entire benchmark. This however does not mean that HTTP/3 as a protocol is unstable, but rather that our benchmark is CPU bound. If we were to run our request rate benchmark on hardware with more computational resources, we would see a more stable request rate throughout the whole benchmark.

Our experiments show a clear difference between the two protocols in regard to performance, but just looking at raw performance is not enough to determine if HTTP/3 is a viable option for practical use at this point in time. Our experiments model a simplistic system with the sole purpose of measuring performance. In reality, systems exhibit far more degrees of complexity than the one used for our experiments. As complexity increases, many more factors need to be considered when committing to a new technology, especially if that technology is as young as the one we explored in our research. Before implementing HTTP/3 in a more complex system we need to ask ourselves, does the benefit of more performance outweigh the cost of implementation. Unfortunately, the answer to this question is not universal. But what we can do is discuss aspects of HTTP/3 and the ecosystem surrounding it, which we have noticed while designing our experiments, that might limit companies like WirelessCar in their efforts to implement it in their systems.

A. *Cloud limitations*

At the time of writing, support for HTTP/3 on AWS is quite limited. Amazon CloudFront does have native HTTP/3 support, however its functionality is not relevant for the use case we're exploring. Communication between two services built on top of AWS infrastructure is most likely to occur through a load balancer. The Application Load Balancer on AWS does not have native HTTP/3 support, however it is possible to route HTTP/3 traffic through a Network Load Balancer with a UDP listener.

Support for HTTP/3 in Lambda applications is also very limited. Most Python packages we have explored require IPv6 to be enabled on the system the package is being used on. For the moment, Lambda environments do not support IPv6.

B. *Ecosystem maturity*

The HTTP/3 ecosystem within Python is at a very early stage of maturity. From a client perspective, there is a lack of packages with HTTP/3 as their primary focus. However, with the use of QUIC packages, such as *aioquic*, it is possible to implement a standalone HTTP/3 client or add HTTP/3 support to already existing HTTP packages, such as *httpx*. From a server perspective, support for HTTP/3 in existing packages seems to be nonexistent at worst and experimental at best. For our benchmarking environment, we used Hypercorn, a Python web server with experimental HTTP/3 support.

VII. Conclusion

In this study, we conducted a performance analysis of HTTP/3 in the role of an interservice communication mechanism. HTTP/3 is a relatively new technology, with a high research potential. Studies prior to ours mainly focused on its performance in the browser, with the results being quite favourable for HTTP/3. With prior research being browser-focused, we were curious if we would come to similar results within a distributed system. To reach our goal, we conducted a computational experiment designed to compare the performance of HTTP/2 and HTTP/3 in identical environments. Throughout the experiment, several variables, such as the request payload size and the client-server distance, were manipulated. This provided us with an overview of how both protocols perform under different conditions. The results of our experiment suggest that HTTP/3 has increased performance in both short-distance and long-distance networks across multiple request payload sizes. However, these results should not be taken at face value, as they were obtained in a controlled environment under perfect network conditions. This is only a small step in discovering the true potential of HTTP/3 in a distributed system, but a lot remains to be researched.

The next step would be to conduct a case study by implementing HTTP/3 within a production environment, allowing for observation of its performance differences in a real-life setting over an extended duration. This approach will consider factors that were not accounted for in our experiments, leading to more accurate and comprehensive results. By embracing this method, we can gain deeper insights into the practical implications and effectiveness of HTTP/3.

References

[1] Martin Thomson and Cory Benfield. 2022. " HTTP/2." IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc9113.

[2] Mike Bishop. 2022. "HTTP/3." IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc9114.

[3] Jana Iyengar and Martin Thomson. 2021. "QUIC: A UDP-Based Multiplexed and Secure Transport." IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc9000.

[4] Michael Spier and Elliott Organick. 1969. "The MULTICS interprocess communication facility." Proc. ACM Second Symp. on Operating Systems Principles, Princeton University

[5] Roy Fielding. 2000. "Architectural styles and the design of network-based software architectures." University of California, Irvine

[6] Gianluca Perna, Martino Trevisan, Danilo Giordano, and Idilio Drago. 2022. "A first look at HTTP/3 adoption and performance." Computer Communications 187. https://www.sciencedirect.com/science/article/pii/S0140366422000421.

[7] Sreeni Tellakula. 2020. "Comparing HTTP/3 vs. HTTP/2 Performance." Cloudflare. https://blog.cloudflare.com/http-3-vs-http-2.

[8] Konrad Wolsing, Jan Rüth, Klaus Wehrle, and Oliver Hohlfeld. 2019. "A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC." RWTH Aachen University, Germany

[9] Darius Saif, Chung-Horng Lung, and Ashraf Matrawy. 2021. "An Early Benchmark of Quality of Experience Between HTTP/2 and HTTP/3 using Lighthouse." Carleton University, Department of Systems and Computer Engineering

[10] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yegane, Ian Swett, and Van Jacobson. 2022. "BBR Congestion Control." Work in progress. IETF Datatracker. https://datatracker.ietf.org/doc/html/draft-cardwell-iccrg-bbr-congestion-control-02.

[11] Injong Rhee, Lisong Xu, Sangtae Ha, Alexander Zimmermann, Lars Eggert, and Richard Scheffenegger. 2018. "CUBIC for Fast Long-Distance Networks." IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc8312.

[12] Roy Fielding, Mark Nottingham and Julian Reschke. 2022. "HTTP Semantics" IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc9110.

[13] Roberto Peon and Herve Ruellan. 2015. "HPACK: Header Compression for HTTP/2" IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc7541.

[14] Charles Krasic, Mike Bishop and Alan Frindell. 2022. "QPACK: Field Compression for HTTP/3" IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc9204.

[15] Adam Langley et al. 2017. "The QUIC Transport Protocol: Design and Internet-Scale Deployment" Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3098822.3098842.

[16] Wesley Eddy. 2022. "Transmission Control Protocol (TCP)" IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc9293.

[17] Eric Rescorla and Tim Dierks. 2008. "The Transport Layer Security (TLS) Protocol Version 1.2" IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc5246.

[18] Eric Rescorla. 2018. "The Transport Layer Security (TLS) Protocol Version 1.3" IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc8446.

[19] Barrie Sosinsky. 2009. "Networking Bible". Wiley.

[20] Stack Overflow. 2022. "Stack Overflow Developer Survery 2022". https://survey.stackoverflow.co/2022/

[21] Apostolos I. Kyratzis and Panayotis G. Cottis. 2022. "QUIC vs TCP: A Performance Evaluation over LTE with NS-3" Communications and Network, 2022, 14, 12-22.